

The Underground PHP and Oracle Manual

CHRISTOPHER JONES AND ALISON HOLLOWAY

Updated for Oracle Database Express Edition 11g Release 2

ORACLE®

The Underground PHP and Oracle® Manual, Release 2.0, December 2012.
Copyright © 2008, 2012 Oracle. All rights reserved.

The latest edition of this book is available free online at:
<http://www.oracle.com/technetwork/topics/php/underground-php-oracle-manual-098250.html>

Authors: Christopher Jones and Alison Holloway

Contributors and acknowledgments: Vladimir Barriere, Luxi Chidambaran, Robert Clevenger, Antony Dovgal, Wez Furlong, Maurice Gamanho, Deepak Goel, Sue Harper, Manuel Hoßfeld, Ken Jacobs, Srinath Krishnaswamy, Shoaib Lari, Simon Law, Krishna Mohan, Craig Mohrman, Chuck Murray, Kevin Neel, Kant Patel, Charles Poulsen, Karthik Rajan, Richard Rendell, Roy Rossebo, Jeffrey Rubinoff, Michael Sekurski, Sreekumar Seshadri, Mohammad Sowdagar, Makoto Tozawa, Todd Trichler, Simon Watt, Zahi, Shuping Zhou.

The chapter on globalization is derived from the *Oracle Database Express Edition 2 Day Plus PHP Developer Guide*. The chapter on connection pooling is derived from the Oracle white paper *PHP Scalability and High Availability*. The discussion on Client Identifiers is from an OTN article *PHP Web Auditing, Authorization and Monitoring*. The foundation for the chapter on the NetBeans IDE was contributed by Jeffrey Rubinoff. The Oracle Solaris content was contributed by Craig Mohrman. We gratefully acknowledge all the people who contributed to the creation of this book.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

License Restrictions Warranty/Consequential Damages Disclaimer

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

Warranty Disclaimer

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

Restricted Rights Notice

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

Hazardous Applications Notice

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Third-Party Content, Products, and Services Disclaimer

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

CONTENTS

Chapter 1 Introduction.....	1
PHP and Oracle.....	1
Introduction to Oracle Database.....	1
Introduction to PHP.....	1
Chapter 2 Getting Started With PHP.....	5
Creating and Editing PHP Scripts.....	5
PHP Syntax Overview.....	5
Running PHP Scripts.....	8
Running Scripts Via a Browser.....	8
Running Scripts Using the PHP Development Web Server.....	8
Running Scripts With Command-Line PHP.....	9
Debugging PHP Scripts.....	9
Chapter 3 PHP Oracle Extensions.....	11
PHP Oracle Extensions.....	11
PHP OCI8 Extension.....	11
Getting the OCI8 Extension.....	12
OCI8 and Oracle Database Installation Options.....	13
PHP OCI8 History.....	14
PHP PDO Extension.....	15
Getting the PDO Extension.....	15
PHP Frameworks.....	15
PHP Database Abstraction Libraries.....	15
ADODb.....	16
PEAR DB.....	16
PEAR MDB2.....	16
The PHP Release Cycle.....	17
Chapter 4 Installing Oracle Database 11g Express Edition.....	19
Oracle Database Editions.....	19
Oracle Database 11g XE.....	19
Installing Oracle Database 11g XE on Linux.....	20
Installing Oracle Database 11g XE on Windows.....	21
Managing the Oracle Database 11g XE.....	23
Setting the Oracle Database 11g XE Environment Variables on Linux.....	23
Starting and Stopping the Listener and Database.....	24
Basic Monitoring of Oracle Database 11g XE.....	25
Oracle Terminology.....	27
Chapter 5 SQL With Oracle Database.....	29
Oracle SQL*Plus.....	29

Setting the Environment for SQL*Plus.....	29
Starting SQL*Plus.....	29
Using SQL*Plus to Unlock the HR Schema for Demonstration Table Access.....	30
Creating Database Users in SQL*Plus.....	31
Executing SQL and PL/SQL Statements in SQL*Plus.....	31
Controlling Query Output in SQL*Plus.....	32
Running SQL Scripts in SQL*Plus.....	32
Table Metadata in SQL*Plus.....	33
Starting and Stopping the Database With SQL*Plus.....	33
Oracle Application Express.....	34
Creating an Application Express Workspace.....	35
Logging In To Oracle Application Express.....	37
Creating Database Objects in Application Express.....	38
Working With SQL in Application Express.....	41
Creating a PL/SQL Procedure in Application Express.....	43
Oracle SQL Developer.....	46
Creating a Database Connection in SQL Developer.....	46
Editing Data in SQL Developer.....	47
Creating a Table in SQL Developer.....	48
Executing a SQL Query in SQL Developer.....	49
Editing, Compiling and Running PL/SQL in SQL Developer.....	51
Running Reports in SQL Developer.....	52
Creating Reports in SQL Developer.....	53
Chapter 6 NetBeans IDE for PHP.....	57
NetBeans Installation	57
NetBeans Editor Features	57
Development Processes in NetBeans	59
OCI8 Support in NetBeans	59
Oracle Database Support in NetBeans	59
Getting Help for NetBeans	60
Chapter 7 Installing Apache HTTP Server.....	63
Apache HTTP Server Packages on Oracle Linux.....	63
Building Apache HTTP Server on Linux.....	63
Configuring Apache HTTP Server on Linux.....	64
Setting the Apache Server Name.....	64
Setting up an Apache User Directory on Linux.....	64
Environment Variables for PHP in Apache on Oracle Linux.....	65
Permissions for PHP OCI8 in Apache on Oracle Linux.....	66
Installing Apache HTTP Server on Windows.....	66
Starting and Stopping Apache HTTP Server on Windows.....	67
Testing Apache HTTP Server.....	67

Chapter 8 Installing and Configuring PHP.....	69
Installing PHP on Linux.....	69
Installing Linux PHP Packages.....	69
Installing Zend Server Packages on Linux.....	70
Compiling PHP as an Apache Module on Linux.....	71
Installing Oracle Instant Client on Linux for the OCI8 Extension.....	73
Configuration Options for Compiling OCI8 With PHP on Linux.....	74
Installing OCI8 on Linux as a Shared Extension Using PECL.....	75
Manually Installing OCI8 on Linux as a Shared Extension.....	76
Setting the Oracle Environment for PHP on Linux.....	77
Signal Handling and Defunct Processes on Linux.....	80
Using PHP-FPM With Apache on Linux.....	81
Installing PHP With OCI8 on Windows.....	84
Oracle Libraries on Windows.....	84
Installing PHP on Windows.....	85
Installing OCI8 With Oracle HTTP Server.....	86
Installing OCI8 With Oracle HTTP Server 11g on Linux.....	87
Installing OCI8 With Oracle HTTP Server 10g on Linux.....	89
Installing the PDO Extension.....	91
Installing PDO_OCI on Linux.....	91
Installing PDO_OCI on Windows.....	92
Checking OCI8 and PDO_OCI Installation.....	92
Chapter 9 Installing PHP and Apache on Oracle Solaris.....	95
Installing Apache on Oracle Solaris 11.1.....	95
Installing PHP on Oracle Solaris 11.1.....	96
Changing the Version of PHP used by Apache.....	98
Installing Oracle Instant Client on Oracle Solaris 11.1.....	99
Installing OCI8 on Oracle Solaris 11.1.....	99
Chapter 10 Connecting to Oracle Using OCI8.....	101
Oracle Connection Example.....	101
Oracle Connection Types.....	102
Standard Connections.....	103
Unique Connections.....	103
Persistent Connections.....	103
Oracle Database Name Connection Identifiers.....	104
Easy Connect String.....	104
Database Connect Descriptor String.....	105
Database Connect Name.....	106
Commonly Seen Connection and Environment Errors.....	107
Closing Oracle Connections.....	111
Closing Connections and Variable Scope.....	111

Transactions and Connections.....	112
Session State With Persistent Connections.....	113
Optional Connection Parameters.....	113
Connection Character Set.....	113
Connection Session Mode.....	114
Password Handling in PHP Applications.....	116
External Authentication With PHP OCI8.....	116
Tuning Connections to Build Scalable Systems.....	118
Use the Best Connection Function.....	118
Use Connection Pooling.....	119
Minimize the number of database user credentials used.....	119
Connect With a Character Set.....	119
Tune the AUDSESS\$ Sequence Generator.....	120
Do Not Set the Date or Numeric Format Unnecessarily.....	120
Manage Persistent Connections.....	121
Changing the Database Password.....	123
Changing Passwords on Demand.....	123
Changing Expired Passwords.....	124
Authorization and Authentication With Client Identifiers.....	126
Setting Client Identifiers.....	126
A Sample Application Using Client Identifiers.....	127
Using a Client Identifier in PHP for Auditing.....	133
Using a Client Identifier in PHP With a VPD for Restricting Data Access.....	135
Using a Client Identifier in PHP for Monitoring and Tracing.....	138
Client Identifier Summary.....	142
Oracle Network Services and PHP.....	142
Connection Rate Limiting.....	142
Setting Connection Timeouts.....	143
Configuring Authentication Methods.....	143
Detecting Dead PHP Apache Sessions.....	144
Detecting Dead Database Servers.....	144
Other Oracle Net Optimizations.....	144
Tracing Oracle Net.....	144
Chapter 11 Executing SQL Statements With OCI8.....	147
SQL Statement Execution Steps.....	147
Query Example.....	147
Quoting SQL Statement Text.....	148
Oracle Data Types.....	149
Fetch Functions.....	150
Fetching to a Numeric Array.....	150
Fetching to an Associative Array.....	151

Fetching Case Sensitive Column Names to an Associative Array.....	152
Duplicate Column Names and Associative Arrays.....	152
Fetching to an Object.....	152
Defining Output Variables.....	153
Fetching Nested Cursors.....	154
Fetching and Working With Numbers.....	155
Fetching and Working With Dates.....	156
Insert, Update, Delete, Create and Drop in PHP OCI8.....	157
Transactions in PHP OCI8.....	157
Autonomous Transactions.....	159
The Transactional Behavior of Multiple Connections.....	160
PHP Error Handling.....	161
Handling PHP OCI8 Errors.....	161
Using Bind Variables in Prepared Statements.....	164
Binding in a “for” Loop.....	167
Binding With LIKE and REGEXP_LIKE Clauses.....	168
Binding Multiple Values in an IN Clause.....	169
Using Bind Variables to Fetch Data.....	171
Binding in an ORDER BY Clause.....	172
Using ROWID Bind Variables.....	172
Improving Performance by Prefetching and Caching.....	173
Tuning the Prefetch Size.....	173
Tuning the Statement Cache Size.....	175
Using the Server and Client Query Result Caches.....	176
Monitoring OCI8 SQL Statements.....	178
OCI8 Driver Identification.....	179
Setting Application Information in PHP OCI8.....	179
LIMIT, Auto-Increment, Last Insert ID and Multiple Inserts.....	180
Limiting Rows and Creating Paged Datasets.....	181
Auto-Increment Columns.....	183
Getting the Last Insert ID.....	184
Inserting Multiple Values.....	185
Exploring Oracle.....	185
Case Insensitive Data Matching in Queries.....	185
Analytic Functions in SQL.....	186
External Tables.....	186
Chapter 12 Using PL/SQL With OCI8.....	187
PL/SQL Overview.....	187
Blocks, Procedures, Packages and Triggers.....	188
Anonymous Blocks.....	188
Stored Procedures and Functions.....	188

Packages.....	189
Triggers.....	190
Creating PL/SQL Stored Procedures in PHP.....	190
End of Line Terminators in PL/SQL With Windows PHP.....	191
Calling PL/SQL Code.....	191
Calling PL/SQL Procedures in PHP.....	191
Calling PL/SQL Functions in PHP.....	192
Binding Unsupported PL/SQL Types.....	193
Array Binding and PL/SQL Bulk Processing.....	194
PL/SQL Success With Information Warnings.....	196
Using REF CURSORS for Result Sets.....	197
Closing Cursors.....	200
Prefetching From REF CURSORS and Nested Cursors for Performance.....	201
Converting from REF CURSOR to PIPELINED Results.....	202
Oracle Collections in PHP.....	203
Using PL/SQL and Oracle Object Types in PHP.....	205
Using a PIPELINED Function.....	206
Using a REF CURSOR.....	207
Using an Array Bind.....	208
Using OCI8 Collection Functions.....	209
Getting Output With DBMS_OUTPUT.....	210
PL/SQL Backtraces in a PL/SQL Exception Handler.....	213
PL/SQL Function Result Cache.....	214
Using Oracle Locator for Spatial Mapping.....	214
Inserting Locator Data.....	215
Queries Returning Scalar Values.....	215
Selecting Vertices Using SDO_UTIL.GETVERTICES.....	216
Using a Custom Function.....	217
Scheduling Background or Long Running Operations.....	218
Oracle Streams Advanced Queuing.....	221
Reusing Procedures Written for MOD_PLSQL.....	224
Easy PL/SQL Upgrades With Edition Based Redefinition.....	226
Database Transactions Across Stateless Web Requests.....	230
Chapter 13 Using Large Objects in OCI8.....	233
Working With LOBs.....	233
Inserting and Updating LOBs.....	233
Fetching LOBs.....	234
Temporary LOBs.....	235
Uploading and Displaying an Image.....	236
LOBs and PL/SQL procedures.....	237
Other LOB Methods.....	239

Working With BFILEs.....	240
Chapter 14 Using XML With Oracle and PHP.....	245
Fetching Relational Rows as XML.....	245
Fetching Rows as Fully Formed XML.....	246
Using the SimpleXML Extension in PHP.....	247
Fetching XMLType Columns.....	248
Inserting Into XMLType Columns.....	250
Fetching an XMLType from a PL/SQL Function.....	252
XQuery XML Query Language.....	252
Accessing Data Over HTTP With XML DB.....	254
Chapter 15 PHP Connection Pooling and High Availability.....	257
Database Resident Connection Pooling.....	257
How DRCP Works.....	258
PHP OCI8 Connections and DRCP.....	260
When to use DRCP.....	262
Sharing the Server Pool.....	263
Using DRCP in PHP.....	264
Configuring and Enabling the Pool.....	265
Configuring PHP for DRCP.....	267
Application Deployment for DRCP.....	268
Monitoring DRCP.....	271
DBA_CPOOL_INFO View.....	271
V\$PROCESS and V\$SESSION Views.....	272
V\$CPOOL_STATS View.....	272
V\$CPOOL_CC_STATS View.....	273
High Availability With FAN and RAC.....	276
Configuring FAN Events in the Database.....	276
Configuring PHP for FAN.....	276
Application Deployment for FAN.....	276
RAC Connection Load Balancing With PHP.....	278
Chapter 16 PHP and TimesTen In-Memory Database.....	279
Installing TimesTen on Linux.....	279
Managing TimesTen.....	280
Creating the TimesTen Sample Database.....	281
Installing Apache and PHP for TimesTen.....	282
Checking the Installation.....	282
Connecting to TimesTen With PHP OCI8.....	283
Configuring TimesTen.....	283
Chapter 17 PHP and Oracle Tuxedo.....	285
Installing Tuxedo 11.1 and SALT for PHP Web Applications.....	285
Installing Oracle Tuxedo.....	286

Installing Oracle SALT.....	290
Installing PHP for Oracle Tuxedo.....	293
Installing Oracle Tuxedo into Apache.....	293
Configuring Oracle Tuxedo for PHP.....	294
Starting and Managing Tuxedo.....	296
Verifying PHP and Tuxedo.....	298
Chapter 18 Globalization.....	301
Establishing the Environment Between Oracle and PHP.....	301
Setting the Language, Territory and Character Set With NLS_LANG.....	301
Setting the Oracle Number Format With NLS_NUMERIC_CHARACTERS.....	303
Setting the Oracle Date Format With NLS_DATE_FORMAT.....	305
Setting the Default Session Time Zone With ORA_SDTZ.....	307
Manipulating Strings.....	307
Determining the Locale of the User.....	307
Developing Locale Awareness.....	308
Encoding HTML Pages.....	309
Specifying the Page Encoding for HTML Pages.....	309
Specifying the Encoding in the HTTP Header.....	309
Specifying the Encoding in the HTML Page Header.....	309
Specifying the Page Encoding in PHP.....	309
Organizing the Content of HTML Pages for Translation.....	310
Strings in PHP.....	310
Static Files.....	310
Data from the Database.....	310
Presenting Data Using Conventions Expected by the User.....	310
Oracle Linguistic Sorts.....	311
Oracle Error Messages.....	312
Chapter 19 Testing PHP and the OCI8 Extension.....	313
Running OCI8 Tests.....	313
Running a Single Test.....	315
Tests that Fail.....	315
Creating OCI8 Tests	315
OCI8 Test Helper Scripts.....	317
Configuring the Database For Testing.....	317
Testing PHP Applications.....	319
Appendix A Tracing OCI8 Internals.....	321
Enabling OCI8 Debugging output.....	321
Appendix B OCI8 php.ini Parameters.....	323
Enabling PHP OCI8 in php.ini.....	323
PHP OCI8 php.ini Parameters.....	324
Appendix C OCI8 Function Names in PHP 4 and PHP 5.....	327

Appendix D The Obsolete Oracle Extension.....	331
Oracle and OCI8 Comparison.....	331
Appendix E Resources.....	335
General Information and Forums.....	335
Oracle Documentation and Whitepapers.....	335
Selected PHP and Oracle Books.....	336
Software and Source Code.....	337
PHP Links.....	338
Glossary.....	339

INTRODUCTION

This book is designed to bridge the gap between the many PHP scripting language and the many Oracle Database books available. It contains unique material about PHP's OCI8 extension for Oracle Database, and about other components in the PHP-Oracle ecosystem. It shows PHP developers how to use PHP and Oracle together, efficiently and easily.

The *Underground PHP and Oracle Manual* is not a complete PHP syntax or Oracle SQL guide. It also does not describe overall application architecture. For these, Oracle documentation is freely available online, as are extensive PHP documentation and application development resources. For newcomers we suggest reading the *Oracle Database Express Edition 2 Day + PHP Developer's Guide* which walks through building a PHP application.

Since the first release of the *Underground PHP and Oracle Manual*, many commercial books on PHP and Oracle have been published. They are worthwhile additions to your library. Each has a different viewpoint and shows something new about the technologies.

PHP and Oracle

A number of Oracle products contribute to the rich ecosystem for PHP applications. These include the NetBeans IDE, Oracle Database, Oracle TimesTen In-Memory Database, Oracle Tuxedo, Oracle Application Server, and various SQL development tools. There are many users of PHP and Oracle all around the world using these technologies.

Introduction to Oracle Database

The main Oracle product discussed in this book is Oracle Database. It is well known for its scalability, reliability and features. It is the leading relational database and is available on many platforms. The installation discussion highlights the free Oracle Database 11g Express Edition (known as "Oracle XE"). Since Oracle XE is a subset of the full Oracle Database bundle, the PHP applications you write for Oracle XE can be run, without change, against all other editions of the Oracle 11g database. PHP also supports some of the advanced functionality of Oracle Database 11gR2 Enterprise Edition; this is covered here too.

Most of the core information in the book is relevant to previous versions of Oracle Database.

Introduction to PHP

PHP is a hugely popular, interpreted scripting language commonly used for web applications. It is open source, free, and has a BSD-style license making it corporation-friendly. PHP is perfect for rapidly developing applications both big and small. It powers millions of web sites on the Internet and has a huge user community behind it. It runs on many platforms.

The language is dynamically typed and easy to use. PHP 5 introduced strong object orientated capabilities. PHP comes with many extensions offering all kinds of functionality from system operations to numerical processing. PHP includes the OCI8 extension which, when linked with Oracle client libraries, enables access to Oracle Database.

Introduction

A PHP command line interface (CLI) can be used to execute PHP scripts from an operating system shell window, or a simple in-built development web server can be used to serve script output to a browser. This is common for quick development testing.

For production applications, PHP is typically installed as an Apache module, or run by a web server using FastCGI.

Consider the script *hello.php*:

Script 1: hello.php

```
<?php  
echo "<p>hello</p>";  
query_cities();  
  
function query_cities() {  
    $c = oci_connect("hr", "welcome", "localhost/XE");  
    $s = oci_parse($c, "select city from locations");  
    oci_execute($s);  
    echo "<table border='1'>\n";  
    while (($row = oci_fetch_array($s, OCI_ASSOC)) != false) {  
        echo " <tr>\n";  
        echo "   <td>".htmlentities($row['CITY'])."</td>\n";  
        echo " </tr>\n";  
    }  
    echo "</table>\n";  
}  
?  
>
```

When you enter the URL of the script (see step 1 in Figure 1) in your browser, the web server invokes PHP to processes the file. The PHP code is loaded and executed (2). Calls to the database (3) return data which is formatted and sent as output. Finally, this HTML is returned to the browser (4), which formats and displays the page.

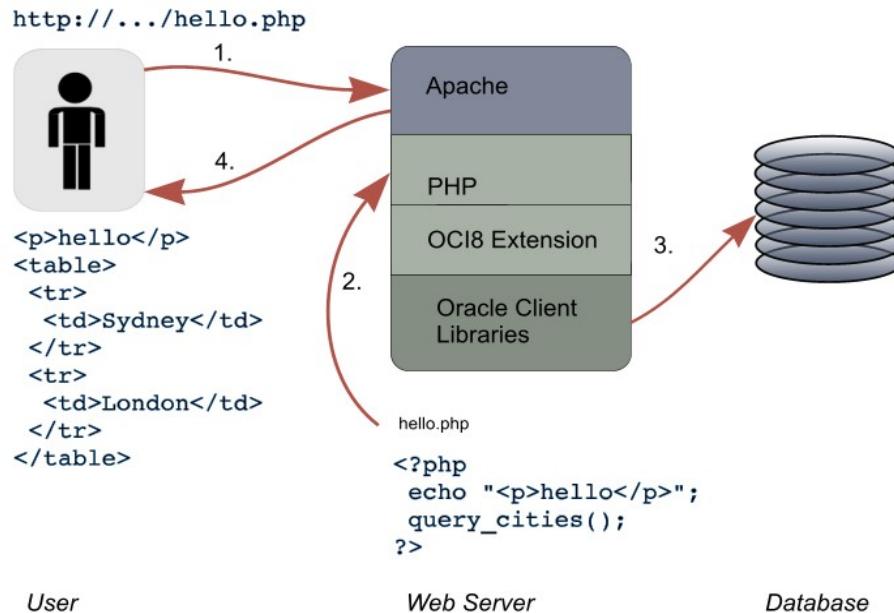


Figure 1: The four stages of processing a PHP script.

PHP is an open source project. It is maintained by a wide-spread community using an open source development methodology that has self-imposed goals and deadlines. Tasks are completed by developers who contribute their spare time and effort on sub-projects that interest them.

Any issues with PHP itself or with using the PHP OCI8 extension to access Oracle Database should be reported through community channels such as the OTN PHP forum and the PHP bug database.

Introduction

GETTING STARTED WITH PHP

This chapter gives a very brief overview of the PHP language. Basic PHP syntax is simple to learn. It has familiar loop, test and assignment constructs.

Creating and Editing PHP Scripts

There are a number of specialized PHP editors available, including Oracle's NetBeans IDE which is very highly regarded for PHP development. Some developers still prefer basic text editors with modes that highlight code syntax and aid development. In general this manual does not assume any particular editor or debugger is being used.

PHP scripts often have the file extension `.php`, but sometimes `.phtml` is also used. Web servers can be configured to recognize any extension(s) that you choose and send those files to PHP for processing.

PHP Syntax Overview

PHP scripts are enclosed in `<?php` and `?>` tags. Lines are terminated with a semi-colon:

```
<?php  
echo 'Hello, World!';  
?>
```

Blocks of PHP code and HTML code may be interleaved. The PHP code can also explicitly print HTML tags:

```
<?php  
echo '<h3>';  
echo 'Full Results';  
echo '</h3>';  
$output = "no results available";  
?>  
<table border="1">  
  <tr>  
    <td>  
      <?php echo $output ?>  
    </td>  
  </tr>  
</table>
```

The output when running this script is:

```
<h3>Full Results</h3><table border="1">  
  <tr>  
    <td>  
      no results available    </td>  
  </tr>  
</table>
```

Getting Started With PHP

A browser would display it as:

Full Results

```
no results available
```

Figure 2: PHP script output.

PHP strings can be enclosed in single or double quotes:

```
'A string constant'  
"another constant"
```

Variable names are prefixed with a dollar sign. Things that look like variables inside a double-quoted string will be expanded:

```
"A value appears here: $v1"
```

Strings and variables can also be concatenated using a period.

```
'Employee ' . $ename . ' is in department ' . $dept
```

Variables do not need types declared:

```
$count = 1;  
$ename = 'Arnie';
```

Arrays can have numeric or associative indexes:

```
$a1[1] = 3.1415;  
$a2['PI'] = 3.1415;
```

Strings and variables can be displayed with an `echo` or `print` statement. Formatted output with `printf()` is also possible.

```
echo 'Hello, World!';  
echo $v, $x;  
print 'Hello, World!';  
printf("There is %d %s", $v1, $v2);
```

Code flow can be controlled with tests and loops. PHP also has a `switch` statement. The `if`, `elseif` and `else` statements look like:

```
if ($sal > 900000) {  
    echo 'Salary is way too big';  
} elseif ($sal > 500000) {  
    echo 'Salary is huge';  
} else {  
    echo 'Salary might be OK';  
}
```

This also shows how blocks of code are enclosed in braces.

A traditional incrementing loop is:

```
for ($i = 0; $i < 10; ++$i) {
    echo $i . "<br>\n";
}
```

This prints the numbers 0 to 9, each on a new line. The value of `$i` is incremented in each iteration. The loop stops when the test condition evaluates to true. You can also loop with `while` or `do while` constructs.

The `foreach` command is useful to iterate over arrays:

```
$a3 = array('Aa', 'Bb', 'Cc');
foreach ($a3 as $v) {
    echo $v;
}
```

This sets `$v` to each element of the array in turn.

Functions may be defined:

```
function myfunc($p1, $p2) {
    echo $p1, $p2;
    return $p1 + $p2;
}
```

The previous function could be called using:

```
$v3 = myfunc(1, 3);
```

Functions may have variable numbers of arguments. Function calls may appear earlier than the function definition. Procedures use the same *function* keyword but do not have a *return* statement.

Classes look like:

```
class myclass {
    private $state = 1;
    public function set($v)
    {
        $this->state = $v;
    }
    public function get()
    {
        return $this->state;
    }
}

$v = new myclass();
echo $v->get();
$v->set(3);
```

Sub-files can be included in PHP scripts with an `include()` or `require()` statement.

```
include('foo.php');
require('bar.php');
```

A `require()` will generate a fatal error if the script is not found. The `include_once()` and `require_once()` statements prevent multiple inclusions of a file.

Comments are either single line:

Getting Started With PHP

```
// a short comment
```

or multi-line:

```
/*
 A
 longer
 comment
*/
```

Running PHP Scripts

PHP scripts can be loaded in a browser, or executed at a command prompt in a terminal window. Because browsers interpret HTML tags and they also compress white space including new-lines, this means script output can differ between command-line and browser invocation of the same script. For a similar reason pure-PHP scripts often omit the closing `?>` tag because any accidental trailing white space after it will be sent to the browser, possibly affecting style layouts. PHP doesn't require the closing tag.

Many aspects of PHP are controlled by settings in a `php.ini` configuration file. The location of the file is system specific. Its location, the list of extensions loaded, and the value of all the initialization settings can be found using the `phpinfo()` function:

```
<?php
phpinfo();
?>
```

Initialization settings can be changed by editing `php.ini` and restarting the web server. Some values can also be changed within scripts at run time by using the `ini_set()` function.

To connect to Oracle Database, some Oracle environment variables need to be set before the web server starts. This is discussed in the installation chapters of this book.

Running Scripts Via a Browser

PHP scripts are commonly run by loading them in a browser:

`http://localhost/myphpinfo.php`

Requesting a script causes the web server to invoke PHP to execute the requested code file. All of the script output is sent to the browser which formats and displays the output. Typically web servers such as Apache are used. The web server needs to be configured to call PHP and to map the user given URL to its equivalent PHP file.

Running Scripts Using the PHP Development Web Server

PHP 5.4 has a small in-built command-line interface (“CLI”) web server which can help testing by being immediately accessible. The web server is suitable for development and testing only.

If your PHP code is in a file `myphpinfo.php`, and the PHP executable is in your path, start the PHP CLI web server from the command line with:

```
$ php -S localhost:8888
```

The file can be loaded in a browser with the URL:

`http://localhost:8888/myphpinfo.php`

Running Scripts With Command-Line PHP

If your PHP code is in a file *myphpinfo.php*, and the PHP executable is in your path, run it with:

```
$ php myphpinfo.php
```

Various options to the PHP executable control its behavior. The [-h](#) option gives the help text. Common options when first using PHP are [--ini](#) which displays the location of the *php.ini* file, and [-i](#) which displays the value of the *php.ini* settings.

Debugging PHP Scripts

If you are not using the NetBeans IDE then debugging will be an old-fashioned matter of printing variables to check code flow.

The [var_dump\(\)](#) function is particularly useful for debugging because it formats and prints complex variables:

```
$a2['PI'] = 3.1415;  
var_dump($a2);
```

The output is:

```
array(1) {  
    ["PI"]=>  
        float(3.1415)  
}
```

The formatting is apparent when using command-line PHP. In a browser, to prevent white space and new lines from coalescing, you will need to do:

```
echo '<pre>';  
$a2['PI'] = 3.1415;  
var_dump($a2);  
echo '</pre>';
```

Some examples in this manual use [var_dump\(\)](#) to simplify the code being demonstrated or to show the type and contents of a variable.

Getting Started With PHP

PHP ORACLE EXTENSIONS

PHP has several extensions that let applications use Oracle Database. There are also database libraries written in PHP which are popular. These abstract the use of the underlying extension.

Database access in each extension and abstraction library is fundamentally similar. The differences are in support for advanced features and the programming methodology promoted. If you want to make full use of Oracle's features and want high performance then use OCI8, which is PHP's main Oracle extension. If you really need database independence, then the PHP Data Object (PDO) extension or the ADOdb abstraction library are available.

Many applications that value conformity over performance will benefit from using one of the increasingly popular PHP frameworks. These abstract much of the data access task in an application.

PHP Oracle Extensions

The PHP extensions that connect to Oracle Database are written in C and linked into the PHP binary. The extensions are:

- OCI8
- PDO_OCI driver for PDO

You can also use the ODBC extension; this is not covered in this book.

The three extensions are implemented independently and have no database access code in common. The extensions can be enabled separately or at the same time.

PHP OCI8 Extension

OCI8 is the recommended extension for accessing Oracle Database and is the focus of this book. OCI8 has been included in PHP since PHP 3. It is open source and maintained by the PHP community. Oracle is a member of the community looking after OCI8.

There have been major and minor changes to the OCI8 extension in the history of PHP. If you are using PHP versions 4 to 5.2, it is *highly* recommended to upgrade from the default OCI8 extension.

An example script that finds city names from the LOCATIONS table using OCI8:

Script 1: intro.php

```
<?php
$c = oci_connect('hr', 'welcome', 'localhost/XE');
$s = oci_parse($c, 'select city from locations');
oci_execute($s);
while (($res = oci_fetch_array($s, OCI_ASSOC)) != false) {
    echo htmlentities($res['CITY']) . "<br>";
}
?>
```

PHP Oracle Extensions

When invoked in a web browser, it connects as the demonstration user HR of the Oracle “XE” database running on the local machine. The query is executed and a web page of results is displayed in the browser:

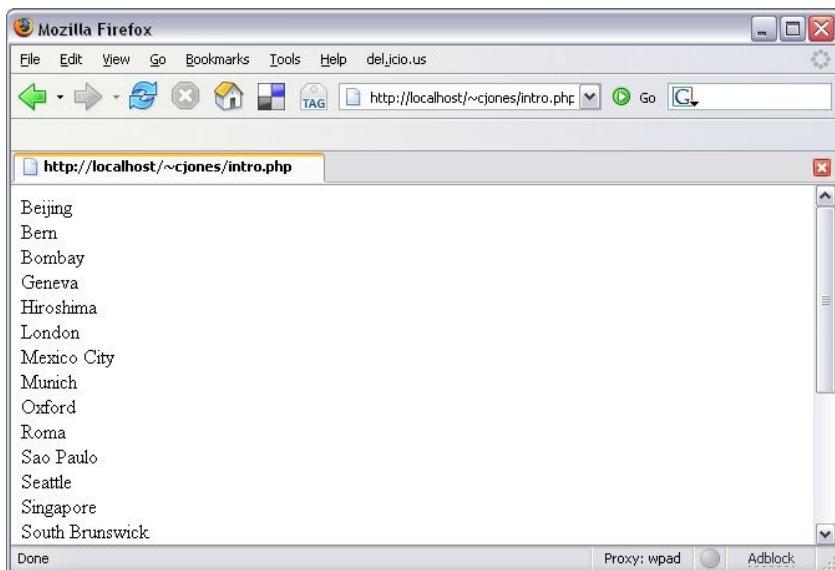


Figure 3: PHP output in a web browser.

In PHP 5 some extension function names were standardized. PHP 4 functions like `OCILogin()` became `oci_connect()`, the function `OCIParse()` became `oci_parse()` and so on. The old names still exist as aliases, so PHP 4 scripts do not need to be changed. A table showing old and new names appears in Appendix C.

The name “OCI8” is also the name for Oracle’s Call Interface API used by C programs such as the PHP OCI8 extension itself. All unqualified references to OCI8 in this book refer to the PHP extension.

Getting the OCI8 Extension

The OCI8 extension can be obtained in various ways. The PHP source code release is the canonical location for OCI8. It is also available from the PHP Extension Community Library (PECL) site, which contains PHP extensions as individual source code downloads. PECL is commonly used to install OCI8 on top of Linux distribution packages of PHP. Pre-built Windows PHP binaries from <http://php.net> include the OCI8 extension DLL. The OCI8 extension is also included in the Zend Server product, which is available for various platforms.

Table 1 shows where OCI8 can be downloaded from the PHP project.

Table 1: OCI8 Availability from the PHP Project and Sample Package Names.

Bundle Containing OCI8	Location and Sample Release
PHP Source Code	http://www.php.net/downloads.php <code>php-5.4.8.tar.bz2</code> Compiles and runs on many platforms

Bundle Containing OCI8	Location and Sample Release
PHP Windows Binaries	http://windows.php.net/download/ php-5.4.8-Win32-VC9-x86.zip
PECL Source Code	http://pecl.php.net/package/oci8 oci8-1.4.9.tgz Used to add or upgrade OCI8 for an existing PHP installation

OCI8 and Oracle Database Installation Options

To provide Oracle database access, the PHP binary must be linked with Oracle Client libraries. These libraries provide underlying connectivity to the database, which may be local or remote on your network.

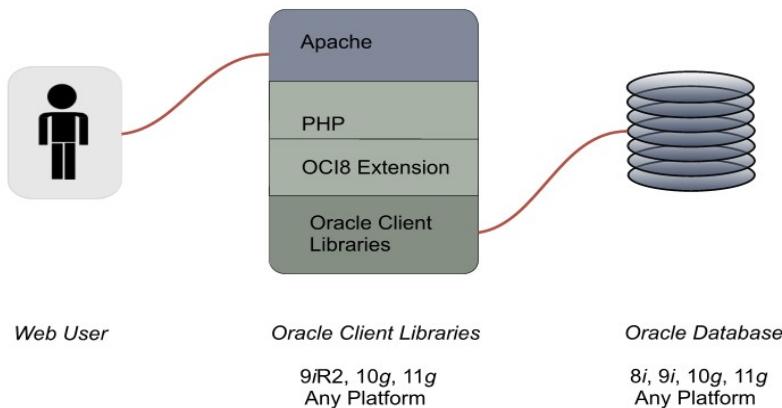


Figure 4: PHP links with Oracle client libraries.

Oracle has cross-version compatibility. For example, if PHP OCI8 is linked with Oracle Database 10g client libraries, then PHP applications can connect to Oracle Database 8i , 9i, 10g or 11g. If OCI8 is linked with Oracle Database 11g libraries, then PHP can connect to Oracle Database 9iR2 onwards.

If the database is installed on the same machine as the web server and PHP, then PHP can be linked with Oracle libraries included in the database software. If the database is installed on another machine, then link PHP with the small, free Oracle Instant Client libraries.

Full OCI8 functionality is not available unless the Oracle client libraries and database server are the latest version.

Table 2 shows the compatibility of the Oracle client libraries with the current OCI8 extension and PHP. Older versions of PHP have different compatibility requirements.

PHP Oracle Extensions

Table 2: OCI8 and Oracle Compatibility Matrix.

Software Bundle	PHP Version	OCI8 Version Included	Oracle Client Libraries Usable with OCI8
PHP Release Source Code	Current release is 5.4	OCI8 1.4	9iR2, 10g, 11g
PHP Release Windows Binaries	Current release is 5.4	OCI8 1.4	10gR2, 11gR2
PECL OCI8 Source Code	Builds with PHP 4.3.9 onwards	Current release is OCI8 1.4	9iR2, 10g, 11g

If PHP 4, 5.0, 5.1 or 5.2 are being used, you should replace the default OCI8 code with the latest version from PECL to get improved stability, behavior and performance optimizations. This is particularly important for PHP 4 and 5.0 because their versions of OCI8 are notoriously unstable.

PHP OCI8 History

PHP OCI8 has undergone continual development since it was first introduced in PHP 3. Table 3 shows the major features in recent revisions of OCI8, ever since the package was given an individual version number.

Table 3: Major Revisions of OCI8.

OCI8 Version	Main Features
OCI8 1.0	First PECL release. Based on PHP 4.3 OCI8 code.
OCI8 1.1	Beta releases that became OCI8 1.2.
OCI8 1.2	A major refactoring of the extension for PHP 5.1. It greatly improved stability, added control over persistent connections, and introduced performance features such as the ability to do statement caching and a new array bind function.
OCI8 1.3	Refactored connection management introduced support for Oracle Database 11g DRCP connection pooling and support for FAN giving high scalability and availability. Included support for Oracle's external authentication.
OCI8 1.4	New <code>oci_set_*</code> functions for better integration with database authentication, auditing and monitoring, and also for helping migration by allowing multiple versions of PL/SQL packages to be used concurrently. Introduces OCI_NO_AUTO_COMMIT as an alias of OCI_DEFAULT.

PHP PDO Extension

PHP Data Objects (PDO) is a data abstraction extension that provides PHP functions for accessing databases using a common core of database independent methods. Each database has its own driver, which may also support vendor specific functionality. PDO_OCI provides the Oracle functionality for PDO. The PDO extension and PDO_OCI driver are open source and included in PHP 5.1 onwards. Oracle does not contribute to PDO_OCI.

The PHP community has let the PDO project languish and Oracle recommends using OCI8 instead whenever possible because of its better feature set, performance, reliability and stability. Use of PDO_OCI for general purpose applications is *not* recommended. However PDO is used by some frameworks and higher level packages, such as content management systems so you may need to use it.

An example script that finds city names from the LOCATIONS table using PDO_OCI is:

Script 1: connectpdo.php

```
<?php
$dbh = new PDO('oci:dbname=localhost/XE', 'hr', 'welcome');
$s = $dbh->prepare("select city from locations");
$s->execute();
while (($r = $s->fetch(PDO::FETCH_ASSOC)) != false) {
    echo htmlentities($r['CITY']) . "<br>";
}
?>
```

The output is the same as the OCI8 example in Figure 4.

The Data Source Name (DSN) prefix `oci:` must be lowercase. The value of `dbname` is the Oracle connection identifier for your database.

Getting the PDO Extension

PDO and the PDO_OCI driver are included with PHP source code. Recent versions of PHP install the generic PDO extension code by default, but the database drivers such as PDO_OCI need to be explicitly added while building, or they can be installed manually afterward.

Only a few minor changes have been made by the community to PDO_OCI since its introduction. The version of PDO_OCI in the PECL repository has not been updated with these fixes, or with other changes needed to be compatible with recent versions of PHP. The PECL repository PDO_OCI release should not be used.

PHP Frameworks

PHP Frameworks are increasingly popular. Describing them is out of scope of this book. They almost all are written in PHP and abstract the use of PDO or native drivers. If you are choosing a PHP framework, consider one that uses OCI8 instead of PDO_OCI because not only is OCI8 more stable, it has caching, pooling and other scalability features that PDO_OCI does not.

PHP Database Abstraction Libraries

There are some older, but still viable, abstraction libraries for PHP:

PHP Oracle Extensions

- ADODb
- PEAR DB
- PEAR MDB2

You can freely download and use the PHP code for these libraries. They all use OCI8 functions in their implementations.

ADODb

The ADODb library is available from <http://adodb.sourceforge.net>. There is an optional C extension plug-in if you need extra performance.

An example script that finds city names from the LOCATIONS table using ADODb:

Script 2: connectadodb.php

```
<?php
require_once("adodb.inc.php");
$db = ADONewConnection("oci8");
$db->Connect("localhost/XE", "hr", "welcome");
$s = $db->Execute("select city from locations");
while ($r = $s->FetchRow()) {
    echo htmlentities($r['CITY']) . "<br>";
}
?>
```

There is an *Advanced Oracle Tutorial* at:

<http://phplens.com/lens/adodb/docs-oracle.htm>

PEAR DB

The PHP Extension and Application Repository (PEAR) contains many useful packages that extend PHP's functionality. PEAR DB is a package for database abstraction. It is available from <http://pear.php.net/package/DB>. PEAR DB has been superseded by PEAR MDB2 but is still sometimes used.

PEAR MDB2

The PEAR MDB2 package is available from <http://pear.php.net/package/MDB2>. It is a library aiming to combine the best of PEAR DB and the PHP Metabase abstraction packages.

An example script that finds city names from the LOCATIONS table using MDB2:

Script 3: connectpear.php

```
<?php
require("MDB2.php");
$MDB2 = MDB2::connect('oci8://hr:welcome@//localhost/XE');
$res = $MDB2->query("select city from locations");
while ($row = $res->fetchRow(MDB2_FETCHMODE_ASSOC)) {
    echo htmlentities($row['city']) . "<br>";
}
?>
```

The PHP Release Cycle

PHP's source code is under continual development in a source code control system viewable at <http://git.php.net/>. This is the only place bug fixes are merged. The code is open source and anyone can read the code in Git or seek approval to contribute. Git "pull" requests can be submitted at <https://github.com/php/php-src>.

The code in Git is used to create the various PHP distributions:

- Two-hourly snap-shots are created containing a complete set of all PHP's source in the Git repository at the time the snapshot was created. You can update your PHP environment by getting this source code and recompiling, or by downloading the Windows binaries. The snapshots may be relatively unstable because the code is in flux. The snapshots are located at <http://smaps.php.net/>.
- The PHP release manager frequently releases a new stable version of PHP. It uses the most current Git code at the time of release. Currently the release cycle is monthly.
- PECL OCI8 source code snapshots are taken from Git. Recently snapshots have been made concurrently at the time of a PHP release (when OCI8 has changed).
- Various operating systems bundle the version of PHP current at the time the OS is released and provide critical patch updates.

The schedules of PHP releases, the PECL source snapshots, and third party distributions are not fully synchronized.

INSTALLING ORACLE DATABASE 11G EXPRESS EDITION

This chapter contains an overview of, and installation instructions for, Oracle Database 11g Express Edition (“Oracle XE”). The installation instructions are given for Linux and Windows.

Oracle Database Editions

There are a number of editions of the Oracle database, each with different features, licensing options and costs. The editions are:

- Express Edition
- Standard Edition One
- Standard Edition
- Enterprise Edition

All the editions are built using the same code base. That is, they all have the same source code, but different features are available in each edition. Enterprise Edition has all the bells and whistles, whereas the free Express Edition has a limited feature set, but still has all the reliability and performance of the Enterprise Edition. The PHP code you write for Express Edition can be used unchanged with Enterprise Edition.

You could start off with the Express Edition, and, as needed, move up to another edition as your scalability and support requirements change. You could do this without changing any of your underlying table structure or code. Just change the Oracle software and you’re away.

There is a comprehensive list of the features for each Oracle edition at
<http://www.oracle.com/us/products/database/product-editions-066501.html>.

This book discusses working with Oracle Database 11g XE.

Oracle Database 11g XE

The free Oracle Database 11g Express Edition is available in 32-bit form on Windows and 64-bit for Linux platforms. Oracle Database 11g XE is a good choice for development of PHP applications that require a free, small footprint database.

Oracle Database 11g XE is available on the Oracle Technology Network at
<http://www.oracle.com/technetwork/products/express-edition/overview/index.html> for the following operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows Server 2003 (all editions), 2003 R8 (all editions), 2008 (Standard, Enterprise, Datacenter, Web and Foundation editions)
- Microsoft Windows 7 (Professional, Enterprise, and Ultimate editions)
- Oracle Linux and RHEL 4 Update 7, or 5 Update 2, or later updates.

Installing Oracle Database 11g Express Edition

- SUSE SLES 10 SP2 and 11

The installation needs 1.5 GB of disk space. A RAM size of 512MB is recommended as the minimum.

There are some limitations on using Oracle Database 11g XE:

- Up to 11GB of data can be stored
- A single database instance is installable
- A single CPU is used, even if multiple CPUs exist
- Only 1GB RAM used, even if more RAM is installed

These limitations do not exist in other editions of Oracle Database.

Oracle Database 11g XE has a browser-based application development tool called Oracle Application Express. It has some management capabilities.

Support for Oracle Database 11g XE is through an Oracle Technology Network discussion forum, which is populated by peers and product experts. You cannot buy support from Oracle for Oracle Database 11g XE. If you need a fully supported version for the Oracle database, you should consider Oracle Standard Edition or Enterprise Edition. You can download all the editions of the Oracle Database from the Oracle Technology Network. Subject to the click through license, you can use them for application development and testing, but when you go production, you will need to purchase a license.

Installing Oracle Database 11g XE on Linux

Most PHP applications are deployed on Linux. You can obtain Oracle Linux and patches for free from <https://linux.oracle.com/>. The yum server at <http://public-yum.oracle.com/> explains how to configure the repository to get updates. If you want to create a virtual machine to run Linux, Oracle's VirtualBox makes it easy. You can download this from <https://www.virtualbox.org/>.

To install Oracle Database 11g XE on Linux:

1. Turn off SELinux by editing /etc/selinux/config on Oracle Linux and setting:

```
SELINUX=disabled
```

Reboot the system after doing this, or also use the *setenforce* program.

2. Install the required packages, of the given versions or later:

Script 2: *glibc* release 2.3.4-2.41

Script 3: *make* release 3.80

Script 4: *binutils* 2.16.91.0.5

Script 5: *gcc* 4.1.2

Script 6: *libaio* 0.3.104

3. Download Oracle Database 11g XE from

<http://www.oracle.com/technetwork/products/express-edition/downloads/index.html>.

4. Log in or *su* as *root*:

```
# su -  
Password:
```

5. Install the RPM:

Installing Oracle Database 11g XE on Linux

```
# rpm -ivh oracle-xe-11.2.0-1.0.x86_64.rpm
```

Oracle Database 11g XE installs in a few minutes.

6. Configure the database:

```
# /etc/init.d/oracle-xe configure
```

7. Accept the ports. The default are **8080** for Application Express, and **1521** for the Database Listener.
8. Enter and confirm the password for the default users.
9. Enter **Y** or **N** for whether you want the database to start automatically on reboot. The database and database listener are configured and started.

If you use the Oracle Unbreakable Linux Network and have the Oracle Software channel enabled, you could instead install Oracle Database 11g XE with:

```
# up2date oracle-xe
```

After this download completes, follow the previous configuration steps from step 5 onwards.

Installing Oracle Database 11g XE on Windows

To install Oracle Database 11g XE on Windows, follow these steps:

1. Log on to Windows as a user with Administrative privileges.
2. Download Oracle Database 11g XE from
<http://www.oracle.com/technetwork/products/express-edition/downloads/index.html>.
3. Unzip *OracleXE112_Win32.zip* and navigate to the Disk1 folder. Double click on the *setup.exe* file.
4. In the Oracle Database 11g XE - Install Wizard welcome window, click **Next**.

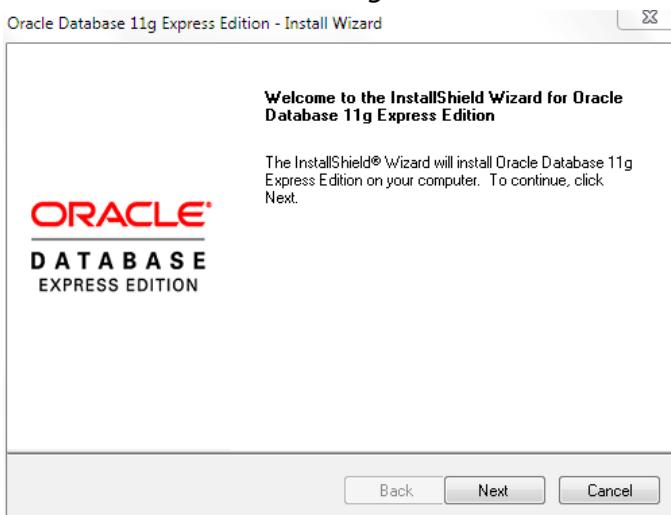


Figure 5: Oracle Database 11g XE install welcome dialog.

Installing Oracle Database 11g Express Edition

5. In the License Agreement window, accept the license and click **Next**.
6. In the Choose Destination Location window, either accept the default or click Browse to select a different installation directory. Click **Next**.

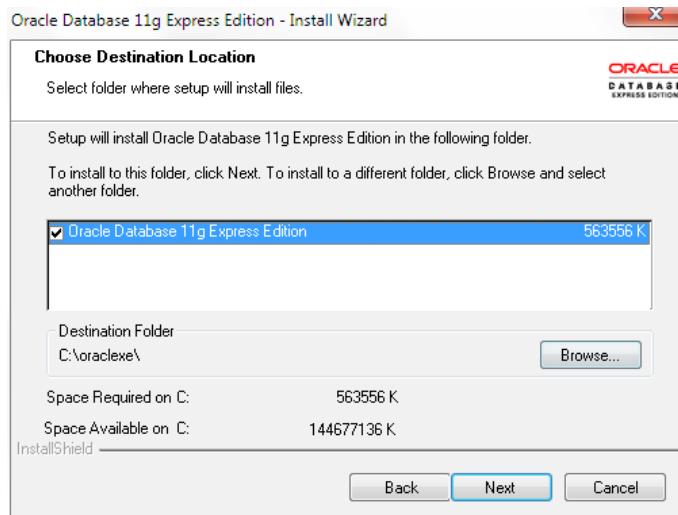


Figure 6: Oracle Database 11g XE install location dialog.

7. Oracle Database 11g XE selects a number of default ports. If these ports are already being used, you are prompted to enter another number.
8. In the Specify Database Passwords window, enter and confirm the password to use for the **SYS** and **SYSTEM** database accounts. Click **Next**.

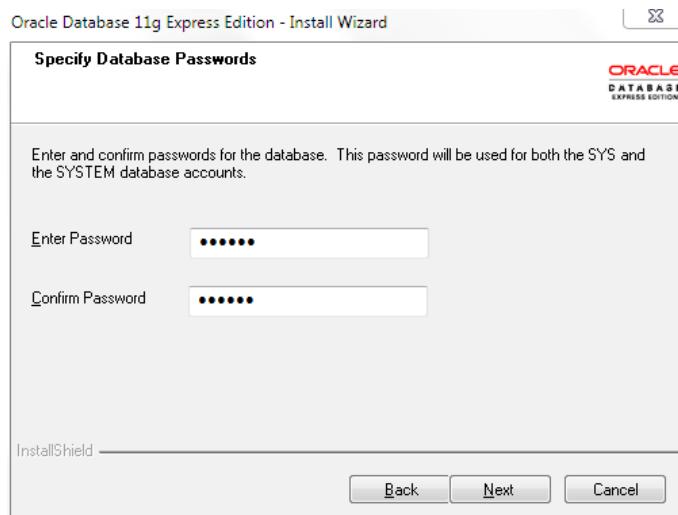


Figure 7: Oracle Database 11g XE database password dialog.

Installing Oracle Database 11g XE on Windows

9. In the Summary window, review the installation settings. Click **Install**.

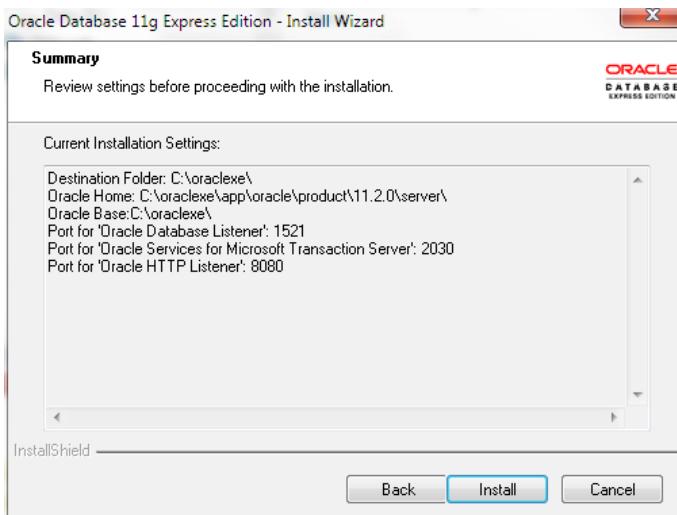


Figure 8: Oracle Database 11g XE install summary dialog.

10. In the InstallShield Wizard Complete window, click **Finish**.

Windows users will understandably want to develop on a familiar platform. However, because most PHP deployments are on Linux, and because of differences in performance and behavior of PHP on Windows, you should strongly consider developing on Linux to avoid unexpected surprises when applications are deployed.

Managing the Oracle Database 11g XE

The Oracle Database 11g XE can be managed with command line tools, Oracle Application Express and Oracle SQL Developer.

Setting the Oracle Database 11g XE Environment Variables on Linux

There are a number of environment settings and configuration options you can set for Oracle Database 11g XE. The more commonly used settings are preconfigured.

On Linux platforms a script is provided to set the Oracle environment variables after you log in. The script for Bourne, Bash and Korn shells:

```
/u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
```

For C and tcsh shells, use *oracle_env.csh*. Run the appropriate script for your shell to set your Oracle Database 11g XE environment variables. You can also add this script to your login profile to have the environment variables set up automatically when you log in.

To add the script to your Bourne, Bash or Korn shell, add the following lines to your *.bash_profile* or *.bashrc* file:

```
source /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
```

Installing Oracle Database 11g Express Edition

In some versions of the shells you may need to use a single period instead of *source*. To add the script to your login profile for C and tcsh shells, use the *oracle_env.csh* file instead. If you ever forget where the Oracle software is installed, look at the file */etc/oratab*. This lists each Oracle Database home directory.

Starting and Stopping the Listener and Database

The database listener is an Oracle Net program that listens for, and responds to, requests to the database. The database listener must be running to handle these requests. The database is another process that runs in memory, and needs to be started before Oracle Net can handle connection requests to it.

After installing Oracle Database 11g XE, the listener and database should already be running, and you may have requested during the installation that the listener and database should be started when the operating system starts up. If you need to manually start or stop the database listener, the options and commands for this are listed below.

Enabling Database Startup and Shutdown on Linux

You may not initially be able to start and stop the database using the menu on Linux platforms unless you are logged in as the root user. This is because your user is not a member of the operating system *dba* group by default. To enable this functionality, add your username to the *dba* group using Linux's System Settings and re-login.

Starting and Stopping the Listener and Database on Linux

The administration options are in an *Oracle Database 11g Express Edition* menu. Depending on your version of Linux this will be located under the *Applications*, *Main* or *K Menu* menu. To start up the listener and database on Linux platforms using the desktop, select **Oracle Database 11g Express Edition > Start Database**. To shut down the database on Linux platforms using the desktop, select **Oracle Database 11g Express Edition > Stop Database**.

To start the listener and database on Linux platforms using the command line, run the following command in a shell as the root user:

```
# service oracle-xe start
```

To stop the listener and database on Linux platforms using the command line, run the following command in your shell:

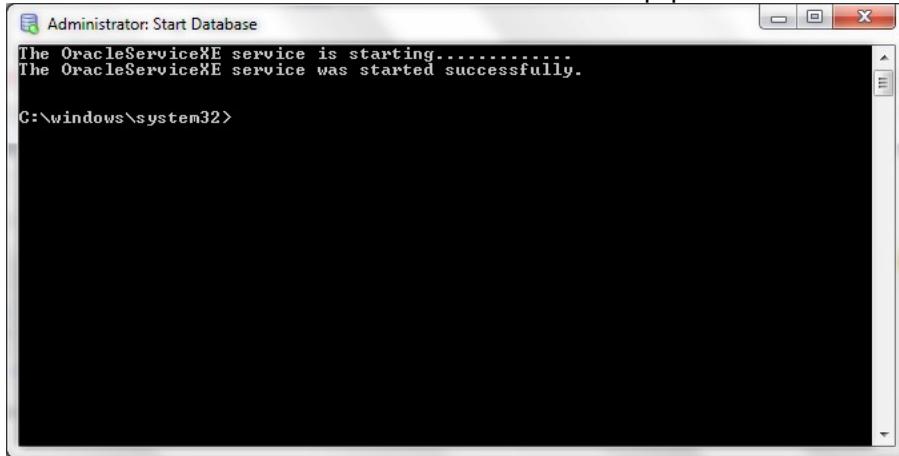
```
# service oracle-xe stop
```

The script is located in */etc/init.d*. If *service* does not exist in your environment, run the script directly.

You can also use the *Services* dialog under the System Administration menu to control the listener and database. For example, to start the listener and database from the Desktop Services dialog on Oracle Linux 5 with the Gnome window manager, select **System > Administration > Server Settings > Services**. Select **oracle-xe** from the list of services and select **Start**.

Starting and Stopping the Listener and Database on Windows

To start the listener and database on Windows platforms, in the **All Programs** menu select **Oracle Database 11g Express Edition > Start Database**. A Window is displayed showing the status of the listener and database startup process.

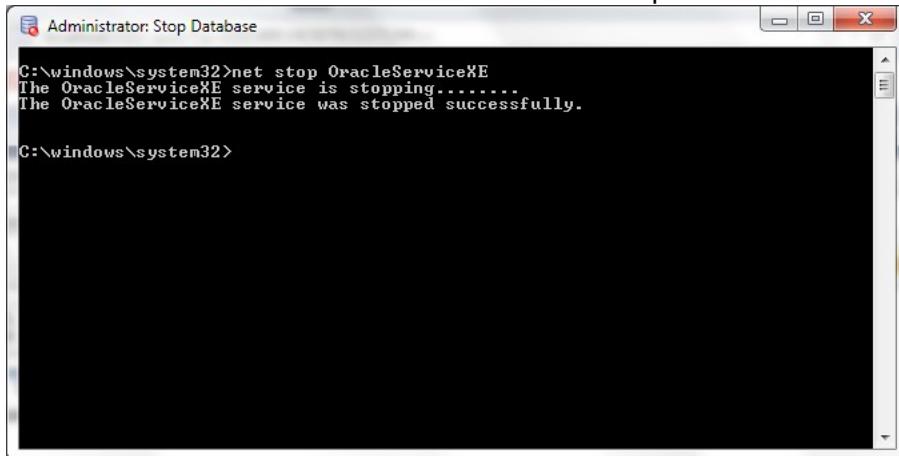


The OracleServiceXE service is starting.....
The OracleServiceXE service was started successfully.
C:\windows\system32>

Figure 9: Start Database dialog.

Type **exit** and press Enter to close the window. The listener and database are now started.

To stop the listener and database on Windows platforms, select **All Programs > Oracle Database 11g Express Edition > Stop Database**. A window is displayed showing the status of the listener and database shutdown process.



C:\windows\system32>net stop OracleServiceXE
The OracleServiceXE service is stopping.....
The OracleServiceXE service was stopped successfully.
C:\windows\system32>

Figure 10: Stop Database dialog.

Type **exit** and press Enter to close the window. The listener and database are now stopped. You can also start and stop the listener separately on Windows platforms using the Services dialog.

Basic Monitoring of Oracle Database 11g XE

The Oracle Application Express “Apex” tool provides some basic database monitoring.

Installing Oracle Database 11g Express Edition

Oracle SQL Developer also has monitoring and management capabilities, discussed in the next chapter.

To use Apex to verify the installation of Oracle Database 11g XE:

1. Open a web browser and enter the home page administration URL, using the port you selected during installation, <http://localhost:8080/f?p=4950>. This URL is also invoked by the **Get Started** sub-menu option of the **Oracle Database 11g Express Edition** menu.
2. The Database administration page is displayed.

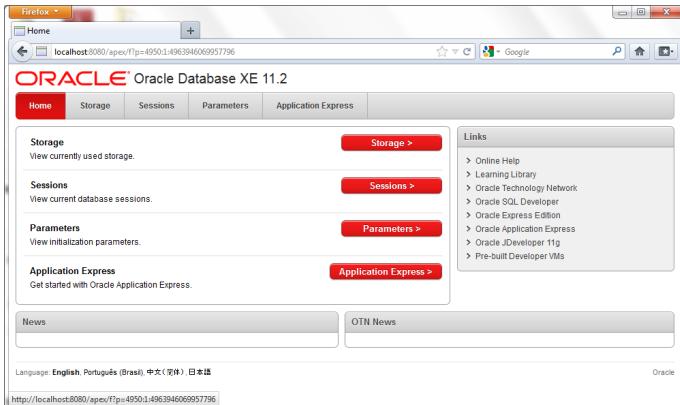


Figure 11: Oracle Database 11g XE home page login screen.

3. Click on *Storage*. Log in as user **SYSTEM** with the password you entered during the installation. You should now see the database configuration parameters that have non-default values. You can also un-check the filter to see all parameter values.

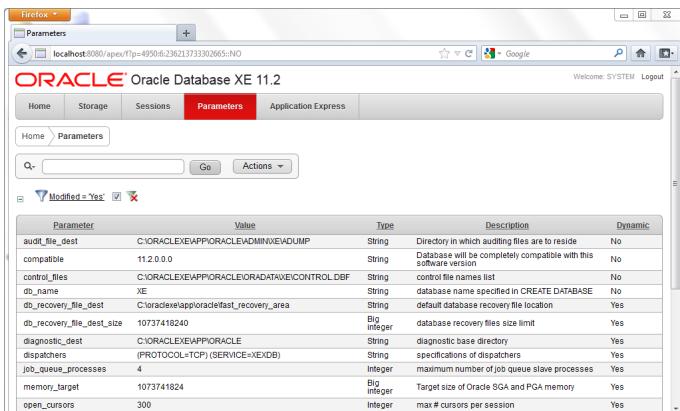


Figure 12: Oracle Database 11g XE parameters page.

Similarly with the other top level menu items on the Database home page you can view the current sessions of database connections, and look at the initialization parameters of the database.

Oracle Terminology

There are some differences between the terminology used when describing an Oracle database and other databases. The following overview covers the main Oracle terminology.

Databases and Instances

An Oracle database stores and retrieves data. Each database consists of one or more data files. An Oracle *database server* consists of an Oracle *database* and an Oracle *instance*. Every time a server is started, a shared memory region called the *system global area* (SGA) is allocated and the Oracle background processes are started. The combination of the background processes and SGA is called an Oracle *instance*. On some operating systems, like Windows, there are no separate background processes. Instead threads run within the Oracle image.

Tablespaces

Tablespaces are the logical units of data storage made up of one or more datafiles. Tablespaces are often created for individual applications because tablespaces can be conveniently managed. Users are assigned a default tablespace that holds all the data the users create. A database is made up of default and DBA-created tablespaces.

Schemas and Users

A schema is a collection of database objects such as tables and indexes. A schema is owned by a database user and has the same name as that user. Many people use the words *schema* and *user* interchangeably.

Once you have installed PHP and want to write scripts that interact with Oracle, you need to connect as the owner of the schema that contains the objects you want to interact with. For example, to connect to the HR schema, you would use the username HR in PHP's connection string.

Although you may have more than one database per machine, typically only a single Oracle database exists containing multiple schemas. Multiple applications can use the same database without any conflict by using different schemas. Instead of using a `CREATE DATABASE` command for new applications, in Oracle use the `CREATE USER` command to create a new schema in the database.

Installing Oracle Database 11g Express Edition

SQL WITH ORACLE DATABASE

This chapter contains an overview of some SQL*Plus, Oracle Application Express and Oracle SQL Developer features you can use to perform database development. These three tools are available with, or for, all editions of Oracle Database. They offer an extensive set of development functionality. This chapter focuses on how to use them for SQL development. Testing and tuning SQL is often more easily done in one of these tools before incorporating it into PHP applications.

Oracle SQL*Plus

SQL*Plus is Oracle's traditional command line tool. It is available whenever the database is installed. It is also available in "Instant Client" form so client machines can easily connect to a remote database. SQL*Plus allows ad-hoc queries, scripting and fundamental database administration. Many books, including this one, use SQL*Plus to show SQL examples.

Setting the Environment for SQL*Plus

Oracle Database 11g XE sets up a menu option to run SQL*Plus. However, in general, if you want to run SQL*Plus from a terminal window, the *sqlplus* executable must be in your PATH and several environment variables need to be set explicitly. These are pre-set in the registry on Windows.

In the Linux bash shell, set the environment for Oracle Database 11g XE with:

```
$ source /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
```

In some versions of the shell you will need to use a period instead of *source*. An equivalent file with the suffix ".csh" exists for users of the C shell.

On other editions of the Oracle database, the */usr/local/bin/oraenv* or */usr/local/bin/coraenv* (for users of C shell) scripts set the environment. In the Bash shell, use:

```
$ source /usr/local/bin/oraenv  
ORACLE_SID = [ ] ?
```

You will be prompted for the system identifier ("SID") of the database that you intend to connect to on this machine. The available SIDs can be seen in */etc/oratab*. Type the desired SID and press enter.

If you are running SQL*Plus on a machine remote from the database server, you need to manually set the environment.

The same environment variables needed for SQL*Plus are also used by the PHP OCI8 extension.

Starting SQL*Plus

Once the environment is set, SQL*Plus can be started with the *sqlplus* command:

```
$ sqlplus
```

SQL With Oracle Database

```
SQL*Plus: Release 11.2.0.2.0 - Production on Sat Jun 2 17:19:09 2012
Copyright (c) 1982, 2011, Oracle. All rights reserved.
```

```
Enter user-name: system
Enter password:
```

```
Connected to:
Oracle Database 11g Express Edition Release 11.2.0.2.0 - 64bit Production
SQL>
```

For Oracle 11g XE, the password to enter is the one set during installation. After starting SQL*Plus, the prompt **SQL>** is shown. At the prompt you can type SQL commands. Enter **HELP INDEX** to find a list of commands. Type **EXIT** to quit.

In development, it is common to put the username, password and database that you want to connect to all on the command line, but beware that entering the password like this is a security risk:

```
sqlplus system/systempwd@localhost/XE
```

A better practice is to run:

```
sqlplus system@localhost/XE
```

This will prompt for the password.

Another way is to start SQL*Plus without attempting to log on, and then use the **CONNECT** command:

```
$ sqlplus /nolog
.
.
.
SQL> connect system/systempwd@localhost/XE
Connected.
SQL>
```

To connect as a privileged user for database administration, first login to the operating system as a user in the operating system *dba* group and then use:

```
$ sqlplus / as sysdba
```

or

```
sqlplus /nolog
SQL> connect / as sysdba
```

Using SQL*Plus to Unlock the HR Schema for Demonstration Table Access

For general database access, it is better not to connect as the SYSTEM user. This book uses the Human Resource sample tables, located in the HR schema. To unlock this account and set the password after the database has been installed, connect as a privileged user and execute an **ALTER USER** command:

```
SQL> connect system/systempwd
SQL> alter user hr identified by welcome account unlock;
```

This sets the password to [welcome](#).

Creating Database Users in SQL*Plus

If you want to create your own schema, start SQL*Plus. Then run SQL statements like:

```
SQL> connect system/systempwd@localhost/XE
SQL> create user cj identified by welcome;
SQL> alter user cj default tablespace users
      temporary tablespace temp
      quota unlimited on users;
SQL> grant create session
      , create table
      , create procedure
      , create sequence
      , create trigger
      , create view
      , create synonym
      , alter session
to cj;
```

This creates a user *cj* with password of *welcome*. The user has the capability to create and use some types of database objects. You can choose which privileges to grant each user depending on what your application does.

Executing SQL and PL/SQL Statements in SQL*Plus

SQL statements such as queries must be terminated with a semi-colon (;):

```
SQL> select * from locations;
```

or with a single slash (/):

```
SQL> select * from locations
  2  /
```

This last example also shows SQL*Plus prompting for a second line of input. Code in Oracle's procedural language, PL/SQL, must end with a slash in addition to the PL/SQL code's semi-colon:

```
SQL> begin
  2    myproc();
  3  end;
  4  /
```

The terminating semi-colon or slash is not part of the statement. Tools other than SQL*Plus will use different methods to indicate "end of statement".

If a blank line (in SQL) or a single period (in SQL or PL/SQL) is entered, SQL*Plus returns to the main prompt and does not execute the statement:

```
SQL> select * from locations
  2
SQL>
```

SQL With Oracle Database

Commands that are local to SQL*Plus like `STARTUP` and `SET` which are not sent to the database do not need a terminating semi-colon or slash.

Controlling Query Output in SQL*Plus

SQL*Plus has various ways to control output display.

The `SET` command controls some formatting. For example, to change the page size (how often table column names repeat) and the line size (where lines wrap):

```
SQL> set pagesize 80
SQL> set linesize 132
```

If you are fetching data from LONG, CLOB or BLOB columns, increase the maximum number of characters that will display (the default is just 80):

```
SQL> set long 1000
```

The column width of queries can be changed with the `COLUMN` command, here setting the `COUNTRY_NAME` output width to 20 characters, and the `REGION_ID` column to a numeric format with a decimal place:

```
SQL> select * from countries where country_id = 'FR';
CO COUNTRY_NAME          REGION_ID
-----
FR France                  1
SQL> column country_name format a20
SQL> column region_id format 99.0
SQL> select * from countries where country_id = 'FR';
CO COUNTRY_NAME          REGION_ID
-----
FR France                  1.0
```

Output can be spooled to a file by using the `SPPOOL` command before commands are executed:

```
SQL> spool /tmp/myfile.log
```

Running SQL Scripts in SQL*Plus

If multiple SQL statements are stored in a script `myscript.sql`, they can be executed with the `START` command or its more common abbreviation `@`. Scripts can be run either from the terminal prompt:

```
$ sqlplus hr@localhost/XE @myscript.sql
```

or from the SQL*Plus prompt:

```
SQL> @myscript.sql
```

Because SQL*Plus doesn't have a full history command, creating scripts in an external editor and running them with `@` is recommended.

Table Metadata in SQL*Plus

Queries from inbuilt views like USER_TABLES and USER_INDEXES will show information about the objects you own. A traditional query from the CAT view gives a short summary:

```
SQL> select * from cat;

TABLE_NAME          TABLE_TYPE
-----
COUNTRIES           TABLE
DEPARTMENTS         TABLE
DEPARTMENTS_SEQ     SEQUENCE
EMPLOYEES           TABLE
EMPLOYEES_SEQ       SEQUENCE
JOBS                TABLE
JOB_HISTORY         TABLE
LOCATIONS           TABLE
LOCATIONS_SEQ       SEQUENCE
REGIONS             TABLE

10 rows selected.
```

To find out about the columns of a table, query the USER_TAB_COLUMNS view, or simply use the [DESCRIBE](#) command to give an overview:

```
SQL> describe countries
Name          Null?    Type
-----
COUNTRY_ID    NOT NULL CHAR(2)
COUNTRY_NAME   VARCHAR2(40)
REGION_ID     NUMBER
```

Starting and Stopping the Database With SQL*Plus

To start up the listener on Linux, open a terminal window as the root user and run the following commands:

```
# su - oracle
$ source /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
$ lsnrctl start
```

Oracle Net starts the listener, which handles communication between the database and client programs like SQL*Plus or PHP. If you later want to shut down the listener manually, use *lsnrctl* like:

```
$ lsnrctl stop
```

On Windows, use the Services dialog to control the listener.

After starting the listener, you need to start the database itself using SQL*Plus. For this, you must start SQL*Plus with the SYSDBA role. To start up a database using SQL*Plus, enter the following at the command line prompt:

```
# su - oracle
$ source /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
```

SQL With Oracle Database

```
$ sqlplus /nolog
```

The SQL*Plus command line starts. You can also start SQL*Plus from the **Oracle Database 11g Express Edition > Run SQL Command Line** on Linux, or **Program Files > Oracle Database 11g Express Edition > Run SQL Command Line** on Windows.

At the SQL*Plus command line prompt, enter the following commands to connect to the database and start it up:

```
SQL> connect / as sysdba  
SQL> startup
```

The database is started.

If you start the database before starting the Oracle Net listener, it can take a short while before the database registers with the listener. Until this happens, connections to the database will fail. You can explicitly force registration with:

```
SQL> alter system register;
```

To shut down the database, you need to log in as a SYSDBA user, and issue the [SHUTDOWN IMMEDIATE](#) command. Log into SQL*Plus as before and issue the following command:

```
SQL> connect / as sysdba  
SQL> shutdown immediate
```

If you use the alternative [SHUTDOWN NORMAL](#) command, the shutdown will hang until all open connections have completed their work. Since PHP OCI8 has the concept of persistent connections that remain open indefinitely, you will first need to shutdown Apache to close those connections. You can also use [SHUTDOWN ABORT](#) to immediately terminate a database, although this is not recommended.

The *SQL*Plus User's Guide and Reference* manual gives you the full syntax and other options for starting up and shutting down the database if you need more help.

The easiest way on Linux to start and stop Oracle XE 11g is with the *service* command as shown in the previous chapter.

Oracle Application Express

Oracle Application Express is a browser-based application builder for the Oracle database. It is installed with Oracle Database 11g XE and is also available for download from Oracle Technology Network as a standalone product for other versions and editions of the database. It contains an application development tool, not covered in this book. The release of Oracle Application Express installed with Oracle Database 11g XE has a small additional module for performing database monitoring.

Creating an Application Express Workspace

1. Open a web browser and enter the home page administration URL, using the port you selected during installation, <http://localhost:8080/f?p=4950>.

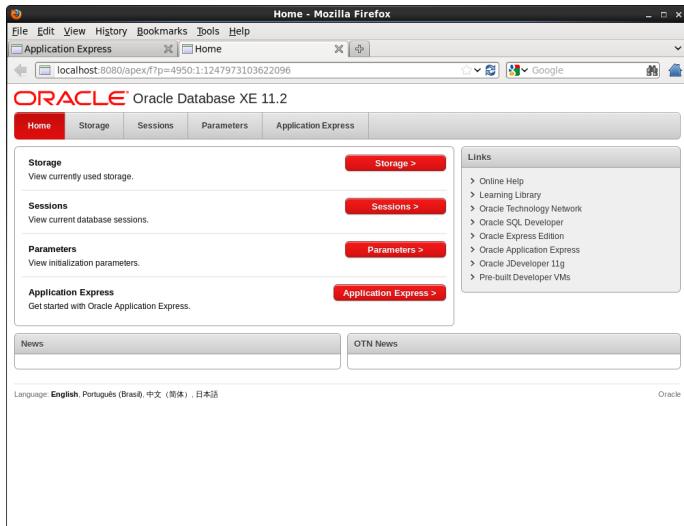


Figure 13: Oracle Application Express Administration Page.

2. Click on *Application Express* and login using the credentials created during database installation.

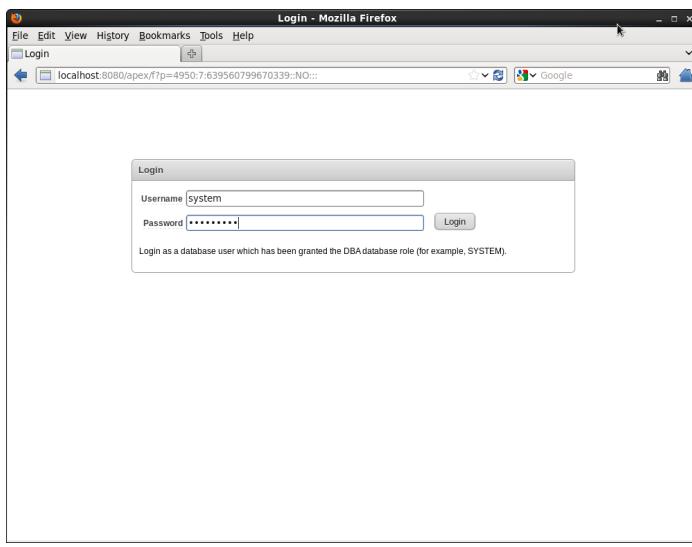


Figure 14: Oracle Application Express Administration Login.

SQL With Oracle Database

3. Create a new workspace with new Database and Application Express users. For example, use *myapexdb* and *myapex*, respectively.

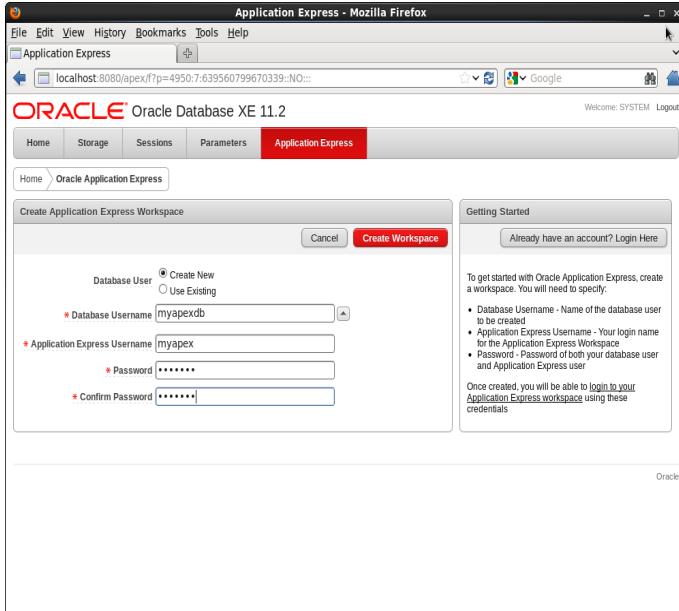


Figure 15: Creating new users with Oracle Application Express

4. Click *Create Workspace* and the workspace is created:

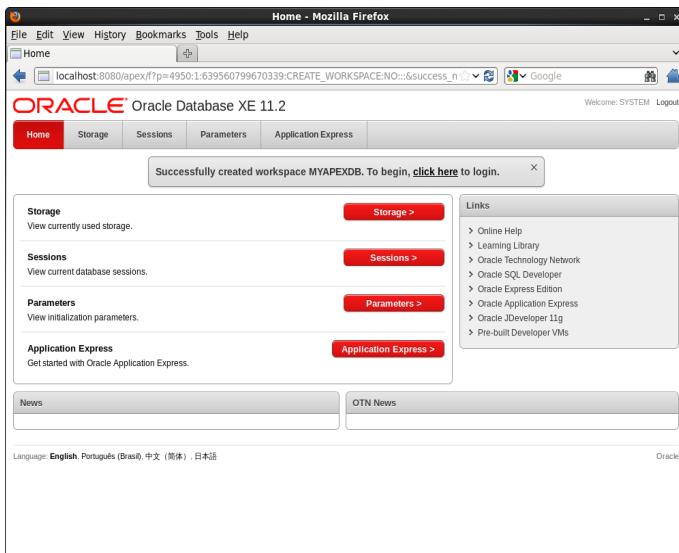


Figure 16: Oracle Application Express Workspace Creation

Logging In To Oracle Application Express

To log in to Oracle Application Express as the new application user:

- Follow the “click here” link on the account confirmation page or open the URL, <http://localhost:8080/apex>. The Oracle Application Express login screen is displayed:

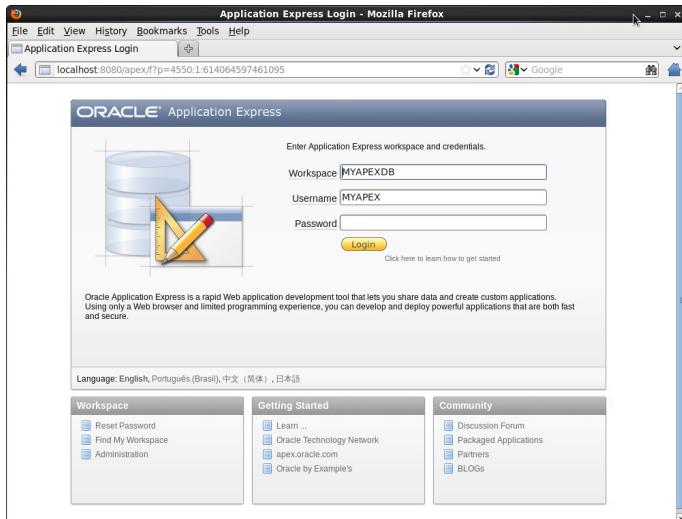


Figure 17: Oracle Application Express login screen.

- Login with the credentials you created in the previous section, *myapexdb* and *myapex*.

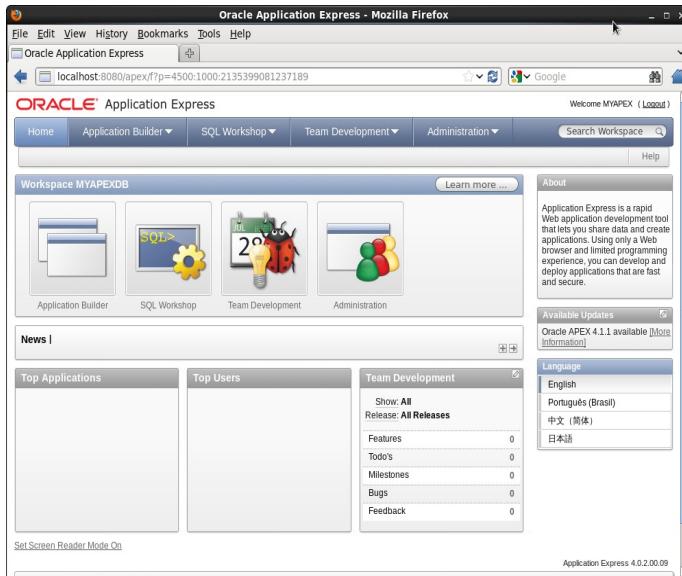


Figure 18: Oracle Application Express interface.

SQL With Oracle Database

Creating Database Objects in Application Express

The Oracle Application Express Object Browser can be used to create or edit many types of database objects, from tables and views, right through to procedures and triggers.

Oracle Application Express uses wizards to guide you through creating these database objects. The following example covers creating a table, but you will see that the interface is wizard-based and creating and editing different objects and queries can all be performed through the SQL Workshop.

To create a new table:

1. On the Database home page, click the **SQL Workshop** menu link.

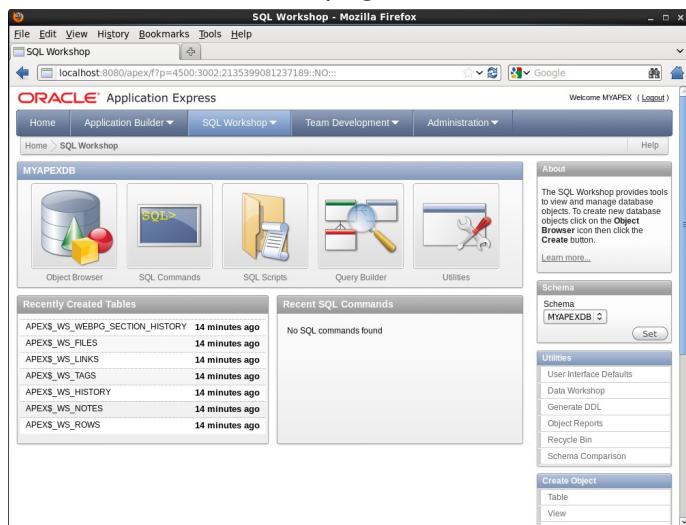


Figure 19: Oracle Application Express SQL Workshop

2. Click the **Object Browser** icon. The Object Browser page is displayed.

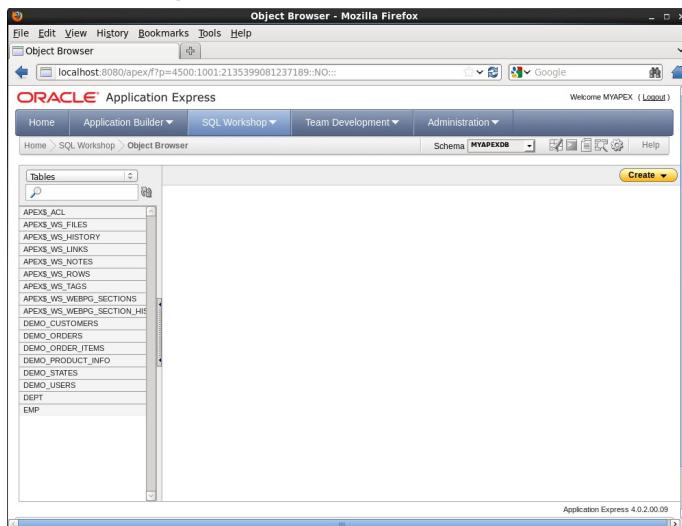


Figure 20: Oracle Application Express object browser screen.

3. Select the yellow **Create** menu on the right hand side and choose **Table**.

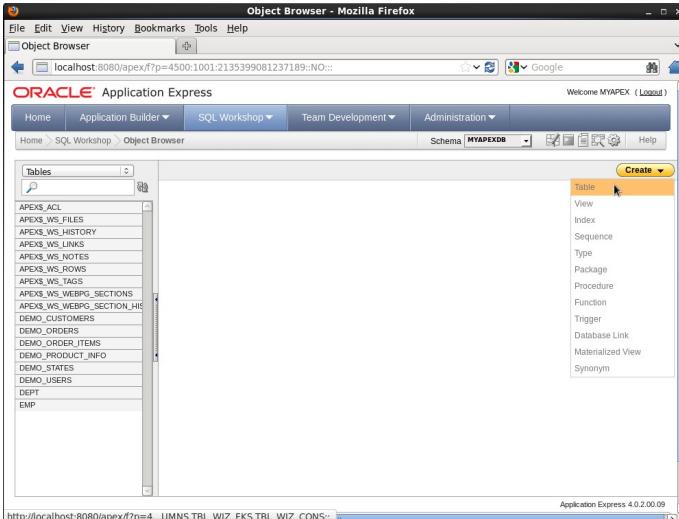


Figure 21: Oracle Application Express object browser screen

4. Enter a table name in the Table Name field, and details for each column. Click **Next >**.

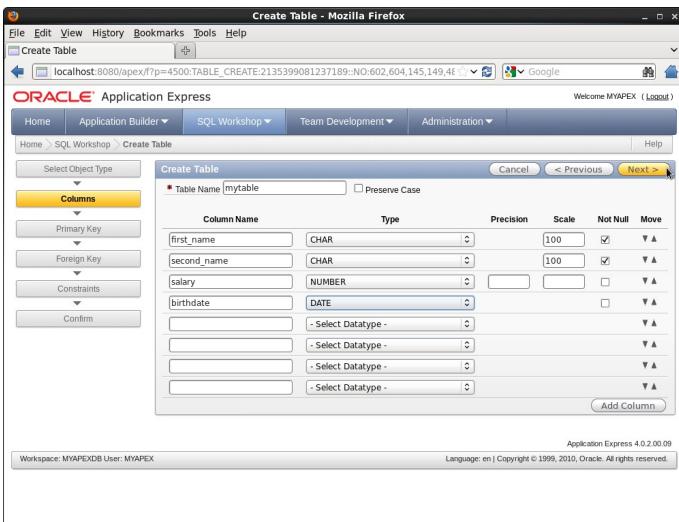


Figure 22: Oracle Application Express table definition screen.

SQL With Oracle Database

5. Leave the default No Primary Key and click **Next**.
6. Leave the Foreign Key unset and click **Next**.
7. Leave the Constraints page unchanged and click **Next**.
8. The SQL **CREATE** statement used to create the table is shown.

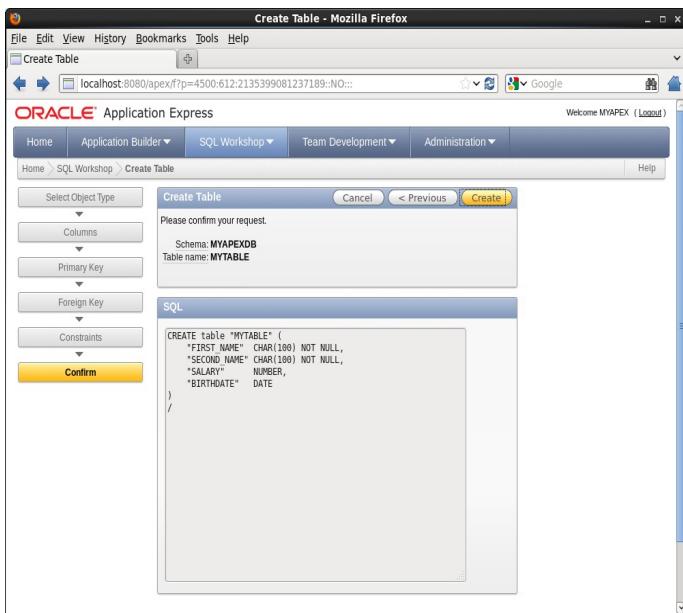


Figure 23: Oracle Application Express Create Table

9. On this confirmation page click **Create** to complete the creation. The table is created and a description of the table is displayed.

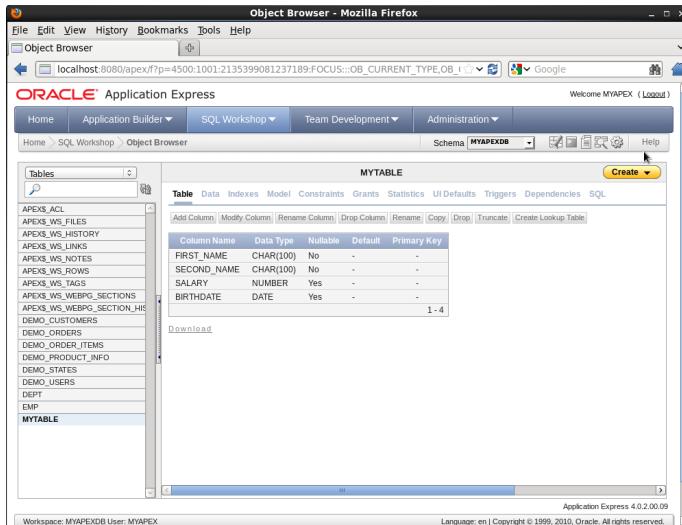


Figure 24: Oracle Application Express table created confirmation screen.

Working With SQL in Application Express

The SQL Workshop also enables you to:

- Write SQL and PL/SQL
- Load and save SQL scripts
- Graphically build SQL

The following example guides you through creating a SQL script. This example uses the Query Builder to graphically create a SQL script to query data. The *myapex* user you created has the sample tables already created. To access Query Builder:

1. On the Application Express home page, click the **SQL Workshop** icon.
2. Click the **Query Builder** icon.

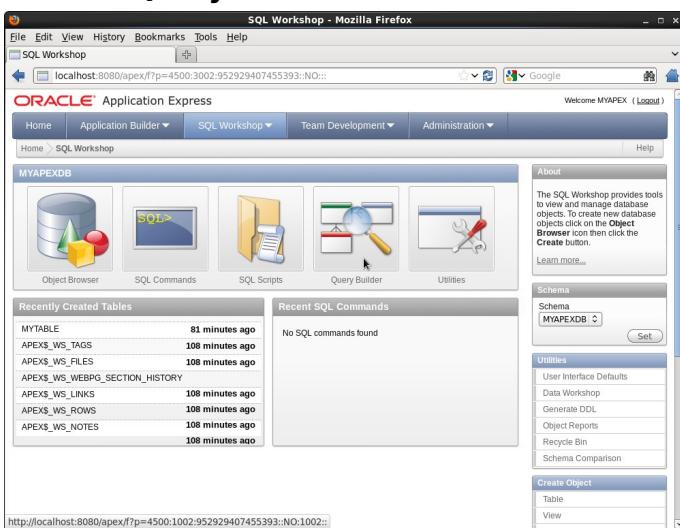


Figure 25: Oracle Application Express SQL options screen.

SQL With Oracle Database

3. Select objects from the Object Selection pane. When you click on the object name, it is displayed in the Design pane.

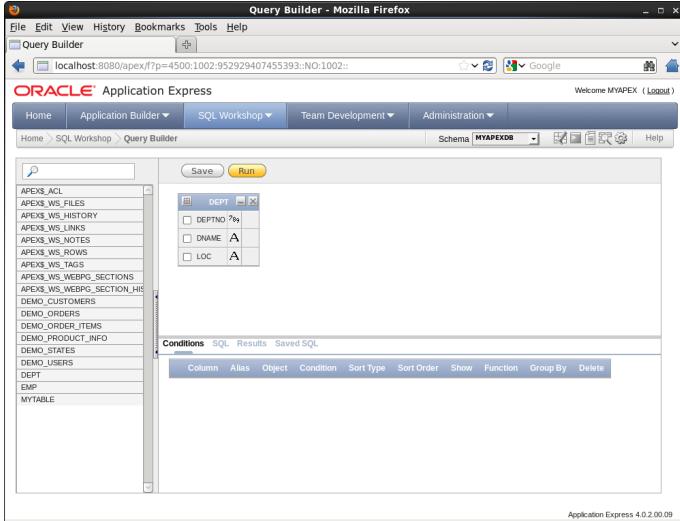


Figure 26: Oracle Application Express SQL query builder screen.

4. Select columns from the objects in the Design pane to select which columns to include in the query results.

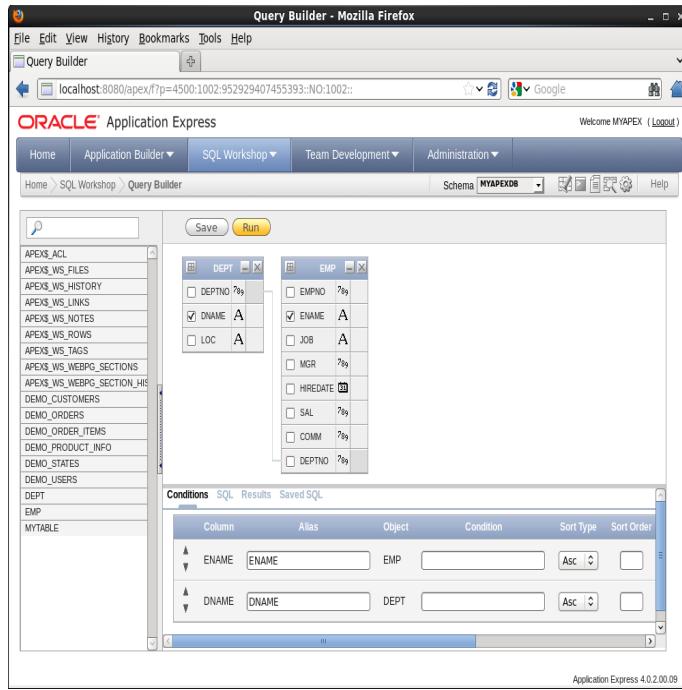


Figure 27: Oracle Application Express Query Builder

5. Establish relationships between objects by clicking on the right-hand column of each table.
6. Click **Run** to execute the query and view the results.

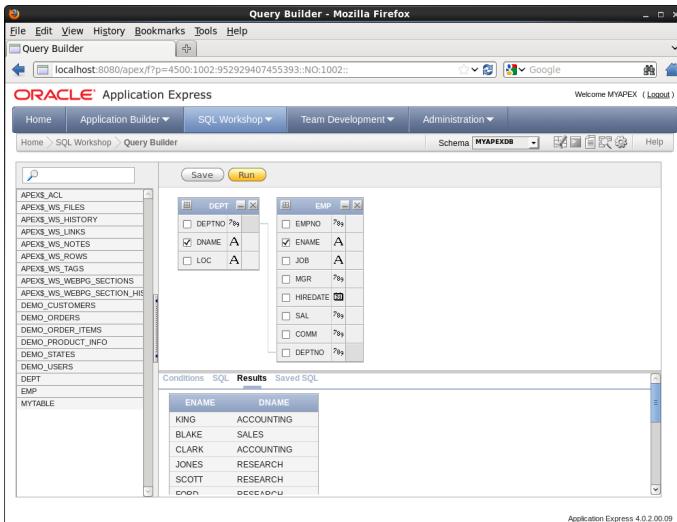


Figure 28: Oracle Application Express SQL query results screen.

7. You can save your query using the **Save** button.

Creating a PL/SQL Procedure in Application Express

To enter and run PL/SQL code in the SQL Commands page:

1. On the Application Express home page, click the **SQL Workshop** icon to display the SQL Workshop page. Then click the **SQL Commands** icon to display the SQL Commands page.

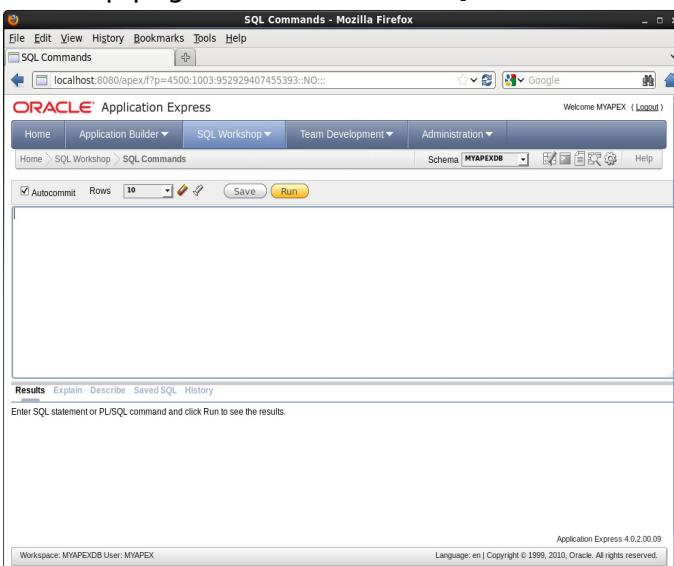


Figure 29: Oracle Application Express SQL Command Page

SQL With Oracle Database

2. On the SQL Commands page, enter some PL/SQL code. For example, the procedure `emp_stat` averages the salary for departments in the HR schema, and is encapsulated in a PL/SQL package called `emp_sal`.

```
create or replace package emp_sal as
procedure emp_stat;
end emp_sal;
```

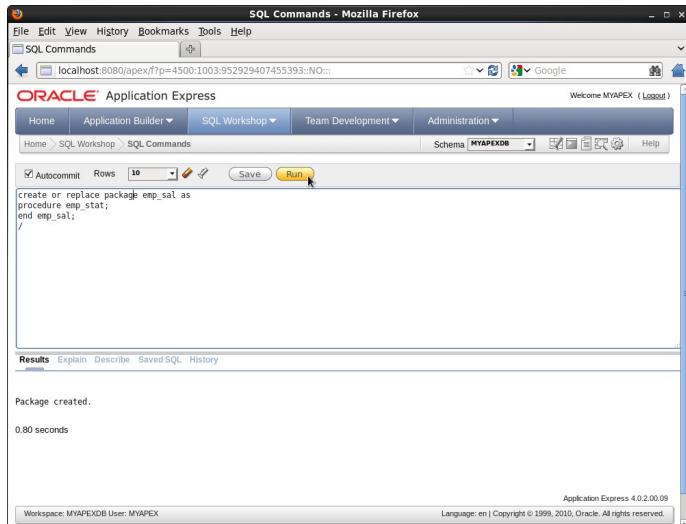


Figure 30: Oracle Application Express PL/SQL and SQL command screen.

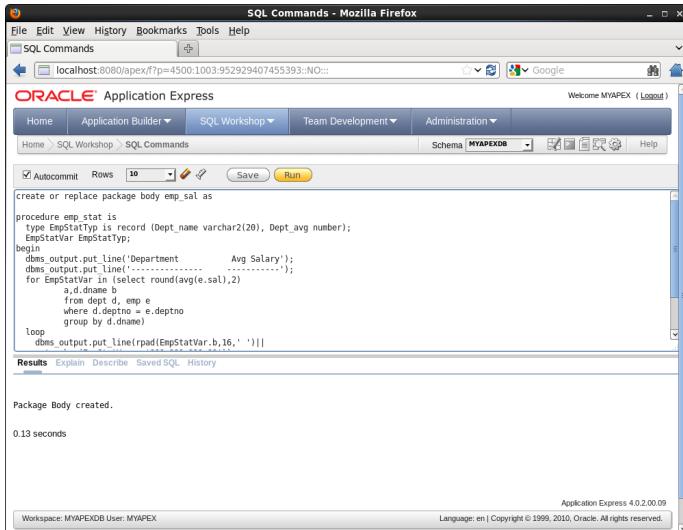
3. Click **Run** to execute the PL/SQL.
4. Now replace the code with the package body specification:

```
create or replace package body emp_sal as

procedure emp_stat is
    type EmpStatTyp is record (Dept_name varchar2(20), Dept_avg number);
    EmpStatVar EmpStatTyp;
begin
    dbms_output.put_line('Department          Avg Salary');
    dbms_output.put_line('-----  -----');
    for EmpStatVar in (select round(avg(e.sal),2)
                        a,d.dname b
                       from dept d, emp e
                      where d.deptno = e.deptno
                     group by d.dname)
    loop
        dbms_output.put_line(rpad(EmpStatVar.b,16, ' ') ||
                           to_char(EmpStatVar.a,'999,999,999.99'));
    end loop;
end;

end emp_sal;
```

5. Click **Run**. The PL/SQL code is executed.



The screenshot shows the Oracle Application Express SQL Commands interface in a Mozilla Firefox browser. The URL is `localhost:8080/apex/f?p=4500:1003:952929407455393:NO::`. The schema is set to MYAPEXDB. The code entered is:

```
create or replace package body emp_sal as
procedure emp_stat is
type EmpStatTyp is record (Dept_name varchar2(20), Dept_avg number);
EmpStatVar EmpStatTyp;
begin
dbms_output.put_line('Department' || 'Avg Salary');
dbms_output.put_line('-----' || '-----');
for EmpStatVar in (select round(avg(e.sal),2)
      ,a.d_name b
     from dept d, emp e
    where d.deptno = e.deptno
   group by d.dname)
loop
dbms_output.put_line(rpad(EmpStatVar.b,16,' '));
end loop;
end;
/

```

The results pane shows the output:

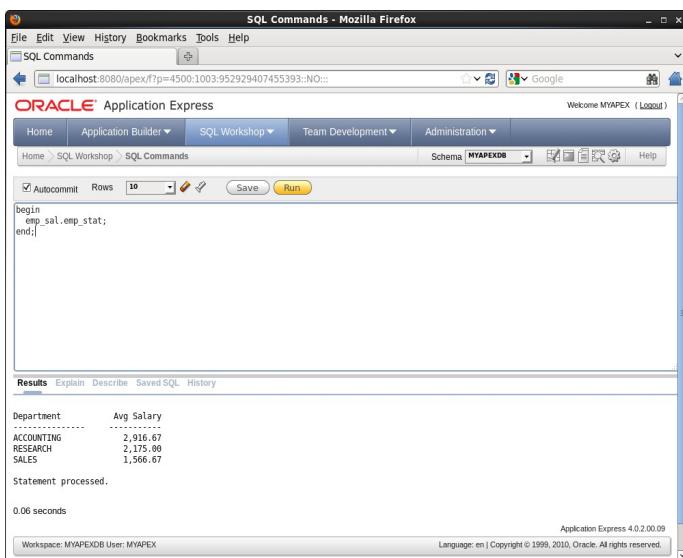
```
Package Body created.
0.13 seconds
```

Application Express 4.0.2.0.09

Figure 31: Oracle Application Express SQL Commands

6. You can save the PL/SQL code for future use, by clicking **Save**, or execute the stored procedure by entering the following PL/SQL code and clicking **Run**.

```
begin
  emp_sal.emp_stat;
end;
```



The screenshot shows the Oracle Application Express SQL Commands interface in a Mozilla Firefox browser. The URL is `localhost:8080/apex/f?p=4500:1003:952929407455393:NO::`. The schema is set to MYAPEXDB. The code entered is:

```
begin
  emp_sal.emp_stat;
end;
```

The results pane shows the output:

Department	Avg Salary
ACCOUNTING	2,916.67
RESEARCH	2,175.80
SALES	1,506.67

Statement processed.
0.06 seconds

Application Express 4.0.2.0.09

Figure 32: Oracle Application Express SQL Commands Screen

Oracle SQL Developer

In addition to Oracle Application Express and SQL*Plus, you can also use Oracle SQL Developer for database development and administration. Oracle SQL Developer is a free, thick-client graphical tool. You can use it to execute SQL statements, view metadata, execute and debug PL/SQL statements, and run some SQL*Plus commands (like `DESCRIBE`). SQL Developer includes a database modeler, and a module to assist migration to Oracle Database.

SQL Developer can connect to Oracle databases from version 9.2.0.1 onwards. Metadata and data from several third party databases, including MySQL Database are also viewable. SQL Developer is available on Linux, Windows and Mac OS X.

You can download SQL Developer from the Oracle Technology Network at <http://www.oracle.com/technetwork/developer-tools/sql-developer>. You can also download extensions, documentation, and other resources from this site. There is also a discussion forum for you to ask questions of other users, and give feedback to the SQL Developer product team.

Creating a Database Connection in SQL Developer

When you start SQL Developer for the first time there are no database connections configured, so the first thing you need to do is create one. The default SQL Developer screen is shown in Figure 33.



Figure 33: Oracle SQL Developer login screen.

To create a database connection to the local Oracle Database 11g XE database:

1. Select **Connections** in the left pane, right click and select **New Connection**. Choose a name to associate with the connection. Enter the login credentials for the Oracle Database 11g XE with the username **hr**, and password you created for the HR user, the hostname **localhost**, and the SID **XE**.

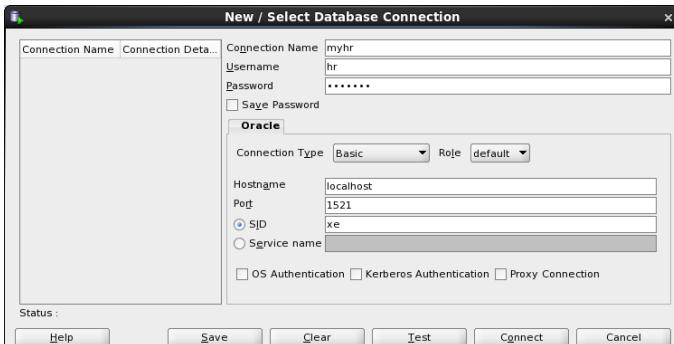


Figure 34: Oracle SQL Developer Connection screen.

2. Click the **Test** button to test the connection. A message is displayed at the bottom left side of the dialog to tell you whether the test connection succeeded.
3. Click the **Connect** button to save the connection and connect to the database. When you have connected to a database, you can browse through the database objects displayed in the left pane, and the right pane shows the contents of the object. In Figure 35, the EMPLOYEES table is displayed.

Figure 35: Oracle SQL Developer main screen.

Editing Data in SQL Developer

You can view and edit data using the Data tab for a table definition. Select the Data tab for the EMPLOYEES table to display the records available. You can add rows, update data and delete rows using the data grid. If you make any changes, the records are marked with an

SQL With Oracle Database

asterisk. Throughout SQL Developer, there are context sensitive menus. Figure 36 shows the choice of context menus available in the data grid.

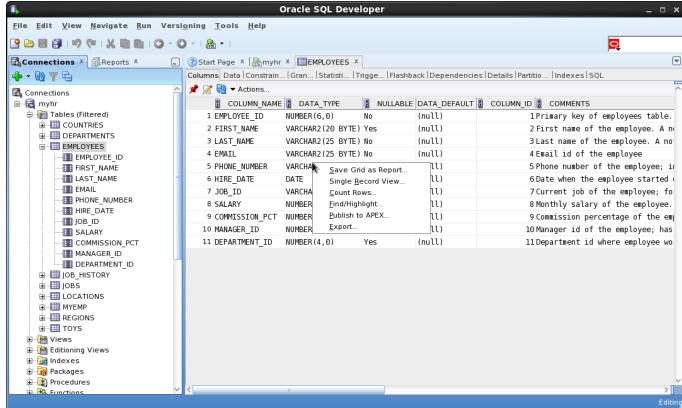


Figure 36: Oracle SQL Developer Data Grid.

Creating a Table in SQL Developer

You can create database objects such as tables, views, indexes, and PL/SQL procedures using SQL Developer. To create a database object, right click on the database object type you want to create, and follow the dialogs. You can use this method to create any of the database objects displayed in the left pane.

To create a new table:

1. Select **Tables** in the left pane, right click and select **New Table**. The Create Table dialog is displayed. Enter the column names, types and other parameters as required.

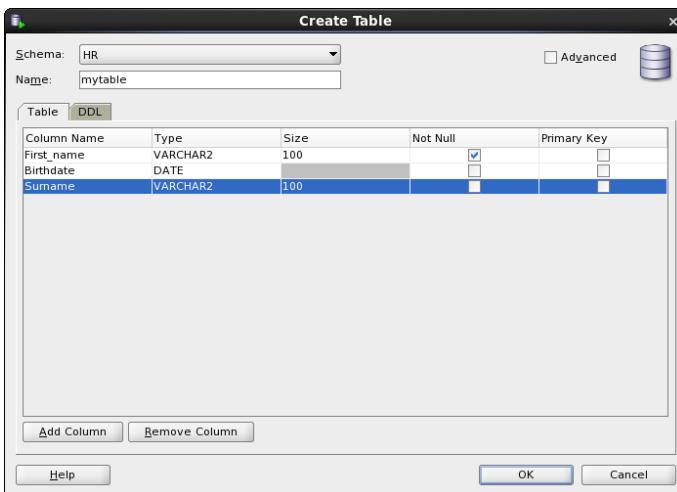


Figure 37: Oracle SQL Developer Create Table screen.

There is an **Advanced** check box that will give more advanced options like constraints, indexes, foreign keys, and partitions.

2. Click **OK** to create the table. The new table *mytable* is now listed in the left pane.

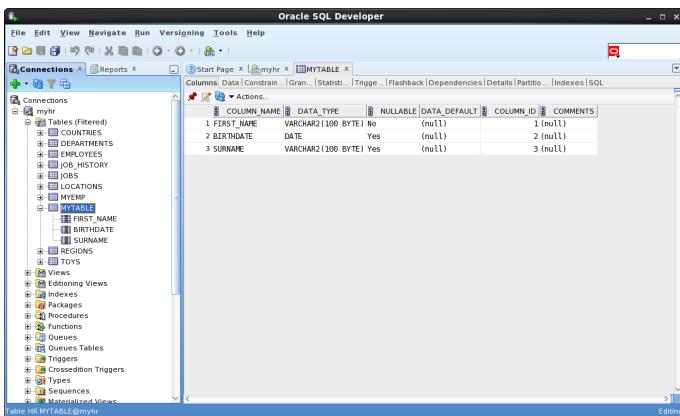


Figure 38: Oracle SQL Developer screen showing the new table, *MYTABLE*.

Click on the tabs displayed in the right pane to see the options available on the table, such as the Data tab, which enables you to add, delete, modify, sort, and filter rows in the table.

Executing a SQL Query in SQL Developer

The SQL Worksheet component included in SQL Developer can be used to execute SQL and PL/SQL statements. Some SQL*Plus commands can also be executed. To execute a SQL statement in SQL Developer:

1. Select the **myhr** tab in the right hand pane. This is the connection created earlier in *Creating a Database Connection in SQL Developer*. The SQL Worksheet component is displayed, and shows an area to enter statements, and a set of tabs below that for further options. If the tab is not available, select the menu **Tools >SQL Worksheet**. You are prompted for the database connection name.

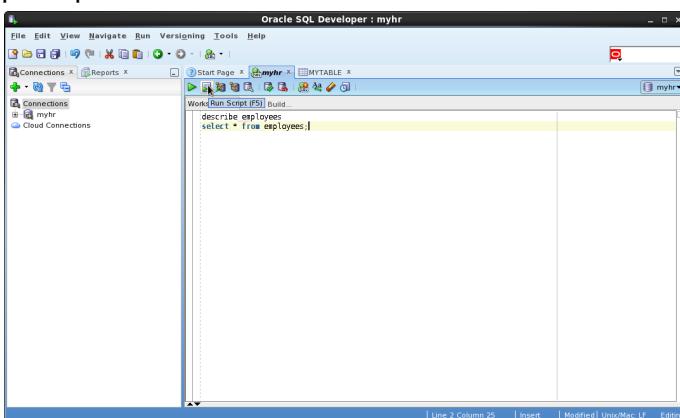


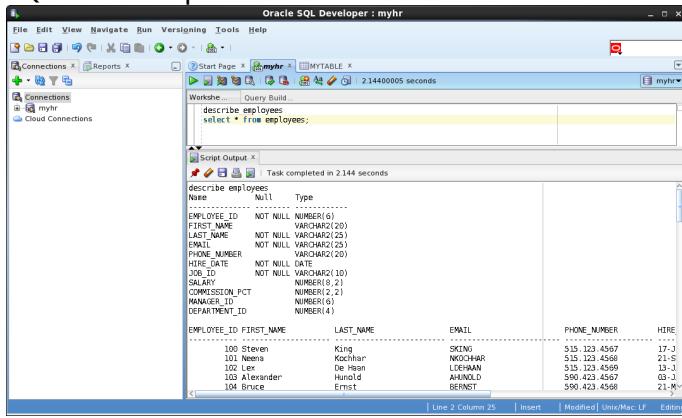
Figure 39: Oracle SQL Developer screen showing the SQL Worksheet.

SQL With Oracle Database

- Enter the following two statements in the SQL Worksheet:

```
describe employees  
select * from employees;
```

Click the **Run Script** icon (the second from the left in the right hand pane), or press **F5**. Both the lines of this script are run and the output is displayed in tabs below. You can view the output of the **SELECT** statement in the **Results** tab, using **F9**, and the output of the whole script (including the **DESCRIBE** and **SELECT** statements) in the **Script Output** tab, by using **F5**. The output in the Script Output window is similar to SQL*Plus output.



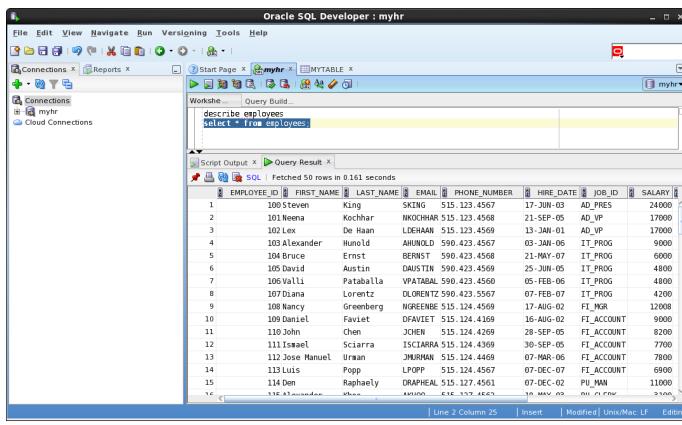
The screenshot shows the Oracle SQL Developer interface. In the top navigation bar, the connection 'myhr' is selected. The main area contains a 'Query Build' tab with the following SQL code:

```
describe employees  
select * from employees;
```

Below the code, there are two tabs: 'Script Output' and 'Query Result'. The 'Script Output' tab displays the results of the 'DESCRIBE' command, showing the structure of the 'employees' table with columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, and JOB_ID. The 'Query Result' tab displays the results of the 'SELECT *' command, listing 14 rows of employee data. The columns shown are: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, and JOB_ID. The data includes entries like Steven King (EMPLOYEE_ID 1), Neena Kochhar (EMPLOYEE_ID 2), Lex De Haan (EMPLOYEE_ID 3), Alexander Hunold (EMPLOYEE_ID 4), Bruce Ernst (EMPLOYEE_ID 5), David Austin (EMPLOYEE_ID 6), Patricia Adams (EMPLOYEE_ID 7), Luis Ernsterz (EMPLOYEE_ID 8), Nancy Greenberg (EMPLOYEE_ID 9), Daniel Faviet (EMPLOYEE_ID 10), John Chien (EMPLOYEE_ID 11), Ismael Sciarra (EMPLOYEE_ID 12), Jose Manuel Urtan (EMPLOYEE_ID 13), Luis Popp (EMPLOYEE_ID 14), and Den Raphaely (EMPLOYEE_ID 15).

Figure 40: Oracle SQL Developer screen showing SQL Worksheet with output.

- If you want to execute a single line of the two-line block, select the line you want to execute and click on the **Execute Statement** icon (the first from the left in the right hand pane), or press **F9**. In this case, the **SELECT** statement (second line) is selected, and the results are shown in the Results tab.



This screenshot is identical to Figure 40, showing the Oracle SQL Developer interface with the same connection 'myhr'. The 'Query Result' tab is active, displaying the results of the executed 'SELECT *' statement from the 'employees' table. The data is identical to Figure 40, listing 14 rows of employee information.

Figure 41: Oracle SQL Developer screen showing SQL Worksheet with output.

Editing, Compiling and Running PL/SQL in SQL Developer

You can use SQL Developer to browse, create, edit, compile, run, and debug PL/SQL.

1. In the **Connections** pane, right click the **Procedures** node for your connection and select **New Procedure** from the context menu. Add the name ADD_DEPT and click **OK**.

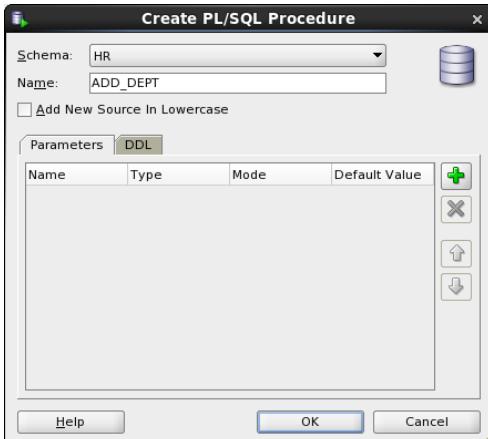


Figure 42: SQL Developer new procedure dialog.

2. The PL/SQL Editor for ADD_DEPT is opened. Change the text so that the code is as follows:

```
create or replace procedure add_dept
(name in departments.department_name%type,
 loc in departments.location_id%type) is
begin
    insert into departments(department_id, department_name, location_id)
    values(departments_seq.nextval, name, loc);
end add_dept;
```

3. Compile the code by selecting the compile icon in the toolbar. If there are errors, they will appear in a Message-Log window below the code.

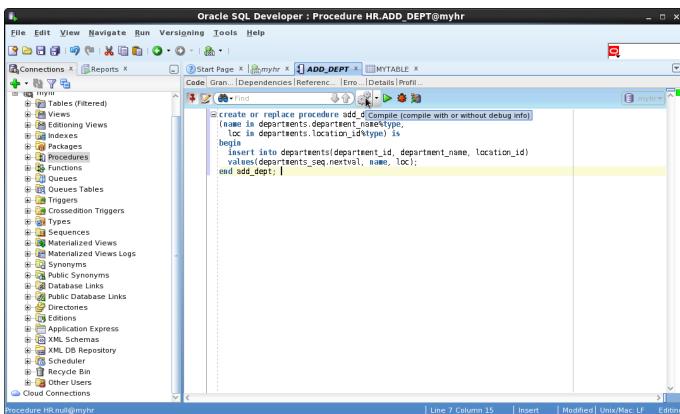


Figure 43: SQL Developer PL/SQL Code Editor.

4. Run the procedure. The green arrow icon runs the code.

SQL With Oracle Database

5. SQL Developer provides a dialog with an anonymous block to help test and run your code. Find and replace the NULL values with values of your own, for example, 'Training' and 1800.

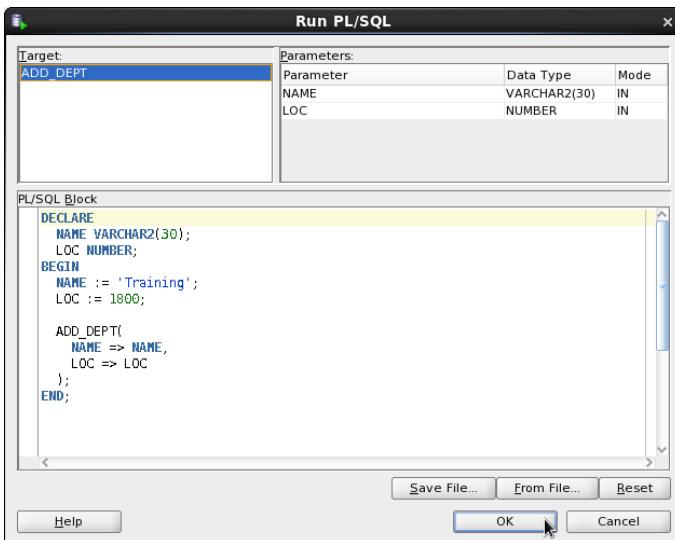


Figure 44: SQL Developer Run PL/SQL dialog.

Click **OK** to run the procedure. Afterward browse the Departments table and go to the Data tab to see the results added.

Running Reports in SQL Developer

There are a number of reports included with SQL Developer that may be useful to you, for example, getting lists of all users in a database, all the tables owned by a user, or all the views available in the data dictionary.

You can also create your own reports (using SQL and PL/SQL), and include them in SQL Developer.

To display the version numbers of the database components using one of the supplied reports:

1. Select the **Reports** tab in the left hand pane. Navigate down to **Reports > Data Dictionary Reports > About Your Database > Version Banner**. After confirming the connection to use, the report is run and the results are displayed in the right hand pane.

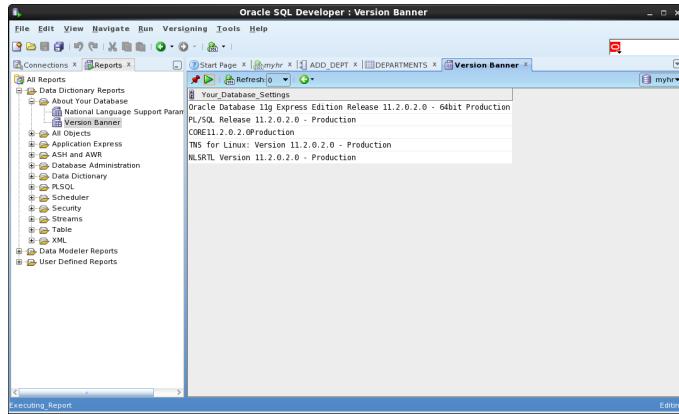


Figure 45: Oracle SQL Developer screen showing output from a report.

2. Click the **Run Report in SQL Worksheet** icon (next to Refresh). The source code is written to the SQL Worksheet, where you could edit it and create a new report, if you wished.

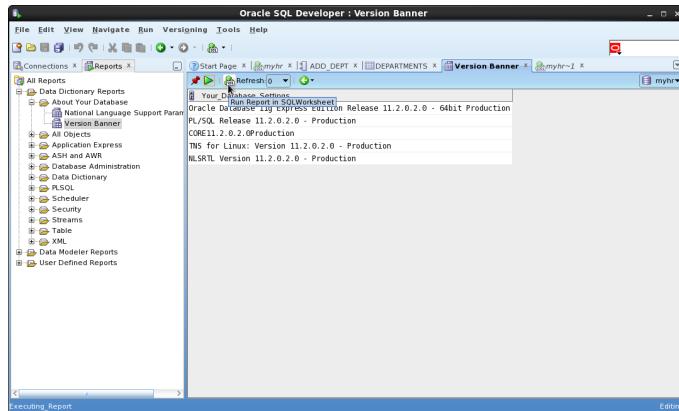


Figure 46: Oracle SQL Developer screen showing the source code of a report.

Creating Reports in SQL Developer

To create your own report, select **User Defined Reports** in the **Reports** pane, and right click. It is good practice to create folder for your reports, to categorize them.

SQL With Oracle Database

3. Right click the **User Defined Reports** node and select **New Folder**. Complete the details in the dialog. You can use this folder to import or add new reports.

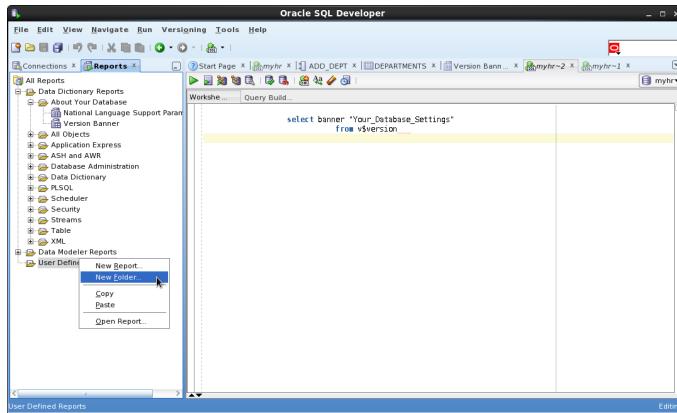


Figure 47: SQL Developer User Defined Reports

4. Select the folder you created and right click to select **New Report**.
5. Give the report a name and description and enter the following SQL text.

```
select last_name,
       department_name,
       city
  from departments d,
       locations l,
       employees e
 where (d.location_id = l.location_id)
   and (d.manager_id = e.employee_id)
   and (e.department_id = d.department_id)
  order by city, department_name
```

6. Notice that the default style is *table*. You can test the report to ensure your SQL is correct, without leaving the dialog. Click the **Test Report** button above the SQL.

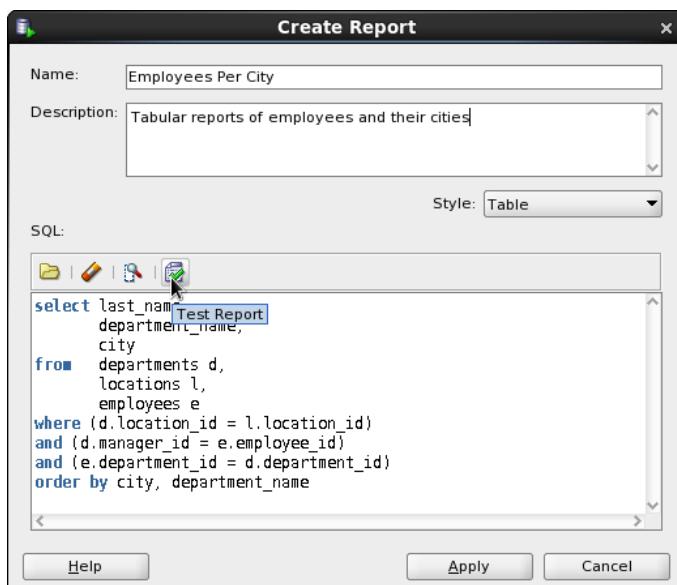


Figure 48: SQL Developer Create Report Dialog

7. Click **Apply** to save and close.
 8. To run the report, select it in the Reports Navigator. You are prompted for a database connection.

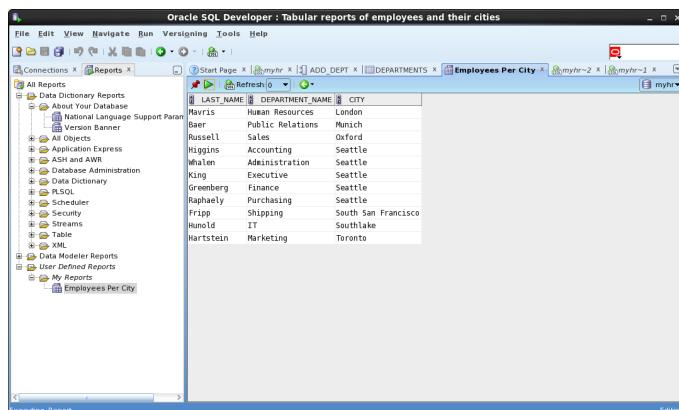


Figure 49: SQL Developer User Defined, Tabular Report

You can create tabular reports, charts, drill down reports and master/detail reports. For all reports, supplied and user defined, you can export, import and share your reports.

SQL With Oracle Database

NETBEANS IDE FOR PHP

The NetBeans IDE (Integrated Developer Environment) for PHP provides tools to make PHP development productive and effective.

The core of NetBeans PHP support is the PHP Editor, which includes code completion and other programming aids. The IDE is also integrated with many third party components including PHP frameworks, the Xdebug debugger, the PHPUnit tester, Jenkins continuous integration, and the ApiGen documentation generator.

This chapter gives a high level overview of NetBeans. NetBeans is extremely popular and there is a large amount of documentation and information already available. Refer to these excellent resources, notably those at <http://netbeans.org/kb/trails/php.html>.

NetBeans Installation

To install NetBeans IDE, go to <http://netbeans.org/downloads>. Download the NetBeans PHP bundle for your operating system or get the platform-independent version. Follow the *NetBeans IDE 7.2 Installation Instructions* at <http://netbeans.org/community/releases/72/install.html>.

If you need additional language support, you can install individual plugins from inside the IDE.

NetBeans Editor Features

Some of the commonly used NetBeans editor features are mentioned below. For more information and to discover other editor features, see the *NetBeans IDE for PHP Editor: Brief Overview* <http://netbeans.org/kb/docs/php/editorguide.html>.

Syntax Highlighting and Code Navigation

Comprehensive highlighting is available for PHP and other languages, including HTML, JSON and XML:

```
// verify user's credentials
if ($_SERVER['REQUEST_METHOD'] == "POST") {
    $logonSuccess = (WishDB::getInstance()->verify_wisher_credentials
        [$_POST['user'], $_POST['userpassword']]);
    if ($logonSuccess == true) {
        session_start();
        $_SESSION['user'] = $_POST['user'];
        header('Location: editWishList.php');
        exit;
    }
}
```

Figure 50: NetBeans IDE syntax highlighting

NetBeans IDE for PHP

NetBeans also gives parameter hints, identifies syntax errors and provides error descriptions.

Code completion and parameter hints

Strong code completion functionality is provided.

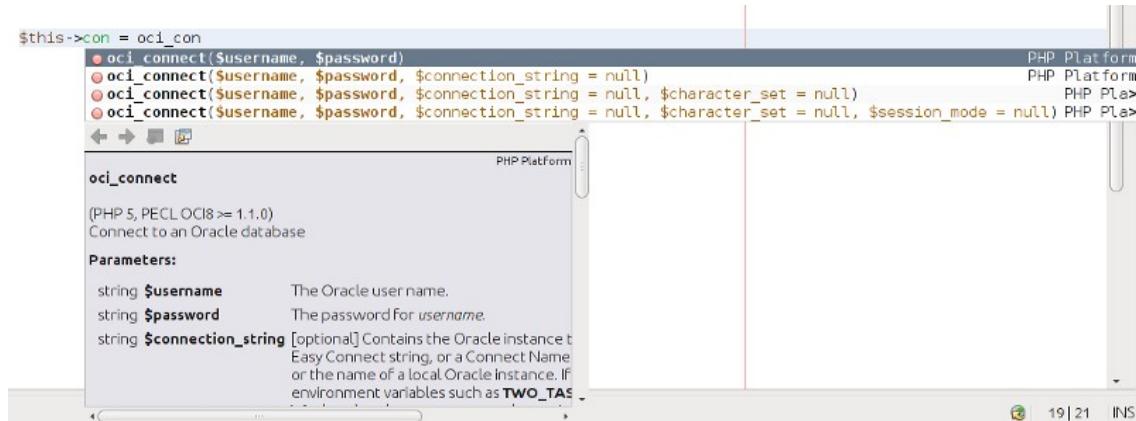


Figure 51: NetBeans code completion

Other IDE features include code folding, smart indenting, formatting, and bracket completion.

Navigator

The Navigator allows easy interpretation of code, and allows direct access to function and method definitions.

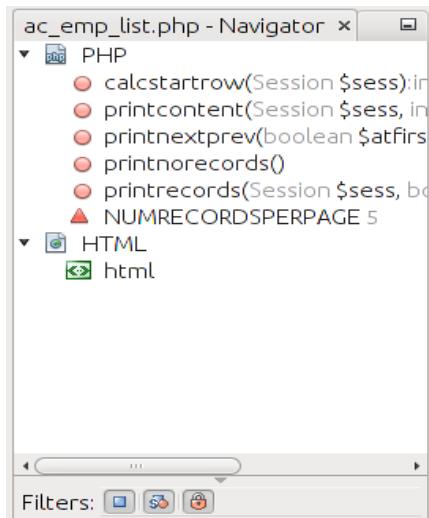


Figure 52: NetBeans Navigator

You can also navigate directly from an occurrence of a variable to the line where the variable is declared or initialized.

Rename refactoring and instant renaming allows safe renaming of all occurrences of an element across multiple files, and has a preview feature.

Third Party Component integration

NetBeans IDE can utilize many standard PHP development tools, including:

- ApiGen documentation generator support
- Code generators to generate class getters, setters, and to override implemented methods
- PHPUnit and Selenium automated testing tools
- Symfony2, Zend Framework and Doctrine2 Frameworks

Development Processes in NetBeans

The NetBeans IDE supports both complex projects and single file development. The IDE has support for Subversion, Mercurial, and Git versioning systems. You can also add CVS support as a plugin.

PHP files can be run as scripts, run with PHP's built-in webserver or deployed remotely. NetBeans supports FTP/SFTP for remote files access.

OCI8 Support in NetBeans

NetBeans IDE for PHP automatically recognizes OCI8 for code completion and provides documentation.

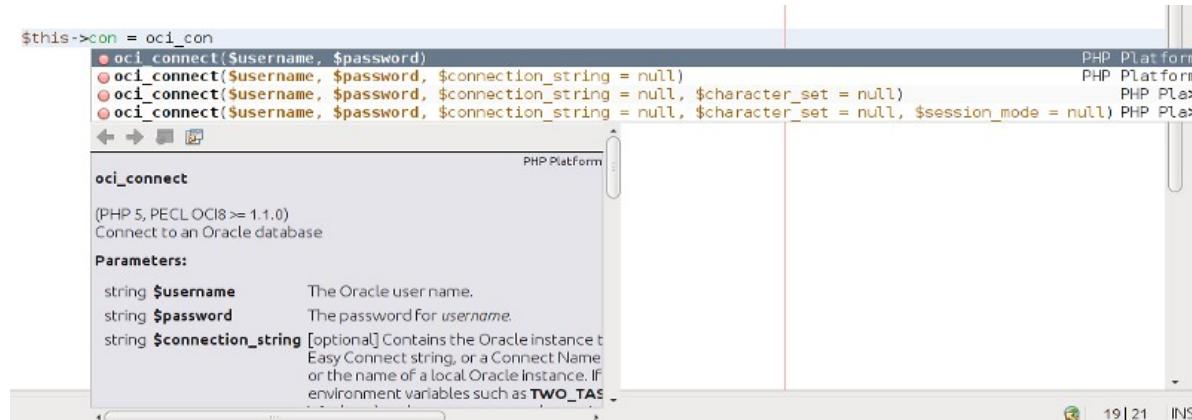


Figure 53: NetBeans code completion

If the local PHP binary registered with NetBeans has the OCI8 extension then scripts using OCI8 will execute in command line or with the local PHP development web server. If you are deploying to a remote host, then OCI8 needs to be installed there.

Oracle Database Support in NetBeans

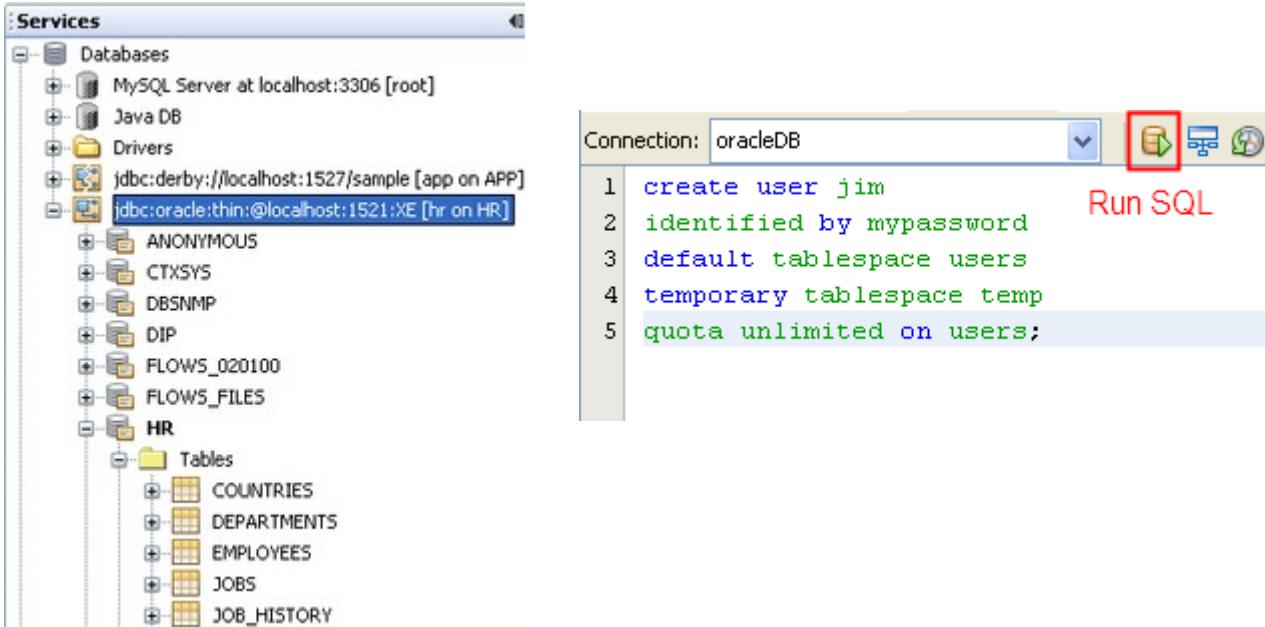
The NetBeans IDE can connect to Oracle Database, view data and meta-data, and also execute SQL commands.

To connect to Oracle, open the Services window and register your database by right-

NetBeans IDE for PHP

clicking the *Databases* node and selecting *New Connection*. You may need to add the path to your database driver as well. For instructions see *Connecting to Oracle Database from NetBeans IDE*, <http://netbeans.org/kb/docs/ide/oracle-db.html>.

NetBeans IDE provides a tree view of database contents and an SQL editor.



Text 1: NetBean Database connection navigator and Oracle SQL area

Note that you do not need to create a connection to your Oracle database from the IDE. You can use another tool to work with SQL and Oracle database objects, and use the NetBeans IDE only for editing PHP, if this is your preferred development process.

NetBeans OCI8 Tutorial

For a full example of PHP development in NetBeans using Oracle Database and the OCI8 extension, see *Creating a Database-Driven Application with NetBeans IDE PHP Editor* <http://netbeans.org/kb/docs/php/wish-list-tutorial-main-page.html>.

Daily and Development NetBeans Builds

To try out upcoming features being added to NetBeans, you can install daily development builds of the IDE. Both public release and development builds can be installed from <http://netbeans.org/downloads/index.html>. Look for the *Development* link.

Getting Help for NetBeans

NetBeans IDE for PHP has great sources of information:

- **Built-in help.** Inside the IDE, click F1 or navigate to *Help > Help Contents* and a help window opens. You can also click the Help button in a dialog to see a description of the options in that dialog. Context-sensitive help is also available for the Editor, Projects,

Getting Help for NetBeans

Services, Files, and other windows. Select any item in a window and press F1 to see the help for that window.

- **Web-based tutorials.** See the PHP Learning Trail, <http://netbeans.org/kb/trails/php.html>.
- **NetBeans PHP Users Forum and Mailing List.** A forum for PHP users is available at <http://forums.netbeans.org/php-users.html>. This forum is mirrored as a mailing list. Subscribe to the mailing list or the forum at <http://netbeans.org/community/lists/index.html>.
- **NetBeans PHP Blog.** The NetBeans PHP development team keeps a blog where you can learn about future plans and the latest NetBeans features. Comments welcome! See <http://blogs.oracle.com/netbeansphp/>.

INSTALLING APACHE HTTP SERVER

This chapter gives you the steps needed to install and configure the Apache HTTP Server for use with PHP. If you already have Apache installed, for example from a Linux software repository, you can skip this chapter.

Steps are given for Oracle Linux and Windows. The procedure to install on other Linux platforms is the same as for Oracle Linux.

Adjust the file names and commands below if the version of Apache you have differs from the instructions.

Apache HTTP Server Packages on Oracle Linux

Apache is available in the *httpd* package on Oracle Linux. If you want to rebuild PHP will also need the *httpd-devel* package installed:

```
# yum install httpd httpd-devel
```

Use the *service* command to control Apache:

```
# service httpd start  
# service httpd stop
```

Note that *service* passes only a limited range of environment variables from your current shell to Apache.

Alternatively you can control Apache with:

```
# /usr/sbin/apachectl start  
# /usr/sbin/apachectl stop
```

Building Apache HTTP Server on Linux

The default version of Apache on your Linux distribution may not be the most recent available. To manually install Apache HTTP Server for use with PHP on Oracle Linux:

1. Download the Apache HTTP Server from <http://httpd.apache.org/download.cgi>. The version used in this installation example is *httpd-2.4.2.tar.bz2*.
2. Download the Apache Portable Runtime (APR) and APR utilities from <http://apr.apache.org/download.cgi>. The files used in this installation are *apr-1.4.6.tar.bz2* and *apr-util-1.4.1.tar.bz2*.
3. Log in as the *root* user and extract the files:

```
# tar -jxvf httpd-2.4.2.tar.bz2  
# tar -jxvf apr-1.4.6.tar.bz2  
# tar -jxvf apr-util-1.4.1.tar.bz2
```

If you downloaded the *.gz* gzipped files, use the *-z* flag instead of the *-j* flag.

Installing Apache HTTP Server

4. Move the APR directories into the web server source, removing the version suffix:

```
# mv apr-1.4.6 httpd-2.4.2/src/lib/apr  
# mv apr-util-1.4.1 httpd-2.4.2/src/lib/apr-util
```

5. Configure and build the web server:

```
# cd httpd-2.4.2  
# ./configure --prefix=/opt/apache --with-included-apr \  
--with-mpm=prefork --enable-mpms-shared  
# make  
# make install
```

When configuring the web server, the options indicate the default processing module is *prefork*, a multi-process mode which is generally recommended because PHP thread safety isn't guaranteed. However enabling the processing module as shared allows later swapping if you want to try a different module. The *--prefix* option sets where Apache HTTP Server will be installed during the command *make install*.

After installation, use the *apachectl* script in the Apache *bin* directory to start and stop Apache HTTP Server:

```
# /opt/apache/bin/apachectl start  
# /opt/apache/bin/apachectl stop
```

Configuring Apache HTTP Server on Linux

The configuration files for Apache are in */etc/httpd/conf* for the system Apache, or */opt/apache/conf* if you did a manual install.

Setting the Apache Server Name

If you installed on a machine with a dynamic IP address, you might see a warning when Apache starts: "*Could not reliably determine the server's fully qualified domain name*". You can prevent this warning by editing *httpd.conf* in the configuration directory and setting:

```
ServerName localhost:80
```

Setting up an Apache User Directory on Linux

You might like to set up the *UserDir* directive so you can run PHP scripts from your own *\$HOME/public_html* directory. Otherwise you will need access to the document root directory, */var/www/html* or */opt/apache/htdocs*. How you enable user directories depends on the Apache version. Typically it requires the *userdir_module* to be loaded and *UserDir* enabled. Your home and *public_html* directories, and PHP or HTML files will also need read and/or execute permissions for the "other" group.

With the *httpd* package on Oracle Linux:

1. Edit */etc/httpd/conf/httpd.conf*
2. Change the *mod_userdir* section to:

```
<IfModule mod_userdir.c>
```

Configuring Apache HTTP Server on Linux

```
UserDir public_html  
</IfModule>
```

3. Set SELinux to *permissive*, otherwise files in the user directory will not be accessible:

```
# setenforce permissive
```

Alternatively */etc/selinux/config* can be updated.

4. Restart the web server:

```
# service httpd restart
```

5. Login as your normal, non-privileged user and make a *public_html* sub-directory:

```
$ mkdir $HOME/public_html  
$ chmod 755 $HOME $HOME/public_html
```

To create a user directory with a manually installed Apache in */opt/apache*:

1. Edit */opt/apache/conf/httpd.conf*

2. Uncomment the line:

```
LoadModule userdir_module modules/mod_userdir.so
```

3. Uncomment the line:

```
Include conf/extra/httpd-userdir.conf
```

4. Restart the web server:

```
# /opt/apache/bin/apachectl restart
```

5. Login as your normal, non-privileged user and make a *public_html* sub-directory:

```
$ mkdir $HOME/public_html  
$ chmod 755 $HOME $HOME/public_html
```

Environment Variables for PHP in Apache on Oracle Linux

When you use Apache HTTP Server with PHP OCI8, you must set some Oracle environment variables before starting the web server. Which variables you need to set are determined by how PHP is installed, how you connect to the database, and what optional settings are desired.

Never set Oracle environment variables in PHP scripts with [putenv\(\)](#). The web server may load Oracle libraries and initialize Oracle data structures before running your script. Using [putenv\(\)](#) causes hard to track errors as the behavior is not consistent for all variables, web servers, operating systems, or OCI8 functions. Variables should be set prior to Apache starting.

Installing Apache HTTP Server

To set environment variables for use by the Oracle Linux packaged Apache, add them to `/etc/sysconfig/httpd`. These variables will be passed to Apache even if you control it with the `service` command. For *Apache* in `/opt/apache` set them in the `/opt/apache/bin/envvars` script. For example, if you install PHP using Oracle Database 11g XE, set:

```
export ORACLE_HOME=/u01/app/oracle/product/11.2.0/xe  
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ORACLE_HOME/lib
```

For more information about setting the environment see *Setting the Oracle Environment on Linux* in the chapter *Installing and Configuring PHP*.

Permissions for PHP OCI8 in Apache on Oracle Linux

When you install PHP OCI8 in Apache (see the next chapter), you will need to make sure that the Apache processes can access the Oracle libraries, messages and globalization data. Without this, Apache may fail to start and you will see errors in the Apache log files. With Oracle 11g or with Oracle Instant Client, giving access can be as simple as making the top level directory readable.

If you are using a non XE edition of Oracle Database 10g Release 10.2, more complex permission changes are needed. With Oracle Database 10g Release 10.2.0.2 onwards, there is a script located in `$ORACLE_HOME/install/changePerm.sh` to change permissions.

Installing Apache HTTP Server on Windows

The following procedure describes how to install the Apache HTTP Server on Windows. The common recommendation is to use PHP in a non-threaded mode via the FastCGI module.

6. Download Apache `httpd-2.2.22-win32-x86-no_ssl.msi` from
<http://httpd.apache.org/download.cgi>
7. Double click the MSI to start the installation wizard. Install "for All Users, on Port 80" because the "only for the Current User" alternative will clash with Oracle Database 11g XE's default port 8080. Do a typical install into the default destination folder, which is *C:\Program Files <x86>\Apache Software Foundation\Apache2.2* on Windows 7, or *C:\Program Files\Apache Software Foundation\Apache2.2* on Windows XP.
8. Download the Apache `mod_fcgid` FastCGI `mod_fcgid-2.3.6-win32-x86.zip` from:
http://httpd.apache.org/download.cgi#mod_fcgid
You may need to navigate to the download directory listing
<http://www.apache.org/dist/httpd/binaries/win32/> if there is no direct link for the Windows binaries on this page.
9. Unzip `mod_fcgid` to the installed Apache 2.2 directory. The *C:\Program Files\Apache Software Foundation\Apache2.2\modules* directory should now have `mod_fcgid.so` and `mod_fcgid.pdb` files.
10. Edit *C:\Program Files\Apache Software Foundation\Apache2.2\conf\httpd.conf* and add:

```
LoadModule fcgid_module modules/mod_fcgid.so
```
11. In `httpd.conf` locate the `<Directory>` section for `htdocs` and add `ExecCGI` to `Options`:

```
<Directory "C:/Program Files/Apache Software Foundation/Apache2.2/htdocs">
```

Installing Apache HTTP Server on Windows

```
...  
Options Indexes FollowSymLinks ExecCGI  
...  
</Directory>
```

You can use the Start menu option to start Apache. This opens a console window showing any error messages. Error messages may also be written to *C:\Program Files\Apache Software Foundation\Apache2.2\logs\error.log*.

You can also use the *ApacheMonitor* utility to start Apache. If you chose to install Apache as a service for all users, it will appear as an icon in your System Tray.

If you have errors, double check your *httpd.conf* file

Starting and Stopping Apache HTTP Server on Windows

Your system tray has an Apache Monitor control that makes it easy to stop and restart the Apache HTTP Server when needed. Alternatively, use the Apache options added to your Windows Start menu.

Testing Apache HTTP Server

You should now test that Apache HTTP Server has been installed properly by starting it on your machine and opening your web browser to *http://localhost/* or *http://127.0.0.1/*. This will display an Apache banner page.

If there is no output or there are errors when starting Apache, review the error log which may be in */var/log/httpd/error_log* (for the Oracle Linux *httpd* package), */opt/apache/logs/error_log* (for a custom install on Linux), or in *C:\Program Files <x86>\Apache Software Foundation\Apache2.2\logs\error_log* (on Windows 7).

Installing Apache HTTP Server

INSTALLING AND CONFIGURING PHP

This chapter discusses the main ways of installing PHP on Linux and Windows. The next chapter discusses installation on Solaris. Installation on other UNIX-like systems is similar to the instructions for Linux.

The Apache HTTP Server generally needs to be installed before installing PHP, and a database should be accessible in your network.

Installing PHP on Linux

There are three main ways to install PHP on Linux:

- Linux distribution packages
- Zend Server packages
- PHP source code

Depending on the PHP installation, the OCI8 extension for Oracle Database access is already installed, or can be installed separately. A common deployment method when OCI8 is compiled separately is to install it as a shared library. However OCI8 can be statically compiled into PHP when building all of PHP from source code. To build the extension you need to have Oracle source header files available. At runtime, OCI8 needs access to Oracle libraries. The headers and libraries can be used from an Oracle database home or from Oracle Instant Client.

The installation scenarios are covered below.

Installing Linux PHP Packages

Most Linux variants supply PHP packages that make installation easy. The caveat is that Linux distribution maintainers generally prefer keeping users' applications stable, so often the packages lag behind the most recent PHP source code. Using older versions of PHP is not recommended by the PHP community.

In Oracle Linux 5 the *php* RPM packages are PHP version 5.1. Oracle Linux 5.6 introduced an additional set of *php53* packages for PHP 5.3. In Oracle Linux 6 the *php* packages contain PHP 5.3. When newer versions of PHP become stable but are not available as packages, you should compile your own binaries, if possible.

The basic PHP installation from Oracle Linux 6 distribution RPMs is described here.

1. Install the PHP command line and Apache modules:

```
# yum install php php-cli
```

Check your local package repository for other extension packages that you may want. Install the *php-devel* RPM package if you plan to install PHP extensions that are only available in source code from PHP's PECL repository or elsewhere.

2. Once PHP is installed, edit */etc/php.ini* and set the *date.timezone* directive to your timezone, for example:

```
date.timezone = America/Los_Angeles
```

Installing and Configuring PHP

See php.net/manual/timezones.php for the available values.

For testing it is helpful to have the directive:

```
display_errors = On
```

This lets you see any problems in your code without having to review the web server logs. Make sure you change this configuration option to Off before releasing your application to users because it is a security risk when errors “leak” information about your configuration.

3. Apache should now be restarted:

```
# service httpd restart
```

Note that *service* clears the environment, so if you need to pass in Oracle environment variables from your shell, restart Apache explicitly with:

```
# /etc/init.d/httpd restart
```

Generally it is better not to rely on the variables being set correctly in the invoking environment. Instead, set them in Apache configuration files such as */etc/sysconf/httpd*. This is discussed later in the chapter.

The bundled packages do not have the PHP OCI8 extension. Use PECL to install it, as described below. Alternatively, Oracle Linux users with an Unbreakable Linux Network (ULN) subscription can install an OCI8 RPM from <https://linux.oracle.com>.

Installing Zend Server Packages on Linux

Zend Server is a pre-built release of PHP from Zend that comes with the OCI8 extension and PDO_OCI driver. It also comes ready with the Oracle Instant Client libraries. Zend Server is available for several platforms, including Oracle Linux. It includes a browser-based management console for configuration and getting updates.

Zend Server CE is free to download and use. A supported edition with enterprise features such as advanced caching and monitoring is also available.

There are several ways to use Zend Server on Oracle Linux:

1. Using an Oracle VirtualBox VM: A quick evaluation, pre-built developer VM with Zend Server and Oracle Database 11g XE can be downloaded from Zend.
2. Using Oracle's Unbreakable Linux Network: ULN subscribers can install Zend Server by subscribing to the "Enterprise Linux 5 Add ons (i386)" or "Enterprise Linux 5 Add ons (x86_64)" channels and installing the zend-server-repo RPM. Equivalent channels for Oracle Linux 6 exist. Zend Server can then be installed:

```
# up2date zend-server-repo  
# yum install zend-server-php-5.3
```

3. Using Oracle's free, public Yum Repository: Follow the steps to enable the Yum repository given in <http://public-yum.oracle.com>. Make sure to enable one of the base channels, for example "el5_u5_base" if you are using Oracle Linux 5 update 5. Also in your new repository configuration file, enable the "Addons" channel. For example:

```
# vi /etc/yum.repos.d/public-yum-el5.repo
```

In the section with the heading [el5_u5_base] enable the channel:

```
enabled=1
```

In the section with the heading [el5_addons] enable the channel:

```
enabled=1
```

Save the file and then install Zend Server with:

```
# yum install zend-server-repo  
# yum install zend-server-php-5.3
```

See the Zend Server page for up to date information on Zend Server and Oracle:

<http://www.oracle.com/technetwork/topics/php/zend-server-096314.html>

Compiling PHP as an Apache Module on Linux

Compiling PHP from source code allows the packages and build options to be explicitly customized. When building PHP from source code, it is common to build PHP first without most extensions, and then later add the extensions as shared libraries. This allows individual extensions to be upgraded or patched without impacting the entire PHP infrastructure. Some people go so far as to configure PHP using `--disable-all` and explicitly enable only the absolutely smallest set of extensions needed.

The basic steps for building PHP as an Apache module are:

1. Login as the `root` user and shutdown Apache:

```
# service httpd stop
```

On Ubuntu use `service apache2 stop`. On other Linux variants you may need to use `/etc/init.d/httpd stop` or run `apachectl stop`.

2. Install the Apache development tools to get the `apxs` utility:

```
# yum install httpd-devel
```

3. Download PHP from <http://www.php.net/downloads.php>.

4. Extract the PHP source code, for example with PHP 5.4.4:

```
# tar -jxf php-5.4.4.tar.bz2  
# cd php-5.4.4
```

If you downloaded the bigger `.tar.gz` file, extract it with `tar -zxf php-5.4.4.tar.gz`.

5. Configure PHP with any desired options, for example:

```
# ./configure --prefix=/opt/php --with-apxs2=/usr/sbin/apxs
```

The example above installs PHP in `/opt/php` and builds PHP's Apache "SAPI" for the packaged `httpd` server. Command line PHP will also be built.

If you installed your own Apache, use the appropriate path to the Apache extension tool, for example `/opt/apache/bin/apxs`. If you have Apache 1.3 instead of Apache 2,

Installing and Configuring PHP

change the `--with-apxs2` option to `--with-apxs`. Other desired options and extensions can be used in the `configure` command. To list all the options, use the command:

```
# ./configure --help
```

You could build the OCI8 extension now by including the `--with-oci8` option as described in detail later in this chapter. Alternatively, you can build OCI8 separately which allows easier upgrades. This is recommended and is also described in this chapter.

To build PHP with a special compiler or compiler options, set any options before running `configure`. Make sure to also remove any configuration cache. For example:

```
# export CC=/bin/cc
# export CFLAGS=-DD64
# rm -rf autom4te.cache config.cache
# ./configure ...
```

PHP 5.4 requires `autoconf` 2.59 or later. Prior to PHP 5.4, older versions of the operating system build tools were needed. If you have both old and new `autoconf` packages installed, PHP builds can be forced to use the appropriate version, for example with:

```
# export PHP_AUTOCONF=autoconf-2.13
# export PHP_AUTOHEADER=autoheader-2.13
```

6. Make and install PHP:

```
# make
# make install
```

7. Copy one of the supplied initialization files `php.ini-development` or `php.ini-production` for PHP. To find the destination configuration file directory, use the `--ini` option to command line PHP:

```
# php --ini
Configuration File (php.ini) Path: /opt/php/lib
Loaded Configuration File:          (none)
Scan for additional .ini files in: (none)
Additional .ini files parsed:     (none)
```

This shows the path is `/opt/php/lib`. Copy one of the template files to `php.ini` in that directory:

```
# cp php.ini-development /opt/php/lib/php.ini
```

8. Edit `php.ini` and set the `date.timezone` directive to your timezone, for example:

```
date.timezone = America/Los_Angeles
```

See <http://php.net/manual/timezones.php> for the available values.

For testing it is helpful to have `display_errors=On` so you see any problems in your code without having to review the web server logs. Make sure you change this configuration option to Off before making your application available to users.

9. Edit Apache's configuration file `/etc/httpd/conf/httpd.conf` and add the following lines:

```
#  
# This section will call PHP for .php and .phps files  
#  
<FilesMatch \.php$>  
    SetHandler application/x-httdp-php  
</FilesMatch>  
<FilesMatch \.phps$>  
    SetHandler application/x-httdp-php-source  
</FilesMatch>
```

Files with `.phps` extension will be shown as highlighted source code. For production systems, this setting should be omitted.

10. If a `LoadModule` line was not already inserted by the PHP install, add it too:

```
LoadModule php5_module /usr/lib64/httpd/modules/libphp5.so
```

11. If OCI8 was configured, set any required Oracle environment variables, such as `ORACLE_HOME`, `LD_LIBRARY_PATH` and `NLS_LANG`. See *Setting the Oracle Environment on Linux* later in this chapter.

When running PHP, set the environment to use the same version of the Oracle libraries as were used during the build process.

12. Restart Apache:

```
# service httpd start
```

You can experiment with PHP configuration, compiler and install options until you have a binary built to your standards. For example, you could benchmark different compilers and customize the optimizer flags. Production sites sometimes reduce binary sizes with the `strip` command.

Installing Oracle Instant Client on Linux for the OCI8 Extension

The PHP OCI8 extension needs access to Oracle "client" libraries. These are contained in the `$ORACLE_HOME/lib` directory. If the database is hosted on another machine or you don't have access to its directories, then install the Oracle Instant Client. This small set of libraries is available through ULN package management for Oracle Linux users, or over HTTP from the Oracle Technology Network. Zend Server already includes Instant Client.

To install Instant Client:

1. Download the Basic and SDK Instant Client packages from ULN or OTN:

Installing and Configuring PHP

<http://www.oracle.com/technetwork/database/features/instant-client/index-100365.html>

Get either the RPM or ZIP files. The even smaller Basic Lite package can be substituted for Basic if its character set and error message language restrictions do not impact your application.

2. If you are using the RPMs, install them as the root user:

```
# rpm -Uvh oracle-instantclient11.2-basic-11.2.0.3.0-1.x86_64.rpm  
# rpm -Uvh oracle-instantclient11.2-devel-11.2.0.3.0-1.x86_64.rpm
```

The first RPM puts the Oracle libraries in `/usr/lib/oracle/11.2/client64/lib` and the second creates headers in `/usr/include/oracle/11.2/client64`.

3. If you are using the Instant Client ZIP files, unzip the Basic and the SDK packages to a directory of your choice, for example `/opt/instantclient_11_2`. The files should be unzipped together so the SDK is in `/opt/instantclient_11_2/sdk`.
4. If Instant Client was installed from the ZIP files, create a symbolic link:

```
# cd /opt/instantclient_11_2  
# ln -s libclntsh.so.11.1 libclntsh.so
```

In Oracle 11.2 the `libclntsh` file suffix retained the 11.1 name to keep compatibility with that first release.

If you use Oracle Instant Client then don't set the `ORACLE_HOME` environment variable while configuring PHP or at runtime.

Configuration Options for Compiling OCI8 With PHP on Linux

To build the OCI8 extension statically into the PHP binary when you compile PHP (as described in the section *Compiling PHP as an Apache Module on Linux*), you will need to use the `--with-oci8` option when you run `configure`. The help description for it is brief:

```
--with-oci8[=DIR]      Include Oracle (OCI8) support. DIR defaults to $ORACLE_HOME.  
                      Use --with-oci8=instantclient,/path/to/instant/client/lib  
                      to use an Oracle Instant Client installation
```

The list below describes how to use the option.

- **--with-oci8**
Without any argument, this looks for Oracle Database libraries in `$ORACLE_HOME`. The variable must previously have been set. It links the OCI8 extension statically into PHP. If `$ORACLE_HOME` is not set, it will look for the Instant Client RPM libraries. PHP must have access to the Oracle libraries and configuration files while building and subsequently when running.
- **--with-oci8=shared**
Same as `--with-oci8` but creates an `oci8.so` shared library. This library can be added to, or removed from, PHP without having to recompile the core PHP executable.

- `--with-oci8=/path/to/full/oracle/home`
Same as `--with-oci8` but uses the specified path instead of looking up `$ORACLE_HOME`.
- `--with-oci8=shared,/path/to/full/oracle/home`
Uses the specified Oracle Database libraries and creates a shared `oci8.so` extension.
- `--with-oci8=instantclient`
Looks for Oracle Instant Client RPMs and uses the most recent version installed. Links the OCI8 extension statically into PHP.
- `--with-oci8=shared,instantclient`
Like the previous option but builds a shared `oci8.so` extension.
- `--with-oci8=instantclient,/path/to/instantclient/libs`
Builds with the Instant Client in the specified directory and links the OCI8 extension statically into PHP.
- `--with-oci8=shared,instantclient,/path/to/instantclient/libs`
Uses the specified Instant Client and creates a shared `oci8.so` extension.

To use a shared `oci8.so` library, edit `php.ini` and add `extension=oci8.so`. Also set `extension_dir` to the directory containing the library.

Before building OCI8 from the PHP source code, navigate to the `ext/oci8/php_oci8.h` file and check whether the version number in the `PHP_OCI8_VERSION` macro is less than the current release of PECL OCI8. If the PECL OCI8 source code is more recent, then install OCI8 as a shared extension using the PECL code.

Installing OCI8 on Linux as a Shared Extension Using PECL

If you have PHP's `pecl` command then it can be used to fetch and install OCI8 with one step.

Since PECL packages are compressed, PHP needs to have the `zlib` extension installed (with the `--with-zlib` option to `configure`) otherwise automatic installation will fail with the error: *The extension 'zlib' couldn't be found*.

If you don't have the `zlib` extension installed, you can still download OCI8 from the PECL website and use `phpize`, as shown in a later section.

PHP 5.4 needs `autoconf` 2.59 or later; earlier versions of PHP need `autoconf` 2.13. Use `PHP_AUTOCONF` to set the version, as shown previously in this chapter.

The PECL OCI8 code can be installed on PHP 4.3.9 onwards. Upgrading OCI8 is **strongly** recommended if you must use PHP 4. Note old PHP versions are no longer maintained by the PHP community and should not be used for new projects.

To install PHP OCI8 using the PECL channel:

1. Shutdown Apache:

```
# service httpd stop
```

2. Remove any existing OCI8 extension:

```
# pecl uninstall oci8
```

3. If you are behind a firewall, set the PEAR proxy which is used by `pecl`, for example:

```
# pear config-set http_proxy http://example.com:80/
```

Installing and Configuring PHP

4. Download and install OCI8:

```
# pecl install oci8
```

Respond to the prompt as if it were a *configure --with-oci8* option. If you have a local database, type the full path to the ORACLE_HOME software location, for example:

```
/u01/app/oracle/product/11.2.0/xe
```

Otherwise if you have Oracle Instant Client 11.2 RPMs, type:

```
instantclient,/usr/lib/oracle/11.2/client64/lib
```

On 32-bit Linux with Instant Client RPMs the line would be

```
instantclient,/usr/lib/oracle/11.2/client/lib
```

Use the absolute path because variables like \$ORACLE_HOME will not be expanded. In some intermediate versions of *pecl* it would first prompt “1-1, ‘all’, ‘abort’, or *Enter to continue*”. If you get this prompt, enter “1” before giving the real response.

5. Edit *php.ini* and add:

```
extension=oci8.so
```

If *extension_dir* is not set, set it to the directory where *oci8.so* was installed, for example:

```
extension_dir=/usr/lib64/php/modules
```

The messages from the *pecl* install will show the correct directory.

6. Set any required Oracle environment variables such as LD_LIBRARY_PATH and NLS_LANG. See *Setting the Oracle Environment on Linux* later in this chapter.

7. Restart Apache:

```
# service httpd start
```

8. Check the installation, as shown at the end of this chapter.

Manually Installing OCI8 on Linux as a Shared Extension

The following steps are the manual equivalent to the previous *pecl install oci8* command. They can be used if you don't have the *pecl* command.

The steps use *phpize*. This will have been installed when you built PHP or it can be found in the *php-devel* or *php53-devel* RPM packages.

To install OCI8 on an existing PHP installation as a shared library:

1. Shutdown Apache:

```
# service httpd stop
```

2. If OCI8 was previously installed, backup or remove the *oci8.so* file

```
# mv /usr/lib64/php/modules/oci8.so /usr/lib64/php/modules/oci8.so.old
```

3. Download the OCI8 extension from PECL, <http://pecl.php.net/package/oci8>
4. Extract and prepare the new code:

```
# tar -zxf oci8-1.4.9.tgz  
# cd oci8-1.4.9  
# phpize
```

5. Configure OCI8. If you have a local database, use:

```
# ./configure --with-oci8=shared,$ORACLE_HOME
```

Otherwise if you have Oracle Instant Client 11.2 use the location of the libraries, for example:

```
# ./configure --with-oci8=\> shared,instantclient,/usr/lib/oracle/11.2/client64/lib
```

6. Build and install the shared library:

```
# make  
# make install
```

7. Edit */etc/php.ini* and add this line:

```
extension=oci8.so
```

8. If *extension_dir* is not set, set it to the directory where *oci8.so* was installed, for example:

```
extension_dir=/usr/lib64/php/modules
```

9. Set any required Oracle environment variables such as *LD_LIBRARY_PATH* and *NLS_LANG*. See the next section in this chapter.

10. Restart Apache:

```
# service httpd start
```

11. Check the installation, as shown at the end of this chapter.

Setting the Oracle Environment for PHP on Linux

The Oracle environment need to be set before PHP loads Oracle libraries so that Oracle configuration files can be found and data structures can be initialized correctly. Environment variables need to be exported in any shell that runs command-line PHP. For web applications they need to be exported before Apache starts. If Apache is started automatically when your machine starts you will need to make sure the environment is set at boot time.

Some common ways to set the environment for Apache are:

- On Oracle Linux and similar distributions using the default Apache package, add environment variables to */etc/sysconfig/httpd*:

```
...  
export ORACLE_HOME=/u01/app/oracle/product/11.2.0/xe
```

Installing and Configuring PHP

```
export LD_LIBRARY_PATH=$ORACLE_HOME/lib:$LD_LIBRARY_PATH
export NLS_LANG=AMERICAN_AMERICA.WE8MSWIN1252
```

- On systems like Ubuntu you can put the variables in `/etc/apache2/envvars`.
- Some Apache 2 installations have a `bin/envvars` script in the Apache directory.
- Sometimes users decide to set the variables in `/etc/init.d/httpd` and always use that script to start the web server.
- Other users use Apache's `PassEnv` directive in `httpd.conf` to pass the environment to PHP. Setting values with Apache's `SetEnv` directive will not work.
- The most generic way to set the whole environment is to create a shell script such as `start_apache.sh` and run it whenever you want to start Apache.

Script 4: start_apache.sh

```
#!/bin/sh

ORACLE_HOME=/u01/app/oracle/product/11.2.0/xe
LD_LIBRARY_PATH=$ORACLE_HOME/lib:$LD_LIBRARY_PATH
NLS_LANG=AMERICAN_AMERICA.WE8MSWIN1252
export ORACLE_HOME LD_LIBRARY_PATH NLS_LANG
echo "Oracle Home: $ORACLE_HOME"

echo Starting Apache
#export > /tmp/envvars # uncomment to debug
/usr/sbin/apachectl start
```

Note: Do not set Oracle environment variables in PHP scripts with `putenv()`. The web server may load Oracle libraries and initialize Oracle data structures before running your script. With persistent connections the environment from one script may affect subsequent scripts. Using `putenv()` causes hard to track errors as the behavior is not consistent for all variables, web servers, operating systems, or OCI8 functions.

If PHP was built with Oracle Instant Client, instead of setting `LD_LIBRARY_PATH` it can be convenient to set the library path system wide. Create a file `/etc/ld.so.conf.d/instantclient.conf` containing the path to the Instant Client libraries:

```
/usr/lib/oracle/11.2/client64/lib
```

Run `ldconfig` to rebuild the system's library search path. Only do this if there is no other Oracle software in use on the machine. It removes the need to set `LD_LIBRARY_PATH` everywhere, but upgrading Instant Client does require remembering to update `instantclient.conf`.

If you have environment related problems such as unexpected connection errors, then create a script `phpinfo.php`:

Script 5: phpinfo.php

```
<?php
phpinfo();
?>
```

Check the output from *phpinfo.php*. Look at the *Environment* section (not the *Apache Environment* section) and make sure the Oracle variables are set to the values you expect. It is useful to check the output using both command line PHP and via a browser.

If you are expecting shell environment variables to be passed to Apache and PHP, do not use the Linux service command to start Apache, since this clears the environment. Instead, directly restart Apache by executing */etc/init.d/httpd restart* or the equivalent on your platform.

Common Oracle Environment Variables on Linux

The variables needed by OCI8 depend on how PHP is installed, how you connect to the database, and what optional settings are desired.

Table 4: Common Oracle environment variables on Linux.

Oracle Environment Variable	Purpose
ORACLE_HOME	The directory containing the Oracle database software. This directory must be accessible by the Apache process. This variable should not be set if PHP uses Oracle Instant Client.
ORACLE_SID	The Oracle Net connect name of the database. Only used when PHP is on the same machine as the database and the connection identifier is not specified in the PHP connect function. Not often set for PHP applications. Not used when PHP is linked with Oracle Instant Client.
LD_LIBRARY_PATH	Set this to include the Oracle libraries, for example <i>\$ORACLE_HOME/lib</i> or <i>/opt/instantclient_11_2</i> . Not needed if the libraries are located by an alternative method, such as with the <i>/etc/ld.so.conf</i> linker path file. On UNIX platforms you will need to set the OS specific equivalent, such as <i>LIBPATH</i> or <i>SHLIB_PATH</i> .
NLS_LANG	Determines the “national language support” globalization options for OCI8. See the chapter <i>Globalization</i> for more details. If not set, a default value will be chosen by Oracle. Setting this is recommended.

Installing and Configuring PHP

Oracle Environment Variable	Purpose
NLS_NUMERIC_CHARACTERS	Commonly set in PHP applications to force Oracle number-to-string conversions to use a period for the decimal separator. Otherwise numeric data values returned from the database in string format will not correctly cast to a number in PHP in locales where the decimal separator is not a period. The variable is ignored if NLS_LANG is not set.
NLS_DATE_FORMAT	Often set in PHP applications to force a consistent date format independent of the locale. The variable is ignored if NLS_LANG is not set.
TNS_ADMIN	The location of the <i>tnsnames.ora</i> and <i>sqlnet.ora</i> configuration files. Needed by PHP if a database connect name from a <i>tnsnames.ora</i> file is used in the OCI8 connect functions and you are using Oracle Instant Client, or if the <i>tnsnames.ora</i> file is not in <i>\$ORACLE_HOME/network/admin</i> . Also needed if using a <i>sqlnet.ora</i> file with Oracle Instant Client, or using files not in <i>\$ORACLE_HOME/network/admin</i> .

With Oracle Database 11g XE, you can set the Oracle environment in a shell by using the *oracle_env.sh* script:

```
$ source /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
```

The *source* command allows the script to set the environment of the shell itself. In some shells use a single period in place of *source*. In C or tcsh shells use the *oracle_env.csh* file.

On other editions of the Oracle database, the */usr/local/bin/oraenv* or */usr/local/bin/coraenv* scripts set the environment. Run one of these scripts before starting Apache. You will be prompted for the database to connect to:

```
$ source /usr/local/bin/oraenv  
ORACLE_SID = [] ? orcl
```

If your database is on a remote machine, you will have to set your local environment manually.

After setting the variables in the shell, you can copy the settings to the appropriate Apache configuration file to make sure they are set no matter how Apache is started.

Signal Handling and Defunct Processes on Linux

In the early days of PHP some Oracle users reported seeing defunct "zombie" processes. The issue has not been reported for a very long time. If it does happen to you, then start Apache with the Oracle Net option **BEQUEATH_DETACH=YES** in your *sqlnet.ora*. Or, to keep the setting specific to PHP, set the environment before starting Apache:

```
export BEQUEATH_DETACH=YES
```

If this doesn't help, only then consider building PHP with **--enable-sigchild**.

Defunct processes might happen if the Oracle code in PHP forks the Oracle server process (called "bequeathing") and the signal handlers in PHP, as the parent of the server process, do not correctly clean up the server process when a database connection is closed. The **BEQUEATH_DETACH=YES** option causes the server processes to do a double fork and be inherited

by "init", which prevents defunct zombies. Oracle Database 10g onwards checks for signal handler clashes and automatically turn on BEQUEATH_DETACH, reducing the need for you to set it. It is not on by default because it adds a few extra CPU cycles and the OS parent process id becomes 1, making it slightly harder to identify the relationship between a server process and the application process using it. You might need to manually set BEQUEATH_DETACH if signal handlers are changed after Oracle has done its heuristic check.

There are some side effects with `--enable-sigchild`. PHP's `pclose()` may return failure. Several PHP bug reports exist for this and PHP code has been patched and broken again in this area.

If PHP is "remote" from the database server you should never need to configure PHP with `--enable-sigchild` or set `BEQUEATH_DETACH=YES`. PHP can be remote physically or because the PHP binary is linked with different Oracle libraries to those used by the database, for example with Oracle Instant Client.

You also don't need to change signal handling if you use Oracle shared or pooled servers. These are the cases where PHP does not fork the Oracle server process directly. The Oracle Net listener does the forking instead and PHP's signal handling won't affect Oracle server process cleanup.

Using PHP-FPM With Apache on Linux

All of the previous Linux configuration examples use a PHP Apache module. The relatively new PHP-FPM "FastCGI Process Manager" module is an alternative that is gaining popularity. PHP-FPM has some advanced process management and isolation controls.

PHP-FPM is documented in the PHP manual and at <http://php-fpm.org/>. It is often used with the `nginx` web server.

Building PHP With PHP-FPM

This example shows building PHP-FPM on 64-bit Oracle Linux using the system Apache. To vary from previous examples, it builds a shared OCI8 extension directly from the PHP source. It assumes Instant Client RPMs are installed. You could choose to build OCI8 statically into the PHP binary, or add it later from PECL using methods previously discussed.

1. Build and install PHP:

```
# tar -jxf php-5.4.4.tar.bz2
# cd php-5.4.4
# ./configure --enable-fpm --prefix=/opt/php544 \
              --with-oci8=shared,instantclient
# make
# make install
```

2. Set up the PHP configuration:

```
# cp php.ini-development /opt/php544/lib/php.ini
```

Edit `php.ini`. Add the OCI8 extension shared module and set `date.timezone` to your timezone, for example:

```
extension=oci8.so
date.timezone = America/Los_Angeles
```

Installing and Configuring PHP

3. Create the PHP-FPM service:

```
# cp sapi/fpm/init.d.php-fpm /etc/init.d/php-fpm  
# chmod +x /etc/init.d/php-fpm  
# chkconfig --add php-fpm
```

4. Create the PHP-FPM configuration file:

```
# cp /opt/php544/etc/php-fpm.conf.default /opt/php544/etc/php-fpm.conf
```

5. Edit *php-fpm.conf* and uncomment the line:

```
pid = run/php-fpm.pid
```

6. Download FastCGI from http://www.fastcgi.com/dist/mod_fastcgi-current.tar.gz

7. Build FastCGI for Apache 2:

```
# tar -zxf mod_fastcgi-current.tar.gz  
# cd mod_fastcgi-2.4.6  
# cp Makefile.AP2 Makefile  
# make top_dir=/usr/lib64/httpd
```

8. Install FastCGI into Apache:

```
# make top_dir=/usr/lib64/httpd install
```

9. Set up Apache by enabling PHP-FPM in */etc/httpd/conf/httpd.conf*:

```
<IfModule mod_fastcgi.c>  
    FastCGIExternalServer /opt/php544/sbin/php-fpm -host 127.0.0.1:9000  
    AddHandler php-fastcgi .php  
    Action php-fastcgi /mycgi  
    ScriptAlias /mycgi /opt/php544/sbin/php-fpm  
  
    # Uncomment for the statistics page (also set pm.status_path in php-fpm.conf)  
    #     <LocationMatch "/status">  
    #         SetHandler php-fastcgi-virt  
    #         Action php-fastcgi-virt /mycgi virtual  
    #     </LocationMatch>  
</IfModule>
```

10. Start PHP-FPM and Apache:

```
# service php-fpm start  
# service httpd start
```

PHP scripts in your *DocumentRoot* directory, for example in */var/www/html*, will be executed by PHP-FPM.

PHP-FPM Statistics

The `php-fpm.conf` file shows a number of PHP-FPM's features, including pool and logging options. One straight-forward PHP-FPM feature is its built in statistics gathering. To use this, first uncomment the `LocationMatch` section in `httpd.conf`:

```
<LocationMatch "/status">
    SetHandler php-fastcgi-virt
    Action php-fastcgi-virt /mycgi virtual
</LocationMatch>
```

1. Edit `php-fpm.conf` and uncomment the `status_path` directive:

```
pm.status_path = /status
```

2. Restart PHP-FPM and Apache:

```
# service php-fpm start
# service httpd start
```

3. Now the statistics page can be called with `http://localhost/status?full`.

The output will be like:

```
pool:          www
process manager: dynamic
start time:    02/Oct/2012:21:55:44 -0700
start since:   4076
accepted conn: 10
listen queue:  0
max listen queue: 0
listen queue len: 128
idle processes: 1
active processes: 1
total processes: 2
max active processes: 1
max children reached: 0

*****
pid:          9322
state:        Idle
start time:   02/Oct/2012:21:55:44 -0700
start since:  4076
requests:     5
request duration: 354
request method: GET
request URI:   /status?full
content length: 0
user:          -
script:        -
last request cpu: 0.00
last request memory: 262144
```

Other options to `/status` allow the statistics to be returned nicely marked up in HTML or in JSON.

Installing and Configuring PHP

Oracle Environment Variables in PHP-FPM

To set Oracle environment variables for PHP-FPM, add them to *php-fpm.conf* like:

```
env[NLS_LANG] = AMERICAN_AMERICA.AL32UTF8  
env[NLS_DATE_FORMAT] = YYYY-MM-DD
```

Instead of using constants, you can assign terminal environment variables to the *env* array. However, beware that *service* clears the environment so you will need to restart PHP-FPM explicitly like:

```
# /etc/init.d/php-fpm restart
```

It is better to set the values explicitly so they always take the expected values.

Installing PHP With OCI8 on Windows

This section describes how to install PHP and Apache on Windows.

Before you do this, consider if you are intending to write PHP code that will run on a different operating system. Most web sites run their production web servers on Linux. Because there are numerous subtle differences between PHP on Windows and PHP on Linux, particularly in file handling and performance, you should avoid developing on Windows unless necessary. You can also run into library clash issues when you have multiple versions of Oracle installed on Windows. A great solution in these cases is to use Oracle's free VirtualBox product and create a virtual machine that runs Oracle Linux.

This example shows installing PHP 5.4.0 using the FastCGI model in Windows. This preferred method of installing avoids any thread-safety issues that PHP's "ZTS" (Zend Thread Safe) Windows binaries are suspected to have.

Before continuing, install Apache as shown in the previous chapter.

Oracle Libraries on Windows

The PHP OCI8 DLL on Windows requires Oracle client DLLs from 10gR2 or later. You should install Oracle Instant Client if you don't have an Oracle Database installed. Otherwise PHP OCI8 can use the client libraries contained in a 32-bit database install such as Oracle 11g XE. If you have 9/R2 libraries, you will have to compile PHP OCI8 yourself, which is out of scope of this book.

Having multiple copies of Oracle libraries installed on Windows can cause hard to resolve conflicts unless your environment is cleanly set. Trying to use PHP with a 64-bit Oracle Database is another issue, since PHP is 32-bit. There are some ugly hacks people have used to resolve the conflicts, including copying the Instant Client libraries to the system directory. These are not recommended. Instead, write a script that sets the environment and then starts Apache. There are tips on this in

https://blogs.oracle.com/opal/entry/using_php_ocil8_with_32-bit_php

Installing Oracle Instant Client on Windows

If you are not using Oracle libraries from a database installation, install Oracle Instant Client with the following steps:

1. Download the Instant Client Basic package for Microsoft Windows (32-bit) from the Instant Client page on the Oracle Technology Network:

Installing PHP With OCI8 on Windows

<http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html>

The Windows 32-bit ZIP file is called *instantclient-basic-nt-11.2.0.3.0.zip* and is around 50 MB in size. If you need to connect to Oracle Database 8*i*, then install the Oracle 10gR2 Instant Client. On some Windows installations, to use Oracle 10gR2 Instant Client you may need to locate a copy of *msvcr71.dll* and put it in your PATH.

Create a new directory, for example, *C:\instantclient_11_2*. Unzip the downloaded file into the new directory.

2. Edit the Windows environment and add the location of the Oracle Instant Client files, *C:\instantclient_11_2*, to the PATH environment variable, before any other Oracle directories. For example, on Windows XP, use **Start > Settings > Control Panel > System > Advanced > Environment Variables**, and edit PATH in the System Variables list. Reboot to make this take effect. Not rebooting is a common source of installation teething troubles.
3. If you are using a *tnsnames.ora* file to define Oracle Network connect names, copy your *tnsnames.ora* file to *C:\instantclient_11_2*, and set the user environment variable TNS_ADMIN to *C:\instantclient_11_2*.
4. Set any other required Oracle globalization language environment variables, such as NLS_LANG. If nothing is set, the default local environment is used. See the *Globalization* chapter, for more information on globalization with PHP and Oracle.
Unset any Oracle environment variables that are not required and should not be set with Oracle Instant Client, such as ORACLE_HOME and ORACLE_SID.

Installing PHP on Windows

Canonical PHP builds are distributed from <http://windows.php.net/download> in ZIP files. In the past, PHP MSI installer files existed, but they are not available from PHP 5.4 onwards.

To install PHP, perform the following steps, substituting the current version of PHP. You must be an administrative user:

1. Download the PHP 5.4.0 "VC9 x86 Non Thread Safe" ZIP package *php-5.4.0-nts-Win32-VC9-x86.zip* from <http://windows.php.net/download/>.

Use the non-thread safe bundle because it will be used with FastCGI.

In Windows Explorer, go to the directory where you downloaded the zip file.

2. Unzip the PHP package to a directory called *C:\php-5.4.0*
3. Copy *php.ini-development* to *C:\php-5.4.0\php.ini*
4. Edit *php.ini* and make the following changes:

Add your localhost timezone, for example:

```
date.timezone = America/Los_Angeles
```

Add the directory containing the PHP extensions:

```
extension_dir = "C:\php-5.4.0\ext"
```

Also in this file remove the semicolon from the beginning of the line:

```
extension=php_oci8_11g.dll
```

Installing and Configuring PHP

This extension can be used when your Oracle client libraries (either from the database install or from Instant Client) are Oracle 11gR2 or greater. If you are only using 10gR2 Oracle client libraries, then instead uncomment the line:

```
extension=php_oci8.dll
```

Only one of *php_oci8.dll* and *php_oci8_11g.dll* can be enabled at any time.

5. Edit the *httpd.conf* file, for example *C:\Program Files\Apache Software Foundation\Apache2.2\conf\httpd.conf*, and add the following lines. Make sure you use forward slashes '/' and not back slashes '\':

```
FcgidInitialEnv PHPRC "C:/php-5.4.0"  
AddHandler fcgid-script .php  
FcgidWrapper "C:/php-5.4.0/php-cgi.exe" .php
```

The PHPRC option tells PHP which directory the *php.ini* file is located in. The other options set up the FastCGI handler.

6. Make sure *mod_fcgid.so* is loaded and the ExecCGI option set as described in the previous chapter.
7. Restart the Apache Server so that you can test your PHP installation. If you have errors, double check your *httpd.conf* and *php.ini* files. If you altered the PATH environment variable make sure you reboot the machine and check PATH includes the Oracle library directory.
8. Check the installation, as shown at the end of this chapter.

Installing OCI8 With Oracle HTTP Server

The Oracle HTTP Server (OHS) is based on Apache. It was part of Oracle's Application Server, and is one of the web servers that can be used by Oracle WebLogic Server (WLS). You can install or upgrade PHP on OHS.

Note: Installing or upgrading PHP in Oracle HTTP Server is not supported (and hence is not recommended) but is technically possible in some circumstances. For any support calls, regardless of whether they are PHP related, Oracle Support may ask you to revert the changes before beginning investigation.

The two sections below show how to build PHP as an Apache module for OHS on Linux.

If you want to install PHP on Oracle WebLogic Server and do not have a web tier, then consider using one and installing PHP into Apache or nginx. Your load balancer or web server can direct HTTP requests as appropriate to WLS or PHP. This will make licensing, installation, management and upgrading of WLS easier. You can also use a WLS web server plug-in, as described in the WLS documentation. The plug-in can proxy requests from, for example, Apache to WLS. You can install PHP into Apache as shown earlier in this chapter.

Installing OCI8 With Oracle HTTP Server 11g on Linux

With OHS 11g, PHP is not included and you must build it yourself. The technical problem faced with building PHP is that the Oracle libraries with Oracle HTTP Server do not include header files. This can be overcome by linking PHP with Oracle Instant Client but care needs to be taken so that AS itself does not use the Instant Client libraries. Otherwise you will get errors or unpredictable behavior.

These steps are very version and platform specific. They may not be technically feasible in all deployments.

To install PHP on OHS 11g:

1. Log on as the *oracle* user and shutdown OHS:

```
$ $ORACLE_INSTANCE/bin/opmnctl stopproc ias-component=ohs1
```

2. Change to the home directory:

```
$ cd $HOME
```

3. Set the Oracle environment:

```
$ export ORACLE_HOME=/your/path/to/your/Oracle/Home
$ export ORACLE_INSTANCE=/your/path/to/OHS
$ export CONFIG_FILE_PATH=$ORACLE_INSTANCE/config/OHS/ohs1
$ export LD_LIBRARY_PATH=$ORACLE_HOME/lib:$ORACLE_HOME/ohs/lib:\>
> $LD_LIBRARY_PATH
```

ORACLE_HOME is the directory with the OHS binaries in `$ORACLE_HOME/ohs/bin`.
ORACLE_INSTANCE is your directory containing the Oracle HTTP Server component "ohs1".

4. Optionally set the PHP compiler environment, for example if you are using a non GNU tool chain:

```
$ export CFLAGS=your-compiler-flags
$ export CC=/path/to/your/compiler
```

5. Download and extract the Oracle Instant Client 11.1.0.7 SDK ZIP file from
<http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html>

6. Copy the new header files from Instant Client to the Oracle home:

```
$ cp instantclient_11_1/sdk/include/*.h $ORACLE_HOME/rdbms/demo
```

7. Download PHP from <http://php.net/downloads.php> and extract to a working directory, for example with PHP 5.3:

```
$ tar -jzf php-5.3.13.tar.bz2
```

8. Change to the extracted PHP directory

```
$ cd php-5.3.13
```

9. If your version of PHP is 5.3.17 or earlier, then patch your *configure* script. Change all

Installing and Configuring PHP

occurrences of:

```
APACHE_VERSION=`expr $4 \* 1000000 + $5 \* 1000 + $6`
```

to

```
APACHE_VERSION=`expr $6 \* 1000000 + $7 \* 1000 + $8`
```

This is because OHS's Apache reports its version number differently from pure Apache. Without this change, PHP will think Apache is version 1.3 and the installation will fail.

10. Configure PHP with your choice of extensions. As a starting point begin with the basic configuration:

```
$ ./configure --disable-all --with-apxs2=$ORACLE_HOME/ohs/bin/apxs \
  --with-oci8=$ORACLE_HOME --disable-rpath \
  --prefix=$ORACLE_HOME --with-config-file-path=$CONFIG_FILE_PATH
```

11. Make the PHP command line binary and Apache module:

```
$ make
```

12. Copy a default *php.ini* file for PHP:

```
$ cp php.ini-development $CONFIG_FILE_PATH/php.ini
```

or

```
$ cp php.ini-production $CONFIG_FILE_PATH/php.ini
```

13. Edit *\$CONFIG_FILE_PATH/php.ini* and add a timezone line:

```
date.timezone = America/Los_Angeles
```

14. If you want to run OCI8 tests, edit *php.ini* and add [E](#) to *variables_order*. This allows PHP's *run-tests.php* script to pass the Oracle environment correctly:

```
variables_order = "EGPCS"
```

This change can be reverted later.

15. Edit *ext/oci8/tests/details.inc* and set the SYSTEM password and connection string to your Oracle database.

16. Test the PHP command-line binary:

```
$ make test
```

17. If all is OK, then install the PHP binaries:

```
$ make install
```

The installation copies the binaries and updates *\$CONFIG_FILE_PATH/httpd.conf*, automatically adding the line:

Installing OCI8 With Oracle HTTP Server

```
LoadModule php5_module libexec/libphp5.so
```

18. Edit `$CONFIG_FILE_PATH/httpd.conf` and add the lines:

```
<FilesMatch \.php$>
    SetHandler application/x-httpd-php
</FilesMatch>
```

19. Restart OHS:

```
$ $ORACLE_INSTANCE/bin/opmnctl startproc ias-component=ohs1
```

Installing OCI8 With Oracle HTTP Server 10g on Linux

Oracle included PHP with its mid-tier Application Server 10g Release 3, giving an out-of-the-box method of using the same web server for PHP and for J2EE applications.

Version 10.1.3.0 of the Application Server (AS) comes with PHP 4.3.11. The AS 10.1.3.2 patchset adds PHP 5.1.2. To use a different version of PHP you may be able to compile your own PHP release using these steps. The same caveats about changing AS exist as for version 11.

A previous installation of AS 10.1.3 is assumed in the steps below. To upgrade the version of PHP in this installation:

1. Log on as the `oracle` user and shut down the Oracle HTTP Server:

```
$ $ORACLE_HOME/opmn/bin/opmnctl stopproc ias-component=HTTP_Server
```

2. Change to the home directory:

```
$ cd $HOME
```

3. Set the `ORACLE_HOME` environment variable to your AS install directory:

```
$ export ORACLE_HOME=$HOME/product/10.1.3/OracleAS_1
```

4. Download the Oracle 10g or 11g Basic and SDK Instant Client ZIP packages from the Instant Client page on the Oracle Technology Network:

<http://www.oracle.com/technology/tech/oci/instantclient/instantclient.html>

5. Extract the ZIP files, for example, if on a 32-bit Linux:

```
$ unzip basic-linux-10.2.0.5.0.zip
$ unzip sdk-linux-10.2.0.5.0.zip
```

6. Change to the Instant Client directory and symbolically link `libclntsh.so.10.1` to `libclntsh.so`:

```
$ cd instantclient_10_2
$ ln -s libclntsh.so.10.1 libclntsh.so
```

The Instant Client RPMs could also be used, in which case this last step is unnecessary.

Installing and Configuring PHP

Be wary of having Instant Client in `/etc/ld.so.conf` since Instant Client libraries can cause conflicts with AS. The `opmnctl` tool may fail with the error *Main: NLS Initialization Failed!!.*

7. Download PHP from <http://php.net/downloads.php> and extract the file to a working directory, for example with PHP 5.2.7:

```
$ tar -jxf php-5.2.17.tar.bz2
```

8. Edit `$ORACLE_HOME/Apache/Apache/conf/httpd.conf` and comment out the PHP `LoadModule` line by prefixing it with `#`:

```
# LoadModule php4_module libexec/libphp4.so
```

If you had enabled PHP 5 for AS 10.1.3.2, the commented line should be:

```
# LoadModule php5_module libexec/libphp5.so
```

9. Back up the `libphp4.so` or `libphp5.so` library in `$ORACLE_HOME/Apache/Apache/libexec` since it will be replaced.

10. Set environment variables required for the build to complete:

```
$ export PERL5LIB=$ORACLE_HOME/perl/lib  
$ export LD_LIBRARY_PATH=$ORACLE_HOME/lib:$LD_LIBRARY_PATH  
$ export CFLAGS=-DLINUX
```

There is no need to set `CFLAGS` if you have AS 10.1.3.2. It is needed with AS 10.1.3.0 to avoid a duplicate prototype error with `gethostname()` that results in compilation failure.

11. Configure PHP:

```
$ cd php-5.2  
$ ./configure \  
> --prefix=$ORACLE_HOME/php \  
> --with-config-file-path=$ORACLE_HOME/Apache/Apache/conf \  
> --with-apxs=$ORACLE_HOME/Apache/Apache/bin/apxs \  
> --with-oci8=instantclient,$HOME/instantclient_10_2
```

With AS 10.1.2 and older Instant Client releases, some users also specify `--disable-rpath`.

12. Make PHP:

```
$ make
```

13. Test the PHP command-line binary:

```
$ make test
```

14. If all is OK, then install PHP:

```
$ make install
```

The installation copies the binaries and updates `$ORACLE_HOME/Apache/Apache/conf/httpd.conf`, automatically adding the line:

```
LoadModule php5_module libexec/libphp5.so
```

15. Back up and update `$ORACLE_HOME/Apache/Apache/conf/php.ini` with options for PHP 5, for example all the new OCI8 directives. Refer to `php.ini-recommended` for new options. Particularly if you are using PHP 5.3, set the `date.timezone` directive to your timezone, for example:

```
date.timezone = America/Los_Angeles
```

See <http://php.net/manual/en/timezones.php> for the available values.

16. The Oracle HTTP Server can now be restarted:

```
$ $ORACLE_HOME/opmn/bin/opmnctl startproc ias-component=HTTP_Server
```

Reminder: these steps invalidate all support for AS, not just for the PHP component, and they should not be used in production environments.

The Oracle HTTP Server document root is

```
$ORACLE_HOME/Apache/Apache/htdocs
```

(Yes, *Apache* is repeated twice). Files with `.php` extensions in this directory will be executed by PHP. Files with a `.phps` extension will be displayed as formatted source code.

Installing the PDO Extension

The PDO_OCI driver for PHP's PDO extension is a separate interface to the Oracle Database. Using it is not recommended but you may wish to experiment with it. PDO_OCI misses some very important scaling and performance features available in OCI8, it isn't stable, and currently it has no maintainer.

To use PDO_OCI with Oracle, install the PDO extension and the PDO_OCI database driver. The PDO extension and drivers are included in PHP from release 5.1.

You can install PDO_OCI and OCI8 at the same time by combining the appropriate options to `configure`. No PHP code is shared by the OCI8 extension and PDO_OCI driver.

Installing PDO_OCI on Linux

The steps shown below show compiling PDO for PHP 5.4 on Oracle Linux. This procedure works for all versions of PHP after release 5.1.

1. Download PHP 5.4.4 from <http://php.net/downloads.php>.
2. Log in as the *root* user and extract the PHP source code using the following commands:

```
# tar -jxf php-5.4.4.tar.bz2  
# cd php-5.4.4
```

If you downloaded the `.tar.gz` file, extract it with `tar -zxf`.

3. Configure PHP with options like:

Installing and Configuring PHP

```
# export ORACLE_HOME=/u01/app/oracle/product/11.2.0/xe
# ./configure \
>   --with-apxs2=/usr/sbin/apxs \
>   --enable-pdo \
>   --with-pdo-oci=$ORACLE_HOME
```

Note there is no “8” in the `--with-pdo-oci` option name. Review the output of `configure` and check that the extension was enabled successfully before continuing.

If you want to build PDO_OCI with the Oracle Instant Client RPMs, change the `--with-pdo-oci` option to:

```
--with-pdo-oci=instantclient,/usr,11.2
```

This indicates to use the Instant Client in `/usr/lib/oracle/11.2`.

If Instant Client ZIP files are used, the option should be, for example:

```
--with-pdo-oci=instantclient,/opt/instantclient_11_2,11.2
```

The trailing version number in this command for ZIP installs of Instant Client is only used for sanity checking and display purposes.

With Instant Client ZIP files, if the error *I'm too dumb to figure out where the libraries are in your Instant Client install* is shown it means the symbolic link from `libclntsh.so` to `libclntsh.so.11.1` (or the appropriate version) is missing. This link needs to be manually created for the ZIP file install.

4. Build and install PHP.

```
# make
# make install
```

5. Following steps similar to the previous OCI8 sections to create the `php.ini` configuration file and configure the Oracle environment for Apache.

Installing PDO_OCI on Windows

If you want to use the PDO_OCI driver for the PDO extension follow the previous steps for installing OCI8 on Windows and add the extension to `php.ini`:

```
extension=php_pdo_oci.dll
```

You can have both PDO_OCI and OCI8 enabled together.

Checking OCI8 and PDO_OCI Installation

To confirm PHP was installed correctly, create a script `phpinfo.php` that shows the PHP configuration settings. The file should be in a directory Apache can read, such as the document root. This is specified by the `DocumentRoot` setting in the `httpd.conf` file. On Oracle Linux the directory is `/var/www/html`.

Script 6: `phpinfo.php`

```
<?php
```

Checking OCI8 and PDO_OCI Installation

```
phpinfo();  
?>
```

Load the script in your browser using <http://localhost/phpinfo.php>. Alternatively, the script can be run in a terminal window with command line PHP, after adding the directory containing PHP to your PATH environment variable, and adding the Oracle library directory to LD_LIBRARY_PATH for Linux:

```
$ php phpinfo.php
```

or you could simply run

```
$ php -i
```

If you have more than one version of PHP on your system beware the output in a browser and from the command line might be different.

In the output, check the *Loaded Configuration File* entry shows the *php.ini* that the previous installation steps created.

The Environment section should show the Oracle environment variables. See *Setting the Oracle Environment on Linux* earlier in this chapter.

If OCI8 was installed, there will be a section for it:

oci8

OCI8 Support	enabled
Version	1.3.4
Revision	\$Revision: 1.269.2.16.2.38.2.23 \$
Active Persistent Connections	0
Active Connections	0
Oracle Instant Client Version	11.1
Temporary Lob support	enabled
Collections support	enabled

Directive	Local Value	Master Value
oci8.connection_class	<i>no value</i>	<i>no value</i>
oci8.default_prefetch	100	100
oci8.events	Off	Off
oci8.max_persistent	-1	-1
oci8.old_oci_close_semantics	Off	Off
oci8.persistent_timeout	-1	-1
oci8.ping_interval	60	60
oci8.privileged_connect	Off	Off
oci8.statement_cache_size	20	20

Figure 54: *phpinfo()* output when OCI8 is enabled.

The parameter values are discussed in later chapters.

If PDO_OCI was installed, its configuration section will look like:

Installing and Configuring PHP

PDO	
PDO support	enabled
PDO drivers	sqlite, oci, sqlite2
PDO_OCI	
PDO Driver for OCI 8 and later	enabled

Figure 55: `phpinfo()` when `PDO_OCI` is enabled.

The chapter *Connecting to Oracle Using OCI8* shows how to use OCI8 to connect to Oracle Database.

INSTALLING PHP AND APACHE ON ORACLE SOLARIS

This chapter discusses installing Apache and PHP IPS packages on Oracle Solaris 11.1. Solaris includes Apache 2.2, PHP 5.2 and PHP 5.3 IPS packages, along with several PHP extensions. PHP is currently only available for 32-bit.

At time of writing, no IPS packages are available for Oracle Instant Client or PHP OCI8 so they must be installed manually. Check OTN for the latest information, <http://www.oracle.com/technetwork/server-storage/solaris11/overview/index.html>

Oracle Database XE is not available on Solaris, but you can install and connect to another edition of Oracle, or connect to a database on a different machine.

Installing Apache on Oracle Solaris 11.1

The Apache webserver can be installed from an IPS repository. To install Apache, follow these steps:

6. Check that a Solaris IPS repository is configured. For example:

```
$ pkg publisher
PUBLISHER          TYPE      STATUS P LOCATION
solaris            origin    online F
http://ipkg.us.oracle.com/solaris11/release/
```

7. Check if Apache is already installed:

```
$ pkg info apache-22
  Name: web/server/apache-22
  Summary: Apache Web Server V2.2
  Description: The Apache HTTP Server Version 2.2
  Category: Web Services/Application and Web Servers
  State: Installed
  Publisher: solaris
  Version: 2.2.22
  Build Release: 5.11
  Branch: 0.175.1.0.0.24.0
  Packaging Date: August 20, 2012 03:34:45 PM
  Size: 9.15 MB
  FMRI: pkg://solaris/web/server/apache-22@2.2.22,5.11-
  0.175.1.0.0.24.0:20120820T153445Z
```

Commonly Apache will be present, so the next step can be skipped.

8. If Apache is not installed, then install it using the appropriate privileges, for example with *su*, *sudo*, or *pfexec*:

```
# pkg install web/server/apache-22
```

Installing PHP and Apache on Oracle Solaris

9. Apache can be started with:

```
# svcadm enable apache22
```

10. To shut Apache down, disable it with:

```
# svcadm disable apache22
```

11. The status of Apache can be checked with:

```
# svcs apache22
STATE      STIME      FMRI
disabled   15:44:11  svc:/network/http:apache22
```

If Apache is running, its STATE will be *online*.

Installing PHP on Oracle Solaris 11.1

Oracle Solaris 11.1 currently includes PHP versions 5.2 and 5.3. Either or both may be installed, however only one of them can be used by Apache at any one time. Solaris 11.0 only has PHP 5.2 IPS packages.

1. To find all of PHP's IPS packages in the remote repository do:

```
$ pkg search -o pkg.name -r pkg.fmri:web/php* OR \
    pkg.fmri:web/server/*php*
PKG.NAME
web/php-52
web/php-52/documentation
web/php-52/extension/php-apc
web/php-52/extension/php-idn
web/php-52/extension/php-memcache
web/php-52/extension/php-mysql
web/php-52/extension/php-pear
web/php-52/extension/php-suhosin
web/php-52/extension/php-tcpwrap
web/php-52/extension/php-xdebug
web/php-53
web/php-53/documentation
web/php-53/extension/php-apc
web/php-53/extension/php-idn
web/php-53/extension/php-memcache
web/php-53/extension/php-mysql
web/php-53/extension/php-pear
web/php-53/extension/php-suhosin
web/php-53/extension/php-tcpwrap
web/php-53/extension/php-xdebug
web/php-common
web/server/apache-22/module/apache-php5
web/server/apache-22/module/apache-php52
web/server/apache-22/module/apache-php53
```

2. Install PHP 5.3 with *pkg* using appropriate privileges such as via *su*, *sudo*, or *pfexec*:

Installing PHP on Oracle Solaris 11.1

```
# pkg install web/php-53 web/server/apache-22/module/apache-php53
      Packages to install: 10
      Packages to update: 1
      Mediators to change: 1
      Create boot environment: No
Create backup boot environment: No

      DOWNLOAD          PKGS       FILES      XFER (MB)
      SPEED
Completed        11/11      606/606    22.2/22.2
2.0M/s

      PHASE           ITEMS
      Removing old actions      1/1
      Installing new actions   962/962
      Updating package state database      Done
      Updating image state       Done
      Creating fast lookup database      Done
```

- Similarly, PHP 5.2 packages can be installed with:

```
# pkg install web/php-52 web/server/apache-22/module/apache-php52
```

- Review all the PHP packages installed:

```
# pkg list '*php*'
NAME (PUBLISHER)          VERSION          IFO
web/php-52                5.2.17-0.175.1.0.0.24.0  i--
web/php-52/extension/php-apc 3.0.19-0.175.1.0.0.24.0  i--
web/php-52/extension/php-idn 0.2.0-0.175.1.0.0.24.0  i--
web/php-52/extension/php-memcache 2.2.5-0.175.1.0.0.24.0  i--
web/php-52/extension/php-mysql 5.2.17-0.175.1.0.0.24.0  i--
web/php-52/extension/php-pear 5.2.17-0.175.1.0.0.24.0  i--
web/php-52/extension/php-suhosin 0.9.29-0.175.1.0.0.24.0  i--
web/php-52/extension/php-tcpwrap 1.1.3-0.175.1.0.0.24.0  i--
web/php-52/extension/php-xdebug 2.0.5-0.175.1.0.0.24.0  i--
web/php-53                5.3.14-0.175.1.0.0.24.0  i--
web/php-53/extension/php-apc 3.1.9-0.175.1.0.0.24.0  i--
web/php-53/extension/php-idn 0.2.0-0.175.1.0.0.24.0  i--
web/php-53/extension/php-memcache 3.0.6-0.175.1.0.0.24.0  i--
web/php-53/extension/php-mysql 5.3.14-0.175.1.0.0.24.0  i--
web/php-53/extension/php-pear 5.3.14-0.175.1.0.0.24.0  i--
web/php-53/extension/php-suhosin 0.9.33-0.175.1.0.0.24.0  i--
web/php-53/extension/php-tcpwrap 1.1.3-0.175.1.0.0.24.0  i--
web/php-53/extension/php-xdebug 2.2.0-0.175.1.0.0.24.0  i--
web/php-common              11.1-0.175.1.0.0.24.0  i--
web/server/apache-22/module/apache-php52 5.2.17-0.175.1.0.0.24  i--
web/server/apache-22/module/apache-php53 5.3.14-0.175.1.0.0.24  i--
```

- As required in PHP 5.3, edit */etc/php/5.3/php.ini* and set the timezone, for example:

```
date.timezone = America/Los_Angeles
```

See <http://php.net/manual/timezones.php> for available values.

- Start Apache:

Installing PHP and Apache on Oracle Solaris

```
# svcadm enable apache22
```

7. Confirm Apache started with:

```
# svcs apache22
STATE      STIME      FMRI
online     15:43:01  svc:/network/http:apache22
```

8. Check PHP's configuration with the command line:

```
$ php -i
```

Alternatively create a script in the webserver document root directory, `/var/apache2/2.2/htdocs/phpinfo.php`:

```
<?php
phpinfo();
?>
```

Now, in a web browser, open the URL <http://localhost/phpinfo.php>. If PHP is installed, the output will show the version of PHP and the extensions that are configured.

Changing the Version of PHP used by Apache

If more than one version of PHP installed, you can choose which one is used by Apache. To see the current version of PHP use `phpinfo()` or the package mediator command:

```
$ pkg mediator
MEDIATOR VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
java    system   1.7    system
php    system   5.3    system
python vendor   2.6    vendor
```

This example shows that Apache is using PHP 5.3.

To see all the possible values for the package mediator:

```
$ pkg mediator -a
MEDIATOR VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
java    system   1.6    system
java    system   1.7    system
php    system   5.2    system
php    system   5.3    system
python vendor   2.6    vendor
```

To change the version of PHP used by Apache, first disable Apache:

```
# svcadm disable apache22
```

Then switch to PHP 5.2:

```
# pkg set-mediator -V 5.2 php
```

Restart Apache:

```
# svcadm enable apache22
```

Installing Oracle Instant Client on Oracle Solaris 11.1

Oracle client libraries must be available for OCI8. If there is no Oracle Database installed on your machine, install the lightweight Oracle Instant Client:

1. Download the Oracle Instant Client for Solaris 32-bit from OTN, choosing SPARC or x86 as appropriate: <http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html>
Install the *instantclient-basic-solaris* and *instantclient-sdk-solaris* packages.
Alternatively, instead of the “basic” package, you can install *instantclient-basiclite-solaris*.
2. Unzip the packages into */opt/instantclient_11_2*, for example:

```
# cd /opt
# unzip instantclient-basic-solaris.sparc32-11.2.0.3.0.zip
# unzip instantclient-sdk-solaris.sparc32-11.2.0.3.0.zip
```

3. Create a symbolic link for the client shared library:

```
# cd /opt/instantclient_11_2
# ln -s libclntsh.so.11.1 libclntsh.so
```

The final directory layout for the “basic” and “sdk” packages should look like:

```
# ls /opt/instantclient_11_2
adrci          libclntsh.so.11.1 libocijdbc11.so uidrvci
BASIC_README   libnnz11.so      ojdbc5.jar    xstreams.jar
genezi         libocci.so.11.1  ojdbc6.jar
libclntsh.so   libociei.so    sdk
```

Installing OCI8 on Oracle Solaris 11.1

The OCI8 extension can be installed from PECL as a shared extension.

1. Make sure *autoconf* and *system/header* are installed.

```
# pkg install autoconf
# pkg install system/header
```

2. A C compiler is needed to build OCI8. Install Solaris Studio by downloading it and following the documentation:

<http://www.oracle.com/technetwork/indexes/downloads/index.html#tools>

If necessary, add the Solaris Studio binary directory to your PATH, for example,

```
# export PATH=$PATH:\n/opt/SolarisStudio12.3-solaris-x86-bin/solarisstudio12.3/bin
```

Solaris Studio is preferred because of its optimizations but you could alternatively, install GCC:

Installing PHP and Apache on Oracle Solaris

```
# pkg install gcc-dev
```

3. Shutdown Apache:

```
# svcadm disable apache22
```

4. If you are behind a firewall, set the PEAR proxy which is used by PECL. For example:

```
# /usr/php/bin/pear config-set http_proxy http://your-proxy.com:80
```

5. Download and install OCI8:

```
# /usr/php/bin/pecl install oci8
```

When prompted for the path to ORACLE_HOME respond with the full path to the Oracle home directory or, if you installed Oracle Instant Client, enter [instantclient](#), [/opt/instantclient_11_2](#). Note the installer will not expand environment variables in your response.

6. For PHP 5.3 edit */etc/php/5.3/php.ini* and add:

```
extension=oci8.so
```

If you installed PHP 5.2 then add this directive to */etc/php/5.2/php.ini*.

7. Set any required environment variables, such as LD_LIBRARY_PATH, in your shell. For Apache the variables must be set in */etc/apache2/2.2/envvars*:

```
export LD_LIBRARY_PATH=/opt/instantclient_11_2
```

See *Setting the Oracle Environment Variables on Linux* in the previous chapter. Oracle Solaris uses similar variables as Linux, such as NLS_LANG and TNS_ADMIN.

8. Restart Apache:

```
# svcadm enable apache22
```

Calling the previous *phpinfo.php* script will show the OCI8 extension and its options. The variable LD_LIBRARY_PATH should be seen in the *Environment* section.

You can remove the OCI8 extension by reverting the extension entry in *php.ini* and physically removing the extension with:

```
# /usr/php/bin/pecl uninstall oci8
```

The next chapter shows how to use OCI8 to connect to Oracle Database.

CONNECTING TO ORACLE USING OCI8

This chapter covers connecting to an Oracle database from your PHP application, showing the forms of Oracle connection and how to tune them. A later chapter *PHP Connection Pooling and High Availability* discusses connection pooling and how it applies to connection management.

Before attempting to connect, review the section *Setting the Oracle Environment on Linux* in Chapter 8 and make sure that your environment is configured appropriately.

The examples use the “human resources” HR schema, a demonstration user account installed with the database. Use SQL*Plus to unlock the account and set a password, as described in the chapter *SQL With Oracle Database*.

Depending on the version of the Oracle Client libraries that OCI8 is linked with, you can connect to various versions of Oracle Database. Oracle Support Note 207303.1 details the full Oracle client-server compatibility matrix.

Oracle Connection Example

Once you have installed PHP OCI8, Apache, and have the credentials to access a local or remote database, then you can test an OCI8 script. Create *oci8.php* in Apache's DocumentRoot directory, where you created *phpinfo.php* in the previous chapter:

Script 7: oci8.php

```
<?php

$c = oci_pconnect("hr", "welcome", "localhost/XE");
if (!$c) {
    $e = oci_error();
    trigger_error('Could not connect to database: '. $e['message'], E_USER_ERROR);
}

$s = oci_parse($c, "select city from locations order by city");
if (!$s) {
    $e = oci_error($c);
    trigger_error('Could not parse statement: '. $e['message'], E_USER_ERROR);
}
$r = oci_execute($s);
if (!$r) {
    $e = oci_error($s);
    trigger_error('Could not execute statement: '. $e['message'], E_USER_ERROR);
}

echo "<table border='1'>\n";
$ncols = oci_num_fields($s);
echo "<tr>\n";
for ($i = 1; $i <= $ncols; ++$i) {
    $colname = oci_field_name($s, $i);
    echo "  <th><b>".htmlentities($colname, ENT_QUOTES)."</b></th>\n";
}
```

Connecting to Oracle Using OCI8

```
}

echo "</tr>\n";

while (($row = oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_NULLS)) != false) {
    echo "<tr>\n";
    foreach ($row as $item) {
        echo "  <td>" . ($item === null ? htmlentities($item,
                                                ENT_QUOTES) : "&nbsp;") . "</td>\n";
    }
    echo "</tr>\n";
}
echo "</table>\n";
?>
```

The script connects as the HR user to the local installation of Oracle 11g XE. The password is “welcome”. Load it in a browser with: <http://localhost/oci8.php>. The output is like:

CITY
Beijing
Bern
Bombay
Geneva
Hiroshima
London
Mexico City
Munich
Oxford

Figure 56: Output from *oci8.php*.

If you have errors, read further in this chapter about connections and troubleshooting.

Oracle Connection Types

There are three ways to connect to an Oracle database in a PHP application: using standard connections, unique connections, or persistent connections. Each method returns a connection *resource* that is used in subsequent OCI8 calls.

Independent of which one of these PHP connection methods are used in applications, the database server can be configured to handle its end of the connection in different ways. The best practice for most PHP web applications is for the database to use Database Resident Connection Pooling (DRCP) connection pooling. This is discussed in the chapter *PHP Connection Pooling and High Availability*.

Standard Connections

For basic connection to Oracle use PHP's `oci_connect()` call:

```
$c = oci_connect($username, $password, $dbname);
```

You can call `oci_connect()` more than once in a script. If you do this and use the same connection details, then you get a pointer to the original connection.

Unique Connections

To get a totally independent connection use `oci_new_connect()`:

```
$c = oci_new_connect($username, $password, $dbname);
```

Each connection is separate from any other. This lets you have more than one database session open at the same time, which is useful when you want to do database operations independently from each other.

Persistent Connections

Persistent connections can be made with `oci_pconnect()`:

```
$c = oci_pconnect($username, $password, $dbname);
```

Persistent connections are not automatically closed at the end of a PHP script when the HTTP request finishes. They remain open in PHP's persistent connection cache for reuse by later scripts. This makes `oci_pconnect()` fast for frequently used web applications. Reconnection does not require re-authentication to the database.

Each cache entry uses the database username, the database connect string, the hashed password, the character set and the connection privilege to ensure reconnection in later (or the same) PHP scripts reuses the correct cached database connection.

Connecting to Oracle Using OCI8

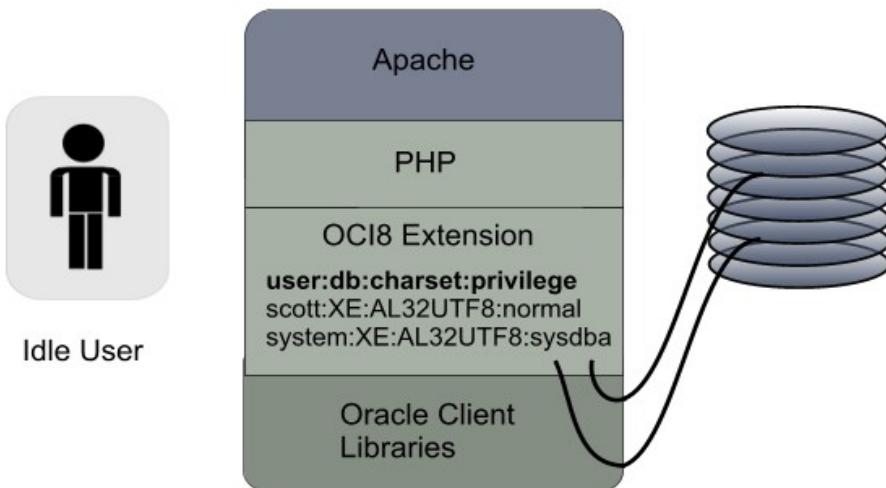


Figure 57: Persistent connections are cached in PHP and held open to the database when the web user is idle.

Limits on the number of persistent connections in the cache can be set, and connections can be automatically expired to free up resources. The parameters for tuning persistent connections are discussed later in this chapter.

When the PHP process terminates, the connection cache is destroyed and all database connections closed. This means that for command line PHP scripts, persistent connections are equivalent to normal connections and there is no performance benefit.

If the database is shutdown with the NORMAL option, the shutdown will hang until all database connections are closed. If persistent connections have been used in PHP, Apache will first need to be shutdown to close those connections.

Oracle Database Name Connection Identifiers

The `$dbname` connection identifier is the name of the local or remote database that you want to attach to. It is interpreted by Oracle Net, the component of Oracle that handles the underlying connection to the database and establishes a connection through to the network "listener" on the database server. The connection identifier can be one of:

- An Easy Connect string
- A Connect Descriptor string
- A Connect Name

Easy Connect String

If you are running Oracle Database XE on a machine called *mymachine*, you could connect to the HR schema with:

```
$c = oci_connect('hr', 'welcome', 'mymachine/XE');
```

In this guide, we assume the database is on the same machine as Apache and PHP so we use *localhost*:

Oracle Database Name Connection Identifiers

```
$c = oci_connect('hr', 'welcome', 'localhost/XE');
```

Depending on your network configuration, you may need to use the equivalent IP address:

```
$c = oci_connect('hr', 'welcome', '127.0.0.1/XE');
```

The Easy Connect string is JDBC-like. The Oracle 10g syntax is:

```
[//]host_name[:port][/service_name]
```

If the PHP binary is linked with Oracle 11g client libraries, the enhanced 11g syntax can be used:

```
[//]host_name[:port][/service_name][:server_type][/instance_name]
```

The prefix `//` is optional. The port number defaults to Oracle's standard port, 1521. The service name defaults to the same name as the database's host computer name. The server is the type of process that Oracle uses to handle the connection, see the chapter *PHP Connection Pooling and High Availability* covering Database Resident Connection Pooling for an example. The instance name is used when connecting to a specific machine in a clustered environment.

While it is common for Oracle database sites to use port 1521, it is relatively rare that a database will be installed with the service name set to the host name. You will mostly need to specify the connection identifier as `host_name/service_name`.

The `lsnrctl` command on the database server shows the service names that the Oracle Net listener accepts requests for. The example below shows the service XE is available.

```
$ lsnrctl services

LSNRCTL for Linux: Version 11.2.0.2.0 - Production on 05-Jun-2011 16:24:52
Copyright (c) 1991, 2011, Oracle. All rights reserved.

Connecting to (DESCRIPTION=(ADDRESS=(PROTOCOL=IPC)(KEY=EXTPROC_FOR_XE)))
Services Summary...
Service "PLSExtProc" has 1 instance(s).
  Instance "PLSExtProc", status UNKNOWN, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:0 refused:0
        LOCAL SERVER
Service "XE" has 1 instance(s).
  Instance "XE", status READY, has 1 handler(s) for this service...
    Handler(s):
      "DEDICATED" established:18 refused:0 state:ready
        LOCAL SERVER
. . .
```

More information on the syntax can be found in the *Oracle® Database Net Services Administrator's Guide 11g Release 2 (11.2)*.

Database Connect Descriptor String

The full Oracle Net *connect descriptor* string gives total flexibility over the connection.

```
$db = '(DESCRIPTION =
```

Connecting to Oracle Using OCI8

```
(ADDRESS = (PROTOCOL = TCP)
  (HOST = mymachine.mydomain)(PORT = 1521))
  (CONNECT_DATA =
    (SERVER = DEDICATED)
    (SERVICE_NAME = MYDB.MYDOMAIN))';

$c = oci_connect($username, $password, $db);
```

The syntax can be more complex than this example, depending on the Oracle Net features used. For example, you can enable features like load balancing and tweak packet sizes. The Easy Connect syntax does not allow this flexibility.

Database Connect Name

You can store the *connect descriptor* string in a commonly used Oracle file called *tnsnames.ora* and refer to it in PHP using a *connect name*:

```
# tnsnames.ora
MYD = (DESCRIPTION =
      (ADDRESS = (PROTOCOL = TCP)
        (HOST = mymachine.mydomain)(PORT = 1521))
      (CONNECT_DATA =
        (SERVER = DEDICATED)
        (SERVICE_NAME = MYDB.MYDOMAIN)))
```

In PHP you would use the *connect name* MYD to connect to the database:

```
$c = oci_connect($username, $password, 'MYD');
```

PHP needs to be able to find the *tnsnames.ora* file to resolve the MYD name. The directory paths that Oracle searches for *tnsnames.ora* depend on your operating system. On Linux, the search path includes:

```
$TNS_ADMIN/tnsnames.ora
/etc/tnsnames.ora
$ORACLE_HOME/network/admin/tnsnames.ora
```

If PHP was compiled using the Oracle libraries in an ORACLE_HOME-style install, then set ORACLE_HOME before starting the web server. The pre-supplied

\$ORACLE_HOME/network/admin/tnsnames.ora will then automatically be found. In Oracle Database XE, the value of \$ORACLE_HOME is:

```
/u01/app/oracle/product/11.2.0/xe
```

If PHP was built with Oracle Instant Client then put *tnsnames.ora* in */etc*, or set TNS_ADMIN to the directory containing it prior to starting the web server.

Make sure Apache has read permissions on *tnsnames.ora*. In some ORACLE_HOME-style installs, the default permissions on the file are restrictive.

Commonly Seen Connection and Environment Errors

Establishing a database connection is the common place for installation and configuration errors to manifest themselves.

Common errors when running PHP with the OCI8 extension are due to an incorrectly set environment. Check the environment variables are set and exported, see *Setting the Oracle Environment on Linux*, in the chapter *Installing and Configuring PHP*. On Windows the common problem is having multiple versions of Oracle libraries installed.

The bottom line is that your environment should be set correctly and consistently. The Apache process must have access to Oracle libraries and configuration files. Environment variables must be set in the shell that starts Apache, not in PHP scripts.

Check the Apache Error Log File

Start troubleshooting by checking the Apache *error_log* file for errors so you try to resolve the underlying cause, not some consequential symptom.

Check Runtime Errors

If *php.ini* is not configured to display errors in normal output then your web pages may simply show as blank page in your browser. Enable output with *display_errors* in *php.ini*. Alternatively, check the web server or PHP logs for the error messages.

Check *php.ini* is Loaded

Check that the correct *php.ini* file is being loaded. To find the *php.ini* location run a script *phpinfo.php*:

Script 8: phpinfo.php

```
<?php  
phpinfo();  
?>
```

Near the top of its output are lines like:

```
...  
Configuration File (php.ini) Path => /opt/php54/lib  
Loaded Configuration File => /opt/php54/lib/php.ini  
...
```

This immediately indicates the *php.ini* directory and, in this example, that it is being correctly read. If the directory did not contain a *php.ini* file, the loaded value would read “(none)”.

Older versions of PHP may only show one line:

```
Configuration File (php.ini) Path => /opt/php/lib/php.ini
```

If *php.ini* was not found, the old output would only show the directory name:

```
Configuration File (php.ini) Path => /opt/php/lib
```

Check the *phpinfo.php* output with both command line PHP and via a browser.

Connecting to Oracle Using OCI8

You can override the default location of *php.ini* by adding the desired file location to your *httpd.conf*:

```
PHPIniDir "/your/path/to/php.ini"
```

The path should include the filename.

Check OCI8 is enabled in php.ini

Check that OCI8 is enabled in *php.ini*, if required.

On Windows exactly one of the lines `extension=php_oci8.dll` or or `extension=php_oci8_11g.dll` must be in *php.ini*.

On Linux, if OCI8 was statically compiled into the PHP binary then no directive is needed but if OCI8 was built as a shared library the line should be `extension=oci8.so`.

If all is well in your *php.ini*, the output of `phpinfo()` will contain a section showing OCI8 parameters. See the section on *Checking OCI8 and PDO_OCI Installation* in the chapter *Installing and Configuring PHP*.

Set Oracle Environment Variables

Never use `PutEnv()` to set Oracle environment variable in PHP scripts. This is the number one cause of unexpected user problem. Variables should be set and exported in the environment that starts the web server.

If you think your environment variables are not set, refer to *Setting the Oracle Environment on Linux* in the chapter *Installing and Configuring PHP*.

If you are expecting shell environment variables to be passed to PHP when starting Apache, do not use the Linux service command to start Apache, since this clears the environment.

Check Apache Has Oracle File Access

The OCI8 extension always needs to find Oracle libraries, globalization data, error message data and optionally needs the Oracle Network *tnsnames.ora* and *sqlnet.ora* files. Not finding the libraries can lead to Apache startup errors about OCI8 not being initialized, such as “*OCIEnvNlsCreate() failed*”, or to script runtime errors like:

```
PHP Fatal error: Call to undefined function oci_connect()
```

This particular error means that the OCI8 extension is not loaded in PHP.

Check that `ORACLE_HOME` and/or `LD_LIBRARY_PATH` on Linux, or `PATH` on Windows are valid. If you built PHP with an `ORACLE_HOME`, then check the Oracle home directory is readable by the Apache process owner. If you are using an Oracle Database 10g Release 2 database other than the Express Edition, refer to the `$ORACLE_HOME/install/changePerm.sh` script in later Oracle patch sets.

Check that the *tnsnames.ora* file (if you use one for connection) is readable. Restart Apache after making changes to Oracle Network configuration files so that PHP re-reads the updated files.

A tool such as *strace* or *truss* can be used on PHP to see which files PHP needs access to.

Avoid Multiple Oracle Library Clashes

A potential source of problems on Windows is having multiple installations of Oracle libraries. Using mismatched versions of Oracle libraries and files can lead to PHP returning errors such as:

```
ORA-12705: Cannot access NLS data files or invalid environment specified
```

or:

```
OCIEnvNlsCreate() failed. There is something wrong with your system
```

Users of older versions of OCI8 may see the one of the equivalent errors:

```
OCIEnvCreate() failed. There is something wrong with your system
```

or

```
OCIEnvInit() failed. There is something wrong with your system
```

Instead of starting Apache as a service that inherits the mixed Oracle environment, write a script that sets PATH with the correct Oracle libraries and then starts Apache. With the IIS web server, use the IIS manager and set PATH in the *php-cgi.exe* entry in the FastCGI settings.

Some users move the Instant Client DLLs to the System, Apache, or PHP directory as a quick solution, but this is not recommended as a long term solution.

Using a dependency checker program can help determine which Oracle libraries are being loaded on Windows.

Use the Right PHP Binary

On Windows 64-bit if you use Oracle 64-bit libraries you might get an `OCIEnvNLSCreate()` error or you might see:

```
Unable to load dynamic library 'C:\Program Files (x86)\PHP\ext\php_oci8_11g.dll'  
- %1 is not a valid Win32 application.
```

There is only a 32-bit version of PHP on Windows so you need to make sure to use Oracle 32-bit client libraries.

Use the Right Connection String

Your `oci_connect()` calls need to know which database to connect to. If an error like this occurs:

```
Error while trying to retrieve text for error ORA-12154
```

it means two problems happened. First, a connection error ORA-12154 occurred. The second problem is the “Error while trying to retrieve text” message, indicating Oracle’s message files were not found, most likely because `ORACLE_HOME` is not correctly set.

The expected description for ORA-12154 is actually:

```
ORA-12154: TNS:could not resolve service name
```

Connecting to Oracle Using OCI8

This error indicates that the connection string is not valid, or the *tnsnames.ora* file (if one is being used) wasn't readable. The result is that OCI8 does not know which machine to connect to. A similar error:

```
ORA-12514 TNS:listener does not currently know of service requested in connect descriptor
```

means that OCI8 was able to contact a machine hosting Oracle, but the expected database is not running on that computer. For example, if Oracle Database 11g XE is currently running on your computer and you try to connect to *localhost/abc* then you will get this error.

Other Troubleshooting Tips

Here are some further things to check if you are having problems connecting or configuring OCI8:

- Turn on error reporting in *php.ini* (remember to turn it off for production deployment) or do it in each script with:

```
error_reporting(E_ALL); // In PHP 5.3 use E_ALL|E_STRICT  
ini_set('display_errors', 'On');
```

- You might get "*OCIEnvNlsCreate() failed*" if PHP links with an older Oracle version at runtime than the one it was compiled with. If you have multiple versions of Oracle installed on Linux then you might be able to use LD_PRELOAD or equivalent to force Apache to load the desired Oracle *libclntsh.so* file.
- If scripts sometime fail with *ORA-12516 TNS:listener could not find available handler with matching protocol stack* then use DRCP connection pooling or increase the Oracle Database processes parameter. Pooling is discussed in the chapter *PHP Connection Pooling and High Availability*. Changing the database parameter is discussed in the chapter *Testing PHP and the OCI8 Extension*.
- On Windows make sure to reboot after modifying the PATH environment variable.
- If you copied *tnsnames.ora* from Windows to Linux you will get an error if it still contains:

`SQLNET.AUTHENTICATION_SERVICES = (NTS)`
- If there is a client *sqlnet.ora* (which would be in the same directory as *tnsnames.ora*), check what the `names.default_domain` value is. This value will be automatically appended to the TNS string in PHP before being looked up in *tnsnames.ora*. For example, if *sqlnet.ora* had `names.default_domain = oracle.com` and the connection call was `oci_connect($u, $p, 'XE')`, then *tnsnames.ora* would need to have an entry '`'XE.oracle.com = (DESCRIPTION ...)'`.
- Only one version of the PHP module can be loaded in Apache at a time. It is possible to get the error "*OCIEnvNlsCreate() failed*" if *httpd.conf* has two LoadModule lines like:

```
LoadModule php5_module modules/libphp5.so  
LoadModule php6_module modules/libphp6.so
```

Closing Oracle Connections

At the end of each script, connections opened with `oci_connect()` or `oci_new_connect()` are automatically closed. You can also explicitly close these connections by calling:

```
oci_close($c);
```

Any uncommitted data is rolled back. The function has no effect on persistent connections. (See the section on connection pooling in the *PHP Connection Pooling and High Availability* chapter for caveats).

If a long running script only spends a small amount of time interacting with the database, close connections as soon as possible to free database resources for other users. When the Apache or PHP command line process terminates, all database connections are closed.

The `oci_close()` function was a “no-op” prior to the re-factoring of OCI8 in PHP 5.1. That is, it had no functional code, and never actually closed a connection. You could not explicitly close connections even if you wanted to! You can revert to this old behavior with a *php.ini* setting:

```
oci8.old_oci_close_semantics = On
```

Closing Connections and Variable Scope

PHP resources such as connection resources work by reference counting. Only when all PHP references to the database connection are finished will it actually be closed and database resources freed. This example shows the effect of reference counting:

Script 9: close.php

```
<?php

$c = oci_connect("hr", "welcome", "localhost/XE");
$s = oci_parse($c, "select * from locations");
oci_execute($s);
oci_fetch_all($s, $res);

// oci_free_statement($s); // Uncomment this for the oci_close() to work
oci_close($c);

echo "Sleeping . . .";
sleep(10);
echo "Done";

?>
```

While *close.php* is sleeping, if you query the database as a privileged user:

```
SQL> select username from v$session where username is not null;
```

you will see that HR is still shown as connected until the `sleep()` finishes and the script terminates. This is because the `oci_parse()` call creating the statement resource `$s` internally increases the reference count on `$c`. The database connection is not closed until PHP's end-of-script processing destroys `$s`.

Connecting to Oracle Using OCI8

An `oci_free_statement($s)` call will explicitly decrease the reference count on `$c` allowing the `oci_close()` to have an immediate effect. If this freeing call is uncommented in the example, the SQL*Plus query will show the database connection was explicitly closed before the `sleep()` starts.

Another commonly seen idiom to close resources is to assign null to them:

```
$s = null;
```

Variables and other kinds of resources may also increase the reference count on a connection, and in turn have their own reference count which must be zero before they can be destroyed. The reference count will decrease if the variables goes out of scope or are assigned new values.

In the next example script, `close2.php`, the variables `$c1` and `$c2` are the same database connection because `oci_connect()` returns the same connection resource when called more than once in a script with the same credentials. If you query `v$session` while the script is running you will only see one open connection. The physical database connection is released only when `$c1` and `c2` are both closed. Also the statement resource must be freed, which happens automatically when `do_query()` completes and `$s` goes out of scope.

Script 10: close2.php

```
<?php

function do_query($c, $query)
{
    $s = oci_parse($c, $query);
    oci_execute($s);
    oci_fetch_all($s, $res);
    echo "<pre>";
    var_dump($res);
    echo "</pre>";
}

$c1 = oci_connect('hr', 'welcome', 'localhost/XE');
$c2 = oci_connect('hr', 'welcome', 'localhost/XE'); // Reuses $c1 DB connection

do_query($c1, 'select user from dual'); // Query 1 works
oci_close($c1); // DB connection doesn't get closed
do_query($c1, 'select user from dual'); // Query 2 fails
do_query($c2, 'select user from dual'); // Query 3 works
oci_close($c2); // DB connection is now closed

?>
```

Variable `$c1` is not usable after `oci_close($c1)` is executed: PHP has dissociated it from the connection. But the database connection remains open because `$c2` still references it. The script outcome is that the first and third queries succeed but the second one fails.

Transactions and Connections

Uncommitted data is rolled back when a connection is closed or at the end of a script. For `oci_pconnect()` this means subsequent scripts reusing a cached database connection will not see any data that should not be shared.

Avoid letting database transactions remain open if a second `oci_connect()` or `oci_pconnect()` call with the same user credentials is executed within a script, or if `oci_close()` is used. Making sure data is committed or rolled back first can prevent hard to debug edge cases where data is not being stored as expected.

The next chapter covers database transactions in more detail.

Session State With Persistent Connections

It is possible for a script to change session attributes for `oci_pconnect()` that are not reset at the end of the script. (The Oracle term *session* is effectively the same as the PHP term *connection*). One example is the globalization setting for the date format:

```
$c = oci_pconnect("hr", "welcome", "localhost/XE");
do_query($c, "select sysdate from dual");
$s = oci_parse($c, "alter session set nls_date_format='YYYY-MM-DD HH24:MI:SS'");
$r = oci_execute($s);
do_query($c, "select sysdate from dual");
```

The first time this is called in a browser, the two dates returned by the queries are:

```
18-APR-07
2007-04-18 14:21:09
```

The first date has the Oracle default format. The second is different because the ALTER SESSION command changed the format.

Calling the script a second time gives:

```
2007-04-18 14:21:10
2007-04-18 14:21:10
```

The persistent connection has retained the session setting and the first query no longer uses the system default format. This only happens if the same Apache process serves both HTTP requests. If a new Apache process serves the second request then it will open a new connection to the database, which will have the original default date format. (Also the system will have two persistent connections left open instead of one.)

Session changes like this may not be a concern. Your applications may never need to do anything like it, or all connections may need the same values anyway. If there is a possibility incorrect settings will be inherited, make sure your application resets values after connecting.

Optional Connection Parameters

The `oci_connect()`, `oci_new_connect()` and `oci_pconnect()` functions take an optional extra two parameters:

- Connection character set
- Connection session mode

Connection Character Set

The character set is a string containing an Oracle character set name, for example, [JA16UEC](#) or [AL32UTF8](#):

Connecting to Oracle Using OCI8

```
$c = oci_connect("hr", "welcome", "localhost/XE", "AL32UTF8");
```

When not specified or `NULL` is passed, the `NLS_LANG` environment variable setting is used.

The character set determines how Oracle translates data when it is transferred from the database to PHP. If the database character set is not equivalent to the OCI8 character set, some data may get converted abnormally. It is recommended to set this parameter to improve performance and guarantee a known value is used.

It is up to your application to handle returned data correctly, perhaps by using PHP's `mb_string`, `iconv` or `intl` extensions. Globalization is discussed in more detail in the *Globalization* chapter.

Connection Session Mode

The session mode parameter allows privileged or externally authenticated connections to be made. For example:

```
$c = oci_connect("hr", "welcome", "localhost/XE", "AL32UTF8", OCI_SYSDBA);
```

Connection Privilege Level

The OCI8 extension allows privileged SYSDBA and SYSOPER connections. Privileged connections are disabled by default. They can be enabled in `php.ini` using:

```
oci8.privileged_connect = 1
```

The SYSDBA and SYSOPER privileges give you the ability to change the state of the database, perform data recovery, and even access the database when it has not fully started. Be very careful about exposing this on customer facing web sites, that is, do not do it! It might be useful for command line PHP scripts in very special circumstances.

When you installed Oracle, the SYS administrative user account was automatically created with the password that you supplied. All base tables and views for the database data dictionary are stored in the SYS schema – they are critical for the operation of Oracle. By default, the SYSDBA privilege is assigned only to user SYS, but it and SYSOPER can manually be granted to other users.

Operating System Authenticated Privileged Connections

You can have the operating system perform the authentication for privileged connections based around the operating system user that is running the web server system process. An operating system authenticated privileged connection in PHP is equivalent to the SQL*Plus connection:

```
$ sqlplus / as sysdba
```

For `/ as sysdba` access (where no username and password is used) in PHP, all these must be true:

- The operating system process user is a member of the OS `dba` group
- PHP is linked with the same ORACLE_HOME software that the database is using (not Oracle Instant Client)

Optional Connection Parameters

- The database is your default local database, for example, specified by the ORACLE_SID environment variable
- `oci8.privileged_connect = 1` in `php.ini`.

Scripts that contain operating system authenticated privileged connection calls will connect successfully:

```
$c = oci_connect("/", "", null, null, OCI_SYSDBA);
```

If PHP is invoked by Apache, the library path needs to contain the same Oracle libraries as used by the database. Also the *nobody* user must be in the privileged Oracle group, for example, in the operating system *dba* group. This is not recommended.

Similarly, AS SYSOPER access is available for members of the *oper* group. In PHP use OCI_SYSOPER in `oci_connect()`.

On Windows, the operating system groups are called *ORA_DBA* and *ORA_OPER*.

Remote Privileged Access

When OCI8 uses Oracle Instant Client, a username and password must be given when connecting to a database. These connections are considered “remote” from the database because the libraries used by PHP are not those used by the running database.

Remote users can make privileged connections only when they have been given the appropriate Oracle access. In SQL*Plus a privileged session would be started like:

```
$ sqlplus username/password@sid as sysdba
```

The database will not permit the (possibly physically) “remote” operating system to authorize access. An extra Oracle password file needs to be created and a password needs to be used in the database connection.

To set up a password file, check the database initialization parameter `remote_login_passwordfile` is EXCLUSIVE. This is the default value. To do this, log in to the operating system shell as the Oracle database software owner, and start SQL*Plus:

```
$ sqlplus / as sysdba
SQL> show parameter remote_login_passwordfile
NAME                      TYPE        VALUE
-----                    -----
remote_login_passwordfile    string      EXCLUSIVE
```

A setting of EXCLUSIVE means the password file is only used with one database and not shared among several databases on the host. It enables you to have multiple users connect to the database as themselves, and not just as SYS. If this parameter is not set to EXCLUSIVE, you can change the value in SQL*Plus by entering a command similar to:

```
SQL> alter system set remote_login_passwordfile='exclusive'
  2   scope=spfile sid='*';
```

From the operating system shell, create an Oracle password file:

```
$ $ORACLE_HOME/bin/orapwd file=$ORACLE_HOME/dbs/acct.pwd \
> password=secret entries=10
```

Connecting to Oracle Using OCI8

This creates a password file named `acct.pwd` that allows up to 10 privileged users with different passwords (this number can be changed later). The file is initially created with the password `secret` for users connecting with the username `SYS`.

To add a new user to the password file use SQL*Plus:

```
SQL> create user c1 identified by c1pw;
SQL> grant connect to c1;
SQL> grant sysdba to c1;
SQL> select * from v$pwfile_users;
USERNAME          SYSDBA  SYSOPER
-----
SYS                TRUE    TRUE
C1                 TRUE    FALSE
```

Now in PHP you can use the following connection command:

```
$c = oci_connect("c1", "c1pw", 'localhost/XE', null, OCI_SYSDBA);
```

One feature of a privileged connection is that if you issue a `SELECT USER FROM DUAL` statement, any `OCI_SYSDBA` connection will show the user as `SYS` not `C1`. A connection made with `OCI_SYSOPER` will show a user of `PUBLIC`.

Password Handling in PHP Applications

The examples in this book hard code the database password in each PHP script. Real applications need to be more careful with password management.

PHP operates in a stateless way but a typical application needs database access over a number of sequential HTTP requests. Once the user has logged into the web application you don't want to store their plaintext password in a cookie and reuse it for database connection on their next web request because storing it this way is a security risk. So web applications typically connect to the database using one pre-determined database schema and do their own application-specific authorization.

The first problem is how to protect that schema. A typical PHP approach to avoid hard coding the database password in scripts is to set it in an environment variable. The variable is passed through to PHP with Apache's `PassEnv`. Whenever Apache is started or the machine rebooted, the variable must be manually set. The value is then available to PHP scripts in the `$_ENV` global:

```
$c = oci_connect('hr', $_ENV['HRPASSWORD'], 'localhost/XE');
```

Note that on Linux the service `httpd start` command only passes the `LANG` and `TERM` variables, so Apache must be started with `/usr/sbin/apachectl` for this to work on Oracle Linux.

External Authentication With PHP OCI8

Instead of storing the database username and password in PHP scripts or environment variables, database access can be authenticated by an outside system. Once this authentication system is configured, PHP scripts connect like:

```
$c = oci_connect("/", "", "mynetalias", null, OCI_CRED_EXT);
```

where `mynetalias` is a connect name configured in a `tnsnames.ora` file. No username or password is stored in the PHP code.

One such external storage mechanism is Oracle Wallet which creates a secret wallet store. Only the operating system user running the Apache process needs to be granted read access to the wallet, which can be done using Access Control Lists (ACL). This further helps protect the database password.

The following example is an overview of using Oracle Wallet Manager. Because the details vary from version to version and the options may be specific to your application, verify the steps before trusting this. Note this is not supported in Oracle XE, and also that the external authentication feature of OCI8 is not available on Windows.

1. First create a wallet directory as the Oracle user:

```
$ mkstore -wrl /home/oracle/wallet_dir -create
```

This will prompt for a new password for the store.

2. Create the wallet for the username and password that are currently hardcoded in your PHP scripts:

```
$ mkstore -wrl "/home/oracle/wallet_dir" -createCredential my112 hr welcome
```

This will prompt for the wallet password previously set. The alias key *my112* immediately following the *-createCredential* option will be the connect name to be used in PHP scripts. If your application connects with multiple different database users, you could create a wallet entry with different connect names for each.

You can see the newly created credential with:

```
$ mkstore -wrl "/home/oracle/wallet_dir" -listCredential
```

3. Create a *tnsnames.ora* file, for example in */opt/oracle/tnsnames.ora*. This will be used by PHP applications:

```
my112 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 1521))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = orcl)
    )
  )
```

The file uses the description for your existing database and sets the connect name alias to *my112*, which is the new identifier used in the wallet.

4. In the same directory as the *tnsnames.ora* file, create a *sqlnet.ora* file containing the wallet location:

```
WALLET_LOCATION =
  (SOURCE =
    (METHOD = FILE)
    (METHOD_DATA =
      (DIRECTORY = /home/oracle/wallet_dir)
    )
  )
SQLNET.WALLET_OVERRIDE = TRUE
```

Connecting to Oracle Using OCI8

```
SSL_CLIENT_AUTHENTICATION = FALSE  
SSL_VERSION = 0
```

5. Set TNS_ADMIN to the directory containing these two files, and export the variable. On Oracle Linux it would be exported in /etc/sysconfig/httpd for PHP, for example:

```
export TNS_ADMIN=/opt/oracle
```

6. Apache needs access to the wallet:

```
# setfacl -m u:apache:rx /home/oracle/wallet_dir  
# setfacl -m u:apache:r /home/oracle/wallet_dir/{cwallet.sso,ewallet.p12}
```

7. Restart Apache:

```
# service httpd restart
```

8. Create an OCI8 script of your choice, passing the flag OCI_CRED_EXT as the session_mode parameter to [oci_connect\(\)](#), [oci_new_connect\(\)](#) or [oci_pconnect\(\)](#):

```
$c = oci_connect("/", "", "my112", null, OCI_CRED_EXT);
```

The OCI_CRED_EXT mode can only be used with a username of "/" and an empty password. The *php.ini* parameter *oci8.privileged_connection* may be *On* or *Off*. The OCI_CRED_EXT mode may be combined with the OCI_SYSOPER or OCI_SYSDBA modes, for example:

```
$c = oci_connect("/", "", $db, null, OCI_CRED_EXT+OCI_SYSOPER);
```

The *php.ini* directive *oci8.privileged_connection* does need to be *On* for OCI_SYSDBA and OCI_SYSOPER use.

9. Load your script in a browser.

Tuning Connections to Build Scalable Systems

Oracle achieved its well-known scalability in part through a multi-threaded architecture. PHP instead has a multi-process architecture. This design difference means care is required when designing scalable applications because, instead of connections being shared via a mid-tier connection pool, large numbers of PHP processes each have one or more database connections. The number of connections open is a factor for database server memory requirements. Creating new connections is also relatively slow in Oracle. Connecting pooling in OCI8 and Oracle Database helps resolve these issues.

A general connection tip is to make applications as efficient as possible. This minimizes the length of time connections are held. The following sub-sections give some detailed tips.

Use the Best Connection Function

Using [oci_pconnect\(\)](#) makes a big improvement in overall connection speed of frequently used applications because it uses the connection cache in PHP. A new, physical connection to the database does not have to be created if one already exists in PHP's cache. Using persistent connections is common for web sites that have high numbers of connections being

established. Reusing a previously opened connection is significantly faster than opening a fresh one.

Overall database load may be reduced if idle connections are closed with Apache process timeouts, although this needs to be balanced against the expense of creating new connections at peak user login time.

If unused-but-still-open persistent connections consume too much memory on the database server, consider using connection pooling.

Use Connection Pooling

The general best practice suggestion is to use persistent OCI8 connections with Oracle Database Resident Connection Pooling. DRCP dramatically reduces the database server memory used for each connection, allowing more memory to be allocated to shared structures in the database itself. DRCP can be useful even for small sites because it allows database servers to handle relatively large numbers of connections. If DRCP is not available then Oracle Shared Servers also known as "Multi Threaded Servers" (MTS) might give some benefits. See the chapter *PHP Connection Pooling and High Availability* for information on using DRCP with OCI8.

Minimize the number of database user credentials used

Each persistent connection that uses a different set of credentials will create a separate process on the database host. If the application connects with a large number of different schemas, then the number of persistent connections can be reduced by connecting as one user who has been granted permission to the original schemas' objects.

The application can either be recoded to use explicit schema names in queries:

```
SQL> select * from olduser.mytable;
```

Or, if the application is too extensive to modify, the first statement executed can set the default schema:

```
SQL> alter session set current_schema = olduser;
```

Setting the default schema this way requires an extra database operation per connection but, depending on the application, it may be bundled in a PL/SQL block or trigger that does other operations.

Connect With a Character Set

Explicitly passing the client character set name as the fourth parameter to the connection functions reduces the time to connect:

```
$c = oci_connect("hr", "welcome", "localhost/XE", "WE8DEC");
```

If you do not enter a character set, PHP has to determine one to use. This may involve a potentially expensive environment lookup. Use the appropriate character set for your globalization requirements.

Connecting to Oracle Using OCI8

Tune the AUDSES\$ Sequence Generator

For sites with hundreds of connections per second, tune the cache size of the internal sequence generator, SYS.AUDSES\$. A starting point is to change it to perhaps 10000:

```
SQL> alter sequence sys.audses$ cache 10000;
```

This is also recommended if you are using Oracle RAC ("Real Application Clusters").

Do Not Set the Date or Numeric Format Unnecessarily

Avoid executing ALTER SESSION statements after each connection. Applications commonly set NLS_DATE_FORMAT and NLS_NUMERIC_CHARACTERS to guarantee data values are returned from Oracle in a known format. Consider an existing connection routine that always sets the date format:

```
function my_connect($un, $pw, $db)
{
    $c = oci_pconnect($un, $pw, $db);
    $s = oci_parse($c, "alter session set nls_date_format='YYYY-MM-DD'");
    oci_execute($s);
    return $c;
}
```

One way to optimize this is simply to set the environment variable NLS_DATE_FORMAT in the shell that starts the web server. Each PHP connection will have the required date format automatically. Note, when setting Oracle NLS_* globalization environment variables you also need to set NLS_LANG otherwise they will be ignored.

Sometimes different database users should have different session values so setting NLS_DATE_FORMAT globally is not possible. With persistent connections the ALTER SESSION can be moved to a logon trigger. This is because session settings are retained in cached connections ready for the next script that makes an `oci_pconnect()` call. Using a trigger means the date format is only set when the physical database connection is created the very first time `oci_pconnect()` is called in the lifetime of the Apache/PHP process. The trigger does not fire when subsequent `oci_pconnect()` calls return a cached connection. It is also a database procedure and does not require any interaction between the PHP script and the database. This reduces load on the whole system. The same solution will help when a standard connection `oci_connect()` is called multiple times in the one script

A logon trigger can be created using SQL*Plus by connecting as a privileged database user:

```
$ sqlplus system@localhost/XE
```

Then run `logontrig.sql`:

Script 11: logontrig.sql

```
create or replace trigger my_set_date after logon on database
begin
    if (user = 'HR') then
        execute immediate 'alter session set nls_date_format = ''YYYY-MM-DD''';
    end if;
end my_set_date;
/
```

This trigger sets the session's date format every time HR connects to the database from any client tool. Note the use of single quotes. The date format string is enclosed in a pair of two quotes, which is the Oracle method of nesting single quotes inside a quoted string.

With the trigger, Oracle does all the work setting the date format when the physical database connection is originally established and first used. When PHP later uses a cached connection it will already have the desired date format.

In PHP, the connection function from the start of this section can be simplified and performance improved because it no longer needs to set the date format.

Script 12: *logontrig.php*

```
<?php
function my_connect($un, $pw, $db)
{
    return(oci_pconnect($un, $pw, $db));
}

$c = my_connect('hr', 'welcome', 'localhost/XE');
$s = oci_parse($c, 'select sysdate from dual');
oci_execute($s);
$row = oci_fetch_array($s, OCI_ASSOC);
echo $row['SYSDATE'] . "<br>\n";
?>
```

The script *logontrig.php* connects as HR using the now trivialized *my_connect()* and queries the current date. It shows the new format set by the trigger:

2007-05-04

If the connection is any user other than HR the standard default date format will be displayed, for example:

04-MAY-07

Using a trigger like this only works when the required session setting is the same for all PHP application users that share the same database user name.

The suggested practice is to use LOGON triggers only for setting session attributes and not for executing per PHP-connection logic such as custom logon auditing.

If you cannot use a trigger because each PHP invocation needs different settings, and you need more than one SQL statement executed, you can put the statements inside a PL/SQL procedure. After connecting you can call the PL/SQL procedure, which is one *oci_execute()* call to the database, instead of multiple calls to execute other SQL statements.

Manage Persistent Connections

Persistent connections are great if the cost of opening a connection is high. What you consider high depends on your application requirements and on implementation issues such as whether the web server and database are on the same host, which will affect the time taken to establish a connection, and on memory availability. The drawback is persistent connections use Oracle resources even when no one is accessing the application or database. And if Apache spawns a number of server processes, each of them may have its own set of connections to the database. The proliferation of connections can be controlled to some

Connecting to Oracle Using OCI8

extent with *php.ini* directives and Apache configuration settings. If connection pooling is used, the number of connections can be kept small.

Maximum Number of Persistent Connections Allowed

`oci8.max_persistent`

This parameter limits the number of persistent connections cached by each individual Apache-PHP process. It is not a system-wide restriction on database usage. When the limit is reached by a PHP process, then all new `oci_pconnect()` calls are treated like `oci_connect()` calls and are closed at the end of the script. Setting it to -1 (the default) means there is no limit. If your PHP scripts connect using the same database credentials, each PHP process will only have one connection entry in its cache so this setting will have no effect on your application's resource usage.

Timeout for Unused Persistent Connections

`oci8.persistent_timeout`

This parameter is the length in seconds that an Apache process maintains an idle persistent connection. Setting this parameter to -1 (the default) means there is no timeout. If a connection has been expired, the next time `oci_pconnect()` is called a new connection is created.

It is not an asynchronous timer. The expiry check happens whenever any PHP script finishes, regardless of whether OCI8 calls were made. Only the persistent connections in the invoked Apache process will be checked. This is an unresolvable weakness with PHP: you want idle connections to be closed, but if PHP is idle then no scripts execute and the timeout is not triggered. Luckily Oracle 11g DRCP makes this issue irrelevant.

Pinging for Closed Persistent Connections

`oci8.ping_interval`

There is no guarantee that the connection descriptor returned by `oci_pconnect()` represents a usable connection to the database. During the time PHP stored an unaccessed connection resource in its cache, the connection to the database may have become unusable due to a network error, a database error, or being expired by the DBA. If this happens, `oci_pconnect()` appears to be successful but an error is thrown when the connection is later used, for example in `oci_execute()`. The ping interval is an easy way to improve connection reliability for persistent connections.

This parameter is the number of seconds that pass before OCI8 does a ping during a `oci_pconnect()` call. If the ping determines the connection is no longer usable, a new connection is transparently created and returned by `oci_pconnect()`. To disable pinging, set the value to -1. When set to 0, PHP checks the database each time `oci_pconnect()` is called. The default value is 60 seconds.

Regardless of the value of `oci8.ping_interval`, an `oci_pconnect()` call will always check an internal Oracle client-side value to see if the server was known to be available the last time anything was received from the database. This is a quick operation. Setting `oci8.ping_interval`

physically sends a message to the server, causing a “round-trip” over the network. This is a “bad thing” for scalability.

Good application design gracefully recovers from failures. In any application there are a number of potential points of failure including the network, the hardware and user actions such as shutting down the database. Oracle itself may be configured to close idle connections and release their database resources. The database administrator may have installed user profiles with CREATE PROFILE IDLE_TIMEOUT, or the Oracle network layer may time out the network.

You need to balance performance (no pings) with having to handle disconnected Oracle sessions (or other changes in the Oracle environment) in your PHP code. For highest reliability and scalability it is generally recommended that you do not use `oci8.ping_interval`, but do error recovery in your application code.

Apache Configuration Parameters

You can tune Apache to kill idle processes, which will free up Oracle resources used by persistent connections. Table 5 lists some Apache pre-fork model configuration parameters that can be used to tune PHP.

Table 5: Apache pre-fork configuration parameters.

Parameter	Purpose
MaxRequestsPerChild	Sets how many requests Apache will serve before restarting.
MaxSpareServers	Sets how many servers to keep in memory that are not handling requests.
KeepAlive	Defines whether Apache can serve a number of documents to the one user over the same HTTP connection.

Setting `MaxRequestsPerChild` too low will cause persistent connections to be closed more often than perhaps necessary, removing any potential performance gain of caching. Many sites use `MaxRequestsPerChild` to restart PHP occasionally, avoiding any potential memory leaks or other unwanted behaviors.

Changing the Database Password

The OCI8 extension allows Oracle database passwords to be changed.

Be cautious about changing passwords when PHP persistent connections are used. This is because a persistent connection is cached in a PHP process using the original password as part of the look-up key. This key is not updated when the password is changed. PHP applications would therefore allow an open persistent DB connection to be reused only when `oci_pconnect()` is given the old password. This is a security issue since knowing the old password is sufficient to get access to the database. It can also cause extra database load because users connecting with the new password will create a new persistent connection to the database, leaving the original connection idle.

Changing Passwords on Demand

After connecting, a password can be changed with `oci_password_change()`:

Connecting to Oracle Using OCI8

```
$c = oci_connect('hr', 'welcome', 'localhost/XE');
oci_password_change($c, 'hr', 'welcome', 'new_password');
```

Subsequent scripts may now connect using:

```
$c = oci_connect('hr', 'new_password', 'localhost/XE');
```

Changing Expired Passwords

Sometimes connection may fail because the password is no longer valid. For example, the DBA may have set a password policy using CREATE PROFILE to expire passwords at a certain time, or they may have expired a password immediately with ALTER USER, forcing the user to choose a new password the next time they connect. When the user tries to connect, their password is recognized but they get an *ORA-28001: the password has expired* message and will not be able to complete their log on. In this case, instead of the user having to bother the DBA to manually reset the expired password, `oci_password_change()` can be used to re-connect and change the password one operation. The next example shows this in action.

Script 13: connectexpired.sql

```
drop user peregrine cascade;
create user peregrine identified by abc;
grant create session to peregrine;
alter user peregrine password expire;
```

Script 14: connectexpired.php

```
<?php

$un = "peregrine";           // New temporary user to be created.
$pw = "abc";                 // Initial password for $un
$db = "localhost/XE";        // Database to connect to

function do_connect($un, $pw, $db)
{
    echo "Calling oci_connect()<br>\n";
    $c = oci_connect($un, $pw, $db);
    if ($c) {
        echo "Connected successfully<br>\n";
    }
    else {
        $e = oci_error();
        if ($e['code'] == 28001) {
            // Connect and change the password to a new one formed by
            // appending 'x' to the original password.
            // In an application you could prompt the user to choose
            // the new password.
            echo "Connection failed: the password for $un has expired<br>\n";
            $c = change_and_connect($un, $pw, $pw."x", $db);
        }
        else {
            echo "Error: ", $e["message"], "<br>\n";
        }
    }
}
```

Changing the Database Password

```
        exit;
    }
}
return($c);
}

function change_and_connect($un, $oldpw, $newpw, $db)
{
    echo "Calling oci_password_change() to connect<br>\n";
    // Note $db is a connection identifier string, not a PHP connection resource
    $c = oci_password_change($db, $un, $oldpw, $newpw);
    if (!$c) {
        $e = oci_error();
        echo "Error: ", $e["message"], "<br>\n";
    }
    else {
        echo "Connected and changed password to $newpw<br>\n";
    }
    return($c);
}

function show_user($c)
{
    $s = oci_parse($c, "select user from dual");
    oci_execute($s);
    oci_fetch_all($s, $res);
    echo "You are connected as {$res['USER'][0]}<br>\n";
}

// Connect as $un and confirm connection succeeded
$c = do_connect($un, $pw, $db);
show_user($c);

?>
```

Before running the PHP script, first run *connectexpired.sql* as a privileged user:

```
$ sqlplus system@localhost/XE @connectexpired.sql
```

In the PHP script, when `oci_connect()` in `do_connect()` fails with an *ORA-28001: the password has expired* error, `change_and_connect()` is called to change the password and connect in a single step. In this example, the new password is simply formed by concatenating an “x” to the current password. In an application, the user would be prompted for the new password.

The output of *connectexpired.php* is:

```
Calling oci_connect()
Connection failed: the password for peregrine has expired
Calling oci_password_change() to connect
Connected and changed password to abcx
You are connected as PEREGRINE
```

The password change call `oci_password_change($db, $un, $oldpw, $newpw)` differs from the example in the previous section *Changing Passwords On Demand* in that it passes a database connection identifier identifier string, '`localhost/XE`', as the first parameter instead of

Connecting to Oracle Using OCI8

passing the connection resource of an already opened connection. This new usage connects to the database and changes the password to `$newpw` all at the same time. Subsequent scripts will be able to connect using the new password.

This method of connecting with `oci_password_change()` also works if the password has *not* expired.

Authorization and Authentication With Client Identifiers

Applications that connect to the database via the one set of Oracle credentials should use `oci_set_client_identifier()` so the database can distinguish between individual web application users.

The "client identifier" is a small string identifier token you set for each connection and which is passed into the database. For example, if your web application physically connects to the database as the database user `phpuser`, and if two different people 'Chris' and 'Alison' are using the site, these two user names could be set as their respective client identifiers.

By associating a unique client identifier with each web user, Oracle Database can:

- Provide an audit trail on individual web users, for example on 'Chris' and 'Alison'
- Automatically apply rules to individual web users to restrict their data access
- Monitor and trace applications per web user

If you don't set client identifiers, all database activity is only recorded as coming from `phpuser`.

Setting Client Identifiers

Each PHP file in a typical Oracle PHP application calls `oci_pconnect()` with an identical database user name. Once the application's own authentication system decides a particular web user is OK, then a unique token is passed back and forth in HTTP responses and requests so that the web user doesn't have to re-authenticate each time a new web page is loaded..

Client identifiers should be set with `oci_set_client_identifier()` after connecting but before executing any statements or OCI8 calls on behalf of the web user. At its most basic, the client identifier could be the web user's name which was validated and stored in PHP's session data by a previous authentication script request:

```
session_start();
$c = oci_pconnect('phpuser', 'welcome', 'localhost/orcl');
oci_set_client_identifier($c, $_SESSION['app_user_name']);
. . .
```

If the identity of the end user alters during the run time of the script (perhaps if PHP is executing a long running command-line process, or perhaps in an administrative web page that runs different components representing different end users) then `oci_set_client_identifier()` can be called at each point the end-user identity changes:

```
$c = oci_pconnect('phpuser', 'welcome', 'localhost/orcl');

$myuser = 'Chris';
oci_set_client_identifier($c, $myuser);
. . .
$myuser = 'Alison';
oci_set_client_identifier($c, $myuser);
```

In practice, consider using more secure values for identifiers.

Client identifiers can be set when using `oci_connect()`, `oci_new_connect()`, or `oci_pconnect()` connection calls. Identifiers can be used when the database is configured to use any of the three types of server processes: "Dedicated" servers, "Shared" servers, and when using Database Resident Connection Pooling (DRCP) servers.

The `oci_set_client_identifier()` function was introduced in PHP OCI8 1.4 (first included in PHP 5.3.1). With older versions of OCI8 you can use the PL/SQL DBMS_SESSION package instead:

```
session_start();
$c = oci_pconnect('phpuser', 'welcome', 'localhost/orcl');
$s = oci_parse($c, "begin dbms_session.set_identifier(:id); end;");
oci_bind_by_name($s, ":id", $_SESSION['app_user_name']);
oci_execute($s);
```

The `oci_set_client_identifier()` function is preferred because unlike DBMS_SESSION.SET_IDENTIFIER it doesn't force a database round-trip request-and-response. With the PHP function, the identifier is piggy-backed on any subsequent OCI8 call that actually does reach the database from PHP. This avoids unnecessary round-trips which slow down each PHP page and impact application scalability.

PHP OCI8 does not clear the client identifier at the end of an HTTP request since the overhead of a round-trip to clear the value would impact scalability of every application. This is not detrimental for standard `oci_connect()` connections since the database connection is destroyed at the end of the HTTP request and the identifier value is cleared as a result. However identifiers may remain in effect across web requests that use `oci_pconnect()` persistent connections. To avoid an incorrect or no identifier being recorded by the database, all PHP files that connect to the database should set the identifier so it is correct for the duration of the request's execution. If many connections are idle causing monitoring to be affected by showing apparently still connected web users, then every script that sets the client identifier should forcefully clear it at the script end with:

```
$s = oci_parse($c, "begin dbms_session.clear_identifier; end;");
oci_execute($s);
```

This causes a round-trip to the database, which will impact scalability.

PHP's `oci_set_client_identifier()` corresponds to setting Oracle's C level OCI_ATTR_CLIENT_IDENTIFIER attribute. Oracle literature on this, and on PL/SQL's equivalent DBMS_SESSION.SET_IDENTIFIER, provides good references about client identifiers.

A Sample Application Using Client Identifiers

A sample PHP "Parts" application illustrates how client identifiers can be used in the OCI8 extension. Overall, the application shows an inventory of electrical and plumbing parts. An application-level authentication system handles web user logins. For successful logins, an identifier that is unique for each web user is passed between HTTP requests in PHP's session data. It is used for the client identifier value. The application has just enough complexity so the Oracle technologies being discussed are not abstract, but it is *no where near* a production example. The sample application is simply intended to show the relationship between the web user and the database user, and to show how a client identifier can be used in the database. PHP session management requires careful design to minimize security issues. There are many

Connecting to Oracle Using OCI8

external references discussing this problem which should be closely studied by every PHP developer. Michael McLaughlin's OTN article *Database-Based Authentication for PHP Apps* is a good place to begin reading more.

The core of the Parts application is a *setup.sql* file that creates the database objects. All the PHP scripts in the application will connect to the database using the PHPUSER schema, which owns the PARTS application table. The SQL script creates a second user PHP_SEC_ADMIN to hold security information about the application. This user is given some extra database privileges needed for the auditing example, shown later. The PHP_AUTHENTICATION table contains the application user names and passwords. Query access on this table is granted to the PHPUSER user so the PHP application only has to open one connection to the database, but that connection cannot modify the security information.

Script 15: setup.sql

```
set echo on

-- Create PHP application user
connect system/sysempwd

-- Create the PHP application user
drop user phpuser cascade;
create user phpuser identified by welcome;
grant connect, resource to phpuser;
alter user phpuser default tablespace users
    temporary tablespace tnt unlock;

-- Create user owner security information about the application
drop user php_sec_admin cascade;
create user php_sec_admin identified by welcome;
alter user php_sec_admin default tablespace system
    temporary tablespace temp account unlock;
grant create procedure, create session, create table,
    resource, select any dictionary to php_sec_admin;

connect phpuser/welcome

-- "Parts" table for the application demo
create table parts
    (id      number primary key,
     category varchar2(20),
     name    varchar2(20));

insert into parts values (1, 'electrical', 'lamp');
insert into parts values (2, 'electrical', 'wire');
insert into parts values (3, 'electrical', 'switch');
insert into parts values (4, 'plumbing', 'pipe');
insert into parts values (5, 'plumbing', 'sink');
insert into parts values (6, 'plumbing', 'toilet');
commit;

connect php_sec_admin/welcome

-- Authentication table with the web user user names & passwords.
-- A real application would NEVER store plain-text passwords but this
```

Authorization and Authentication With Client Identifiers

```
-- article is about uses of client identifiers and not about
-- authentication.
create table php_authentication
  (app_username varchar2(20) primary key,
   app_password varchar2(20) not null);

insert into php_authentication values ('chris', 'tiger');
insert into php_authentication values ('alison', 'red');
commit;

grant select on php_authentication to phpuser;
```

Production applications would not use such simple passwords and would *never* store clear text passwords in tables. Applications could do end user authentication in a number of ways, including using LDAP.

Each script in the PHP application needs to know the Oracle DB credentials so they are stored in a common include file *dbinfo.inc.php*:

Script 16: dbinfo.inc.php

```
<?php
// All connections to the database use these credentials
define("ORA_CON_UN", "phpuser");
define("ORA_CON_PW", "welcome");
define("ORA_CON_DB", "localhost/orcl");
?>
```

In real life, consider using Oracle Wallet Manager and connecting with OCI_CRED_EXT instead of hard coding the database password.

The application login page is a typical simple PHP script that when first loaded displays a form:

Login Page

Welcome

Username:

Password:

Figure 58: Login page of login.php.

The code that generates this form looks like:

Script 17: login.php

```
<?php

require_once('./dbinfo.inc.php');
session_start();

function login_form($message)
```

Connecting to Oracle Using OCI8

```
{  
    echo <<<EOD  
    <body style="font-family: Arial, sans-serif;">  
  
        <h2>Login Page</h2>  
        <p>$message</p>  
        <form action="login.php" method="POST">  
            <p>Username: <input type="text" name="username"></p>  
  
            <p>Password: <input type="text" name="password"></p>  
            <input type="submit" value="Login">  
        </form>  
    </body>  
EOD;  
}  
  
if (!isset($_POST['username']) || !isset($_POST['password'])) {  
    login_form('Welcome');  
} else {  
    // Check validity of the supplied username & password  
    $c = oci_pconnect(ORA_CON_UN, ORA_CON_PW, ORA_CON_DB);  
    // Use a "bootstrap" identifier for this administration page  
    oci_set_client_identifier($c, 'admin');  
  
    $s = oci_parse($c, 'select app_username  
                            from php_sec_admin.php_authentication  
                            where app_username = :un_bv  
                                and app_password = :pw_bv');  
    oci_bind_by_name($s, ":un_bv", $_POST['username']);  
    oci_bind_by_name($s, ":pw_bv", $_POST['password']);  
    oci_execute($s);  
    $r = oci_fetch_array($s, OCI_ASSOC);  
  
    if ($r) {  
        // The password matches: the user can use the application  
  
        // Set the user name to be used as the client identifier in  
        // future HTTP requests:  
        $_SESSION['username'] = $_POST['username'];  
  
        echo <<<EOD  
        <body style="font-family: Arial, sans-serif;">  
            <h2>Login was successful</h2>  
            <p><a href="application.php">Run the Application</a><p>  
        </body>  
EOD;  
    }  
    else {  
        // No rows matched so login failed  
        login_form('Login failed. Valid usernames/passwords ' .  
                    'are "chris/tiger" and "alison/red"');  
    }  
}
```

```
?>
```

If you copy this code, make sure the **EOD** "heredoc" tokens are at the very start of their lines.

For form submission, the script calls back to itself, which now validates the entered user name and password against the users in the PHP_AUTHENTICATION table. A client identifier of **admin** is set as a bootstrap value since at this initial point we don't know if we have a valid end user and also the login script is an administrative component not doing any actual application work on behalf of an end user.

From the login page, authenticated users can click to the application inventory page:

Login was successful

[Run the Application](#)

Figure 59: Successful login in login.php

The user name is passed to the application page in PHP session data as **\$_SESSION['username']**. This value will be used as the client identifier for the web user. In a real application a less obvious identifier would be recommended. For example, as part of application authentication for a successful end-user login, an initial look-up query or PL/SQL function could return a pre-computed obscure value to be used as the user's client identifier. This value would then be stored in the PHP session information for use in subsequent "real" application work. An obscure value would make it harder for attackers to predict identifier values. Also an identifier could be quickly changed if there was ever a concern about the authenticity of HTTP requests using it.

The application page *application.php* checks that the user is authenticated - this application's definition of an authenticated user is simply that a user name is set. The code then sets the client identifier and shows the inventory list by querying the PARTS table:

Script 18: application.php

```
<?php

require_once('./dbinfo.inc.php');
session_start();

// Check the user is logged in according to our application authentication
if (!isset($_SESSION['username'])) {
    echo <<<EOD
        <h2>Unauthorized</h2>
        <p>You are not authenticated.<br>
        Valid usernames/passwords are "chris/tiger" and "alison/red"<p>

        <p><a href="login.php">Login Page</a><p>
EOD;
    exit;
}

// Generate the application page

$c = oci_pconnect(ORA_CON_UN, ORA_CON_PW, ORA_CON_DB);
```

Connecting to Oracle Using OCI8

```
// Set the client identifier after every connection call
// using a value unique for the web end user.
oci_set_client_identifier($c, $_SESSION['username']);

$username = htmlentities($_SESSION['username'], ENT_QUOTES);
echo <<<EOD
<body style="font-family: Arial, sans-serif;">
<h2>Parts Company</h2>
<table border='1'>

<caption><b>Inventory for $username </b></caption>
EOD;

$s = oci_parse($c, "select * from parts order by id");
oci_execute($s);
while (($row = oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_NULLS))
      != false) {
    echo "<tr>\n";
    foreach ($row as $item) {
        echo "    <td>" .
            ($item!==null?htmlentities($item, ENT_QUOTES):" ") .
            "</td>\n";
    }
    echo "</tr>\n";
}

echo <<<EOD

</table>
<p><a href="logout.php">Logout</a></p>
</body>
EOD;
?>
```

When logged in as Chris, the application shows:

Parts Company

Inventory for chris

1	electrical	lamp
2	electrical	wire
3	electrical	switch
4	plumbing	pipe
5	plumbing	sink
6	plumbing	toilet

[Logout](#)

Figure 60: Parts report for Chris.

A logout script clears PHP's session information:

Script 19: logout.php

```
<?php  
  
session_start();  
unset($_SESSION['username']);  
  
echo <<<EOD  
<body style="font-family: Arial, sans-serif;">  
<h2>Goodbye</h2>  
  
<p>You are logged out.<p>  
  
<p><a href="login.php">Login Page</a><p>  
</body>  
EOD;  
  
?>
```

The logout page does not call `dbms_session.clear_identifier` to clear the database connection's identifier: that would need to be done at the end of all the files that use a database connection if there was concern about the effect on monitoring due to the identifiers not being cleared.

The same application code will be used in the next sections without any modifications.

To summarize, this simple application is designed to show the relationship between database users and end users so that client identifiers can be discussed. It does not constitute a suitable example for production use. The application sets a client identifier with `oci_set_client_identifier()` immediately after each `oci_pconnect()` connection call. This identifier uniquely identifies the end user who is sitting at his or her web browser. For existing real-life applications, adding a call to `oci_set_client_identifier()` with a unique identifier per web user is the only application change that needs to be made to take advantage of client identification.

Using a Client Identifier in PHP for Auditing

Auditing lets you:

- Identify inappropriate database changes
- Investigate suspicious activity
- Verify authorization or access control policies
- Satisfy business compliance regulations
- Gather data about database activities for use in capacity and resource allocation planning

Oracle auditing is powerful and multi-faceted. You can audit general activities such as the type of SQL statement executed. You can audit fine grained activities such as when specific values occur, or what IP address initiated a request. Auditing can occur on both successful and failed activities. The audit trail can be stored inside the database or outside it, suitable for analysis with various tools. Auditing is available in various forms in different editions of the database. The full “Fine Grained Auditing” feature is available with Oracle Database Enterprise Edition.

Connecting to Oracle Using OCI8

Setting a client identifier allows auditing to be associated with unique web users, and not just with the database schema owner who authenticated the PHP OCI8 `oci_pconnect()` call to the database.

The `auditon.sql` script is a basic example of query auditing on the PARTS table:

Script 20: auditon.sql

```
-- Turn on object auditing for the PARTS table
connect system/sysempwd
audit select on phpuser.parts by access;
```

Run `autiton.sql` in SQL*Plus. Then run the application and login as 'chris' or 'alison' (their passwords are set to 'tiger' and 'red' respectively in `setup.sql`). You can even query the table as the SYSTEM user in SQL*Plus outside the application:

```
SQL> select * from phpuser.parts;
```

This returns the expected parts list.

To show the audit trail from all these table accesses, the SQL script `auditreport.sql` queries the DBA_AUDIT_TRAIL view, which contains the audit data when the database initialization parameter `AUDIT_TRAIL` is set to DB.

Script 21: auditreport.sql

```
-- View the audit trail for the PARTS table

connect system/sysempwd

set pagesize 100

col app_username format a13
col username format a13
col extended_timestamp format a37
col action_name format a13

select auth.app_username,
       dat.username,
       extended_timestamp,
       action_name
  from  dba_audit_trail dat
  left outer join
    php_sec_admin.php_authentication auth
   on auth.app_username = client_id
 where obj_name = 'PARTS'
 order by extended_timestamp;
```

Running the report shows the time each web user accessed the PARTS table:

APP_USERNAME	USERNAME	EXTENDED_TIMESTAMP	ACTION_NAME
chris	PHPUSER	16-AUG-10 12.25.42.846153 PM -07:00	SELECT
alison	PHPUSER	16-AUG-10 12.25.50.870773 PM -07:00	SELECT
	SYSTEM	16-AUG-10 12.25.58.660922 PM -07:00	SELECT

Authorization and Authentication With Client Identifiers

There is no APP_USERNAME shown for the SYSTEM user because there was no client identifier set in the SQL*Plus session. Sometimes identifying data accesses where the client identifier is not correctly set is the desired auditing goal. Oracle's Fine-Grained Auditing can be used to audit specific events like this, helping monitor suspicious activity. This can be useful when client identifiers are used by Virtual Private Databases to restrict data access but complete auditing is not required.

When you are finished exploring the example, you can turn auditing off using the NOAUDIT command in SQL*Plus:

Script 22: auditoff.sql

```
-- Turn off object auditing for the PARTS table
connect system/sysempwd
noaudit all on phpuser.parts;
```

More information about auditing can be found in the *Verifying Security Access with Auditing* chapter of the *Oracle Database Security Guide 11g Release 2 (11.2)* manual.

Using a Client Identifier in PHP With a VPD for Restricting Data Access

Limiting access to avoid misuse of sensitive data is an architectural goal of all applications. Oracle PHP applications can use the client identifier to restrict data access in a manually coded or an automatic way. The manual way is to modify every SQL and PL/SQL statement to use `sys_context()`, which returns the client identifier of the PHP connection. For example, queries could be written to return rows from PARTS only when the identifier of the current connection is 'chris':

```
select * from parts
where sys_context('userenv', 'client_identifier') = 'chris';
```

When Alison (or any user with a different client identifier) is connected, then the WHERE clause evaluates to false and no rows will be returned. This kind of logic is cumbersome to code and error prone to consistently implement everywhere. Oracle Database Enterprise Edition's Virtual Private Database (VPD) technology comes to the rescue. It will automatically add a WHERE predicate to each statement the application executes.

To set up VPD, a PL/SQL function that returns the desired text of the restrictive WHERE clause needs to be created. To automatically restrict data returned from the query `select * from parts` the PL/SQL function would just need to return the string:

```
sys_context('userenv', 'client_identifier') = 'chris'
```

With VPD enabled to use such a function, the query from PARTS would be executed by Oracle as if it had the restrictive WHERE clause, resulting in the same application behavior as discussed above for the manual implementation. Technically Oracle uses a transient view that enforces the WHERE clause and rewrites the application query to use the view instead of the base table, as described in the VPD documentation. Regardless of the implementation details, Oracle transparently handles the authorization, so security is consistent and programmers can be more productive on other tasks. Remember that from the database perspective, client identifiers are "insecure" because the database has to rely on externally provided information for policy enforcement. This is the outcome of using shared database connections and middle-tier authentication in a stateless web architecture and it places a reliance on having correct application code.

Connecting to Oracle Using OCI8

For the Parts application, the SQL script *vpdon.sql* sets up VPD. First it creates an application specific table of privileges. In this example Chris can only see electrical items but Alison can see electrical and plumbing supplies. The VPD policy function F_POLICY_PARTS returns a subquery that checks the current client identifier has access to the part category of the row. Although F_POLICY_PARTS is passed the schema and table name that the policy is being applied to, in this example the policy is only used for one table so the function parameters are not referenced. With the policy function defined, the DBMS_RLS.ADD_POLICY procedure is used to enable it for the PARTS table.

Script 23: *vpdon.sql*

```
set echo on

connect / as sysdba
grant execute on sys.dbms_rls to php_sec_admin;

connect php_sec_admin/welcome

-- Application policy table
drop table php_privs;
create table php_privs (username varchar2(64), category varchar2(20));

-- Chris should only see electrical items. Alison can see
-- electrical and plumbing items
insert into php_privs values ('chris', 'electrical');
insert into php_privs values ('alison', 'electrical');
insert into php_privs values ('alison', 'plumbing');
commit;

grant select on php_privs to phpuser;

-- Policy function F_POLICY_PARTS returns a "WHERE" clause to restrict access

create or replace function f_policy_parts
  (schema in varchar2, tab in varchar2) return varchar2
as
  predicate varchar2(400);

begin
  predicate :=
    'category in
     (select category
      from php_sec_admin.php_privs
      where username = sys_context(''userenv'', ''client_identifier''))';
  return predicate;
end;
/
show errors

begin
  dbms_rls.add_policy (
    object_schema  => 'PHPUSER',
    object_name    => 'PARTS',
    policy_name    => 'ACCESS_CONTROL_PARTS',
```

```
function_schema => 'PHP_SEC_ADMIN',
policy_function  => 'F_POLICY_PARTS',
policy_type      => DBMS_RLS.STATIC);
end;
/
```

The policy functions in an application can be as complex as needed. Your own web sites can implement policy rules in the most suitable way for them, which is likely to be completely different to that used in this example. Take care with the POLICY_TYPE argument. Here the policy function returns a simple string, making the function identical for all uses. This means the type can be specified as DBMS_RLS.STATIC allowing the function to be cached. The client identifier is not considered part of the user defined application context so if the policy function logic evaluates the identifier value then you will need to set the type to DBMS_RLS.DYNAMIC.

Login to the Parts application as Chris see how the inventory list now only shows electrical supplies:

Parts Company

Inventory for chris

1	electrical	lamp
2	electrical	wire
3	electrical	switch

[Logout](#)

Figure 61: VPD automatically restricts the report for Chris.

When logged in as Alison you can continue to see everything:

Parts Company

Inventory for alison

1	electrical	lamp
2	electrical	wire
3	electrical	switch
4	plumbing	pipe
5	plumbing	sink
6	plumbing	toilet

[Logout](#)

Figure 62: The VPD policy allows Alison to see everything.

Connecting to Oracle Using OCI8

Oracle Database has a number of views for VPD management. One is the V\$VPD_POLICY view that can be used to find the policies that were applied to executed SQL statements. This can be useful for debugging the values returned by the policy function.

A fun thing to do with VPD is to login to SQL*Plus as the owner of the PARTS table and check its contents:

```
SQL> connect phpuser/welcome
SQL> select * from parts;
```

No rows will be returned because the policy function F_POLICY_PARTS is applied even for the table owner. Without having an identifier validly set, the condition can never be satisfied. To remove this restriction and make administration of objects easier, Oracle has an EXEMPT ACCESS POLICY privilege for exempting users from VPD policies.

VPD is useful for more than personnel access control. The model can be extended to allow "shared hosting". The infrastructure for one application can be shared between multiple different groups of people who are never authorized to see data from any other group.

When you are finished with the VPD example, you can remove the policy by dropping it:

Script 24: vpdoff.sql

```
set echo on
connect php_sec_admin/welcome
begin
    dbms_rls.drop_policy (
        object_schema => 'PHPUSER',
        object_name   => 'PARTS',
        policy_name   => 'ACCESS_CONTROL_PARTS');
end;
/
```

More information on VPD can be found in the *Using Oracle Virtual Private Database to Control Data Access* chapter of the *Oracle Database Security Guide 11g Release 2 (11.2)* manual.

Using a Client Identifier in PHP for Monitoring and Tracing

While many tuning projects start with the automatic performance diagnostics run by Oracle Database, or analyze overall system performance manually using AWR snapshots, this may not be possible in all environments. Sometimes on a shared system, monitoring and analyzing the behavior of one web user is more practical and simplifies the process of diagnosing performance problems in PHP. During development, the behavior of a proposed application patch can be isolated from what else is happening on the system. The client identifier allows focused monitoring via Oracle's End to End Application Tracing, a feature introduced for multi-tier applications.

To collect database statistics about a user's database resource usage, the database administrator can execute `dbms_monitor.client_id_stat_enable()` in SQL*Plus:

```
SQL> connect system/systempwd
SQL> execute dbms_monitor.client_id_stat_enable(client_id => 'chris');
```

Authorization and Authentication With Client Identifiers

The PHP application can then be run normally for any chosen actions and duration. Statistics it produces can be accessed in various way, including from the V\$CLIENT_STATS view. After Chris looks at the Parts application inventory once, this view might contain:

STAT_NAME	VALUE
user calls	1
DB time	943
DB CPU	2000
parse count (total)	1
parse time elapsed	91
execute count	1
sql execute elapsed time	494
opened cursors cumulative	1
session logical reads	7
physical reads	0
physical writes	0
...	

This particular example shows a single SQL statement was parsed and executed. Standard Oracle manuals and literature describes interpreting all the values, and describe the other statistics views.

Statistics can be turned off and reset to zero with:

```
execute dbms_monitor.client_id_stat_disable(client_id => 'chris');
```

A database trace to show the SQL "Explain Plan" output for analyzing executed statements can also be turned on for each web user. It will show how statements actually got optimized - not just how you thought they would be run. The database administrator can enable tracing with the DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE procedure in SQL*Plus:

```
SQL> connect system/systempwd
SQL> execute dbms_monitor.client_id_trace_enable(client_id => 'chris',
                                                 waits => true, binds => true);
```

The application can then be run normally. After completion of the analysis period, tracing can be turned off with:

```
SQL> execute dbms_monitor.client_id_trace_disable(client_id => 'chris');
```

To examine the created trace files, find the trace directory using SHOW PARAMETER in SQL*Plus:

```
SQL> show parameter user_dump_dest
```

This gives output like:

NAME	TYPE	VALUE
user_dump_dest	string	/home/oracle/app/diag/rdbms/orcl/orcl/trace

The trace directory typically contains many trace files from normal operation. The *trcsess* utility can consolidate any of those created by Chris's use of the application. The consolidation (or an individual file) can then be formatted with *tkprof*. For example, start a terminal window as the Oracle software owner and run:

Connecting to Oracle Using OCI8

```
$ cd /home/oracle/app/diag/rdbms/orcl/orcl/trace  
$ trcsess output=/tmp/all.trc clientid=chris *.trc  
$ tkprof /tmp/all.trc /tmp/tkprof.out explain=phpuser/welcome
```

This looks through all the trace files in the directory and aggregates those created by Chris. If you need to run *trcsess* on a subset of files, such as the files for a particular day, search for the client identifier near the top of the files and pass the relevant file names to *trcsess*. Individual trace files contain a section like:

```
*** 2010-08-16 15:29:12.481  
*** SESSION ID:(143.943) 2010-08-16 15:29:12.481  
*** CLIENT ID:(chris) 2010-08-16 15:29:12.481  
*** SERVICE NAME:(orcl) 2010-08-16 15:29:12.481  
*** MODULE NAME:(httpd@localhost (TNS V1-V3)) 2010-08-16 15:29:12.481  
*** ACTION NAME:() 2010-08-16 15:29:12.481
```

Oracle Database can also name files with a given suffix, for example *myphp*, to make them easier to identify. Do this by executing the SQL command `alter session set tracefile_identifier = 'myphp'` in PHP after connecting. Database server trace file names would then look like *orcl_ora_9414_myphp.trc*. Logic would need to be added to each PHP file to decide the trace file suffix to use, and what conditions to set it.

The output from *tkprof* in *tkprof.out* contains analysis of the executed statements. Here is a section of the file analyzing results on a small system:

```
SQL ID: af69s0fa3cjnp  
Plan Hash: 3769467330  
select *  
from  
  parts order by id  
  
call      count        cpu   elapsed         disk      query     current         rows  
-----  -----  -----  -----  -----  -----  
Parse       5        0.00    0.16          0          8          0          0  
Execute     33        0.00    0.00          0          0          0          0  
Fetch       33        0.00    0.00          0        231          0        198  
-----  
total      71        0.01    0.17          0        239          0        198  
  
Misses in library cache during parse: 1  
Optimizer mode: ALL_ROWS  
Parsing user id: 1602  (PHPUSER)  
  
Rows      Row Source Operation  
-----  
 6  SORT ORDER BY (cr=7 pr=0 pw=0 time=5 us cost=4 size=222 card=6)  
 6  TABLE ACCESS FULL PARTS (cr=7 pr=0 pw=0 time=5 us cost=3 size=222  
card=6)  
  
Rows      Execution Plan  
-----  
 0  SELECT STATEMENT  MODE: ALL_ROWS  
 6  SORT (ORDER BY)
```

Authorization and Authentication With Client Identifiers

```
6    TABLE ACCESS (FULL) OF 'PARTS' (TABLE)
```

Elapsed times include waiting on following events:

Event waited on	Times Waited	Max. Wait	Total Wasted
SQL*Net message to client	33	0.00	0.00
SQL*Net message from client	32	2219.78	5720.55
cursor: mutex S	1	0.02	0.02
library cache lock	1	0.01	0.01
cursor: pin S wait on X	1	0.01	0.01
Disk file operations I/O	4	0.00	0.00

The section *Understanding SQL Trace and TKPROF* in the *Oracle Database Performance Tuning Guide 11g Release 2 (11.2)* manual describes how to interpret the trace output.

For lovers of GUIs, Enterprise Manager 11g (Database Control) has a number of ways to check the impact of the application on the database. For example, to see statistics for a particular client identifier, start the Enterprise Manager console <http://localhost:5500/em/> in a browser and navigate to *Performance > Top Consumers > Top Clients*. Set the *View* dropdown to "Clients with Aggregation Enabled". Click *Add Client* and specify the client identifier 'chris'. You can then select the row 'chris' and click the *Enable SQL Trace* button (this is same as `DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE`). Aggregation allows each run to be totaled together:

The screenshot shows the 'Top Consumers' section of the Enterprise Manager interface. The 'Top Clients' tab is selected. A dropdown menu 'View' is set to 'Clients with Aggregation Enabled'. Below it are buttons for 'Enable SQL Trace', 'Disable SQL Trace', 'View SQL Trace File', and 'Disable Aggregation'. Under 'Select All | Select None', there is a table with one row for 'chris'. The table columns are: Client ID, Activity (% for the last 5 minutes), SQL Trace Enabled, Delta Elapsed Time (seconds), and Cumulative Time. The 'chris' row has a checked checkbox in the first column, an activity value of 16.7 in the second, 'TRUE' in the third, and '0' in the fourth. At the bottom, the 'Top Clients' tab is again highlighted.

Figure 63: Enterprise Manager Top Clients

Other areas of Enterprise Manager can also filter by client identifier, including the *Top Activity* report.

A single run of the small Parts application may not make Enterprise Manager's monitoring thresholds or be visible in the aggregation periods.

You might have noticed that `trcsess` and Enterprise Manager also let data be aggregated by *Action* and by *Module*. These values can be set in PHP OCI8 with the functions `oci_set_action()` and `oci_set_module_name()` respectively to identify which parts of a PHP

Connecting to Oracle Using OCI8

application are being executed. The section *Monitoring OCI8 SQL Statements* in the chapter *Executing SQL Statements with OCI8* covers them. Monitoring and tracing can show up the application-wide hot spots, and the SQL statements being executed can easily be identified.

Enterprise Manager is useful for tracing performance bottlenecks and tracking causes of database slowdowns during development. It allows live analysis in a large system without impacting other concurrent web users. For more information on tracing see *Using Application Tracing Tools* in the *Oracle Database Performance Tuning Guide 11g Release 2 (11.2)*.

Client Identifier Summary

Client identifiers should be used by PHP web applications that allow multiple application users to connect to the database via a single database user name. The identifier is a developer chosen value that can be derived from session information about the end user that is already present in most web applications. Client identifiers are set by simply calling the `oci_set_client_identifier()` function in PHP scripts connecting to the database. Oracle Database uses identifiers to audit, automatically restrict access to sensitive data, and allow focused monitoring and tracing of resource usage. Oracle PHP applications should use client identifiers so these Oracle Database features can be utilized at any point in the lifetime of the application.

Oracle Network Services and PHP

Oracle Net is the database component handling communication between the database and its clients. It allows sophisticated control over network connection management for connectivity, performance and has features such as encryption of network traffic. This section gives an overview of some Oracle Net features of interest to PHP applications. Tuning the OS, hardware and TCP/IP stack will also substantially help improve performance and scalability.

Some of the Oracle Net settings are configured in a file called *sqlnet.ora* that you can create. For PHP, it should be put in the same directory as the *tnsnames.ora* file if you use one. Otherwise, set the `TNS_ADMIN` environment variable to the directory containing *sqlnet.ora*. The database server can also have a *sqlnet.ora* file, which should be in `$ORACLE_HOME/network/admin`. This directory on the database server also contains *listener.ora*, a file automatically created during database installation, which configures the Oracle Network listener process.

Connection Rate Limiting

Large sites that have abnormal spikes in the number of users connecting can prevent database host CPU overload by limiting the rate that connections can be established. The database *listener.ora* file can specify a `RATE_LIMIT` clause to set the maximum number of requests per second that will be serviced:

```
LISTENER=(ADDRESS_LIST=
  (ADDRESS=(PROTOCOL=tcp)(HOST=sales)(PORT=1521)(RATE_LIMIT=4))
```

The value will depend on the hardware in use.

Setting Connection Timeouts

From Oracle 10.2.0.3 onwards, you can specify a connection timeout in case there is a network problem during connection. This lets PHP connections return an Oracle error to the user faster, instead of appearing to "hang". Set SQLNET.OUTBOUND_CONNECT_TIMEOUT in the client side (the PHP-side) *sqlnet.ora* file. This sets the upper time limit for establishing a connection right through to the database, including the time for attempts to connect to other database services.

In Oracle 11g, a slightly lighter-weight solution TCP.CONNECT_TIMEOUT was introduced. It is also a *sqlnet.ora* parameter. It bounds just the TCP connection establishment time, which is mostly where connection problems occur.

Configuring Authentication Methods

There are many ways to configure connection and authentication. For example a connection:

```
$c = oci_connect("hr", "welcome", "abc");
```

could be evaluated by Oracle 10g as using the Easy Connect syntax to host machine *abc* (using the default port and database service) or using a net alias *abc* configured in a *tnsnames.ora* file.

The flexibility can cause a delay in getting an error back if the connection details are invalid or a database is not operational. Both internal connection methods may be tried in sequence adding to the time delay before a PHP script gets the error. This depends on your Oracle Net and DNS settings.

How Oracle is configured to authenticate the user's credentials (here a username and password) can also have an effect.

The issue is not specific to PHP. In SQL*Plus the connection:

```
$ sqlplus hr/welcome@abc
```

would be the same.

In a basic Oracle 10g or 11g installation, one way to return an error as soon as possible is to set this in your OCI8 *sqlnet.ora* file:

```
NAMES.DIRECTORY_PATH = (TNSNAMES)
SQLNET.AUTHENTICATION_SERVICES = (NONE)
```

This DIRECTORY_PATH value disables Easy Connect's [hostname:port/service](#) syntax. Instead of using Easy Connect syntax, a connect name and *tnsnames.ora* file would be needed. (This may also add a small measure of security if your scripts accidentally allow arbitrary connection identifiers. It stops users guessing database server hostnames they should not know about.)

Setting AUTHENTICATION_SERVICES to NONE stops different authentication methods being tried. Although this may prevent privileged database connections, which require operating system authorization, this, again, might be beneficial. Check the Oracle Net documentation for details and for other authentication methods and authentication-type specific timeout parameters.

Connecting to Oracle Using OCI8

Detecting Dead PHP Apache Sessions

If a PHP Apache process hangs, its database server process will not be closed. This will not prevent other PHP processes from continuing to work, unless a required lock was not released.

The TCP keepalive feature will automatically detect unusable connections based on the operating system timeout setting, which is typically some hours. This detection is enabled on the database server by default.

Oracle Net itself can also be configured to detect dead connections, which is useful if a much smaller timeout compared with the default TCP keepalive is desired. This is configured by SQLNET.EXPIRE_TIME in the database \$ORACLE_HOME/network/admin/sqlnet.ora. A starting recommendation is to set it to 10 minutes. If a dead or terminated connection is identified, the server process exits.

Both settings will use some resources. Avoid setting them too short, which may interrupt normal user activity.

Detecting Dead Database Servers

If the network drops out – or a RAC node goes down, the connection to the database server from a PHP process is likely to be lost. You can tell PHP to detect and recover from these dropouts by enabling keepalive in the PHP side tnsnames.ora using the ([ENABLE=BROKEN](#)) clause under the DESCRIPTION parameter in a connect string. The operating system settings will be used to determine the timeout period.

Other Oracle Net Optimizations

Oracle Net lets you tune a lot of other options too. Check the *Oracle Net Services Administrator's Guide* and the *Oracle Net Services Reference* for details and more features. A few more tips are mentioned below.

The best session data unit (SDU) size will depend on the application. An 8K size (the new default in Oracle 11g) is suitable for many applications. If LOBs are used, a bigger value might be better. It should be set the same value in both the database server sqlnet.ora and the OCI8 tnsnames.ora file.

For sites that have a large number of connections being made, tune the QUEUESIZE option in the *listener.ora* file.

Keeping the PATH environment variable short for the *oracle* user on the database machine can reduce time for forking a database server process. This is of most benefit for standard PHP connections. Reducing the number of environment variables also helps.

Tracing Oracle Net

Sometimes your network is the bottleneck. If you suspect this is the case, turn on Oracle Net tracing in your OCI8 sqlnet.ora file and see where time is being spent. The example \$ORACLE_HOME/network/admin/sample/sqlnet.ora has some notes on the parameters that help. For example, with a USER level trace in the PHP-side sqlnet.ora:

```
trace_level_client = USER
trace_directory_client = /tmp
```

And the PHP code:

```
$c = oci_connect('hr', 'welcome', '#c'); // invalid db name
```

The trace file, for example `/tmp/cli_3232.trc`, shows:

```
...
[17-JUN-2012 09:54:58:100] nnftmlf_make_system_addrfile: system names file is ...
[17-JUN-2012 09:55:00:854] snlinGetAddrInfo: Name resolution failed for #c
...
```

The left hand column is the timestamp of each low level call. Here, it shows a relatively big time delay doing name resolution for the non-existent host `#c`. The cause is the configuration of the machine network name resolution.

The logging infrastructure of Oracle 11g changed significantly. Look for trace files in a sub-directory of the Oracle diagnostic directory, for example in `$HOME/oradiag_cjones/diag/clients/user_cjones` for command line PHP. For Oracle Net log files created by a web server running PHP, look in `/root/oradiag_root` if no other path was configured. You can also set the environment variable `ADR_BASE` to indicate where tracing and logging files should be stored.

Connecting to Oracle Using OCI8

EXECUTING SQL STATEMENTS WITH OCI8

This chapter discusses using SQL statements with the PHP OCI8 extension. It covers statement execution, the OCI8 functions available, handling transactions, tuning queries, and some useful tips and tricks.

SQL Statement Execution Steps

Queries using the OCI8 extension follow a model familiar in the Oracle world: parse, execute and fetch. Statements like `CREATE` and `INSERT` require only parsing and executing.

The possible steps are:

1. **Parse:** Prepares a statement for execution. Parsing is really just a light-weight local preparatory step, since Oracle's actual text parse occurs in the database at the execution stage.
2. **Bind:** An optional step that lets you bind data values, for example, in the `WHERE` clause, for better performance and security. Binding local values into a statement is similar to the way you use a `%s` print format specification in a string.
3. **Define:** An optional step allowing you to specify which PHP variables will hold query results. This is not commonly used because most scripts use the OCI8 *fetch* functions to return results.
4. **Execute:** The database processes the statement and buffers any results.
5. **Fetch:** Gets any query results back from the database.

There is no one-stop function to do all these steps in a single PHP call but it is trivial to create one in your application and you can then add custom error handling requirements.

To safeguard from run-away scripts, by default PHP will terminate if a script takes longer than 30 seconds. This is set with the `php.ini` parameter `max_execution_time` or the `set_time_limit()` function. The value is the CPU time used by PHP, so long running SQL statements in the database may not get interrupted. You will need to enable resource management in the database to overcome this.

Similarly, if you are manipulating large amounts of data, you may need to increase the `php.ini` parameter `memory_limit`, which caps the amount of memory each PHP process can consume.

Query Example

A basic query in OCI8 is:

Script 25: query.php

```
<?php
```

Executing SQL Statements With OCI8

```
$c = oci_pconnect("hr", "welcome", "localhost/XE");

$s = oci_parse($c, "select city, postal_code from locations");
oci_execute($s);
print '<table border="1">';
while ($row = oci_fetch_array($s, OCI_NUM+OCI_RETURN_NULLS)) {
    print "<tr>";
    foreach ($row as $item)
        print "<td>" . htmlentities($item) . "</td>";
    print "</tr>";
}
print "</table>";

?>
```

The output from the script *query.php* is:

Roma	00989
Venice	10934
Tokyo	1689
Hiroshima	6823
Southlake	26192
South San Francisco	99236
South Brunswick	50090
Seattle	98199
Toronto	M5V 2L7
Whitehorse	Y5W 9T2
Beijing	190518
Bombay	490231
Sydney	2901

Figure 64: Output from *query.php*.

The `htmlentities()` function prevents any user data such as '`<`' from being interpreted as an HTML tag. In many cases you will want to use this function's optional `ENT_QUOTES` parameter. You should also use the optional character set parameter, specifying a value that matches the character set of your document.

Quoting SQL Statement Text

In PHP, single and double quotes can both be used for strings. Strings with embedded quotes can be made by escaping the nested quotes with a backslash, or by using both quoting characters. The next example shows single quotes around the city name. To make the query a valid PHP string it, therefore, must be enclosed in double quotes:

```
$q = "select * from locations where city = 'Sydney'";
$s = oci_parse($c, $q);
```

If you have a query syntax error, echo out the string and verify the quoting is valid for Oracle:

```
echo $q;
```

PHP 5.3 introduced a NOWDOC syntax that is useful for embedding quotes and dollar signs in strings, such as this example that queries one of Oracle's administration views V\$SQL:

```
$sql = <<<'END'  
select parse_calls, executions  
from v$sql  
END;  
  
$s = oci_parse($c, $sql);  
. . .
```

Note the `END` token must appear in the first column of the script, without any indentation.

Freeing Statements

In long scripts it is recommended to close statements when they are complete:

```
oci_free_statement($s);
```

This allows resources to be reused efficiently. For brevity, and because the examples execute quickly or the statement resource is closed automatically at the end of function scope, most code snippets in this book do not follow this practice.

Oracle Data Types

Each column has a data type, which is associated with a specific storage format. The common built-in Oracle data types are:

- CHAR
- VARCHAR2
- NUMBER
- DATE
- TIMESTAMP
- INTERVAL
- BLOB
- CLOB
- BFILE
- XMLType

The CHAR, VARCHAR2, NUMBER, DATE, TIMESTAMP and INTERVAL data types are stored directly in PHP variables. BLOB, CLOB, and BFILE data types use PHP descriptors and are shown in the *Using Large Objects in OCI8* chapter. XMLTypes are returned as strings or LOBs, as discussed in the *Using XML with Oracle and PHP* chapter. Oracle's NCHAR, NVARCHAR2, and NCLOB types are not supported in the OCI8 extension.

Executing SQL Statements With OCI8

Fetch Functions

There are a number of OCI8 fetch functions, all documented in the *PHP Oracle OCI8 Manual* at <http://php.net/manual/en/book.oci8.php>. Table 6 lists the functions.

Table 6: OCI8 fetch functions.

OCI8 Function	Purpose
oci_fetch_all()	Gets all the results at once.
oci_fetch_array()	Gets the next row as an array indexed by an integer or as an associative array, depending on your choice.
oci_fetch_assoc()	Gets the next row as an associative array.
oci_fetch_object()	Gets the next row as an object.
oci_fetch_row()	Gets the next row as an integer indexed array.
oci_fetch()	Used with <code>oci_result()</code> , which returns the result of a given field. Also used with <code>oci_define_by_name()</code> which presets which variable the data will be returned into.

Some of the functions have optional parameters. Refer to the PHP manual for more information.

The function commonly used is `oci_fetch_array()`:

```
$rowarray = oci_fetch_array($statement, $mode);
```

The mode is optional. Table 7 lists the available modes.

Table 7: `oci_fetch_array()` options.

Parameter	Purpose
OCI_ASSOC	Return results as an associative array.
OCI_NUM	Return results as a numerically indexed array.
OCI_BOTH	Return results as both associative and numeric arrays. This is the default.
OCI_RETURN_NULLS	Return PHP NULL value for NULL data.
OCI_RETURN_LOBS	Return the actual LOB data instead of an OCI-LOB resource.

Modes can be used together by adding or binary-or'ing them together:

```
$rowarray = oci_fetch_array($s, OCI_NUM + OCI_RETURN_NULLS);
```

The `oci_fetch_assoc()` and `oci_fetch_row()` functions are special cases of `oci_fetch_array()`.

Fetching to a Numeric Array

A basic example to fetch results into a numerically indexed PHP array is:

```
$s = oci_parse($c, "select city, postal_code from locations");
oci_execute($s);
while ($row = oci_fetch_array($s, OCI_NUM)) {
    echo $row[0] . " - " . $row[1] . "<br>\n";
}
```

The two columns are index 0 and index 1 in the result array. This displays:

```
Roma - 00989
Venice - 10934
Tokyo - 1689
Hiroshima - 6823
Southlake - 26192
. . .
```

Some of the fetch functions do not return NULL data by default. This can be tricky when using numerically indexed arrays. The result array can appear to have fewer columns than selected, and you can't always tell which column was NULL. Either use associative arrays so the column names are directly associated with their values, or specify the [OCI_RETURN_NULLS](#) flag:

```
$row = oci_fetch_array($s, OCI_NUM + OCI_RETURN_NULLS);
```

Fetching to an Associative Array

Associative array keys are the column names in the case shown when describing the table in SQL*Plus:

```
SQL> describe locations
Name                           Null?    Type
-----                         -----
LOCATION_ID                    NOT NULL NUMBER(4)
STREET_ADDRESS                 VARCHAR2(40)
POSTAL_CODE                    VARCHAR2(12)
CITY                          NOT NULL VARCHAR2(30)
STATE_PROVINCE                 VARCHAR2(25)
COUNTRY_ID                     CHAR(2)
```

A script to access the post code column would be:

```
$s = oci_parse($c, "select postal_code from locations");
oci_execute($s);
while ($row = oci_fetch_array($s, OCI_ASSOC)) {
    echo $row["POSTAL_CODE"] . "<br>\n";
}
```

This displays:

```
00989
10934
1689
6823
26192
. . .
```

Executing SQL Statements With OCI8

Fetching Case Sensitive Column Names to an Associative Array

If the table column names were created case sensitively by using quotes at table creation, then the PHP array indices need to match the case:

```
SQL> create table cs_tab ("MyCol" number);
SQL> describe cs_tab
Name          Null?    Type
-----
MyCol          NUMBER
```

This should be fetched in PHP as:

```
while ($row = oci_fetch_array($s, OCI_ASSOC)) {
    echo $row['MyCol'] . "<br>\n";
}
```

Duplicate Column Names and Associative Arrays

In an associative array index there is no table prefix for the column name. If you have to join tables where the same column name occurs with different meanings in both tables, use a column alias in the query. Otherwise only one of the similarly named columns will be returned by PHP.

This contrived example selects two columns called COLOR:

```
$s = oci_parse($c, "select cat_name,
                     cats.color as cat_color,
                     dog_name,
                     dogs.color
                  from cats, dogs");
oci_execute($s);

while ($row = oci_fetch_array($s, OCI_ASSOC)) {
    echo $row["CAT_NAME"] . " " . $row["CAT_COLOR"] . " - " .
        $row["DOG_NAME"] . " " . $row["COLOR"] . " ".
    "<br>\n";
}
```

The associative array has COLOR and CAT_COLOR indices for the two different COLOR columns.

A straightforward alternative is to use [OCI_NUM](#) to return numeric array indexes.

Fetching to an Object

Fetching as objects allows property-style access to be used.

```
$s = oci_parse($c, 'select * from locations');
oci_execute($s);
while ($row = oci_fetch_object($s)) {
    var_dump($row);
}
```

This shows each row is an object and gives its properties. The [var_dump\(\)](#) function prints and automatically formats the variable [\\$row](#). The output is:

```
object(stdClass)#1 (6) {
    ["LOCATION_ID"]=>
        string(4) "1000"
    ["STREET_ADDRESS"]=>
        string(20) "1297 Via Cola di Rie"
    ["POSTAL_CODE"]=>
        string(5) "00989"
    ["CITY"]=>
        string(4) "Roma"
    ["STATE_PROVINCE"]=>
        NULL
    ["COUNTRY_ID"]=>
        string(2) "IT"
}
.
.
```

If the loop is changed to:

```
while ($row = oci_fetch_object($s)) {
    echo "Address is " . $row->STREET_ADDRESS . "<br>\n";
}
```

the output is:

```
Address is 1297 Via Cola di Rie
Address is 93091 Calle della Testa
Address is 2017 Shinjuku-ku
Address is 9450 Kamiya-cho
Address is 2014 Jabberwocky Rd
.
.
```

Defining Output Variables

Explicitly setting output variables can be done with [oci_define_by_name\(\)](#). This example fetches city names:

```
$s = oci_parse($c, 'select city from locations');
oci_define_by_name($s, 'CITY', $city); // column name is uppercase
oci_execute($s);
while (oci_fetch($s)) {
    echo "City is " . $city . "<br>\n";
}
```

The define is done before execution so Oracle knows where to store the output. The column name in the [oci_define_by_name\(\)](#) call must be in uppercase unless the table was created with case sensitive column names. The result is:

```
City is Roma
City is Venice
City is Tokyo
City is Hiroshima
City is Southlake
.
.
```

Executing SQL Statements With OCI8

The `oci_define_by_name()` function has an optional type parameter that is useful, for example, to specify that the PHP variable should be a LOB.

Fetching Nested Cursors

The next example shows a query that fetches two data values. The first is the `DEPARTMENT_NAME` and the second is a nested cursor for the sub-query of people within that department. This second value is given the column alias `NC`:

Script 26: nestedcurl.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$sql =
'select department_name,
       cursor(select first_name
              from employees
             where employees.department_id = departments.department_id) as nc
from departments
where department_id in (10, 20, 30)';

$s = oci_parse($c, $sql);
$r = oci_execute($s);
while (($row1 = oci_fetch_array($s, OCI_ASSOC)) != false) {
    echo "Department: " . $row1['DEPARTMENT_NAME'] . "<br>\n";
    $nc = $row1['NC'];           // treat as a statement resource
    oci_execute($nc);
    while (($row2 = oci_fetch_array($nc, OCI_ASSOC+OCI_RETURN_NULLS)) != false) {
        echo $row2['FIRST_NAME'] . "<br>\n";
    }
    oci_free_statement($nc);
    echo "<br>\n";
}

?>
```

The value in `$nc` is treated like a parsed query. After executing `$nc` any of the fetch functions can be used. The output is:

```
Department: Administration
Jennifer

Department: Marketing
Michael
Pat

Department: Purchasing
Den
Alexander
Shelli
Sigal
Guy
```

Karen

Fetching and Working With Numbers

Number Formatting and String Conversion

When fetching numbers, a conversion to string representation is done by Oracle. This means Oracle formats the data according to its globalization settings. In some regions the decimal separator for numbers might be a comma, causing problems if your PHP script later casts the string to a PHP number for an arithmetic operation. Oracle's default format can be changed easily and it is recommended to explicitly set the number conversion format `NLS_NUMERIC_CHARACTERS` to `'. ,'` if the default value inherited from your territory settings is different. See *Setting the Oracle Number Format with NLS_NUMERIC_CHARACTERS* in the chapter *Globalization*.

Number Accuracy

PHP and Oracle differ in their arithmetic handling and precision so a choice must be made where to do calculations.

If your application depends on numeric accuracy with financial data, do arithmetic in Oracle SQL or PL/SQL, or consider using PHP's `bcmath` extension.

This example shows how by default PHP fetches numbers as strings, and the difference between doing arithmetic in PHP and the database. SQL statements to create the number data are:

```
create table dt (cn1 number, cn2 number);
insert into dt (cn1, cn2) values (71, 70.6);
commit;
```

PHP code to fetch the row is:

```
$s = oci_parse($c, "select cn1, cn2, cn1 - cn2 as diff from dt");
oci_execute($s);
$row = oci_fetch_array($s, OCI_ASSOC);
var_dump($row);
```

The `var_dump()` function shows the PHP data type for numeric columns is *string*:

```
array(3) {
  ["CN1"]=>
  string(2) "71"
  ["CN2"]=>
  string(4) "70.6"
  ["DIFF"]=>
  string(2) ".4"
}
```

Arithmetic calculations are handled with different precision in PHP. The previous example showed the result of the subtraction was the expected value. If the code is changed to do the subtraction in PHP:

```
$row = oci_fetch_array($s, OCI_ASSOC);
```

Executing SQL Statements With OCI8

```
$diff = $row['CN1'] - $row['CN2'];
echo "PHP difference is " . $diff . "\n";
```

The output shows:

```
PHP difference is 0.40000000000001
```

PHP has a *php.ini* parameter *precision* which determines how many significant digits are displayed in floating point numbers. By default it is set to 14.

Fetching and Working With Dates

Oracle has capable date handling functionality, supporting various needs. Dates and times with user specified precisions can be stored. Oracle's date arithmetic makes calendar work easy.

DATE, DATETIME and INTERVAL types are fetched from Oracle as strings, similar to the way PHP returns Oracle's numeric types.

The DATE type has resolution to the second but the default format is often just the day, month and year. This example queries a date column:

```
$s = oci_parse($c, "select hire_date from employees where employee_id = 200");
oci_execute($s);
$row = oci_fetch_array($s, OCI_ASSOC);
echo "Hire date is " . $row['HIRE_DATE'] . "\n";
```

If the default format is the Oracle American standard of DD-MON-YY then the output is:

```
Hire date is 17-SEP-87
```

Dates inserted are expected to be in the default Oracle format too:

```
$s = oci_parse($c, "insert into mydtb (dcol) values ('04-AUG-07')");
oci_execute($s);
```

The default format can be changed with Oracle's globalization setting NLS_DATE_FORMAT before or after PHP starts. See the chapter *Globalization*.

Regardless of the default, any statement can use its own custom format. When querying, use the [TO_CHAR\(\)](#) function. When inserting, use [TO_DATE\(\)](#):

```
// insert a date
$s = oci_parse($c,
    "insert into mydtb (dcol)
        values (to_date('2006/01/01 05:36:50', 'YYYY/MM/DD HH:MI:SS'))");
oci_execute($s);

// fetch a date
$s = oci_parse($c, "select to_char(dcol, 'DD/MM/YY') as dcol from mydtb");
oci_execute($s);
$row = oci_fetch_array($s, OCI_ASSOC);

echo "Date is " . $row["DCOL"] . "\n";
```

The output is:

```
Date is 01/01/06
```

To find the current database server time, use SYSDATE. Here the date and time returned by SYSDATE are displayed:

```
$s = oci_parse($c,
    "select to_char (sysdate, 'YYYY-MM-DD HH24:MI:SS') as now from dual");
$r = oci_execute($s);
$row = oci_fetch_array($s, OCI_ASSOC);
echo "Time is " . $row["NOW"] . "\n";
```

The output is:

```
Time is 2007-08-01 15:28:44
```

Oracle's TIMESTAMP type stores values precise to fractional seconds. You can optionally store a time zone or local time zone. For PHP, the local time zone would be the time zone of the web server, which may not be relevant to users located remotely.

For an example, SYSTIMESTAMP, which is analogous to SYSDATE, gives the current server time stamp and time zone:

```
$s = oci_parse($c, "select systimestamp from dual");
$r = oci_execute($s);
$row = oci_fetch_array($s, OCI_ASSOC);
echo "Time is " . $row["SYSTIMESTAMP"] . "\n";
```

The output is:

```
Time is 01-AUG-07 03.28.44.233887 PM -07:00
```

An INTERVAL type represents the difference between two date values. Intervals are useful for Oracle's analytic functions. In PHP they are fetched as strings, like DATE and TIMESTAMP are fetched.

Insert, Update, Delete, Create and Drop in PHP OCI8

Executing Data Definition Language (DDL) and Data Manipulation Language (DML) statements, like [CREATE](#) and [INSERT](#), simply requires a parse and execute:

```
$s = oci_parse($c, "create table iitest (col1 number)");
oci_execute($s);
```

The only-run-once installation sections of applications should contain almost all the [CREATE TABLE](#) statements used. Applications in Oracle do not commonly need to create temporary tables at run time, and it is expensive to do so. Use inline views, or join tables when required. In some cases "*global temporary tables*" might be useful.

Transactions in PHP OCI8

Using transactions to protect the integrity of data is as important in PHP as any other relational database application. Except in special cases, you want either all your changes to be committed, or none of them.

Unnecessarily committing or rolling back impacts database performance as it causes unnecessary network traffic (*round trips*) between PHP and the database.

Executing SQL Statements With OCI8

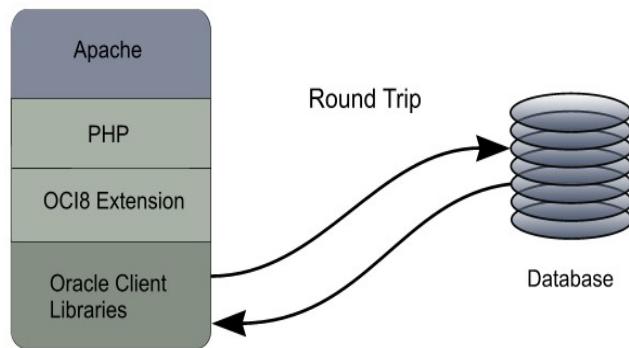


Figure 65: Each round trip between PHP and the database reduces scalability.

It also causes extra processing and more IO to the database files. To maximize efficiency, use transactions where appropriate.

OCI8's default commit behavior is like other PHP extensions but different from Oracle Database's standard. The default mode of `oci_execute()` is `OCI_COMMIT_ON_SUCCESS` to commit changes. This can easily be overridden in OCI8. But take care with committing and rolling back. Hidden transactional consistency problems can be created by not understanding when commits or rollbacks occur. Such problems may not be apparent in normal conditions, but an abnormal event might cause only part of a transaction to be committed. Problems can also be caused by programmers trying to squeeze out absolutely optimal performance by committing or rolling back only when absolutely necessary.

Scripts that call a connection function more than once with the same credentials should make sure transactions are complete before re-connecting. Similarly, be careful at the end of scope if the transaction state is uncommitted.

In the following example a new record is committed when the `oci_execute()` call is called:

```
$s = oci_parse($c, "insert into testtable values ('my data')");
oci_execute($s); // automatically committed by default
```

Other users of the table will immediately be able to see the new record. Auto-committing can be handy for a single `INSERT` or `UPDATE`, but transactional and performance requirements should be thought about before using the default mode everywhere.

You specify to begin a transaction without auto-committing by:

```
$s = oci_parse($c, "insert into testtest values ('my data 2')");
oci_execute($s, OCI_NO_AUTO_COMMIT); // not committed
```

Prior to OCI8 1.4 (prior to PHP 5.3), use `OCI_DEFAULT` instead of the new more obviously named alias `OCI_NO_AUTO_COMMIT`.

To commit any un-committed transactions for your connection, do:

```
oci_commit($c);
```

To rollback, do:

```
oci_rollback($c);
```

Any outstanding transaction is automatically rolled back when a connection is closed or at the end of the script. If you need a transaction that spans multiple HTTP requests, use DBMS_XA which is discussed in the chapter *Using PL/SQL with OCI8*.

Note: Be careful mixing and matching `oci_execute()` calls with both commit modes in one script, since you may commit at incorrect times. In particular, note executing a query will commit an outstanding transaction if OCI_NO_AUTO_COMMIT is **not** used in the query's `oci_execute()` call.

Any `CREATE` or `DROP` statement will automatically commit regardless of the `oci_execute()` mode. This is a feature of Oracle Database that cannot be altered.

If all your database calls in a script are queries, or are calls to PL/SQL packages that handle transactions internally, use:

```
oci_execute($s);
```

If you pass OCI_NO_AUTO_COMMIT, PHP will send an explicit rollback to the database at the end of every script, even though it is unnecessary for your application.

Autonomous Transactions

Oracle's PL/SQL procedural language (covered in detail in the next chapter) allows you to do autonomous transactions, which are effectively sub-transactions. An autonomous transaction can be committed or rolled back without affecting the main transaction. This might be useful for logging data access - an audit record can be inserted even if the user decides to rollback their main change. An example is:

Script 27: logger.sql

```
drop table mytable;
drop table logtable;

create table mytable (c1 varchar2(10));
create table logtable (event varchar2(30));

create or replace procedure updatelog(p_event in varchar2) as
    pragma autonomous_transaction;
begin
    insert into logtable (event) values(p_event);
    commit;
end;
/
```

You could call the PL/SQL function from PHP to log events:

Script 28: logger.php

```
<?php
```

Executing SQL Statements With OCI8

```
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "insert into mytable values ('abc')");
oci_execute($s, OCI_NO_AUTO_COMMIT); // don't commit

$s = oci_parse($c, "begin updatelog('INSERT attempted'); end;");
oci_execute($s, OCI_NO_AUTO_COMMIT); // don't commit

oci_rollback($c);

?>
```

Even though OCI_NO_AUTO_COMMIT is used, the autonomous transaction commits to the log table. This commit does not also commit the PHP script insert into MYTABLE. After running *logger.php*, the tables contain:

```
SQL> select * from mytable;
no rows selected

SQL> select * from logtable;
EVENT
-----
INSERT attempted
```

The Transactional Behavior of Multiple Connections

To see the transactional behavior of the three connection functions, use SQL*Plus to create a table with a single date column:

```
SQL> create table mytable (col date);
```

Create and run *transactions.php* a few times, changing `oci_connect()` to `oci_new_connect()` and `oci_pconnect()`:

Script 29: transactions.php

```
<?php

function do_query($c)
{
    $s = oci_parse($c, 'select col from mytable');
    oci_execute($s, OCI_NO_AUTO_COMMIT);
    $row = oci_fetch_array($s);
    echo "<p>Date is: " . $row['COL'] . "</p>\n";
}

$c1 = oci_connect("hr", "welcome", "localhost/XE"); // first PHP connection
$s = oci_parse($c1, "insert into mytable values ('" . date('j:M:y') . "')");
oci_execute($s, OCI_NO_AUTO_COMMIT);
do_query($c1);

$c2 = oci_connect("hr", "welcome", "localhost/XE"); // second PHP connection
```

```
do_query($c2);
?>
```

The script inserts (but does not commit) using one connection and queries back the results with both the original and a second connection.

Using an `oci_connect()` connection lets you query the newly inserted (but uncommitted) data both times because `$c1` and `$c2` refer to the same Oracle connection. Using `oci_pconnect()` is the same as `oci_connect()`. The output in both cases is:

```
Date is: 16-JUN-12
Date is: 16-JUN-12
```

Using `oci_new_connect()` for `$c2` gives a new connection which cannot see the uncommitted data. The output shows the second query does not fetch any rows:

```
Date is: 16-JUN-12
Date is:
```

PHP Error Handling

The *Installing and Configuring PHP* chapter recommended setting `display_errors` to On in `php.ini` to aid debugging. You can also set the `error_reporting` parameter to `E_ALL` to catch all potential problems during development. In a production system you should make sure error output is logged instead of displayed. You do not want to leak internal information to web users, and you do not want application pages containing ugly error messages.

The `E_STRICT` level was not part of the `E_ALL` level prior to PHP 5.4. You must explicitly “or” the two values together.

Error handling can also be controlled at runtime. For example, to see all errors displayed, scripts can set:

```
error_reporting(E_ALL); // In PHP 5.3 use E_ALL|E_STRICT
ini_set('display_errors', 'On');
```

Depending on the `php.ini` value of `display_errors`, you might consider using PHP’s `@` prefix to completely suppress automatic display of function errors, although this impacts performance:

```
$c = @oci_connect('hr', 'welcome', 'localhost/XE');
```

To trap output and recover from errors, PHP’s output buffering functions may be useful. If an error occurs part way during creation of the HTML page, the partially complete page contents can be discarded and a nicely formatted error page can be created instead.

Handling PHP OCI8 Errors

The error handing of any solid application requires careful design. Expect the unexpected. Check all return codes. Oracle may piggy-back calls to the database to optimize performance. This means that errors may occur during later OCI8 calls than you might expect.

To fetch OCI8 errors, use the `oci_error()` function. The function requires a different argument depending on the calling context, as shown later. It returns an array:

Executing SQL Statements With OCI8

Table 8: Error array after \$e = oci_error().

Variable	Description
\$e["code"]	Oracle error number.
\$e["message"]	Oracle error message.
\$e["offset"]	Column position in the SQL statement of the error. Will be 0 if no SQL statement was involved.
\$e["sqltext"]	The text of the SQL statement. Will be empty if no SQL statement was involved.

For information on getting extra information for errors during creation of PL/SQL procedures, see the chapter *Using PL/SQL with OCI8*.

OCI8 Connection Errors

To get connection error messages, no argument to `oci_error()` is needed:

```
error_reporting(E_ALL);
ini_set('display_errors', 'Off');           // Don't automatically show errors

$c = oci_connect('hr', 'not_welcome', 'localhost/XE');

if (!$c) {
    $e = oci_error();                      // No parameter passed
    ini_set('display_errors', 'On');        // Allow trigger_error() to display
    trigger_error('Could not connect: '. $e['message'], E_USER_ERROR);
}
```

With the invalid password, the output in a browser is:

```
Fatal error: Could not connect: ORA-01017: invalid username/password;
logon denied
```

OCI8 Persistent Connection Errors

An internal reimplementation of connection management in OCI8 1.3 made it much better at automatically recovering from database unavailability errors. However with persistent connections, PHP can still return a cached connection without knowing if the database at the other end of the network is in fact still available. If the database has restarted since the time of first connection, or if the DBA had enabled resource management that limited the maximum connection time for a user, or even if the DBA issued an `ALTER SYSTEM KILL SESSION` command to close the user's database session, then an already cached OCI8 connection returned by a future `oci_pconnect()` call will be unusable. The `oci_pconnect()` call will not return an error in this case. Typically the error will be returned by the first `oci_execute()` call that tries to use the connection resource. However OCI8 will then mark the connection as invalid and the subsequent time PHP calls `oci_pconnect()` a brand new connection to the database will be successfully created and will be usable in the PHP script. (The section on `oci8.ping_interval` in the chapter *Connecting to Oracle Using OCI8* has further discussion on this case).

For example, consider the script `oci8.php` at the start of the chapter *Connecting to Oracle Using OCI8*. After the script is successfully run in a web browser, if the DBA issues an `ALTER DATABASE KILL SESSION` command for the database connection it created, then the next time the script is run in a browser it will display the error after the `oci_execute()` call:

```
Could not execute statement: ORA-00028: your session has been killed
```

The `oci_pconnect()` call doesn't generate an error itself – it just returns a cached connection. If the script is run a third time by the same Apache process it will silently reconnect and run to completion normally. This assumes there is no ping due to `oci8.ping_interval`.

The bottom line is that all OCI8 calls should be checked for errors because architectural and internal optimizations may defer error notification.

OCI8 Parse Errors

To get parse error messages, pass `oci_error()` the connection resource:

```
error_reporting(E_ALL);
ini_set('display_errors', 'Off');           // Don't automatically show errors

$s = oci_parse($c, "select ' city from locations"); // Note stray quote
if (!$s) {
    $e = oci_error($c);                  // Connection resource passed
    ini_set('display_errors', 'On');      // Allow trigger_error() to display
    trigger_error('Could not parse: '. $e['message'], E_USER_ERROR);
}
```

Note the one extra single-quote in the middle of the query string. The result is the error message:

```
PHP Fatal error: Could not parse: ORA-01756: quoted string not properly
terminated
```

OCI8 Execution and Fetching Errors

An example of an execution error is when the table being queried does not exist. For execution errors, pass the statement resource to `oci_error()`:

```
error_reporting(E_ALL);
ini_set('display_errors', 'Off');           // Don't automatically show errors

$s = oci_parse($c, "select city from not_locations");
$r = oci_execute($s);
if (!$r) {
    $e = oci_error($s);                  // Statement resource passed
    ini_set('display_errors', 'On');      // Allow trigger_error() to display
    trigger_error('Could not execute: '. $e['message'], E_USER_ERROR);
}
```

The parse completes successfully but when the statement is sent to the database for execution, the table will not be found. The output is:

```
Fatal error: Could not execute: ORA-00942: table or view does not exist
```

Executing SQL Statements With OCI8

If you call `var_dump($e)`, you will see the array contains the text of the statement and the column offset position of the error in that statement. Column 17 of the query is the table name NOT_LOCATIONS:

```
array(4) {
    ["code"]=>
    int(942)
    ["message"]=>
    string(39) "ORA-00942: table or view does not exist"
    ["offset"]=>
    int(17)
    ["sqltext"]=>
    string(30) "select city from not_locations"
}
```

A fetch error might occur if the network to the database disconnects unexpectedly. For fetch errors, pass the statement resource:

```
$r = oci_fetch_all($s, $results);
if (!r) {
    $e = oci_error($s);           // Statement resource passed
    ini_set('display_errors', 'On'); // Allow trigger_error() to display
    trigger_error('Could not fetch: '. $e['message'], E_USER_ERROR);
}
```

Using Bind Variables in Prepared Statements

Bind variables are just like `%s` print format specifiers. They let you re-execute a statement with different values for the variables and get different results. In the PHP community statements like this are known as prepared statements.

If you do not bind, Oracle must reparse and cache multiple statements. Each statement requires creation of a cursor in the Oracle SGA, causes library latch contention, and causes shared pool contention. The overall result is scalability issues. An application that runs fine in small development and testing environments may not be able to handle the intended number of real users.

Using Bind Variables in Prepared Statements

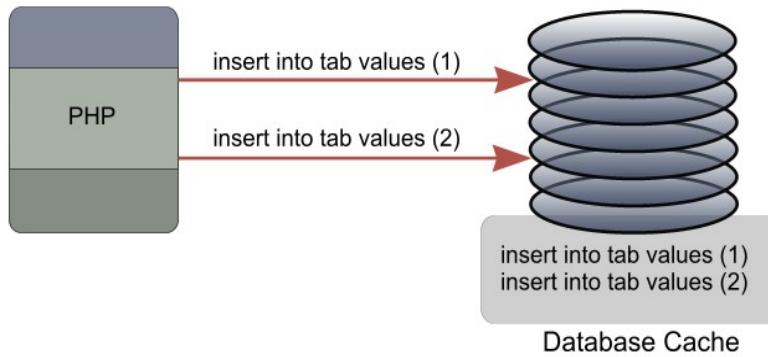


Figure 66: Not binding wastes database resources.

Binding is *highly* recommended. It can improve overall database throughput. Oracle is more likely to find the statement in its cache and be able to reuse the execution plan and context for that statement, even if someone else originally executed it.

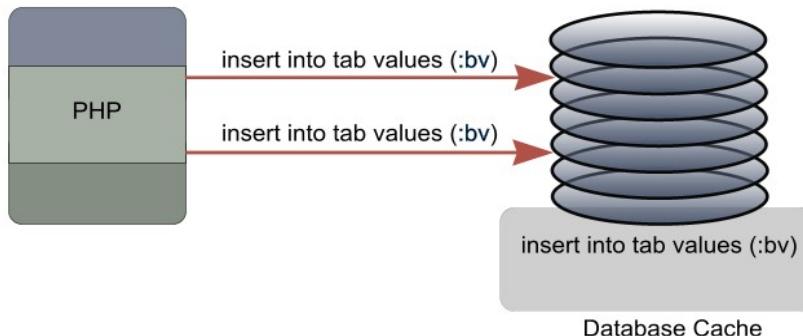


Figure 67: Binding improves performance and security.

Bind variables are also an important way to prevent SQL injection security attacks. SQL injection may occur when SQL statements are constructed from hard-coded text concatenated with user input:

```
$w = "userid = 1"; // emulate "user input"  
$s = oci_parse($c, "select * from mytable where $w");
```

If the user input is not carefully checked, then it may be possible for a malicious user to execute a SQL statement of their choice instead of the one you intended.

In Oracle, a bind variable is a colon-prefixed name in the SQL text. An [oci_bind_by_name\(\)](#) call tells Oracle which PHP variable to actually use when executing the statement.

Executing SQL Statements With OCI8

Script 30: *bindvar.php*

```
<?php  
  
$c = oci_connect("hr", "welcome", "localhost/XE");  
  
$s = oci_parse($c, "select last_name from employees where employee_id = :eidbv");  
$myeid = 101;  
oci_bind_by_name($s, ":eidbv", $myeid);  
oci_execute($s);  
$row = oci_fetch_array($s, OCI_ASSOC);  
echo "Last name is: ". $row['LAST_NAME'] ."  
\n";  
  
?>
```

The output is the last name of employee 101:

```
Last name is: Kochhar
```

There is no need to (and for efficiency you should not) re-parse the SQL statement if you just want to change the value of the bind variable. The following code would work when appended to the end of *bindvar.php*:

```
// No need to re-parse or re-bind  
$myeid = 102;  
oci_execute($s);  
$row = oci_fetch_array($s, OCI_ASSOC);  
echo "Last name is: ". $row['LAST_NAME'] ."  
\n";
```

Re-running *bindvar.php* now gives:

```
Last name is: Kochhar  
Last name is: De Haan
```

You can bind a single value with each [oci_bind_by_name\(\)](#) call. Multiple values can be bound with another function, [oci_bind_array_by_name\(\)](#), and passed to PL/SQL blocks. This is discussed in the chapter on PL/SQL.

Do not change the PHP type of `$myeid` after binding or between executions otherwise the internal representation won't be recognized by OCI8.

The syntax of [oci_bind_by_name\(\)](#) is:

```
$rc = oci_bind_by_name($statement, $bindvarname, $phpvariable, $length, $type)
```

The length and type are notionally optional. The default type is the string type, SQLT_CHR. Oracle will convert most basic types to or from this as needed. For example when binding a number you can omit the type parameter.

You will need to set a length when data is being returned from Oracle as an OUT bind. This is so PHP can allocate a buffer of the correct size. It is also recommended to pass the length if a query is being re-executed in a script with different IN bind values. For example when binding, pass the length of the largest potential string. Passing the length also avoids

Using Bind Variables in Prepared Statements

potential edge-case behavior differences if a script runs with multiple different Oracle character sets, or with different architectures.

Some very old PHP examples use an ampersand '&' with `oci_bind_by_name()` parameters. Do not do this. Since a general clean up of the overall call-by-reference implementation in PHP, this syntax has been deprecated and may cause problems.

A bind call tells Oracle which memory address to read data from. That address needs to contain valid data when `oci_execute()` is called. If the bind call is made in a different scope from the execute call there could be a problem. For example, if the bind is in a function and a function-local PHP variable is bound, then Oracle may read an invalid memory location if the execute occurs after the function has returned. This has an unpredictable outcome.

There is one case where you might decide not to use bind variables. When queries contain bind variables, the optimizer does not have any information about the value you may eventually use when the statement is executed. If your data is highly skewed, you might want to hard code values. But if the data is derived from user input be sure to sanitize it to avoid SQL injection security issues.

Finally, Oracle does not use question mark '?' for bind variable placeholders at all. OCI8 supports only named placeholders with a colon prefix. Some PHP database abstraction layers will simulate support for question marks by scanning your statements and replacing them with supported syntax.

Binding in a “for” Loop

There is a common problem with binding in a `foreach` loop with PHP OCI8:

```
$s = oci_parse($c, 'select *
    from departments
    where department_name = :dname and location_id = :loc');

$ba = array(':dname' => 'IT Support', ':loc' => 1700);
foreach ($ba as $key => $val) {
    oci_bind_by_name($s, $key, $val); // problem here
}
```

The problem here is that `$val` is local to the loop (and is reused). The SQL statement will not execute as expected. Changing the bind call in the loop to use `$ba[$key]` solves the problem:

Script 31: bindloop.php

```
<?php

$c = oci_connect("hr", "welcome", "localhost/XE");

$s = oci_parse($c, 'select *
    from departments
    where department_name = :dname and location_id = :loc');

$ba = array(':dname' => 'IT Support', ':loc' => 1700);
foreach ($ba as $key => $val) {
    oci_bind_by_name($s, $key, $ba[$key]);
}

oci_execute($s);
```

Executing SQL Statements With OCI8

```
while (($row = oci_fetch_array($s, OCI_ASSOC))) {
    foreach ($row as $item) {
        echo htmlentities($item) . " ";
    }
    echo "<br>\n";
}
?>
```

Binding With LIKE and REGEXP_LIKE Clauses

You can bind the value used in a pattern-matching SQL [LIKE](#) or [REGEXP_LIKE](#) clause:

Script 32: bindlike.php

```
<?php

$c = oci_connect("hr", "welcome", "localhost/XE");

$s = oci_parse($c,
    "select city, state_province from locations where city like :bv");
$city = 'South%';
oci_bind_by_name($s, ":bv", $city);
oci_execute($s);
oci_fetch_all($s, $res);

var_dump($res);

?>
```

This uses Oracle's traditional LIKE syntax, where '%' means match anything. An underscore in the pattern string '_' would match exactly one character.

The output from *bindlike.php* is cities and states where the city starts with 'South':

```
array(2) {
    ["CITY"]=>
    array(3) {
        [0]=>
        string(15) "South Brunswick"
        [1]=>
        string(19) "South San Francisco"
        [2]=>
        string(9) "Southlake"
    }
    ["STATE_PROVINCE"]=>
    array(3) {
        [0]=>
        string(10) "New Jersey"
        [1]=>
        string(10) "California"
        [2]=>
        string(5) "Texas"
    }
}
```

Using Bind Variables in Prepared Statements

```
}
```

Oracle also supports regular expression matching with functions like [REGEXP_LIKE](#), [REGEXP_INSTR](#), [REGEXP_SUBSTR](#), and [REGEXP_REPLACE](#).

In a query from PHP you might bind to [REGEXP_LIKE](#) using:

Script 33: bindregexp.php

```
<?php  
  
$c = oci_connect("hr", "welcome", "localhost/XE");  
  
$s = oci_parse($c, "select city from locations where regexp_like(city, :bv)");  
$city = '.*ing.*';  
oci_bind_by_name($s, ":bv", $city);  
oci_execute($s);  
oci_fetch_all($s, $res);  
  
var_dump($res);  
  
?>
```

This displays all the cities that contain the letters 'ing':

```
array(1) {  
  ["CITY"]=>  
  array(2) {  
    [0]=>  
      string(7) "Beijing"  
    [1]=>  
      string(9) "Singapore"  
  }  
}
```

Binding Multiple Values in an IN Clause

User data for a bind variable is always treated as pure data and never as part of the SQL statement. Because of this, trying to use a comma separated list of items in a single bind variable will be recognized by Oracle only as a single value, not as multiple values. The common use case is when allowing a web user to choose multiple options from a list and wanting to do a query on all values.

Hard coding multiple values in an [IN](#) clause in the SQL statement is the anti-example that should be avoided:

```
$s = oci_parse($c,  
              "select last_name from employees where employee_id in (101,102)");  
oci_execute($s);  
oci_fetch_all($s, $res);  
foreach ($res['LAST_NAME'] as $name) {  
  echo "Last name is: ". $name . "<br>\n";  
}
```

Executing SQL Statements With OCI8

This displays both surnames but it leads to the scaling and security issues that bind variables overcome.

The next code snippet shows the naïve equivalent using a single bind variable:

```
$s = oci_parse($c,
    "select last_name from employees where employee_id in (:eidbv)");
$myeids = "101,102";
oci_bind_by_name($s, ":EIDBV", $myeids);
oci_execute($s);
oci_fetch_all($s, $res);
```

The code gives the error *ORA-01722: invalid number* because the `$myeids` string is treated as a single value and is not recognized as a list of numbers.

The solution for a fixed, small number of values in an IN bind clause is to use individual bind variables. A `NULL` can be bound for any unknown values:

Script 34: bindinlist.php

```
<?php

$c = oci_connect("hr", "welcome", "localhost/XE");

$s = oci_parse($c,
    "select last_name from employees where employee_id in (:e1, :e2, :e3)");
$mye1 = 103;
$mye2 = 104;
$mye3 = NULL; // pretend we were not given this value
oci_bind_by_name($s, ":E1", $mye1);
oci_bind_by_name($s, ":E2", $mye2);
oci_bind_by_name($s, ":E3", $mye3);
oci_execute($s);
oci_fetch_all($s, $res);

foreach ($res['LAST_NAME'] as $name) {
    echo "Last name is: ". $name . "<br>\n";
}
?>
```

The output is:

```
Last name is: Ernst
Last name is: Hunold
```

If the number of values to be compared is big, you could dynamically build up the SQL statement, although try to avoid doing this where too many distinct SQL statements are created and executed:

Script 35: dynamicinlist.php

```
<?php

$args = array(100,105,111); // simulate a variable number of arguments
$c = oci_connect('hr', 'welcome', 'localhost/XE');
```

Using Bind Variables in Prepared Statements

```
// Create uniquely named bind variables in the IN list
$sql = 'select last_name from employees where employee_id in (';
$ac = count($args);
for ($i = 0; $i < $ac; ++$i) {
    $sql .= ":".$i.",";
}
$sql = rtrim($sql, ",");
$sql .= ')';

// Bind each variable
$s = oci_parse($c, $sql);
for ($i = 0; $i < $ac; ++$i) {
    oci_bind_by_name($s, $i, $args[$i]);
}

oci_execute($s);
while (($row = oci_fetch_array($s, OCI_ASSOC)) != false) {
    foreach ($row as $item)
        echo $item . "\n";
}
?>
```

For very large numbers of IN bind values, binding may not be possible and alternative solutions such as inserting the values into a global temporary tables and using a join might be required. Tom Kyte discusses the general problem and gives solutions for other cases in the March - April 2007 *Oracle Magazine*.

Using Bind Variables to Fetch Data

As well as what are called IN binds, which pass data into Oracle, there are also OUT binds that return values. These are mostly used to return values from PL/SQL procedures and functions. (See the chapter on using PL/SQL). If the PHP variable associated with an OUT bind does not exist, you need to specify the optional length parameter. Another case when the length should be specified is when returning numbers. By default in OCI8, numbers are converted to and from strings when they are bound. This means the length parameter should also be passed to `oci_bind_by_name()` when returning a number, otherwise digits may be truncated:

```
oci_bind_by_name($s, ":MB", $mb, 10);
```

There is also an optional fifth parameter, which is the data type. This mostly used for binding LOBs and result sets as shown in the chapter on LOBs. One micro-optimization when numbers are known to be integral, is to specify the data type as SQLT_INT. This avoids the type conversion cost:

```
oci_bind_by_name($s, ":MB", $mb, -1, SQLT_INT);
```

In this example, the length was set to -1 meaning use the native data size of an integer.

Executing SQL Statements With OCI8

Binding in an ORDER BY Clause

Some applications allow the user to choose the presentation order of results. Typically the number of variations for an [ORDER BY](#) clause are small and so having different statements executed for each condition is efficient:

```
switch ($v) {
    case 1:
        $ob = ' order by first_name';
        break;
    default:
        $ob = ' order by last_name';
        break;
}

$s = oci_parse($c, 'select first_name, last_name from employees' . $ob);
```

But if your tuning indicates that binding in a [ORDER BY](#) clause is necessary, and the columns are of the same type, you might be able to use a SQL [CASE](#) statement. However this might negatively impact SQL statement optimization:

```
$s = oci_parse($c, "select first_name, last_name
                    from employees
                    order by
                        case :ob
                            when 'FIRST_NAME' then first_name
                            else last_name
                        end");
$vs = "FIRST_NAME";
oci_bind_by_name($s, ':ob', $vs);
oci_execute($s);
```

Using ROWID Bind Variables

The pseudo-column [ROWID](#) uniquely identifies a row within a table. This example shows fetching a record, changing the data, and binding its [ROWID](#) in the [WHERE](#) clause of an [UPDATE](#) statement.

Script 36: rowid.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

// Fetch a record
$s = oci_parse($c,
    'select rowid, street_address
     from locations where location_id = :l_bv');
$locid = 3000; // location to fetch
oci_bind_by_name($s, ':l_bv', $locid);
oci_execute($s);
$row = oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_NULLS);

$rid = $row['ROWID'];
```

Using Bind Variables in Prepared Statements

```
$addr = $row['STREET_ADDRESS'];

// Change the address to upper case
$addr = strtoupper($addr);

// Save new value
$s = oci_parse($c,
    'update locations set street_address = :a_bv where rowid = :r_bv');
oci_bind_by_name($s, ':r_bv', $rid, -1, OCI_B_ROWID);
oci_bind_by_name($s, ':a_bv', $addr);
oci_execute($s);

?>
```

After running *rowid.php*, the address has been changed from

Mertenstrasse 921

to

MURTENSTRASSE 921

Improving Performance by Prefetching and Caching

PHP OCI8 can use several well known Oracle data access features to improve PHP application performance.

Tuning the Prefetch Size

You can tune PHP's overall query performance with the *php.ini* configuration parameter *oci8.default_prefetch*. This parameter sets the number of extra rows returned in a batch when an underlying fetch call across the network to the database occurs. Increasing the prefetch value can significantly improve performance of queries that return a large number of rows. It minimizes database server *round-trips* by returning as much data as possible each time an underlying network fetch request to the database is made.

All the prefetched rows are cached by the Oracle client libraries and there is no interface impact to PHP applications from altering the value. PHP functions like [oci_fetch_array\(\)](#) return one row to the user per call regardless of the prefetch size. Subsequent OCI8 fetches will consume the data from the cache until eventually another batch of records is needed to be retrieved from the database.

Executing SQL Statements With OCI8

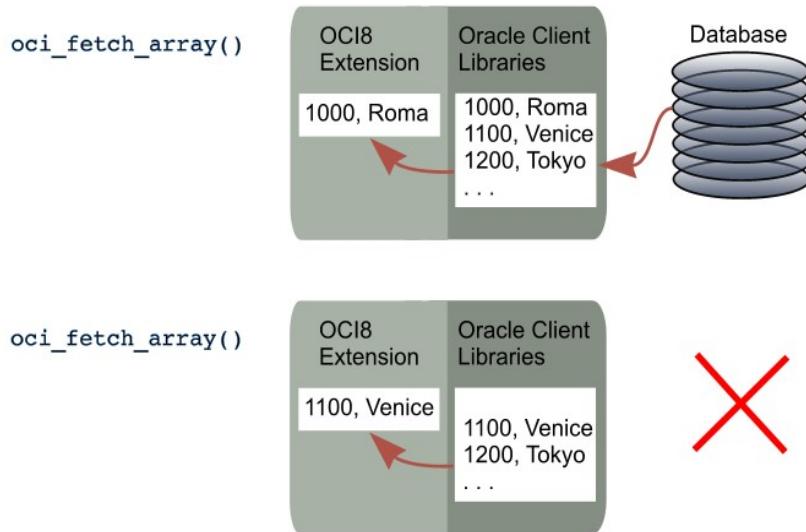


Figure 68: The first request to the database fetches multiple rows to the cache. Subsequent fetches read from the cache without requiring DB access.

You can also change the prefetch value at runtime with the `oci_set_prefetch()` function:

```
$s = oci_parse($c, "select city from locations");
oci_execute($s);
oci_set_prefetch($s, 200);
$row = oci_fetch_array($s, OCI_ASSOC);
```

For `oci_fetch_all()`, which returns all query rows to the PHP script in one call, PHP OCI8 internally fetches the records from the database in batches.

Testing will show the optimal prefetch size for your queries. There is no benefit using too large a prefetch value. Conversely, because Oracle dynamically allocates space, there is little to be gained by reducing the value too small.

The default prefetch value is 100. Prior to OCI8 1.3, the default was 10 and the OCI8 extension also capped the memory used by the prefetch buffer at $1024 * \text{oci8.default_prefetch}$ bytes.

From OCI8 1.4 the prefetch value can be set to 0. The prefetch value is the number of extra rows to be fetched on each underlying database access, so a prefetch value of 0 means only one row is returned each time across the network. When PHP is linked with Oracle 11gR2 libraries, prefetching will also occur for REF CURSOR fetches, see the chapter *Using PL/SQL with OCI8*. Setting the prefetch value to 0 may be useful only in one edge case, which involves REF CURSORS.

Prefetching works for nested cursor columns as long as both the Oracle client libraries linked with PHP and the database are both Oracle Database 11gR2. The default prefetch value is used or it can be overridden prior to executing the nested cursor resource. To change the inner prefetch value for the previous example `nestedcurl.php`, add an `oci_set_prefetch()` call. The outer fetch will still use the `oci8.default_prefetch` value:

Improving Performance by Prefetching and Caching

Script 37: nestedcur2.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$sql =
'select department_name,
       cursor(select first_name
              from employees
             where employees.department_id = departments.department_id) as nc
from departments
where department_id in (10, 20, 30)';

$s = oci_parse($c, $sql);
$r = oci_execute($s);
while (($row1 = oci_fetch_array($s, OCI_ASSOC)) != false) {
    echo "Department: " . $row1['DEPARTMENT_NAME'] . "<br>\n";
    $nc = $row1['NC'];           // treat as a statement resource
    oci_set_prefetch($nc, 20); // override default prefetch if desired
    oci_execute($nc);
    while (($row2 = oci_fetch_array($nc, OCI_ASSOC+OCI_RETURN_NULLS)) != false) {
        echo $row2['FIRST_NAME'] . "<br>\n";
    }
    oci_free_statement($nc);
    echo "<br>\n";
}
?>
```

Prefetching is not used when queries contain LONG or LOB columns. One partial solution is to structure the application so that queries only need to fetch and display part of the LOB data, which can be done efficiently. For example a query fetching a column MYLOBCOL could be changed to fetch `dbms_lob.substr(mylobcol, 1000, 1)` instead of returning all of the data upfront. A drill-down link on the application would then fully query just one selected LOB.

Tuning the Statement Cache Size

Performance is improved with Oracle's "client" (that is, PHP OCI8) statement caching feature. In the PHP extension the default statement cache size is 20 statements. You can change the size with the `php.ini` directive `oci8.statement_cache_size`. The recommendation is to use the number of statements in the application's working set of SQL as the value. Caching can be disabled by setting the size to 0.

The client-side statement cache is in addition to the standard database statement cache. The client statement cache means even the text of the statement does not need to be transmitted to the database more than once, reducing network traffic and database server load. The database can directly look up the statement context in its cache without even having to hash the statement. In turn, the database does not need to transfer meta-data about the statement back to PHP.

Executing SQL Statements With OCI8

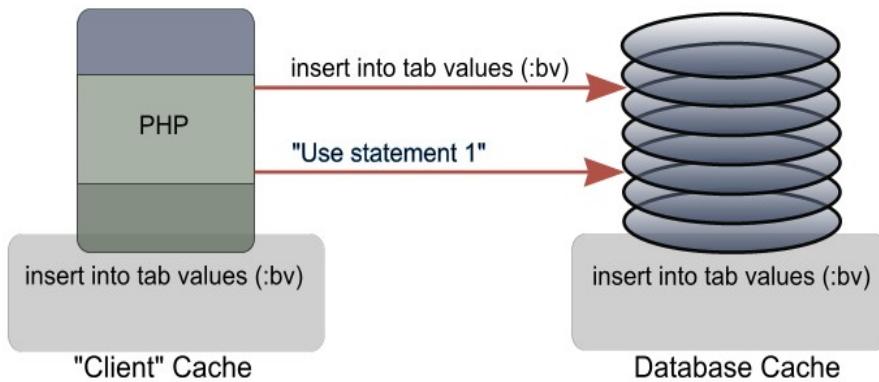


Figure 69: The second time a statement is issued, the statement text is not sent to the database.

The cache is per-Oracle session so this feature is more useful when persistent connections are used. Like many tuning options, there is a time/memory trade-off when tweaking this parameter. The statement cache also means slightly more load is put on the PHP host.

To tune the statement cache size, monitor general web server load and the database statistic "bytes sent via SQL*Net to client". This can be seen, for example, in Oracle Automatic Workload Repository (AWR) reports. When caching is effective, the statistic should show an improvement. Adjust the value of `oci8.statement_cache_size` to your satisfaction.

If the schema changes, such as during the initial design and prototyping phases of an application, you may experience errors such as *ORA-01007 variable not in select list* or *ORA-00932: inconsistent datatypes* because the statement cache can become out of sync. The way to prevent this is to turn off statement caching and restart Apache.

OCI8 will clear the cache if a statement returns a database error.

Using the Server and Client Query Result Caches

Oracle Database 11g introduces "server-side" and "client-side" result caches. These store the final result of queries, reducing the work needed when queries are re-executed.

The database cache is enabled with the `RESULT_CACHE_MODE` database parameter, which has several settings. With Oracle 11gR2, tables which are candidates for query results being cached can be nominated with `CREATE TABLE`, or with an `ALTER TABLE` command :

```
SQL> alter table employees result_cache (mode force);
```

No PHP changes are required in Oracle 11gR2. Applications will immediately benefit from caches when the `EMPLOYEES` table is queried. With Oracle 11gR1, you do need to add a hint to the query:

```
$s = oci_parse($c, "select /*+ result_cache */ * from employees");
```

The client cache is ideal for small queries from infrequently modified tables, such as look-up tables. It can reduce PHP statement processing time and significantly reduce database CPU usage, allowing the database to handle more PHP processes and users. The client-side cache is per PHP process.

Improving Performance by Prefetching and Caching

A key feature is that Oracle automatically handles cache entry invalidation when a data change invalidates the stored results. Oracle will check the client cache validity each time any *round trip* to the database occurs. If no round trip has happened within a configurable “lag” time, the client cache is assumed stale and the next query will go to the database for processing. The cache will be refreshed at this time.

The *Oracle Call Interface Programmer's Guide, 11g Release 2 (11.2)* contains the best description of the feature and has more about when to use it and how to manage it.

Client Query Result Caching (CRC) is best suited for small lookup tables, so commonly the query statement will be static and won't involve bind variables. If queries have string binds, such as the default bind type, PHP queries won't be able to use CRC. However CRC is used when binding numbers with SQLT_INT like:

```
oci_bind_by_name($s, ":bv", $bv, -1, SQLT_INT);
```

To demonstrate client caching, the database parameter CLIENT_RESULT_CACHE_SIZE can be set to a non zero value and the Oracle database restarted:

```
$ sqlplus / as sysdba
SQL> alter system set client_result_cache_size=64M scope=spfile;
SQL> startup force
```

Because the EMPLOYEES table was altered above to always be a candidate for caching, the PHP code needs no magic to take advantage of CRC in Oracle 11gR2:

Script 38: crc.php

```
<?php

$c = oci_pconnect('hr', 'welcome', 'localhost/orcl');

for ($i = 0; $i < 1000; ++$i) {
    $s = oci_parse($c, "select * from employees where rownum < 2");
    oci_execute($s);
    oci_fetch_all($s, $res);
}

?>
```

However, if PHP is linked with Oracle 11.1 client libraries, change the query to use a hint:

```
select /*+ result_cache */ * from employees where rownum < 2
```

Also, prior to Oracle 11.2.0.2 make sure not to auto commit. Instead do:

```
oci_execute($s, OCI_NO_AUTO_COMMIT);
```

Before executing *crc.php*, run this query in the SQL*Plus session:

```
SQL> select parse_calls, executions, sql_text
  2  from v$sql
  3  where sql_text like '%employees%';

PARSE_CALLS EXECUTIONS SQL_TEXT
-----
          1      1 select parse_calls, executions, sql_text from v$sql
```

Executing SQL Statements With OCI8

```
where sql_text like '%employees%'
```

This shows the database being accessed when the query is executed. Initially it shows just the monitoring query itself.

In another terminal window, run *crc.php* from the command line or run it in a browser – it doesn't display any results.

```
$ php crc.php
```

Re-running the monitoring query shows that during the 1000 loop iterations, the database executed the PHP query just twice, once for the initial execution and the second time by a subsequent cache validation check:

```
PARSE_CALLS EXECUTIONS SQL_TEXT
-----
2          2 select * from employees where rownum < 2
2          2 select parse_calls, executions, sql_text from v$sql
                  where sql_text like '%employees%'
```

This means that 998 of the times the statement was performed, the client cache was used for the results, with no database access required.

Now edit *crc.php* and add `/*+ no_result_cache */` to the SQL statement:

```
$s = oci_parse($c,
    "select /*+ no_result_cache */ * from employees where rownum < 2");
```

Re-run the script:

```
$ php crc.php
```

The monitoring query now shows the modified query was executed 1000 times, or once per loop iteration. This means the client query result cache was not used and each iteration had to be sent to the database and processed:

```
PARSE_CALLS EXECUTIONS SQL_TEXT
-----
2          2 select * from employees where rownum < 2
3          3 select parse_calls, executions, sql_text from v$sql
                  where sql_text like '%employees%'

2          1000 select /*+ no_result_cache */ * from employees where
                  rownum < 2
```

A dedicated view `CLIENT_RESULT_CACHE_STATS$` is periodically updated with statistics on client caching. For short tests like this example where the process quickly runs and terminates, it may not give meaningful results and `V$SQL` can be more useful.

Monitoring OCI8 SQL Statements

Monitoring is the first step to tuning applications. The database-centric approach is to first tune the application, next tune the SQL, and finally tune the database.

OCI8 1.4 (PHP 5.3) allows some meta data to be passed to Oracle which can help assess the impact of your application on the system and identify performance issues. The chapter

Connecting to Oracle using OCI8 showed using `oci_set_client_identifier()` for tracing and monitoring. Below are some other OCI8 features that can also be useful.

OCI8 Driver Identification

OCI8 1.4 automatically sets the DRIVER_NAME attribute of V\$SESSION_CONNECT_INFO (when using Oracle 11gR2 client libraries). This allows administrators to monitor which applications connect to the database. The attribute value set is "PHP OCI8" followed by the OCI8 version number. You can test by running a script:

```
$c = oci_connect('hr', 'welcome', 'localhost/XE');
sleep(20); // let the script run, so the trace information can be viewed
```

In SQL*Plus you can query V\$SESSION_CONNECT_INFO:

```
select unique sid, client_driver, authentication_type
from v$session_connect_info
where client_driver like 'PHP OCI8%'
order by sid;
```

This shows something like:

SID	CLIENT_DR
22	PHP OCI8

Note the view truncates the driver name attribute to nine characters in current releases of Oracle. The "UNIQUE" clause is used because V\$SESSION_CONNECT_INFO view is not normalized: each Oracle Net banner for the session appears in one row. Another column in the V\$SESSION_CONNECT_INFO view is CLIENT_CHARSET. This could be used to verify all PHP applications are using the correct character set. At disconnect time, entries used for job cleanup may be shown in this view. It can be useful to join V\$SESSION_CONNECT_INFO with V\$SESSION.

Setting Application Information in PHP OCI8

OCI8 1.4 introduced functions to set user defined attribute values for tracing:

`oci_set_module_name()`, `oci_set_action()` and `oci_set_client_info()`.

The values are visible in various data dictionary views in the database, such as V\$SESSION. Tools like Oracle Enterprise Manager allow searching and grouping on the values. You can also architect your code to use the values. For example they can be viewed and tested in SQL queries using the SYS_CONTEXT() function.

The attributes are sent to the database with the next roundtrip, which occurs most commonly when a SQL or PL/SQL statement is executed. This is the most efficient way to set the values. If Client Query Result Caching is enabled, a roundtrip may not happen. When using persistent connections, the values may not be cleared at the end of the script.

One use for the MODULE and ACTION attributes is to track the SQL statements that each part of your application executes:

Script 39: appinfo.php

```
<?php
```

Executing SQL Statements With OCI8

```
$c = oci_connect('hr', 'welcome', 'localhost/XE');

oci_set_client_info($c, 'My Application Version 2');
oci_set_module_name($c, 'Home Page');
oci_set_action($c, 'Friend Lookup');

// Do some action which touches the database
// The three attribute values will be "piggy backed"
// and sent with the SQL to the database
$s = oci_parse($c, 'select * from dual');
oci_execute($s);
oci_fetch_all($s, $res);

// sleep so the trace information can be viewed in SQL*Plus
sleep(20);

?>
```

The first time the script is run and the SQL statement is executed, the MODULE and ACTION are recorded along with the SQL statement in the V\$SQLAREA view:

```
select sql_text, module, action
from v$sqlarea
where module = 'Home Page';
```

This produces:

SQL_TEXT	MODULE	ACTION
select * from dual	Home Page	Friend Lookup

This can help narrow down where problematic statements are coded and lets performance issues be resolved.

With older versions of OCI8, use PL/SQL to set the attributes. However this method incurs a roundtrip which will slow down the application:

```
$s = oci_parse($c,
  "begin
    dbms_application_info.set_client_info('My Application Version 2');
    dbms_application_info.set_module(
      'Home Page',          // Module
      'Friend Lookup');    // Action
  end;");
oci_execute($s);
```

LIMIT, Auto-Increment, Last Insert ID and Multiple Inserts

If you are migrating from another database to Oracle, there are several common operations you might have used that are handled differently in Oracle.

Limiting Rows and Creating Paged Datasets

Oracle 11g SQL does not have a `LIMIT` keyword. There are several alternative ways to limit the number of rows returned in OCI8.

The canonical paging query for Oracle8i onwards is given on <http://asktom.oracle.com>:

```
select *
from ( select a.* , rownum as rnum
      from (YOUR_QUERY_Goes_Here -- including the order by) a
      where rownum <= MAX_ROW )
where rnum >= MIN_ROW
```

Here, `MIN_ROW` is the row number of first row and `MAX_ROW` is the row number of the last row to return. In PHP you might do this:

Script 40: limit.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$mystmt = "select city from locations order by city";
$minrow = 4; // row number of first row to return
$maxrow = 8; // row number of last row to return

$pagesql = "select *
            from ( select a.* , rownum as rnum
                  from ( $mystmt ) a
                  where rownum <= :maxrow)
            where rnum >= :minrow";

$s = oci_parse($c, $pagesql);
oci_bind_by_name($s, ":maxrow", $maxrow);
oci_bind_by_name($s, ":minrow", $minrow);
oci_execute($s);
oci_fetch_all($s, $res);

var_dump($res);

?>
```

Note that `$mystmt` itself is not bound. Bind data is not treated as code, so you cannot bind the text of the statement and expect it to be executed. Beware of SQL injection security issues if SQL statements are constructed or concatenated.

The output of the script is:

```
array(2) {
  ["CITY"]=>
  array(5) {
    [0]=>
    string(6) "Geneva"
    [1]=>
    string(9) "Hiroshima"
    [2]=>
```

Executing SQL Statements With OCI8

```
    string(6) "London"
[3]=>
string(11) "Mexico City"
[4]=>
    string(6) "Munich"
}
["RNUM"]=>
array(5) {
    [0]=>
    string(1) "4"
    [1]=>
    string(1) "5"
    [2]=>
    string(1) "6"
    [3]=>
    string(1) "7"
    [4]=>
    string(1) "8"
}
}
```

An alternative and preferred query syntax uses Oracle's analytic [ROW_NUMBER\(\)](#) function. The query:

```
select last_name, row_number() over (order by last_name) as myr
from employees
```

returns two columns identifying the last name with its row number:

LAST_NAME	MYR
Abel	1
Ande	2
Atkinson	3
...	

By turning this into a subquery and using a [WHERE](#) condition any range of names can be queried. For example to get the 11th to 20th names the query is:

```
select last_name FROM
  (select last_name,
          row_number() over (order by last_name) as myr
   from employees)
 where myr between 11 and 20
```

In SQL*Plus the output is:

LAST_NAME
Bissot
Bloom
Bull
Cabrio
Cambrault
Cambrault
Chen

LIMIT, Auto-Increment, Last Insert ID and Multiple Inserts

```
Chung
Colmenares
Davies
```

As an anti-example, another way to limit the number of rows returned involves the [oci_fetch_all\(\)](#) function, which has optional arguments to specify a range of results to fetch. This is implemented by the extension, not by Oracle's native functionality. All rows preceding those you want still have to be fetched from the database. These unused rows are discarded, which is wasteful of network and processing resources:

```
$firstrow = 3;
$numrows = 5;
oci_execute($s);
oci_fetch_all($s, $res, $firstrow, $numrows);
var_dump($res);
```

It is more efficient to let Oracle do the row selection and only return the exact number of rows required.

Auto-Increment Columns

Auto-increment columns in Oracle can be created using a sequence generator and a trigger.

Sequence generators are defined in the database and return Oracle numbers. Sequence numbers are generated independently of tables. Therefore, the same sequence generator can be used for more than one table or anywhere that you want to use a unique number. Sequence generation is useful to generate unique primary keys for your data and to coordinate keys across multiple tables. You can get a new value from a sequence generator using the [NEXTVAL](#) operator in a SQL statement. This gives the next available number and increments the generator. The similar [CURRVAL](#) operator returns the current value of a sequence without incrementing the generator.

A trigger is a PL/SQL procedure that is automatically invoked at a predetermined point. In this example a trigger is invoked whenever an insert is made to a table.

In SQL*Plus an auto increment column MYID can be created like:

Script 41: autoinc.sql

```
create sequence myseq;

create table mytable (myid number primary key, mydata varchar2(20));

create trigger mytrigger
before insert on mytable for each row
begin
  :new.myid := myseq.nextval;
end;
/
```

Prior to Oracle Database 11g you need to fetch the value using a [SELECT](#) like:

```
select myseq.nextval into :new.myid from dual;
```

In PHP insert two rows:

Executing SQL Statements With OCI8

Script 42: *autoinc.php*

```
<?php  
$c = oci_connect("hr", "welcome", "localhost/XE");  
  
$s = oci_parse($c, "insert into mytable (mydata) values ('Hello')");  
oci_execute($s);  
$s = oci_parse($c, "insert into mytable (mydata) values ('Bye')");  
oci_execute($s);  
?>
```

Querying the table in SQL*Plus shows the MYID values were automatically inserted and incremented:

```
SQL> select * from mytable;
```

MYID	MYDATA
1	Hello
2	Bye

The identifier numbers will be unique and increasing but may not be consecutive. For example if someone rolls back an insert, a sequence number can be “lost”.

Getting the Last Insert ID

OCI8 does not have an explicit “`insert_id`” function. Instead, use a `RETURN INTO` clause and a bind variable. Using the table and trigger created above in *autoinc.sql*, the insert would be:

Script 43: *insertid.php*

```
<?php  
$c = oci_connect("hr", "welcome", "localhost/XE");  
  
$s = oci_parse($c,  
    "insert into mytable (mydata) values ('Hello') return myid into :id");  
oci_bind_by_name($s, ":id", $id, 20, SQLT_INT);  
oci_execute($s);  
echo "Data inserted with id: $id\n";  
?>
```

This returns the value of the MYID column for the new row into the PHP variable `$id`. The output, assuming the two inserts of *autoinc.php* were previously executed, is:

```
Data inserted with id: 3
```

You could similarly return the ROWID of the new row into a descriptor:

```
$rid = oci_new_descriptor($c, OCI_D_ROWID);  
$s = oci_parse($c,  
    "insert into mytable (mydata) values ('Hello') return rowid into :rid");
```

LIMIT, Auto-Increment, Last Insert ID and Multiple Inserts

```
oci_bind_by_name($s, ":rid", $rid, -1, OCI_B_ROWID);
oci_execute($s);
```

Inserting Multiple Values

Some databases allow multiple values to be inserted in one call similar to:

```
insert into mytab (col_1, col_2) values (1,2),(3,4);
```

In Oracle, you can use `oci_bind_array_by_name()` (see the chapter on PL/SQL) or try the INSERT ALL statement:

```
insert all
  into mytab (col_1, col_2) values (1, 2)
  into mytab (col_1, col_2) values (3, 4)
  select 1 from dual;
```

Don't forget to bind data values unlike this simple example shows.

Benchmark your implementation in an environment representative of your production system. If you have a large number of values, or have to build up the statement text by string concatenation in a loop, or have a fast network, then multiple INSERT statements might be faster.

If you are migrating to Oracle, watch out for differences in transactional behavior, since failure will cause the whole statement to be rolled back unlike some other databases.

One related use of INSERT ALL is to insert into multiple different tables in the one statement.

Exploring Oracle

Explore the SQL and PL/SQL languages. Make maximum reuse of functionality that already exists. Tom Kyte's popular site, <http://asktom.oracle.com>, has a lot of useful information.

Oracle's general guideline is to let the database manage data and to transfer the minimum amount across the network. Avoid shipping data from the database to PHP for unnecessary post processing. Data is a core asset of your business. It should be treated consistently across your applications. Keeping a thin interface between your application layer and the database is also good programming practice.

There are many more useful SQL and Database features than those described in this book. They are left for you to explore. A few are mentioned below.

Case Insensitive Data Matching in Queries

If you want to do queries that sort and match data in a case insensitive manner, change the session attributes NLS_SORT and NLS_COMP first, either with environment variables, or per session:

```
alter session set nls_sort = binary_ci;
alter session set nls_comp = linguistic;
```

Executing SQL Statements With OCI8

Analytic Functions in SQL

Oracle's Analytic functions are a useful tool to compute aggregate values based on a group of rows. Here is an example of a correlation. [CORR\(\)](#) returns the coefficient of correlation of a set of number pairs:

```
select max_extents, corr(max_trans, initial_extent)
from all_tables
group by max_extents;
```

You must be connected as the SYSTEM user in this particular example, since it depends on table data not available to the HR schema.

Other analytic functions allow you to get basic information like standard deviations or do tasks such as ranking (for Top-N or Bottom-N queries) or do linear regressions.

External Tables

Oracle External tables allow data outside the database to be accessed as if it were in a database table. You can read and write data. External tables can be used to load and unload data. In Oracle Database 11g querying an external table can invoke an external program which can generate the data.

USING PL/SQL WITH OCI8

PL/SQL is Oracle's procedural language extension to SQL. It is a database-side language that is easy-to-use. PL/SQL enables you to mix SQL statements with procedural constructs. PHP can call PL/SQL blocks to make use of advanced database functionality, and can use it to efficiently insert and fetch data. As with SQL, the PL/SQL language gives applications easy access to Oracle's better date and number handling, for example to make sure your financial data is not affected by PHP's floating point semantics. You can create stored procedures, functions and packages so your business logic is reusable in all your applications. PL/SQL has an inbuilt native compiler, optimizing and debugging features, and a 'wrap' code obfuscation facility to protect the intellectual property in applications.

PL/SQL Overview

A PL/SQL block has three basic parts:

- An optional declarative part `DECLARE`
- An executable part `BEGIN ... END`
- An optional exception-handling part `EXCEPTION`

An example PL/SQL block is:

```
declare
    sal_1 pls_integer;
begin
    select salary into sal_1 from employees where employee_id = 191;
    dbms_output.put_line('Salary is ' || sal_1);
exception
    when no_data_found then
        dbms_output.put_line('No results returned');
end;
```

You can run this in many tools, including PHP. In Oracle's SQL*Plus it is run by entering the text at the prompt and finishing with a single slash (`/`) to tell SQL*Plus to execute the code. If you turn on `SET SERVEROUTPUT` beforehand, then SQL*Plus will display the output messages after execution:

```
SQL> set serveroutput on
SQL> declare
  2      sal_1 pls_integer;
  3  begin
  4      select salary into sal_1
  5      from employees
  6      where employee_id = 191;
  7      dbms_output.put_line('Salary is ' || sal_1);
  8  exception
  9      when no_data_found then
 10          dbms_output.put_line('No results returned');
```

Using PL/SQL With OCI8

```
11  end;
12 /
Salary is 2500
```

Other tools have different ways to indicate the end of the statements and how to switch server output on.

If a PHP application performs several SQL statements at one time, it can be efficient to bundle the statements together in a single PL/SQL procedure. Instead of executing multiple SQL statements, PHP only needs to execute one PL/SQL call. This reduces the number of round trips between PHP and the database, and can improve overall performance.

There are a number of pre-supplied PL/SQL packages to make application development easier. Packages exist for full text indexing, queuing, change notification, sending emails, job scheduling and TCP access, just to name a few. When deciding whether to write PHP on the mid-tier or PL/SQL in the server, consider your skill level in the languages, the cost of data transfer across the network and the re-usability of the code. If you write in PL/SQL, all your Oracle applications in any tool or client language can reuse the functionality.

Blocks, Procedures, Packages and Triggers

PL/SQL code can be categorized as one of the following:

- Anonymous blocks
- Stored procedures or functions
- Packages
- Triggers

Anonymous Blocks

An anonymous block is a PL/SQL block included in your application that is not named or stored in the database. The previous example is an anonymous block. Because these blocks are not stored in the database, they are generally for one-time use in a SQL script, or for simple code dynamically submitted to the Oracle server.

Stored Procedures and Functions

A stored procedure is a PL/SQL block that Oracle stores in the database. They can be called by name from an application. Functions are similar but also return a value when executed.

Procedures and functions can be used from other procedures or functions. They can be enabled and disabled to prevent them being used. They may also have an invalid state, if anything they reference is not available. They can be created individually, or be part of a package.

When you create a stored procedure or function, Oracle stores its parsed representation in the database for efficient reuse. Procedures can be created in SQL*Plus like:

```
SQL> create table mytab (mydata varchar2(40), myid number);

SQL> create or replace procedure
 2    myproc(d_p in varchar2, i_p in number) as
 3 begin
```

```
4   insert into mytab (mydata, myid) values (d_p, i_p);
5 end;
6 /
```

The procedure is only created, not run. Programs like PHP can run it later.

PL/SQL functions are created in a similar way using the [CREATE OR REPLACE FUNCTION](#) command.

If you have creation errors, use the SQL*Plus [SHOW ERRORS](#) command to display any messages. For example, creating a procedure that references an invalid table causes an error:

```
SQL> create or replace procedure
  2      myproc(d_p in varchar2, i_p in number) as
  3  begin
  4      insert into yourtab (mydata, myid) values (d_p, i_p);
  5  end;
  6 /
```

Warning: Procedure created with compilation errors.

```
SQL> show errors
Errors for PROCEDURE MYPROC:

LINE/COL  ERROR
-----  
4/3      PL/SQl: SQL Statement ignored
4/15      PL/SQl: ORA-00942: table or view does not exist
```

If you are running SQL script files in SQL*Plus, it is helpful to turn [SET ECHO ON](#) to see the line numbers.

See later below for handling PL/SQL errors in PHP.

Packages

Typically, stored procedures and functions are encapsulated into packages. This helps minimizes recompilation of dependent objects. The package specification defines the signatures of the functions and procedures. If that definition is unchanged, code that invokes it will not need to be recompiled even if the implementation of the package body changes.

Script 44: toyshop.sql

```
create table toys (id number, name varchar2(40));
insert into toys (id, name) values (1, 'bicycle');
commit;

create or replace package toyshop as
  function find_toy(id_p in number) return varchar2;
  procedure add_toy(id_p in number, name_p in varchar2);
  procedure find_toy_proc(id_p in number, name_p out varchar2);
end toyshop;
/

create or replace package body toyshop as
  function find_toy(id_p in number) return varchar2 as
    name_1 varchar2(20);
```

Using PL/SQL With OCI8

```
begin
  select name into name_l from toys where id = id_p;
  return name_l;
end;

procedure add_toy(id_p in number, name_p in varchar2) as
begin
  insert into toys (id, name) values(id_p, name_p);
end;

procedure find_toy_proc(id_p in number, name_p out varchar2) as
begin
  select name into name_p from toys where id = id_p;
end;
end toyshop;
/
```

Triggers

A database trigger is a stored procedure associated with a database table, view, or event. The trigger can be called after the event, to record it, or take some follow-up action. A trigger can also be called before an event, to prevent erroneous operations or fix new data so that it conforms to business rules. Triggers were shown earlier as a way to optimize setting date formats (see *Do Not Set the Date or Numeric Format Unnecessarily* in the chapter on connecting) and as a way of creating auto-increment columns (see *Auto-Increment Columns* in the previous chapter).

Creating PL/SQL Stored Procedures in PHP

Procedures, functions and triggers can be created using PHP. For example, to create a procedure BIKE_CREATE the code is:

Script 45: bikecreate.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$plsql = <<<'EOS'
create or replace procedure bike_create(type_p in varchar2) as
begin
  insert into bicycles (style) values (type_p);
end;
EOS; // this must be at the start of the line without leading whitespace

$s = oci_parse($c, $plsql);
$r = oci_execute($s);
if ($r) {
  echo 'Procedure created';
}

?>
```

Note the last character of the PL/SQL statement is a semi-colon (after the PL/SQL keyword `end`), which is different to the way SQL statements are terminated in Oracle.

The example shows a PHP NOWDOC containing the statement. If you are on Windows, make sure you avoid the end-of-line terminator issue mentioned below.

Similar to the earlier performance advice on creating tables, avoid creating packages and procedures at runtime in an application. Pre-create them as part of application installation.

End of Line Terminators in PL/SQL With Windows PHP

With older versions of Oracle on Windows, multi-line PL/SQL blocks won't run if the line terminators are incorrect. The problem happens when the end of line characters in a multi-line PL/SQL string are Windows carriage-return line-feeds:

```
$plsql = "create or replace procedure
          myproc(d_p in varchar2, i_p in number) as
          begin
              insert into mytab(mydata, myid) values (d_p, i_p);
          end;";
```

The typical error is *ORA-24344: success with compilation error*.

If the `showcompilationerrors()` function, shown later, is used, additional Oracle messages will show the error *PLS-00103: Encountered the symbol "" when expecting one of the following*. This error, which may have the symbol ";" or a seemingly empty token representing the unexpected end-of-line syntax, is followed by a list of keywords or tokens the PL/SQL parser was expecting.

Use one of these solutions to fix the problem:

- Write the PL/SQL code on a single line:

```
$plsql = "create or replace procedure myproc . . . end;";
```

- Use PHP string concatenation with appropriate white space padding between string tokens:

```
$plsql = "create or replace procedure "
        . "myproc(d_p in varchar2, i_p in number) as "
        . "begin "
        . "insert into mytab(mydata, myid) values (d_p, i_p); "
        . "end;";
```

- Convert the file to use UNIX-style line-feeds with a conversion utility or editor.

Calling PL/SQL Code

Calling PL/SQL Procedures in PHP

To invoke a PL/SQL procedure from PHP, use `BEGIN` and `END` to create an anonymous block:

Script 46: anonplsql.php

```
<?php
```

Using PL/SQL With OCI8

```
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "begin toyshop.add_toy(2, 'ball'); end;");
oci_execute($s);

?>
```

The block contains a single procedure call, but you could include any number of other PL/SQL statements.

You can also use the SQL `CALL` statement like:

Script 47: callplsql.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "call toyshop.add_toy(3, 'paddling pool')");
oci_execute($s);

?>
```

The `call` command is actually a SQL command and does not have the trailing semi-colon needed for PL/SQL blocks.

Binding Parameters in PL/SQL Procedure Calls

PL/SQL procedure and function arguments can be marked IN, OUT or IN OUT depending on whether data is being passed into or out of PL/SQL. Single value parameters can be bound in PHP with `oci_bind_by_name()`. In the *toyshop.sql* example, the `find_toy_proc()` parameters were IN and OUT. The code could be:

```
$s = oci_parse($c, "begin toyshop.find_toy_proc(:id, :name); end;");
$id = 1;
oci_bind_by_name($s, ":id", $id);
oci_bind_by_name($s, ":name", $name, 40);
oci_execute($s);
echo "Name is: ".$name;
```

The bind call specifies that 40 bytes should be allocated to hold the retrieved toy name. For OUT and IN OUT parameters, make sure the length is specified in the bind call. As mentioned in the previous chapter, specifying the length for IN binds is often a good idea too, if the one statement is executed multiple times in a loop.

Calling PL/SQL Functions in PHP

Calling a PL/SQL function requires a bind variable for the return value. Using the function `find_toy()` created previously in *toyshop.sql*:

Script 48: plsqlfunc.php

```
<?php
```

```
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "begin :name := toyshop.find_toy(1); end;");
oci_bind_by_name($s, ':name', $name, 40);
oci_execute($s);
echo "Name is: " . $name;

?>
```

The `:=` token is the assignment operator in PL/SQL. Here it assigns the return value of the function to the bind variable. The bind call specifies that 40 bytes should be allocated to hold the result. The script output is:

```
Name is: bicycle
```

Binding Unsupported PL/SQL Types

Some PL/SQL data types are internal to PL/SQL and cannot be returned through the C layer used by the OCI8 extension. In these cases some extra PL/SQL code that maps the type to a form usable in PHP is needed. For example, the PL/SQL user function `is_valid()` returns the internal Oracle type BOOLEAN:

Script 49: isvalid.sql

```
create or replace function is_valid(p_uid in number) return boolean as
begin
  if (p_uid < 10) then
    return true;
  else
    return false;
  end if;
end;
/
```

The anti-example PHP code to call this is:

Script 50: isvalid.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$sql = "begin :r := is_valid(:userid); end;"; // will fail
$s = oci_parse($c, $sql);
oci_bind_by_name($s, ':r', $r, 40);
$userid = 4;
oci_bind_by_name($s, ':userid', $userid);
oci_execute($s);
echo "Result is " . ($r ? "true" : "false") . "\n";

?>
```

Using PL/SQL With OCI8

The expectation is that user id 4 is less than 10 and so the result will display as *true*. However, because of the use of the internal PL/SQL type, this script actually gives the error *PLS-00382: expression is of wrong type*. The solution is to change the anonymous PL/SQL statement to make it evaluate the `is_valid()` return value and propagate a type that can be bound in OCI8:

```
$sql = "begin
    if (is_valid(:userid) = true) then
        :r := 1;
    else
        :r := 0;
    end if;
end;";
```

The output of the script is now:

```
Result is true
```

Array Binding and PL/SQL Bulk Processing

OCI8 1.2 (PHP 5.1.2) introduced a function, `oci_bind_array_by_name()`. Used with a PL/SQL procedure, this can be very efficient for insertion or retrieval, requiring just a single `oci_execute()` to transfer multiple values. The following example, *arraybind.sql*, creates a PL/SQL package with two procedures. The first, `myinsproc()`, will be passed a PHP array to insert. It uses Oracle's "bulk" `FORALL` statement for fast insertion. The second procedure, `myselproc()`, selects back from the table using the `BULK COLLECT` clause and returns the array as the OUT parameter `p_arr`. The `p_count` parameter is used to make sure PL/SQL does not try to return more values than the PHP array can handle.

Script 51: arraybind.sql

```
drop table mytab;
create table mytab(name varchar2(20));

create or replace package mypkg as
    type arrtype is table of varchar2(20) index by pls_integer;
    procedure myinsproc(p_arr in arrtype);
    procedure myselproc(p_arr out arrtype, p_count in number);
end mypkg;
/
show errors

create or replace package body mypkg as
    procedure myinsproc(p_arr in arrtype) is
    begin
        forall i in indices of p_arr
            insert into mytab values (p_arr(i));
    end myinsproc;

    procedure myselproc(p_arr out arrtype, p_count in number) is
    begin
        select name bulk collect into p_arr from mytab where rownum <= p_count;
```

```

    end myselproc;
end mypkg;
/
show errors

```

To insert a PHP array `$a` into MYTAB, use:

Script 52: arrayinsert.php

```

<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$a = array('abc', 'def', 'ghi', 'jkl');

$s = oci_parse($c, "begin mypkg.myinsproc(:a); end;");
oci_bind_array_by_name($s, ":a", $a, count($a), -1, SQLT_CHR);
oci_execute($s);

?>

```

The `oci_bind_array_by_name()` function is similar to `oci_bind_by_name()`. As well as the upper data length, it has an extra parameter giving the number of elements in the array. In this example, the number of elements inserted is `count($a)`. The data length `-1` tells PHP to use the actual length of the character data, which is known to PHP.

To query the table in PHP, the `myselproc()` procedure can be called. The number of elements `$numelems` to be fetched is passed into `myselproc()` by being bound to `:n`. This limits the query to return four rows. The value is also used in the `oci_bind_array_by_name()` call so the output array `$r` is correctly sized to hold the four rows returned. The value `20` is the width of the database column. Any lower value could result in shorter strings being returned to PHP.

Script 53: arrayfetch.php

```

<?php

$c = oci_connect("hr", "welcome", "localhost/XE");

$numelems = 4;
$s = oci_parse($c, "begin mypkg.myselproc(:p1, :n); end;");
oci_bind_array_by_name($s, ":p1", $r, $numelems, 20, SQLT_CHR);
oci_bind_by_name($s, ":n", $numelems);
oci_execute($s);

var_dump($r); // print the array

?>

```

The output is:

```

array(4) {
  [0]=>
  string(3) "abc"
  [1]=>

```

Using PL/SQL With OCI8

```
string(3) "def"
[2]=>
string(3) "ghi"
[3]=>
string(3) "jkl"
}
```

A number of other Oracle types can be bound with [oci_array_bind_by_name\(\)](#), for example SQL_FLT for floating point numbers.

There are more examples of [oci_bind_array_by_name\(\)](#) in the automated OCI8 tests bundled with the PHP source code, see `ext/oci8/tests`.

PL/SQL Success With Information Warnings

A common PL/SQL error when creating packages, procedures or triggers is *Warning: oci_execute(): OCI_SUCCESS_WITH_INFO: ORA-24344: success with compilation error*. This message is most likely to be seen during development of PL/SQL which is commonly done in SQL*Plus or SQL Developer. It can also be seen during application installation if PL/SQL packages, procedures or functions have an unresolved dependency.

PHP code to check for informational errors and warnings is shown in the example `plsqlerr.php`. It creates a procedure referencing a non-existent table and then queries the USER_ERRORS table after the ORA-24344 error occurs:

Script 54: `plsqlerr.php`

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

ini_set('display_errors', false); // do not automatically show PHP errors

// PL/SQL statement with deliberate error: not_mytab does not exist
$plsql = "create or replace procedure
          myproc(d_p in varchar2, i_p in number) as
          begin
              insert into not_mytab (mydata, myid) values (d_p, i_p);
          end;";

$s = oci_parse($c, $plsql);
$r = @oci_execute($s);

if (!$r) {
    $m = oci_error($s);
    if ($m['code'] == 24344) { // A PL/SQL "success with compilation error"
        echo "Warning is " . $m['message'] . "\n";
        showcompilationerrors($c);
    } else { // A normal SQL-style error
        echo "Error is " . $m['message'] . "\n";
    }
}

// Display PL/SQL errors
function showcompilationerrors($c)
```

```
{
    $s = oci_parse($c, "SELECT NAME || ':' || ATTRIBUTE
                        || ' at character ' || POSITION
                        || ' of line ' || LINE || ' - ' || TEXT
                   FROM USER_ERRORS
                   ORDER BY NAME,LINE,POSITION,ATTRIBUTE,MESSAGE_NUMBER");
    oci_execute($s);
    print "<pre>\n";
    while ($row = oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_NULLS)) {
        foreach ($row as $item) {
            print ($item?htmlentities($item):"");
        }
        print "\n";
    }
    print "</pre>";
}
?>
```

This displays:

```
Warning is ORA-24344: success with compilation error
MYPROC: ERROR at character 13 of line 4 - PL/SQL: SQL Statement ignored
MYPROC: ERROR at character 25 of line 4 - PL/SQL: ORA-00942: table or view does
not exist
```

Looking at the PL/SQL code creating the procedure, character 13 on line 4 of the PL/SQL code is the [INSERT](#) statement. Character 25 is the table name [NOT_MYTAB](#).

Your output may also include errors from creating earlier blocks. You can insert a [WHERE](#) clause before the [ORDER BY](#) to restrict the error messages:

```
where name = 'MYPROC'
```

Using REF CURSORS for Result Sets

REF CURSORS let you return a set of query results to PHP - think of them like a pointer to results. In PHP you bind an [OCI_B_CURSOR](#) variable to a PL/SQL REF CURSOR procedure parameter and retrieve the rows of the result set in a normal fetch loop.

As an example, we create a PL/SQL package with a procedure that queries the EMPLOYEES table. The procedure returns a REF CURSOR containing the employees' last names.

The PL/SQL procedure contains the code:

Script 55: refcurl.sql

```
create or replace procedure myproc(p1 out sys_refcursor) as
begin
    open p1 for select last_name from employees where rownum <= 5;
end;
/
show errors
```

In PHP the [oci_new_cursor\(\)](#) function returns a REF CURSOR resource. This is bound to [:rc](#) in the call to [myproc\(\)](#). The bind size of -1 means “ignore the size passed”. It is used because

Using PL/SQL With OCI8

the size of the REF CURSOR is fixed by Oracle. Once the PL/SQL procedure has completed then the value in `$refcur` is treated like a prepared statement identifier. It is simply executed and used in a fetch loop like a normal query.

Script 56: `refcur1.php`

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

// Execute the call to the PL/SQL stored procedure
$s = oci_parse($c, "call myproc(:rc)");
$refcur = oci_new_cursor($c);
oci_bind_by_name($s, ':rc', $refcur, -1, OCI_B_CURSOR);
oci_execute($s);

// Execute and fetch from the cursor
oci_execute($refcur); // treat the returned cursor as an OCI8 statement resource

echo "<table border='1'>\n";
while($row = oci_fetch_array($refcur, OCI_ASSOC)) {
    echo "<tr>";
    foreach ($row as $item) {
        echo "<td>". htmlentities($item) . "</td>";
    }
    echo "</tr>\n";
}
echo "</table>\n";

?>
```

The output is:

```
Abel
Ande
Atkinson
Austin
Baer
```

This next example uses a user-defined type for the REF CURSOR, making the cursor “strongly typed”. The type is declared in a package specification.

Script 57: `refcur2.sql`

```
create or replace package emp_pack as

    type contact_info_type is record (
        fname employees.first_name%type,
        lname employees.last_name%type,
        phone employees.phone_number%type,
        email employees.email%type);

    type contact_info_cur_type is ref cursor return contact_info_type;

    procedure get_contact_info(
```

Using REF CURSORS for Result Sets

```
p_emp_id      in number,
p_contact_info out contact_info_cur_type);

end emp_pack;
/
show errors

create or replace package body emp_pack as

procedure get_contact_info(
    p_emp_id      in number,
    p_contact_info out contact_info_cur_type) as
begin
    open p_contact_info for
        select first_name, last_name, phone_number, email
        from employees
        where employee_id = p_emp_id;
    end;

end emp_pack;
/
show errors
```

The PHP code is very similar to *refcurl1.php*, except in the call to the procedure. The procedure name has changed and, for this example, an example employee identifier of 188 is used.

```
...
$s = oci_parse($c, "call emp_pack.get_contact_info(188, :rc)");
...
```

The output would be the record for employee 188. The four values match the CONTACT_INFO_TYPE:

```
Kelly Chung 650.505.1876 KCHUNG
```

A PL/SQL function that returns a REF CURSOR can be called in an anonymous block like used previously, or be used as a query column. For example:

```
create or replace function selectme(eid_p number) return sys_refcursor is
    rc_1 sys_refcursor;
begin
    open rc_1 for
        select first_name, last_name from employees where employee_id = eid_p;
    return rc_1;
end;
/
```

This can be called in PHP with a query like:

```
<?php
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "select selectme(:eid) as rc from dual");
$eid = 101;
```

Using PL/SQL With OCI8

```
oci_bind_by_name($s, ":eid", $eid);
oci_execute($s);
$r = oci_fetch_array($s);

$refcur = $r['RC'];
oci_execute($refcur);
oci_fetch_all($refcur, $res);
var_dump($res);

?>
```

Closing Cursors

To avoid running out of Oracle cursors (which have a database-configured, per-session limit `open_cursors` set by the DBA), make sure to explicitly free cursors. The example in `refcur3.php` is a script that implicitly creates cursors.

Script 58: `refcur3.php`

```
<?php

// Create a table with 400 rows
function initialize($c)
{
    $stmtarray = array("drop table mytab",
                       "create table mytab(col1 varchar2(1))");

    foreach ($stmtarray as $stmt) {
        $s = oci_parse($c, $stmt);
        @oci_execute($s);
    }

    $s = oci_parse($c, "insert into mytab values ('A')");
    for ($i = 0; $i < 400; ++$i) {
        oci_execute($s);
    }
}

$c = oci_connect('hr', 'welcome', 'localhost/XE');

initialize($c);

$s = oci_parse($c, 'select cursor(select * from dual) from mytab');
oci_execute($s);
while ($refcur = oci_fetch_array($s, OCI_NUM)) { // get each REF CURSOR
    oci_execute($refcur[0]); // execute the REF CURSOR
    while ($row = oci_fetch_array($refcur[0], OCI_NUM)) {
        foreach ($row as $item)
            echo "$item ";
        echo "\n";
    }
    oci_free_statement($refcur[0]); // free the ref cursor
}
```

```
?>
```

The outer select from MYTAB returns not rows, but a CURSOR per row of MYTAB. Those cursors each represent the result from the inner query. That is, there are 400 queries from the DUAL table. The outer WHILE loop fetches each of the 400 REF CURSORS in turn. The inner WHILE loop fetches *from* each REF CURSOR. The result is a stream of X's (which is the single row of data in DUAL) being displayed:

```
X  
X  
X  
. . .
```

This script works, but if the `oci_free_statement()` line is commented out:

```
//    oci_free_statement($refcur[0]);           // free the ref cursor
```

then the script can reach the database limit on the number of cursors. After some iterations through the loop, an error is displayed:

```
PHP Warning:oci_fetch_array(): ORA-00604: error occurred at recursive SQL level 1  
ORA-01000: maximum open cursors exceeded
```

The number of iterations before getting the messages depends on the database configuration parameter `open_cursors`.

Prefetching From REF CURSORS and Nested Cursors for Performance

Prefetching of rows from REF CURSORS is supported from Oracle Database 11gR2 onwards. Prefetching is also performed when connected to previous database versions as long as PHP OCI8 is using version 11.2 Oracle client libraries. Taking advantage of the better performance can be as simple as relinking PHP with the latest libraries. The value of `oci8.default_prefetch` is used for prefetching REF CURSORS, or the value can be changed at runtime:

```
$s = oci_parse($c, "call myproc(:rc)");
$refcur = oci_new_cursor($c);
oci_bind_by_name($s, ':rc', $refcur, -1, OCI_B_CURSOR);
oci_execute($s);
oci_set_prefetch($refcur, 200);
oci_execute($refcur);
oci_fetch_all($refcur, $res);
```

Setting the prefetch count on the "parent" resource `$s` does not change the prefetch size for `$refcur`.

If your script retrieves a REF CURSOR, fetches a few records from it, and then passes the REF CURSOR back to the database where a stored procedure continues fetching records from it, you should set the prefetch size to 0. Otherwise the prefetch buffer of records sent to PHP might not be completely consumed by PHP `oci_fetch_*` calls, and those extra rows would not be available to the second PL/SQL stored procedure. This would give the appearance of missing data, which wasn't processed by the PHP application or the stored procedure.

Using PL/SQL With OCI8

Converting from REF CURSOR to PIPELINED Results

If you are using an older version of Oracle client libraries and cannot use REF CURSOR prefetching, evaluate alternatives such as doing direct queries, writing a wrapping function in PL/SQL that has types that can be bound with `oci_bind_array_by_name()`, or writing a wrapping function that pipelines the output. A pipelined PL/SQL function gives the ability to select from the function as if it were a table.

To convert the `myproc()` procedure from `refcurl.sql` to return pipelined data, create a package:

Script 59: `rc2pipeline.sql`

```
create or replace package myplmap as
    type outtype is record (                      -- structure of the ref cursor in myproc
        last_name varchar2(25)
    );
    type outtype_set is table of outtype;
    function maprctopl return outtype_set pipelined;
end;
/
show errors

create or replace package body myplmap as
    function maprctopl return outtype_set pipelined is
        outrow      outtype_set;
        p_rc        sys_refcursor;
        batchsize   pls_integer := 20;    -- fetch batches of 20 rows at a time
    begin
        myproc(p_rc);                  -- call the original procedure
        loop
            fetch p_rc bulk collect into outrow limit batchsize;
            for i in 1 .. outrow.count() loop
                pipe row (outrow(i));
            end loop;
            exit when outrow.count < batchsize;
        end loop;
    end maprctopl;
end myplmap;
/
show errors
```

This calls `myproc()` and pipes each record. It can be called in PHP using a simple query with the `table` operator:

Script 60: `rc2pipeline.php`

```
<?php
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "select * from table(myplmap.maprctopl())");
oci_execute($s);
oci_fetch_all($s, $res);
var_dump($res);
```

```
?>
```

If the REF CURSOR query in `myproc()` selected all columns, then OUTTYPE_SET could simply have been declared in the SQL script as:

```
type outtype_set is table of employees%rowtype;
```

Oracle Collections in PHP

Many programming techniques use collection types such as arrays, bags, lists, nested tables, sets, and trees. To support these techniques in database applications, PL/SQL provides the data types TABLE and VARRAY, which allow you to declare index-by tables, nested tables, and variable-size arrays.

An Oracle collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. Collections work like the arrays found in most third-generation programming languages. Also, collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

Oracle collections can be manipulated in PHP by methods on a collection resource, which is allocated with `oci_new_collection()`.

In a simple email address book example, two VARRAYs are created, one for an array of people's names, and one for an array of email addresses. VARRAYs (short for variable-size arrays) use sequential numbers as subscripts to access a fixed number of elements.

Script 61: addressbook.sql

```
drop table emails;

create table emails (
    user_id      varchar2(10),
    friend_name  varchar2(20),
    email_address varchar2(20));

create or replace type email_array as varray(100) of varchar2(20);
/
show errors

create or replace type friend_array as varray(100) of varchar2(20);
/
show errors

create or replace procedure update_address_book(
    p_user_id      in varchar2,
    p_friend_name  friend_array,
    p_email_addresses  email_array)
is
begin
    delete from emails where user_id = p_user_id;
    forall i in indices of p_email_addresses
        insert into emails (user_id, friend_name, email_address)
        values (p_user_id, p_friend_name(i), p_email_addresses(i));
```

Using PL/SQL With OCI8

```
end update_address_book;
/
show errors
```

The `update_address_book()` procedure loops over all elements of the address collection and inserts each one and its matching name.

The `updateaddresses.php` code creates a collection of names and a collection of email addresses using the `append()` method to add elements to each array. These collections are bound as `OCI_B_NTY` ("named type") to the arguments of the PL/SQL `address_book()` call. The size `-1` is used because Oracle internally knows the size of the type. When `address_book()` is executed, the names and email addresses are inserted into the database.

Script 62: updateaddresses.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$user_name      = 'cjones';
$friends_names  = array('alison', 'aslam');
$friends_emails = array('alison@example.com', 'aslam@example.com');

$friend_coll = oci_new_collection($c, 'FRIEND_ARRAY');
$email_coll  = oci_new_collection($c, 'EMAIL_ARRAY');

for ($i = 0; $i < count($friends_names); ++$i) {
    $friend_coll->append($friends_names[$i]);
    $email_coll->append($friends_emails[$i]);
}

$s = oci_parse($c, "begin update_address_book(:un, :friends, :emails); end;");
oci_bind_by_name($s, ':un', $user_name);
oci_bind_by_name($s, ':friends', $friend_coll, -1, OCI_B_NTY);
oci_bind_by_name($s, ':emails', $email_coll, -1, OCI_B_NTY);

oci_execute($s);

?>
```

The EMAILS table now has the inserted data:

```
SQL> select * from emails;
```

USER_ID	FRIEND_NAME	EMAIL_ADDRESS
cjones	alison	alison@example.com
cjones	aslam	aslam@example.com

Other OCI8 collection methods allow accessing or copying data in a collection. See the PHP OCI8 manual for more information.

Using PL/SQL and Oracle Object Types in PHP

Sometime you have to work with Oracle object types or call PL/SQL procedures that are designed for interacting with other PL/SQL code and that cannot be used directly with OCI8. Previous sections have introduced some methods. This section gives more examples. It is a brief guide, and not exhaustive. Also, some techniques will work better in some situations than in others.

The first example simulates the Oracle Text CTX_THES package procedures. These return Oracle object types. (Oracle Text is a database component that uses standard SQL to index, search, and analyze text and documents stored in the database, in files, and on the web. It can perform linguistic analysis on documents, as well as search text using a variety of strategies including keyword searching and context queries).

This example, `ctx.sql`, sets up an example package with a similar interface to CTX_THES. Here it just returns random data in the OUT parameter:

Script 63: ctx.sql

```
-- Package "SuppliedPkg" simulates Oracle Text's CTX_THES.
-- It has a procedure that returns a PL/SQL type.

create or replace package SuppliedPkg as
  type SuppliedRec is record (
    id      number,
    data    varchar2(100)
  );
  type SuppliedTabType is table of SuppliedRec index by binary_integer;
  procedure SuppliedProc(p_p in out nocopy SuppliedTabType);
end SuppliedPkg;
/
show errors

create or replace package body SuppliedPkg as
  procedure SuppliedProc(p_p in out nocopy SuppliedTabType) is
begin
  -- Create some random data
  p_p.delete;
  for i in 1..5 loop
    p_p(i).id    := i;
    p_p(i).data := 'Random: ' || i || (1+ABS(MOD(dbms_random.random,100000)));
  end loop;
end SuppliedProc;
end SuppliedPkg;
/
show errors
```

Run the file `ctx.sql` in SQL*Plus:

```
$ sqlplus hr/welcome@localhost/XE @ctx.sql
```

This is the “fixed” part of the problem, representing the unchangeable, pre-supplied functionality you need to work with. The next four sections show different techniques for fetching data from [SuppliedProc\(\)](#).

Using PL/SQL With OCI8

Using a PIPELINED Function

A PL/SQL wrapper function using PIPE can be used to fetch the values returned by [SuppliedProc\(\)](#):

Script 64: myplpkg.sql

```
create or replace package myplpkg as
  type pltab is table of SuppliedPkg.SuppliedRec;
  function mywrapper1 return pltab pipelined;
end;
/
show errors

create or replace package body myplpkg as
  function mywrapper1 return pltab pipelined is
    origdata SuppliedPkg.SuppliedTabType;
  begin
    SuppliedPkg.SuppliedProc(origdata);
    for i in 1..origdata.count loop
      pipe row (origdata(i));
    end loop;
  end mywrapper1;
end myplpkg;
/
show errors
```

The PHP code to call the wrapper is:

Script 65: myplpkg.php

```
<?php

$c = oci_connect("hr", "welcome", "localhost/XE");

$s = oci_parse($c, "select * from table(myplpkg.mywrapper1())");
oci_execute($s);
oci_fetch_all($s, $res);
var_dump($res);

?>
```

The output is like:

```
array(2) {
  ["ID"]=>
  array(5) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "3"
    [3]=>
    string(1) "4"
```

```
[4]=>
    string(1) "5"
}
["DATA"]=>
array(5) {
    [0]=>
        string(14) "Random: 174852"
    [1]=>
        string(14) "Random: 210905"
    [2]=>
        string(14) "Random: 341032"
    [3]=>
        string(13) "Random: 41530"
    [4]=>
        string(14) "Random: 540388"
}
```

Using a REF CURSOR

Since data from MYPLPKG.MYWRAPPER1 is fetched with a query, you could also easily return the values to PHP using a REF CURSOR, if this is your preferred coding style:

Script 66: myrcpkg.sql

```
create or replace package myrcpkg as
    procedure mywrapper2 (p_rc out sys_refcursor);
end;
/
show errors

create or replace package body myrcpkg as
    procedure mywrapper2 (p_rc out sys_refcursor) is
    begin
        open p_rc for select * from table(myplpk.mywrapper1());
    end mywrapper2;
end myrcpkg;
/
show errors
```

This can be called in PHP like:

Script 67: myrcpkg.php

```
<?php

$c = oci_connect("hr", "welcome", "localhost/XE");

$s = oci_parse($c, "begin myrcpkg.mywrapper2(:myrc); end;");
$rc = oci_new_cursor($c);
oci_bind_by_name($s, ':myrc', $rc, -1, OCI_B_CURSOR);
oci_execute($s);
oci_execute($rc);
oci_fetch_all($rc, $res);
```

Using PL/SQL With OCI8

```
var_dump($res);
?>
```

The output is the same as from *myrcpkg.php*.

Using an Array Bind

A third solution lets you use the fast `oci_bind_array_by_name()` function. The wrapper procedure looks like:

Script 68: mybapkg.sql

```
create or replace package mybapkg as
procedure mywrapper3 (
    p_id    out dbms_sql.number_table,
    p_data  out dbms_sql.varchar2_table);
end;
/
show errors

create or replace package body mybapkg as
procedure mywrapper3(
    p_id    out dbms_sql.number_table,
    p_data  out dbms_sql.varchar2_table) as
begin
    select id, data
    bulk collect into p_id, p_data
    from hxuntable(myplpkg.mywrapper1());
end mywrapper3;
end mybapkg;
/
show errors
```

This can be called from PHP with:

Script 69: mybapkg.php

```
<?php
$c = oci_connect("hr", "welcome", "localhost/XE");

$s = oci_parse($c, "begin mybapkg.mywrapper3(:myid, :mydata); end;");
oci_bind_array_by_name($s, ":myid", $myid, 10, -1, SQLT_INT);
oci_bind_array_by_name($s, ":mydata", $mydata, 10, 20, SQLT_CHR);
oci_execute($s);

var_dump($myid);
var_dump($mydata);

?>
```

This technique can be used when the number of items to return is known. The output is similar to:

```
array(5) {
  [0] =>
  int(1)
  [1] =>
  int(2)
  [2] =>
  int(3)
  [3] =>
  int(4)
  [4] =>
  int(5)
}
array(5) {
  [0] =>
  string(14) "Random: 113453"
  [1] =>
  string(14) "Random: 298387"
  [2] =>
  string(14) "Random: 375767"
  [3] =>
  string(14) "Random: 450352"
  [4] =>
  string(14) "Random: 599520"
}
```

Using OCI8 Collection Functions

The [SuppliedProc\(\)](#) procedure can also be called using OCI8 collection functions by creating yet another a wrapper function in PL/SQL to convert the PL/SQL type [SuppliedTabType](#) to a pair of SQL types:

Script 70: myclpkg.sql

```
create or replace type MyIdRec as table of number;
/
show errors

create or replace type MyDataRec as table of varchar2(100);
/
show errors

create or replace package myclpkg as
  procedure wrapper4 (p_id in out MyIdRec, p_data in out MyDataRec);
end;
/
show errors

create or replace package body myclpkg as
  procedure wrapper4 (p_id in out MyIdRec, p_data in out MyDataRec)
  as
  begin
    select id, data
    bulk collect into p_id, p_data
```

Using PL/SQL With OCI8

```
    from table(myplpkg.mywrapper1());
  end wrapper4;
end myclpkg;
/
show errors
```

Now you can call `wrapper4()` in PHP:

Script 71: `myclpkg.php`

```
<?php

$c = oci_connect("hr", "welcome", "localhost/XE");

$s = oci_parse($c, 'begin myclpkg.wrapper4(:res_id, :res_data); end;');
$res_id = oci_new_collection($c, 'MYIDREC');
$res_data = oci_new_collection($c, 'MYDATAREC');
oci_bind_by_name($s, ':res_id', $res_id, -1, OCI_B_NTY);
oci_bind_by_name($s, ':res_data', $res_data, -1, OCI_B_NTY);
oci_execute($s);

for ($i = 0; $i < $res_id->size(); $i++) {
    $id = $res_id->getElem($i);
    $data = $res_data->getElem($i);
    echo "Id: $id, Data: $data\n";
}

?>
```

This allocates two collections and binds them as the parameters to `wrapper4()`. After `wrapper4()` has been called, the PHP OCI8 collection method `getElem()` is used to access each value returned. The output is similar to:

```
Id: 1, Data: Random: 155942
Id: 2, Data: Random: 247783
Id: 3, Data: Random: 365553
Id: 4, Data: Random: 487553
Id: 5, Data: Random: 589879
```

This section has shown four methods of fetching PL/SQL types. The best method will depend on personal preference, data size and performance in your environment.

Getting Output With DBMS_OUTPUT

The DBMS_OUTPUT package is the standard way to “print” output from PL/SQL. The drawback is that it is not asynchronous. The PL/SQL procedure or block that calls DBMS_OUTPUT runs to completion before any output is returned to the user.

DBMS_OUTPUT is like a buffer. Your PHP code turns on DBMS_OUTPUT buffering, calls some PL/SQL code that puts output in the buffer, and then later fetches from the buffer. Other database connections cannot access your buffer.

A basic way to fetch DBMS_OUTPUT in PHP is to bind an output string to the PL/SQL `dbms_output.get_line()` procedure:

```
$s = oci_parse($c, "begin dbms_output.get_line(:ln, :st); end;");
```

Getting Output With DBMS_OUTPUT

```
oci_bind_by_name($s, ":ln", $ln, 255);          // output line
oci_bind_by_name($s, ":st", $st, -1, SQLT_INT); // status: 1 means no more lines
while (($succ = oci_execute($s)) && !$st) {
    echo "$ln\n";
}
```

The variable `:ln` is arbitrarily bound as length 255. This was the DBMS_OUTPUT line size limit prior to Oracle Database 10g Release 10.2, when it was then changed to 32KB. (The limitation on the number of lines was also raised). Avoid binding 32KB, especially if the database is running in Oracle's shared server mode. If you bind this size, then it is easy to slow down performance or get memory errors. However, if you bind less than the full size, make sure your application does not print wider lines.

Alternatively, you can use [oci_bind_array_by_name\(\)](#) and call another DBMS_OUTPUT package that returns multiple lines, [get_lines\(\)](#). The performance of this is generally better but it does depend on how big the bind array is, how much data is returned and how the code is structured. In the worst case it might be slower than the simple code, so benchmark your application carefully.

A more consistent and fast DBMS_OUTPUT fetching implementation uses a custom pipelined PL/SQL function. In SQL*Plus create the function:

Script 72: dbmsoutput.sql

```
create or replace type dorow as table of varchar2(4000);
/
show errors

create or replace function mydofetch return dorow pipelined is
    line    varchar2(4000);
    status  integer;
begin
    loop
        dbms_output.get_line(line, status);
        exit when status = 1;
        pipe row (line);
    end loop;
    return;
end;
/
show errors
```

Because we will fetch the data in a query as a SQL string, the maximum length is 4000 bytes.

A function to turn on output buffering is shown in [dbmsoutput.inc.php](#) along with [getdbmsoutput\(\)](#) which returns an array of the output lines:

Script 73: dbmsoutput.inc.php

```
<?php

// Turn DBMS_OUTPUT on
function enabledbmsoutput($c)
{
    $s = oci_parse($c, "begin dbms_output.enable(null); end;");
    $r = oci_execute($s);
    return $r;
```

Using PL/SQL With OCI8

```
}  
  
// Returns an array of DBMS_OUTPUT lines  
function getdbmsoutput($c)  
{  
    $res = false;  
    $s = oci_parse($c, "select * from table(mydofetch())");  
    oci_execute($s);  
    oci_fetch_all($s, $res);  
    return $res['COLUMN_VALUE'];  
}  
  
?>
```

This has the arbitrary file extension “.inc.php” file to indicate it will be included in other PHP scripts.

The next script uses these functions to show “printing” output from PL/SQL blocks:

Script 74: dbmsoutput.php

```
<?php  
  
include("dbmsoutput.inc.php");  
  
$c = oci_connect("hr", "welcome", "localhost/XE");  
  
// Turn output buffering on  
enabledbmsoutput($c);  
  
// Create some output  
$s = oci_parse($c, "call dbms_output.put_line('Hello, world!')");  
oci_execute($s);  
  
// Create more output  
// Any PL/SQL code being run can insert into the output buffer  
$s = oci_parse($c, "begin  
    dbms_output.put_line('Hello again');  
    dbms_output.put_line('Hello finally');  
end;");  
oci_execute($s);  
  
// Display the output  
$output = getdbmsoutput($c);  
if ($output) {  
    foreach ($output as $line) {  
        echo "$line<br>\n";  
    }  
}  
?>
```

The output is all the [dbms_output.put_line\(\)](#) text:

```
Hello, world!  
Hello again
```

```
Hello finally
```

If you expect large amounts of output, you may want to stream results as they are fetched from the database instead of returning them in one array from `getdbmsoutput()`.

If DBMS_OUTPUT does not suit your application, you can also get output from PL/SQL by logging it to database tables or by using packages like UTL_FILE and DBMS_PIPE to asynchronously display output to a separate terminal window.

PL/SQL Backtraces in a PL/SQL Exception Handler

PL/SQL has several functions helpful for problem resolution when dealing with exceptions. Instead of hiding errors or returning abstract messages from an exception handler you can return the exact Oracle problem and location. If your PL/SQL code includes these functions, your PHP application can generate useful backtraces to help debugging. Remember to log these errors and not display them to web users.

Sample SQL code that always generates and then catches an exception is in *backtrace.sql*:

Script 75: backtrace.sql

```
create or replace procedure mybt is
begin
    dbms_output.put_line('In mybt');
    raise no_data_found; // always throw an error
exception
    when others then
        dbms_output.put_line('Displaying the error stack:');
        dbms_output.put(dbms_utility.format_error_stack);
        dbms_output.put_line(dbms_utility.format_error_backtrace);
end;
/
```

PHP code to call this and fetch the output is in *backtrace.php*:

Script 76: backtrace.php

```
<?php
include("dbmsoutput.inc.php");

$c = oci_connect("hr", "welcome", "localhost/XE");
enabledbmsoutput($c);

$s = oci_parse($c, "begin mybt(); end;");
oci_execute($s);

// Display the output
$output = getdbmsoutput($c); // see previous examples
foreach ($output as $line)
    echo "$line<br>\n";

?>
```

Running *backtrace.php* gives:

Using PL/SQL With OCI8

```
In mybt
Displaying the error stack:
ORA-01403: no data found
ORA-06512: at "HR.MYBT", line 4
```

PL/SQL Function Result Cache

Oracle Database 11g introduced a cache for PL/SQL function results, ideal for repeated lookup operations. The cache contains the generated results of previous function calls, for particular sets of input parameters. If the function is re-run with the same parameter values, the result from the cache is returned immediately without needing to re-execute the code. The cached results are available to any user. The cache will age out results if more memory is required.

There are some restrictions including that only basic types can be used for parameters, and they must be IN only. Return types are similar restricted, in particular not using REF CURSOR or PIPELINED results.

To use the cache, a normal function is created with the RESULT_CACHE option:

Script 77: frc.sql

```
create or replace function mycachefunc(p_id in varchar2) return varchar2
  result_cache relies_on(mytab)
as
  l_data varchar2(40);
begin
  select mydata into l_data from mytab where myid = p_id;
  return l_data;
end;
/
show errors
```

The `relies_on()` clause is a comma separated list of tables. If any of these tables change, than the cache is automatically invalidated by Oracle. The next time the function is called, it will execute completely and update the cache appropriately. From Oracle Database 11.2, tables used in the function are automatically detected.

See the *Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)* manual for more details about the feature.

Using Oracle Locator for Spatial Mapping

Oracle Locator is a subset of Oracle Spatial, a comprehensive mapping library. Oracle Locator is powerful itself and is available in all Oracle Database editions. A great introduction to Oracle Locator is in the *Oracle Database Express Edition 2 Day Plus Locator Developer Guide 11g Release 2*.

This section shows some techniques to use Locator data in PHP. Oracle Locator makes use of PL/SQL types such as collections. These can not always be directly fetched into PHP.

The examples here use the tables shown in the above mentioned manual's sample scenario. Create the CUSTOMERS and STORES tables from *Example 1-1 SQL Script for Customers and Stores Scenario* in SQL*Plus before continuing. The URL for the script is:

http://docs.oracle.com/cd/E17781_01/appdev.112/e18750/xe_locator.htm#XELOC560

Inserting Locator Data

Inserting Locator data in PHP is simply a matter of executing the appropriate SQL [INSERT](#) statement:

```
$sql = "insert into customers values
    (100, 'A', 'B',
     '111 Reese Ave', 'Chicago', 'IL', 12345,
     SDO_GEOmetry(2001,
                   8307,
                   SDO_POINT_TYPE(-69.231445,12.001254,NULL), NULL, NULL))";
$s = oci_parse($c, $sql);
oci_execute($s);
```

Queries Returning Scalar Values

Before fetching data, determine if this is, in fact, necessary. Often the data can be processed in Oracle SQL or PL/SQL efficiently and easily.

Queries returning scalar values from Locator objects are no different to other PHP queries. This example finds the three closest customers to the store with CUSTOMER_ID of 101. The query uses the in-built Spatial function [SOD_NN\(\)](#) to determine the *nearest neighbor* relationship.

Script 78: loc1.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$sql = "select /*+ ordered */
        c.customer_id,
        c.first_name,
        c.last_name
       from stores s, customers c
      where s.store_id = :sid
        and sdo_nn(c.cust_geo_location, s.store_geo_location, :nres) = 'TRUE'";

$s = oci_parse($c, $sql);

$sid = 101;
$nres = 'sdo_num_res=3'; // return 3 results

oci_bind_by_name($s, ":sid", $sid);
oci_bind_by_name($s, ":nres", $nres);
oci_execute($s);
oci_fetch_all($s, $res);
var_dump($res);

?>
```

The output is:

```
array(3) {
  ["CUSTOMER_ID"]=>
```

Using PL/SQL With OCI8

```
array(3) {
    [0]=>
    string(4) "1001"
    [1]=>
    string(4) "1003"
    [2]=>
    string(4) "1004"
}
["FIRST_NAME"]=>
array(3) {
    [0]=>
    string(9) "Alexandra"
    [1]=>
    string(6) "Marian"
    [2]=>
    string(6) "Thomas"
}
["LAST_NAME"]=>
array(3) {
    [0]=>
    string(7) "Nichols"
    [1]=>
    string(5) "Chang"
    [2]=>
    string(8) "Williams"
}
}
```

The CUSTOMER_ID, FIRST_NAME and LAST_NAME columns are scalar NUMBER and VARCHAR2 columns returned directly into a PHP array.

Selecting Vertices Using SDO_UTIL.GETVERTICES

For some Locator types, in-built functions will convert objects to scalar values that can be returned to PHP. For example, to fetch the coordinates from a geometry for customer 1001, use the inbuilt [SDO_UTIL.GETVERTICES\(\)](#) function:

Script 79: loc2.php

```
<?php
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$sql = "select t.x, t.y
       from customers,
            table(sdo_util.getvertices(customers.cust_geo_location)) t
      where customer_id = :cid";

$s = oci_parse($c, $sql);
$cid = 1001;
oci_bind_by_name($s, ":cid", $cid);
oci_execute($s);
oci_fetch_all($s, $res);
var_dump($res);
```

```
?>
```

The output is:

```
array(2) {
    ["X"]=>
        array(1) {
            [0]=>
                string(9) "-71.48923"
        }
    ["Y"]=>
        array(1) {
            [0]=>
                string(8) "42.72347"
        }
}
```

Using a Custom Function

Sometimes you may need to create a PL/SQL function to decompose spatial data into simple types to return them to PHP. This example uses the COLA_MARKETS table from section 1.9, *Using Non-Point Geometry Types* in the *Locator* manual:

http://docs.oracle.com/cd/E17781_01/appdev.112/e18750/xe_locator.htm#XELOC575

Before continuing, execute the three statements given in the manual to create the table, insert the meta-data into USER_SDO_GEOM_METADATA table, and create the index.

Next, insert a sample row as shown below in *cm1.sql*. The row describes a polygon of (x,y) ordinates which are given as pairs in the SDO_ORDINATE_ARRAY array:

Script 80: cm1.sql

```
insert into cola_markets values (
    301,          -- market ID number
    'polygon',
    sdo_geometry (
        2003,      -- two-dimensional polygon
        null,
        null,
        sdo_elem_info_array(1,1003,1), -- one polygon (exterior polygon ring)
        sdo_coordinate_array(5,1, 8,1, 8,6, 5,7, 5,1) -- list of X,Y coordinates
    )
);
commit;
```

A decomposition function in *cm2.sql* helps query the coordinates in PHP. Note the alias CM (an alias here is also known as a *correlation name*) for the table in the query. This allows the SDO_ORDINATES collection to be included as a select column:

Using PL/SQL With OCI8

Script 81: cm2.sql

```
create or replace procedure myproc(p_id in number, p_o out sdo_ordinate_array) as
begin
    select cm.shape.sdoordinates
    into p_o
    from cola_markets cm
    where mkt_id = p_id;
end;
/
show errors
```

The coordinates can now be retrieved in PHP as a collection:

Script 82: cm.php

```
<?php
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "begin myproc(:id, :ords); end;");
$id = 301;
oci_bind_by_name($s, ":id", $id);
$ords = oci_new_collection($c, "SDO_ORDINATE_ARRAY");
oci_bind_by_name($s, ":ords", $ords, -1, OCI_B_NTY);
oci_execute($s);

for ($i = 0; $i < $ords->size(); $i++) {
    $v = $ords->getElem($i);
    echo "Value: $v\n";
}
?>
```

The output is the list of coordinates that were inserted in the SDO_ORDINATE_ARRAY:

```
Value: 5
Value: 1
Value: 8
Value: 1
Value: 8
Value: 6
Value: 5
Value: 7
Value: 5
Value: 1
```

Similar techniques to these example given, or those techniques in earlier sections like *Using PL/SQL and Oracle Object Types in PHP* can be used to fetch other Locator data, if required.

Scheduling Background or Long Running Operations

Sometimes a web page starts a database operation that can run in the background while the user continues other work.

Scheduling Background or Long Running Operations

For example, there might be some database cleanup to be run periodically. Another example is when a user of a photo site decides to change the name of a tag associated with images. The photo site application might initiate the name change, but return the user an HTML page saying *Your request is being processed and will soon complete*. The user can continue viewing photos without having to wait for the renaming process to complete. This technique can improve user satisfaction. It can also free up an Apache server that would otherwise be blocked, allowing it to be used by another page request.

The DBMS_SCHEDULER package can be used to start such background database tasks. It has a lot of functionality, including allowing tasks to be repeated at intervals, or started when events are received. It can also be used to invoke operating system programs. In Oracle 9i, the DBMS_JOB package can be used instead of DBMS_SCHEDULER.

For the photo site example, create some data with the tag *weeding*:

Script 83: dschedinit.sql

```
connect system/systempwd  
grant create job to hr;  
connect hr/welcome  
drop table tag_table;  
  
create table tag_table (tag varchar2(20), photo_id number);  
insert into tag_table values ('weeding', 2034);  
insert into tag_table values ('weeding', 2035);  
insert into tag_table values ('sanfrancisco', 4540);  
commit;
```

To change the tag *weeding* to *wedding*, a procedure [changetagname\(\)](#) can be created:

Script 84: dbsched.sql

```
create or replace procedure changetagname(old in varchar2, new in varchar2) as  
  b number;  
begin  
  for i in 1..100000000 loop b := 1; end loop; -- simulate slow transaction  
  update tag_table set tag = new where tag = old;  
  commit;  
end;  
/  
show errors
```

This script creates a sample table and the procedure to update tags. The procedure is artificially slowed down to simulate a big, long running database operation.

The following PHP script uses an anonymous block to create a job calling [changetagname\(\)](#).

Script 85: dsched.php

```
<?php  
  
$c = oci_connect("hr", "welcome", "localhost/XE");  
  
function doquery($c)
```

Using PL/SQL With OCI8

```
{  
    $s = oci_parse($c, "select tag from tag_table");  
    oci_execute($s);  
    oci_fetch_all($s, $res);  
    var_dump($res);  
}  
  
// Schedule a task to change a tag name from 'weeding' to 'wedding'  
  
$stmt =  
"begin  
    dbms_scheduler.create_job(  
        job_name          => :jobname,  
        job_type          => 'STORED_PROCEDURE',  
        job_action         => 'changetagname',      // procedure to call  
        number_of_arguments => 2);  
    dbms_scheduler.set_job_argument_value (  
        job_name          => :jobname,  
        argument_position  => 1,  
        argument_value     => :oldval);  
    dbms_scheduler.set_job_argument_value (  
        job_name          => :jobname,  
        argument_position  => 2,  
        argument_value     => :newval);  
    dbms_scheduler.enable(:jobname);  
end;";  
  
$s = oci_parse($c, $stmt);  
  
$jobname = uniqid('ut');  
$oldval  = 'weeding';  
$newval  = 'wedding';  
oci_bind_by_name($s, ":jobname", $jobname);  
oci_bind_by_name($s, ":oldval",   $oldval);  
oci_bind_by_name($s, ":newval",   $newval);  
  
oci_execute($s);  
  
echo "<pre>Your request is being processed and will soon complete\n";  
doquery($c); // gives old results  
sleep(10);  
echo "Your request has probably completed\n";  
doquery($c); // gives new results  
  
?>
```

The PHP call to the anonymous PL/SQL block returns quickly. The background PL/SQL call to [changetagname\(\)](#) will take several more seconds to complete (because of its `for` loop), so the first [doquery\(\)](#) output shows the original, incorrect tag values. Then, after PHP has given the job time to conclude, the second [doquery\(\)](#) call shows the updated values:

```
Your request is being processed and will soon complete  
array(1) {  
    ["TAG"]=>
```

```
array(3) {
    [0]=>
    string(7) "weeding"
    [1]=>
    string(7) "weeding"
    [2]=>
    string(12) "sanfrancisco"
}
}
Your request has probably completed
array(1) {
    ["TAG"]=>
    array(3) {
        [0]=>
        string(7) "wedding"
        [1]=>
        string(7) "wedding"
        [2]=>
        string(12) "sanfrancisco"
    }
}
```

Oracle Streams Advanced Queuing

Another way to initiate background tasks is to use Oracle Streams Advanced Queuing in a producer-consumer message passing fashion. Oracle AQ is highly configurable. Messages can be queued by multiple producers. Different consumers can filter messages for them. Messages can also be propagated to queues in other databases. Oracle AQ has PL/SQL, Java, C and HTTPS interfaces. From PHP, the PL/SQL interface is used.

The following example simulates an application user registration system where the PHP application queues each new user's street address. An external system can then fetch and process that address. In real life the external system might mail a welcome letter, or do further, slower validation on the address.

The SQL*Plus script *qcreate.sql* creates a new Oracle user *demoqueue* with permission to create and use queues. A payload type for the address is created and a queue set up for this payload.

Script 86: *qcreate.sql*

```
connect / as sysdba
drop user demoqueue cascade;

create user demoqueue identified by welcome;
grant connect, resource to demoqueue;
grant aq_administrator_role, aq_user_role to demoqueue;
grant execute on dbms_aq to demoqueue;
grant create type to demoqueue;

connect demoqueue/welcome@localhost/xe

-- The data we want to queue
create or replace type user_address_type as object (
```

Using PL/SQL With OCI8

```
    name      varchar2(10),
    address   varchar2(50)
);
/

begin
  dbms_aqadm.create_queue_table(
    queue_table      => 'demoqueue.addr_queue_tab',
    queue_payload_type => 'demoqueue.user_address_type');
end;
/

begin
  dbms_aqadm.create_queue(
    queue_name        => 'demoqueue.addr_queue',
    queue_table       => 'demoqueue.addr_queue_tab');
end;
/

begin
  dbms_aqadm.start_queue(
    queue_name        => 'demoqueue.addr_queue',
    enqueue           => true);
end;
/
```

The script *qhelper.sql* creates two helper functions to enqueue and dequeue messages.

Script 87: qhelper.sql

```
-- Set up enqueue/dequeue procedures

connect demoqueue/welcome@localhost/xe

create or replace procedure my_enq(
    user_addr_p in user_address_type,
    priority_p  in number) as
  enqueue_options  dbms_aq.enqueue_options_t;
  message_properties dbms_aq.message_properties_t;
  enq_id          raw(16);
begin
  dbms_aq.enqueue(queue_name      => 'demoqueue.addr_queue',
                  enqueue_options  => enqueue_options,
                  message_properties => message_properties,
                  payload          => user_addr_p,
                  msgid            => enq_id);
  commit;
end;
/
show errors

create or replace procedure my_deq(
    user_addr_p out user_address_type) as
  dequeue_options  dbms_aq.dequeue_options_t;
```

```

message_properties dbms_aq.message_properties_t;
enq_id           raw(16);
begin
  dbms_aq.dequeue(queue_name      => 'demoqueue.addr_queue',
                  dequeue_options   => dequeue_options,
                  message_properties => message_properties,
                  payload          => user_addr_p,
                  msgid            => enq_id);
  commit;
end;
/
show errors

```

The script *newuser.php* handles a new application user and queues a message containing their address:

Script 88: newuser.php

```

<?php
$c = oci_connect("demoqueue", "welcome", "localhost/xe");

// The new application user details
$username = 'Fred';
$address = '500 Oracle Parkway';

// Enqueue the user information for further offline handling
$sql = "begin my_enq(user_address_type('$username', '$address'), 1); end;";
$s = oci_parse($c, $sql);
$r = oci_execute($s);

// Continue processing the new user in the current script
echo "Welcome $username\n";
?>

```

This executes an anonymous PL/SQL block to create and enqueue the address message. The immediate output to the caller is simply the welcome message:

```
Welcome Fred
```

Once this PHP script is executed, any application can dequeue the new message at its leisure. For example, the following SQL*Plus commands call the helper *my_deq()* dequeue function and display the user details:

Script 89: showuser.sql

```

connect demoqueue/welcome@localhost/xe
set serveroutput on
declare
  user_address user_address_type;
begin
  my_deq(user_address);

```

Using PL/SQL With OCI8

```
dbms_output.put_line('Name    : ' || user_address.name);
dbms_output.put_line('Address : ' || user_address.address);
end;
/
```

The output is:

```
Name      : Fred
Address   : 500 Oracle Parkway
```

If this dequeue operation is called without anything in the queue, it will block waiting for a message until the queue wait time expires.

The PL/SQL API has much more functionality than shown in this overview. For example you can enqueue an array of messages, or listen to more than one queue.

Queuing is highly configurable and scalable, providing a great way to distribute workload for a web application. Oracle AQ is available in all editions of the database.

Reusing Procedures Written for MOD_PLSQL

Oracle's MOD_PLSQL gateway allows a Web browser to invoke a PL/SQL stored subprogram through an HTTP listener. This is the interface used by Oracle Application Express. Existing user-created PL/SQL procedures written for this gateway can be called from PHP using a wrapper function. For example, consider a stored procedure for MOD_PLSQL that was created in SQL*Plus:

Script 90: myowa.sql

```
create or replace procedure myowa as
begin
  http.htmlOpen;
  http.headOpen;
  http.title('Greeting Title');
  http.headClose;
  http.bodyOpen;
  http.header(1, 'Salutation Heading');
  http.p('Hello, world!');
  http.bodyClose;
  http.htmlClose;
end;
/
show errors
```

This generates HTML output to the gateway:

```
<HTML>
<HEAD>
<TITLE>Greeting Title</TITLE>
</HEAD>
<BODY>
<H1>Salutation Heading</H1>
Hello, world!
</BODY>
</HTML>
```

Reusing Procedures Written for MOD_PLSQL

To reuse the procedure directly in PHP, use HTP.GET_LINE in a mapping function to pipe the output from the *myowa.sql* HTP calls:

Script 91: *mymodplsql.sql*

```
create or replace type modpsrow as table of varchar2(512);
/
show errors

create or replace function mymodplsql(proc varchar2) return modpsrow pipelined is
  param_val owa_vc_arr;
  line      varchar2(256);
  irows      integer;
begin
  owa.init_cgi_env(param_val);
  http.init;
  execute immediate 'begin'||proc||'; end;';
  loop
    line := http.get_line(irows);
    exit when line is null;
    pipe row (line);
  end loop;
  return;
end;
/
show errors
```

This is fundamentally similar to the previous pipelined examples.

In [modpsrow\(\)](#) you can optionally use PARAM_VAL to set CGI values. See the definition of [init.cgi_env\(\)](#) in *\$ORACLE_HOME/rdbms/admin/privowa.sql* for details.

In PHP, the new wrapper can be called like:

Script 92: *mymodplsql.php*

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$stmt = oci_parse($c, 'select * from table(mymodplsql(:proc))');
$func = 'myowa';
oci_bind_by_name($stmt, ':proc', $func);

oci_execute($stmt);

$content = false;
while ($row = oci_fetch_array($stmt, OCI_ASSOC)) {
    if ($content) {
        print $row["COLUMN_VALUE"];
    } else {
        if ($row["COLUMN_VALUE"] == "\n")
            $content = true;
        else
            header($row["COLUMN_VALUE"]);
    }
}
```

Using PL/SQL With OCI8

```
?>
```

When called in a browser, the output is the expected rendition of the HTML fragment shown earlier.

Easy PL/SQL Upgrades With Edition Based Redefinition

The *Editioning* feature of Oracle Database 11gR2 is very useful for web applications that aim for no downtime when releasing enhanced versions of applications. It allows multiple versions of PL/SQL objects with the same names to be used concurrently. This lets you upgrade stored procedures and test new versions while production users are still accessing the original versions. As well as allowing this safe way to upgrade, you can also use it for A/B testing, where you evaluate the user response to a new version of your application. The application versions that users don't appreciate can quickly be dropped.

The objects you can edition are:

- synonyms
- views
- PL/SQL object types:
 - function
 - library
 - package and package body
 - procedure
 - trigger
 - type and type body

Tables themselves can't be editioned but there is support for moving and viewing data across editions.

The following example shows how editioning can be used to upgrade a live PHP application. The changes can be made and tested on the production database and then enabled for all users with a one keyword change in the application.

As the user SYSTEM, allow the application user HR to use editions:

Script 93: ed1.sql

```
connect system/systempwd@localhost/XE
grant create any edition to hr;
alter user hr enable editions;
```

As the user HR create a table of employees for the application and create the stored function that calculates the number of days of vacation an employee is eligible for. This function is stored in the database so all Oracle applications can reuse the same logic:

Script 94: ed2.sql

```
connect hr/welcome@localhost/XE
drop table myemp;
```

Easy PL/SQL Upgrades With Edition Based Redefinition

```
create table myemp (name varchar2(10), hoursworked number);
insert into myemp (name, hoursworked) values ('alison', 200);
insert into myemp (name, hoursworked) values ('kris', 200);
insert into myemp (name, hoursworked) values ('wenji', 200);
commit;

create or replace function vacationdaysleft(p_name in varchar2) return number as
  vdl number;
begin
  -- For every 40 hours worked, you get 1 day of vacation
  select floor(hoursworked / 40) into vdl
    from myemp
   where myemp.name = p_name;
  return vdl;
end;
/
```

Test the function in SQL*Plus by calling it:

```
SQL> select name, vacationdaysleft(name) from myemp;
```

This returns:

NAME	VACATIONDAYSLEFT(NAME)
alison	5
kris	5
wenji	5

In PHP the function might be used like:

Script 95: edition1.php

```
<?php

oci_set_edition('ora$base');
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "begin :vdl := vacationdaysleft(:name); end;");
oci_bind_by_name($s, ":vdl", $vdl, 10);
oci_bind_by_name($s, ":name", $name, 10);
$name = 'alison';
oci_execute($s);

echo "$name has ".$vdl." days vacation left\n";

?>
```

This makes a simple call to the stored function and returns the number of vacation days person \$name has.

The `ora$base` token in `oci_set_edition()` means to use the root or first edition of objects, i.e the PL/SQL function we just created. Although currently redundant because it sets the default edition, the edition name will be changed in the next version of this file to change the version of the `vacationdaysleft()` procedure being called. Instead of calling

Using PL/SQL With OCI8

`oci_set_edition()` you could alternatively assign an edition to an Oracle Net service with the DBMS_SERVICE PL/SQL procedure and set the PHP connection string to that service..

A DBA can query DBA_EDITIONS to see what editions are available in the database. An application user can query `select sys_context('USERENV', 'CURRENT_EDITION_NAME') from dual` to see the edition they are currently accessing.

The script `edition1.php` produces output like:

```
alison has 5 days vacation left
```

This is good. We can put the application into production and employees can start using it.

Now assume the rate used to calculate vacation hours needs to be changed so that for every 30 hours worked, employees get one day of vacation. Also we now want the calculation to include how many days vacation they have already taken.

First, we need a new column to store the vacation they have previously taken. Adding this column won't affect the running PHP application since it doesn't know about it (this is a good reminder never to do `select *` in an application). The new column is created and, for the purpose of this example, populated with some arbitrary values:

Script 96: ed3.sql

```
-- Run as HR
alter table myemp add daysvacationtaken number;

update myemp set daysvacationtaken = 2 where name = 'alison';
update myemp set daysvacationtaken = 3 where name = 'kris';
update myemp set daysvacationtaken = 0 where name = 'wenji';
commit;
```

(For more complex migration scenarios the editioning feature has *crossedition* triggers and *editioning* views to help ensure the appropriate data is used in the old and new editions.)

Now create the new version of the PL/SQL function. In SQL*Plus as HR create a new edition:

Script 97: ed4.sql

```
-- Run as HR
create edition e2;
```

Create the updated version of the PL/SQL procedure. Because the SQL*Plus session is now running in edition E2 this new procedure won't affect the PHP application which is running using edition ORA\$BASE:

Script 98: ed5.sql

```
-- Run as HR

alter session set edition = e2;

create or replace function vacationdaysleft(p_name in varchar2) return number as
  vdl number;
begin
  -- For every 30 hours worked, you get 1 day of vacation
  select floor(hoursworked / 30) - daysvacationtaken into vdl
  from myemp
```

Easy PL/SQL Upgrades With Edition Based Redefinition

```
    where myemp.name = p_name;
  return vdl;
end;
/
```

Querying it shows the updated values:

```
SQL> select name, vacationdaysleft(name) from myemp;
```

NAME	VACATIONDAYSLEFT(NAME)
alison	4
kris	3
wenji	6

This is all done while others users continue to use the PHP application live and get the original results. The new PL/SQL procedure shows Alison now has only 4 days of vacation. You can run the PHP script and check it still returns the "old" value:

```
alison has 5 days vacation left
```

Now copy the PHP file to *edition2.php* and change the `oci_set_edition()` function to set the new edition:

Script 99: edition2.php

```
<?php

oci_set_edition('e2');
$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, "begin :vdl := vacationdaysleft(:name); end;");
oci_bind_by_name($s, ":vdl", $vdl, 10);
oci_bind_by_name($s, ":name", $name, 10);
$name = 'alison';
oci_execute($s);

echo "$name has ".$vdl." days vacation left\n";;

?>
```

When *edition2.php* is run, the output shows the updated calculation of vacation time:

```
alison has 4 days vacation left
```

You can run both *edition1.php* and *edition2.php* concurrently and they will show different results even though the PHP application logic and the PL/SQL procedure name they call is identical.

Oracle Database 11gR2 Edition Based Redefinition helps PHP applications meet the goals of minimal downtime with the frequent upgrade cycle needed by web applications. It allows PHP applications, their often complex stored logic, and large data sets to be updated ready for rolling out in production without impacting the operation of existing users.

Using PL/SQL With OCI8

Database Transactions Across Stateless Web Requests

The DBMS_XA package can be used to start, resume and complete database transactions. Web applications can rely on the database to store the transaction state. If you have multiple mid-tier web servers, this removes the need to track the state of a set of SQL operations. The only shared information needed by each web request is a numeric key identifying the transaction. This key should be secured and made available to the application via PHP's session handling.

Using the TOYS table from *toyshop.sql* (shown in the introduction to packages at the start of this chapter) run SQL*Plus and check the current records:

```
SQL> select * from toys;
```

ID	NAME
2	ball
3	paddling pool
1	bicycle

A DBMS_XA example starts with *xa1.php*:

Script 100: xa1.php

```
<?php

$id = 123;

$c = oci_connect('hr', 'welcome', 'localhost/XE');

// Start a transaction, insert a Teddy Bear, and suspend the transaction

$sql =
"declare
    rc pls_integer;
begin
    rc := dbms_xa.xa_start(dbms_xa_xid(:id), dbms_xa.tmnoflags);
    insert into toys (id, name) values (4, 'teddy bear');
    rc := dbms_xa.xa_end(dbms_xa_xid(:id), dbms_xa.tmsuspend);
end;";

$s = oci_parse($c, $sql);
oci_bind_by_name($s, ":id", $id);
oci_execute($s);

?>
```

This script begins a transaction arbitrarily identified as 123, and inserts a new toy. For sake of the example, the identifier value is hard coded.

Execute *xa1.php*:

```
$ php xa1.php
```

There is no output.

Database Transactions Across Stateless Web Requests

In SQL*Plus check the rows in the table. The new row has not yet been committed and is not visible yet.

The second script, *xa2.php*, resumes transaction number 123, inserts another new row and commits the transaction:

Script 101: *xa2.php*

```
<?php

$id = 123;

$c = oci_connect('hr', 'welcome', 'localhost/XE');

// Resume the transaction, insert a Stethoscope, and commit

$sql =
"declare
  rc pls_integer;
begin
  rc := dbms_xa.xa_start(dbms_xa_xid(:id), dbms_xa_tmresume);
  insert into toys (id, name) values (5, 'stethoscope');
  rc := dbms_xa.xa_commit(dbms_xa_xid(123), true);
end;";

$s = oci_parse($c, $sql);
oci_bind_by_name($s, ":id", $id);
oci_execute($s);

?>
```

Run this script (there is no output) and then use SQL*Plus to query the table:

```
SQL> select * from toys;
```

ID	NAME
2	ball
3	paddling pool
4	teddy bear
5	stethoscope
1	bicycle

This shows that both new rows were inserted, even though the PHP scripts were run in independent processes, with an unknown time frame. If you don't see the teddy bear row, you probably took longer than 60 seconds between running *xa1.php* and *xa2.php*. This is the default timeout before a transaction is aborted. It can be set with DBMS_XA.XA_SETTIMEOUT.

The DBMS_XA package is a convenient way to conduct transactions across the web. It provides functionality to start, suspend, and cancel transactions. The transaction number is the only piece of metadata that needs to be chosen and shared across web requests. This is easily managed using standard PHP techniques such as with PHP sessions.

Using PL/SQL With OCI8

USING LARGE OBJECTS IN OCI8

Oracle Character Large Object (CLOB) and Binary Large Object (BLOB) types can be used for very large amounts of data. They can be used for table columns and as PL/SQL variables. A pre-supplied DBMS_LOB package makes manipulation in PL/SQL easy. OCI8 LOB methods allow storing and fetching LOB data in PHP.

Oracle also has a BFILE type for large objects stored outside the database.

Working With LOBs

In successive versions, the Oracle database has made it easier to work with LOBs. Along the way “Temporary LOBs” were added, and some string-to-LOB conversions are now transparent so data can be handled directly as strings. Develop and test your LOB application with the Oracle client libraries and database that will be used for deployment so you can be sure all the expected functionality is available.

When working with large amounts of data, set *memory_limit* appropriately in *php.ini* otherwise PHP may terminate early. When reading or writing files to disk, check if *open_basedir* allows file access.

These examples show BLOBs. Using CLOBs is almost identical to using BLOBs: the descriptor type becomes *OCI_D_CLOB*, the bind type becomes *OCI_B_CLOB*, and tables must obviously contain a CLOB column.

The examples use a table created in SQL*Plus containing a BLOB column called BLOBDATA:

```
SQL> create table mytab (blobid number primary key, blobdata blob);
```

Note querying BLOB columns in SQL*Plus is not possible unless SQL*Plus 11g is used, where it will display a hexadecimal version of the data. Tables with CLOB columns can be queried in all versions of SQL*Plus. The output of BLOB and CLOB data can be controlled in SQL*Plus with the **SET LONG** command. The default value of 80 means that only the first 80 characters of data will be displayed by a query.

LOB database storage and access options can be configured at table creation time, or with **ALTER TABLE**. Frequently accessed LOBs will benefit from monitoring their use and adjusting their configuration. One common tuning step is to turn on LOB caching:

```
SQL> alter table mytab modify lob (blobdata) (cache);
```

In Oracle 11g you can optionally create LOBs with the storage option SECUREFILE to take advantage of LOB SecureFile features such as deduplication and compression.

Inserting and Updating LOBs

In PHP, LOBs are generally manipulated using a descriptor. PHP code to insert into MYBTAB is:

Script 102: blobinsert.php

```
<?php  
  
$c = oci_connect('hr', 'welcome', 'localhost/XE');
```

Using Large Objects in OCI8

```
$myblobid = 123;
$myv = 'a very large amount of binary data';

$s = oci_parse($c, 'insert into mytab (blobid, blobdata)
                     values (:myblobid, EMPTY_BLOB())
                     returning blobdata into :blobdata');
$lob = oci_new_descriptor($c, OCI_D_LOB);
oci_bind_by_name($s, ':myblobid', $myblobid);
oci_bind_by_name($s, ':blobdata', $lob, -1, OCI_B_BLOB);
oci_execute($s, OCI_NO_AUTO_COMMIT); // Don't commit so $lob->save() works

$lob->save($myv);
oci_commit($c);
$lob->close(); // close LOB descriptor to free resources

?>
```

The `RETURNING` clause returns the Oracle LOB locator of the new row. By binding as `OCI_B_BLOB`, the PHP descriptor in `$lob` references this locator. The `$lob->save()` method then stores the data in `$myv` into the BLOB column. The `OCI_NO_AUTO_COMMIT` flag is used for `oci_execute()` so the descriptor remains valid for the `save()` method. The commit concludes the insert and makes the data visible to other database users.

If the application uploads LOB data using a web form, it can be inserted directly from the upload directory with `$lob->import($filename)`. PHP's maximum allowed size for uploaded files is set in `php.ini` using the `upload_max_filesize` parameter.

To update a LOB, use the same code with this SQL statement:

```
$s = oci_parse($c, 'update mytab set
                     blobdata = empty_blob()
                     returning blobdata into :blobdata');
```

Fetching LOBs

When fetching a LOB, OCI8 returns a LOB descriptor. The data can be retrieved by using a `load()` or `read()` method:

Script 103: blobfetch.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$myblobid = 123;

$query = 'select blobdata from mytab where blobid = :myblobid';
$s = oci_parse ($c, $query);
oci_bind_by_name($s, ':myblobid', $myblobid);
oci_execute($s);
$arr = oci_fetch_array($s, OCI_ASSOC);
if (is_object($arr['BLOBDATA'])) { // protect against a NULL LOB
    $data = $arr['BLOBDATA']->load();
    $arr['BLOBDATA']->free();
```

```

    echo $data;
}

?>
```

It is important to free all returned LOB locators to avoid leaks:

```

while (($arr = oci_fetch_array($s, OCI_ASSOC))) {
    echo $arr['BLOBDATA']->load(); // do something with the BLOB
    $arr['BLOBDATA']->free(); // cleanup before next fetch
}
```

If LOBS are not freed, the ABSTRACT_LOBS column in the V\$TEMPORARY_LOBS table will show increasing values.

Instead of using locators, LOB data can alternatively be returned as a string:

```

$arr = oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_LOBS);
echo $arr['BLOBDATA'];
```

If the returned data is larger than expected, PHP may not be able to allocate enough memory. To protect against this, use a locator with the `read()` method, which allows the data to be fetched in chunks.

When doing SELECT ... FOR UPDATE, use `oci_execute($s, OCI_NO_AUTO_COMMIT)` otherwise you will get an *ORA-01002: fetch out of sequence* error when fetching the LOB.

In LOB-fetching loops it is good practice to free local PHP variables containing LOB data before fetching the next record. This can reduce the overall memory use of PHP, because only one LOB value needs to be held in memory at a time, instead of the original and new values:

```

while (($arr = oci_fetch_array($s, OCI_ASSOC))) {
    $arr = oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_LOBS);
    echo $arr['BLOBDATA']; // do something with the LOB
    unset($arr); // free PHP's memory before fetching the next LOB.
}
```

Temporary LOBs

Temporary LOBs are created and maintained in the database. PHP accesses the LOB there for reading and writing. Temporary LOBs are not persistent and are available only within the current PHP connection. Temporary LOBs make some operations easier.

Inserting data with a Temporary LOB does not use a `RETURNING INTO` clause, for example:

Script 104: tempblobinsert.php

```

<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$myblobid = 124;
$myv = 'a very large amount of binary data';

$s = oci_parse($c, 'insert into mytab (blobid, blobdata)
                     values (:myblobid, :blobdata)');
$lob = oci_new_descriptor($c, OCI_D_LOB);
oci_bind_by_name($s, ':myblobid', $myblobid);
```

Using Large Objects in OCI8

```
oci_bind_by_name($s, ':blobdata', $lob, -1, OCI_B_BLOB);
$lob->writeTemporary($myv, OCI_TEMP_BLOB);
oci_execute($s, OCI_NO_AUTO_COMMIT);
oci_commit($c);
$lob->close(); // close lob descriptor to free resources

?>
```

Temporary LOBs also simplify updating values:

```
$s = oci_parse($c, 'update mytab set blobdata = :bd where blobid = :bid');
```

If you want to either insert a new row or update existing data if the row turns out to exist already, the SQL statement can be changed to use an anonymous block :

```
$s = oci_parse($c,
  'begin'
  . ' insert into mytab (blobdata, blobid) values(:blobdata, :myblobid);'
  . ' exception'
  . ' when dup_val_on_index then'
  . '     update mytab set blobdata = :blobdata where blobid = :myblobid;'
  . 'end;');
```

Uploading and Displaying an Image

The script *image.php* shows how a JPEG image can be uploaded and inserted into the database. After the data is inserted, it queries the image data back immediately and displays the picture to verify the insert and query work. The same MYBTAB table as created above is used.

Script 105: image.php

```
<?php

if (!isset($_FILES['lob_upload'])) {
?>
<h1>BLOB Example - Uploading a JPEG</h1>
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST"
      enctype="multipart/form-data">
JPEG image filename: <input type="file" name="lob_upload">
<input type="submit" value="Upload">
</form>

<?php
} else {
$c = oci_connect('hr', 'welcome', 'localhost/xe');

$myblobid = 1; // should really be a unique id e.g. a sequence number

// Delete any existing BLOB so the query at the bottom
// displays the new data

$sql = 'delete from mytab where blobid = :myblobid';
$s = oci_parse ($c, $sql);
```

```

oci_bind_by_name($s, ':myblobid', $myblobid);
$e = oci_execute($s, OCI_COMMIT_ON_SUCCESS);

// Read and insert the BLOB from PHP's temporary upload area

$lob = oci_new_descriptor($c, OCI_D_LOB);
$sql = 'insert into mybtab (blobid, blobdata) values(:myblobid, :blobdata)';
$s = oci_parse($c, $sql);
oci_bind_by_name($s, ':myblobid', $myblobid);
oci_bind_by_name($s, ':blobdata', $lob, -1, OCI_B_BLOB);
$myv = file_get_contents($_FILES['lob_upload']['tmp_name']);
$lob->writeTemporary($myv, OCI_TEMP_BLOB);
oci_execute($s, OCI_NO_AUTO_COMMIT);
oci_commit($c);
$lob->close();

// Now query back the uploaded BLOB and display it

$sql = 'select blobdata from mybtab where blobid = :myblobid';
$s = oci_parse ($c, $sql);
oci_bind_by_name($s, ':myblobid', $myblobid);
oci_execute($s, OCI_NO_AUTO_COMMIT);
$arr = oci_fetch_assoc($s);
$result = $arr['BLOBDATA']->load();

// If any text (or whitespace!) is printed before this header is sent,
// the text won't be displayed and the image won't display properly.
// Comment out this line to see the text and debug such a problem.

header("Content-type: image/JPEG");
echo $result;
}
?>
```

When the script is initially loaded in a browser, `$_FILES['lob_upload']` is not set and so the HTML upload form is displayed. Submitting this form calls `$_SERVER['PHP_SELF']` which is the same PHP script. It now executes the second part of the `else` block. This deletes any existing image, uploads the new data from PHP's upload area and inserts it. Finally the image is selected and displayed. If there is any text such as whitespace before the `<?php` tag, or any of the OCI8 functions produce an error then the image will not display. To debug this, comment out the `header()` call to see the text being displayed before the image data.

LOBs and PL/SQL procedures

Temporary LOBs can also be used to pass data into PL/SQL IN parameters, or returned from OUT parameters. Given a PL/SQL procedure that accepts a BLOB and inserts it into MYBTAB:

Script 106: inproc.sql

```

create or replace procedure inproc(pid in number, pdata in blob) as
begin
  insert into mybtab (blobid, blobdata) values (pid, pdata);
end;
```

Using Large Objects in OCI8

```
/  
show errors
```

PHP code to pass a BLOB to INPROC would look like:

Script 107: inproc.php

```
<?php  
  
$c = oci_connect('hr', 'welcome', 'localhost/XE');  
  
$myblobid = 125;  
$myv = 'a very large amount of binary data';  
  
$s = oci_parse($c, 'begin inproc(:myblobid, :myblobdata); end;');  
$lob = oci_new_descriptor($c, OCI_D_LOB);  
oci_bind_by_name($s, ':MYBLOBID', $myblobid);  
oci_bind_by_name($s, ':MYBLOBDATA', $lob, -1, OCI_B_BLOB);  
$lob->writeTemporary($myv, OCI_TEMP_BLOB);  
oci_execute($s);  
$lob->close();  
  
?>
```

If the PL/SQL procedure returns a BLOB as an OUT parameter:

Script 108: outproc.sql

```
create or replace procedure outproc(pid in number, pdata out blob) as  
begin  
    select blobdata into pdata from mytab where blobid = pid;  
end;  
/  
show errors
```

PHP code to fetch and display the BLOB would look like:

Script 109: outproc.php

```
<?php  
  
$c = oci_connect('hr', 'welcome', 'localhost/XE');  
  
$myblobid = 125;  
  
$s = oci_parse($c, "begin outproc(:myblobid, :myblobdata); end;");  
$lob = oci_new_descriptor($c, OCI_D_LOB);  
oci_bind_by_name($s, ':MYBLOBID', $myblobid);  
oci_bind_by_name($s, ':MYBLOBDATA', $lob, -1, OCI_B_BLOB);  
oci_execute($s, OCI_NO_AUTO_COMMIT);  
if (is_object($lob)) { // protect against a NULL LOB  
    $data = $lob->load();  
    $lob->free();  
    echo $data;  
}
```

```
?>
```

Other LOB Methods

A number of other methods on the LOB descriptor allow seeking to a specified offset, exporting data directly to file, erasing data, and copying or comparing a LOB.

This code snippet shows seeking to the 10th position in the result descriptor, and then storing the next 50 bytes in `$result`:

```
$arr['BLOBDATA']->seek(10, OCI_SEEK_SET);
$result = $arr['BLOBDATA']->read(50);
```

The LOB buffering methods allow writes to the database to be deferred and then explicitly flushed. This reduces the number of network round-trips to the database, and allows the database to operate more efficiently.

The full list of LOB methods and functions is shown in Table 9. Check the PHP manual for usage.

Table 9: LOB methods and functions.

PHP Function or Method	Action
OCI-Lob->close	Close a LOB descriptor
OCI-Lob->eof	Test for LOB end-of-file
OCI-Lob->erase	Erases a specified part of the LOB
OCI-Lob->export OCI-Lob->writeToFile	Write a LOB to a file
OCI-Lob->flush	Flushes buffer of the LOB to the server
OCI-Lob->free	Frees database resources associated with the LOB
OCI-Lob->getBuffering	Returns current state of buffering for the LOB
OCI-Lob->import OCI-Lob->saveFile	Loads data from a file to a LOB. OCI8 reads the complete file before transferring it to the database.
OCI-Lob->load	Returns LOB contents
OCI-Lob->read	Returns part of the LOB
OCI-Lob->rewind	Moves the LOB's internal pointer back to the beginning
OCI-Lob->save	Saves data to the LOB
OCI-Lob->seek	Sets the LOB's internal position pointer

Using Large Objects in OCI8

PHP Function or Method	Action
OCI-Lob->setBuffering	Changes LOB's current state of buffering
OCI-Lob->size	Returns size of LOB
OCI-Lob->tell	Returns current pointer position
OCI-Lob->truncate	Truncates a LOB
OCI-Lob->write	Writes data to the LOB
OCI-Lob->writeTemporary	Writes a temporary LOB
oci_lob_copy	Copies a LOB
oci_lob_is_equal	Compare two LOB locators for equality

Working With BFILEs

A BFILE is an Oracle large object (LOB) data type for files stored outside the database. BFILEs are a handy way for using relatively static, externally created content. They are also useful for loading text or binary data into Oracle tables.

If you are evaluating features, compare BFILEs with Oracle's External Table feature, not covered in this book.

In SQL and PL/SQL, a BFILE is accessed via a locator, which is simply a pointer to the external file. There are numerous pre-supplied functions that operate on BFILE locators.

To show how BFILEs work in PHP this section creates a sample application that accesses and displays a single image. The image will not be loaded into the database but the picture description is loaded so it can be queried. The BFILE allows the image to be related to the description. Also the application could be extended in future to use PL/SQL packages to read and manipulate the image.

In this example, the image data is not loaded and printed in PHP. Instead, the browser is redirected to the image URL of the external file. This significantly reduces the amount of data that needs to be handled by the application.

To allow Apache to serve the image, edit *httpd.conf* and map a URL to the directory containing the file. For example if the file is */tmp/cj.jpg* add:

```
Alias /tmp/ "/tmp/"
<Directory "/tmp/">
    Options None
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

Using */tmp* like this is not recommended for anything except testing!

Restart Apache and use a browser to check that *http://localhost/tmp/cj.jpg* loads the picture in */tmp/cj.jpg*.

In Oracle, create a DIRECTORY alias for `/tmp`. This is Oracle's pointer to the operating system and forms part of each BFILE. The directory must be on the same machine that the database server runs on. Run SQL*Plus as:

```
$ sqlplus system@localhost/XE @bfile.sql
```

Where `bfile.sql` is:

Script 110: bfile.sql

```
create directory TestDir AS '/tmp';
grant read on directory TestDir to hr;
connect hr/welcome@localhost/XE
create table FileTest (
    FileNum  number primary key,
    FileDesc varchar2(30),
    Image    bfile);
```

This gives the HR user access to the `/tmp` directory and creates a table FILETEST containing a file number identifier, a text description of the file, and the BFILE itself. The image data is not loaded into this table; the BFILE in the table is a pointer to the file on your file system.

PHP code to insert the image name into the FILETEST table looks like:

Script 111: bfileinsert.php

```
<?php
$c = oci_connect("hr", "welcome", "localhost/XE");

$fnum = 1;
$fdsc = "Some description to search";
$name = "cj.jpg";

$s = oci_parse($c, "insert into FileTest (FileNum, FileDesc, Image) "
    . "values (:fnum, :fdsc, bfilename('TESTDIR', :name))");
oci_bind_by_name($s, ":fnum", $fnum, -1, SQLT_INT);
oci_bind_by_name($s, ":fdsc", $fdsc, -1, SQLT_CHR);
oci_bind_by_name($s, ":name", $name, -1, SQLT_CHR);
oci_execute($s, OCI_NO_AUTO_COMMIT);
oci_commit($c);

?>
```

The `bfilename()` constructor inserts into the BFILE-type column using the TESTDIR directory alias created earlier. Bind variables are used for efficiency and security.

This new BFILE can be queried back in PHP:

Script 112: bfilequery1.php

```
<?php
$c = oci_connect("hr", "welcome", "localhost/XE");
$fnum = 1;

$s = oci_parse($c, "select Image from FileTest where FileNum = :fnum");
```

Using Large Objects in OCI8

```
oci_bind_by_name($s, ":fnum", $fnum);
oci_execute($s);
$row = oci_fetch_assoc($s);
$bf = $row['IMAGE'];           // This is a BFILE descriptor
echo "<pre>"; var_dump($bf); echo "</pre>";
?>
```

This displays the BFILE descriptor:

```
object(OCI-Lob)#1 (1) {
    ["descriptor"]=>
        resource(7) of type (oci8 descriptor)
}
```

For simplicity, the query condition is the file number of the new record. In real life it might use a regular expression on the FILEDESC column like:

```
select Image from FileTest where regexp_like(FileDesc, 'somepattern')
```

In this example the file name is needed so the browser can redirect to a page showing the image. Unfortunately there is no direct method in PHP to get the filename from the descriptor. However, an Oracle procedure can be created to do this.

Instead of executing the query in PHP and using PL/SQL to find the filename, a more efficient method is to do both in PL/SQL. Here an anonymous block is used. Alternatively, a procedure could be used.

The previous query code in *bfilequery1.php* can be replaced with:

Script 113: showpic.php

```
<?php

$c = oci_connect("hr", "welcome", "localhost/XE");

$s = oci_parse($c,
    'declare '
    . '    b_1 bfile;'
    . '    da_1 varchar2(255);'
    . 'begin '
    . '    select image into b_1 from filetest where filenum = :fnum;'
    . '    dbms_lob.filegetname(b_1, da_1, :name);'
    . 'end;');
$fnum = 1;
oci_bind_by_name($s, ":fnum", $fnum);
oci_bind_by_name($s, ":name", $name, 255, SQLT_CHR);
oci_execute($s);

header("Location: http://localhost/tmp/$name");

?>
```

The filename *cj.jpg* is returned in `$name` courtesy of the `:name` bind variable argument to the `DBMS_LOB.FILEGETNAME()` function. The `header()` function redirects the user to the image. If any text is printed before the `header()` is output, the HTTP headers will not be correct and the

Working With BFILEs

image will not display. If you have problems, comment out the `header()` call and echo `$name` to check it is valid.

BFILEs are easy to use in PL/SQL because the pre-supplied DBMS_LOB package has a number of useful functions. For example `DBMS_LOB.LOADFROMFILE()` reads BFILE data from the file system into a PL/SQL BLOB or CLOB. This could be loaded into a BLOB table column, manipulated in PL/SQL, or even returned to PHP using OCI8's LOB features. Another example is `DBMS_LOB.FILEEXISTS()`, which can be used to check whether the FILETEST table contains references to images that do not exist.

BFILEs are very useful for many purposes including loading images into the database, but BLOBS may be better in some circumstances. Changes to BFILE locators can be rolled back or committed but since the files themselves are outside the database, BFILE data does not participate in transactions. You can have dangling references to BFILEs because Oracle does not check the validity of BFILEs until the data is explicitly read (this allows you to pre-create BFILEs or to change the physical data on disk). BFILE data files are read-only and cannot be changed within Oracle. Finally, BFILEs need to be backed up manually. Because of these points, there might be times you should use BLOBS to store images inside the database to ensure data and application consistency but BFILEs are there if you want them.

Using Large Objects in OCI8

USING XML WITH ORACLE AND PHP

Both Oracle and PHP 5 have excellent XML capabilities. All editions of Oracle contain what is known as “XML DB”, the XML capabilities of the database. When tables are created, XML can be stored in linear LOB format, or according to the structure of your XML schema.

This chapter covers the basics of using XML data with Oracle and PHP. It also shows how to access data over HTTP directly from the database.

Fetching Relational Rows as XML

One useful feature of XML DB is that existing relational SQL tables can automatically be retrieved as XML:

Script 114: xmlfrag.php

```
<?php  
$c = oci_connect('hr', 'welcome', 'localhost/XE');  
  
$query =  
    'select xmlelement("Employees",  
                      xmlelement("Name", employees.last_name),  
                      xmlelement("Id", employees.employee_id)) as result  
    from employees  
    where employee_id > 200';  
  
$s = oci_parse($c, $query);  
oci_execute($s);  
while ($row = oci_fetch_array($s, OCI_NUM)) {  
    foreach ($row as $item) {  
        echo htmlentities($item)."<br>\n";  
    }  
}  
?  
>
```

The returned values are XML fragments, and not fully formed XML documents. The output is:

```
<Employees><Name>Hartstein</Name><Id>201</Id></Employees>  
<Employees><Name>Fay</Name><Id>202</Id></Employees>  
<Employees><Name>Mavris</Name><Id>203</Id></Employees>  
<Employees><Name>Baer</Name><Id>204</Id></Employees>  
<Employees><Name>Higgins</Name><Id>205</Id></Employees>  
<Employees><Name>Gietz</Name><Id>206</Id></Employees>
```

Using XML With Oracle and PHP

Tip: Watch out for the quoting of XML queries. The XML functions can have embedded double-quotes. This is the exact opposite of standard SQL queries, which can have embedded single quotes. Use a PHP HEREDOC or NOWDOC to help construct queries.

There are a number of other XML functions that can be similarly used. See the *Oracle Database SQL Language Reference*.

Fetching Rows as Fully Formed XML

Another way to create XML from relational data is to use the PL/SQL package DBMS_XMLGEN. This package returns a fully formed XML document, with the XML header.

Queries that use DBMS_XMLGEN return a CLOB column, so the result in PHP needs to be treated as a LOB descriptor. There is effectively no length limit for CLOBs. The following example queries the first name of employees in department 30 and stores the XML marked-up output in `$mylob`:

Script 115: getxml.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$query = "select dbms_xmlgen.getxml(
            select first_name
            from employees
            where department_id = 30) xml
            from dual";

$s = oci_parse($c, $query);
oci_execute($s);
$row = oci_fetch_array($s, OCI_NUM);
$x = $row[0]->load(); // treat result as a LOB descriptor
$row[0]->free();

echo "<pre>\n";
echo htmlentities($x);
echo "</pre>\n";

?>
```

The output in a browser is:

```
<?xml version="1.0"?>
<ROWSET>
<ROW>
<FIRST_NAME>Den</FIRST_NAME>
</ROW>
<ROW>
<FIRST_NAME>Alexander</FIRST_NAME>
</ROW>
```

```
<ROW>
<FIRST_NAME>Shelli</FIRST_NAME>
</ROW>
<ROW>
<FIRST_NAME>Sigal</FIRST_NAME>
</ROW>
<ROW>
<FIRST_NAME>Guy</FIRST_NAME>
</ROW>
<ROW>
<FIRST_NAME>Karen</FIRST_NAME>
</ROW>
</ROWSET>
```

Using the SimpleXML Extension in PHP

You can use PHP's *SimpleXML* extension to convert XML to a PHP object. Following on from the previous example the query results can be converted with:

```
$xo = simplexml_load_string((binary)$x);
```

Note the cast to [binary](#) which ensures consistent encoding. The value in `$xo` is a PHP object:

```
object(SimpleXMLElement)#2 (1) {
  ["ROW"]=>
  array(6) {
    [0]=>
    object(SimpleXMLElement)#3 (1) {
      ["FIRST_NAME"]=>
      string(3) "Den"
    }
    [1]=>
    object(SimpleXMLElement)#4 (1) {
      ["FIRST_NAME"]=>
      string(9) "Alexander"
    }
    [2]=>
    object(SimpleXMLElement)#5 (1) {
      ["FIRST_NAME"]=>
      string(6) "Shelli"
    }
    [3]=>
    object(SimpleXMLElement)#6 (1) {
      ["FIRST_NAME"]=>
      string(5) "Sigal"
    }
    [4]=>
    object(SimpleXMLElement)#7 (1) {
      ["FIRST_NAME"]=>
      string(3) "Guy"
    }
    [5]=>
    object(SimpleXMLElement)#8 (1) {
```

Using XML With Oracle and PHP

```
    ["FIRST_NAME"]=>
        string(5) "Karen"
    }
}
}
```

This object can be accessed with array iterators and properties:

```
foreach ($xo->ROW as $r) {
    echo "Name: " . $r->FIRST_NAME . "<br>\n";
}
```

This output from the loop is:

```
Name: Den
Name: Alexander
Name: Shelli
Name: Sigal
Name: Guy
Name: Karen
```

There are more examples of using SimpleXML with XML data in PHP's test suite under the `ext/simplexml/tests` directory of the PHP source code bundle.

To trap and gracefully handle SimpleXML errors use [libxml_use_internal_errors\(\)](#):

```
<?php
libxml_use_internal_errors(true);
$xo = simplexml_load_string("not XML");
if (!$xo) {
    echo "XML load failed:\n";
    foreach(libxml_get_errors() as $error)
        echo " ", $error->message;
}
?>
```

This gives:

```
XML load failed:
Start tag expected, '<' not found
```

If you comment out [libxml_use_internal_errors\(true\)](#) you would see a normal warning raised:

```
PHP Warning:  simplexml_load_string(): Entity: line 1: parser error :
Start tag expected, '<' not found in xml.php on line 3
```

Fetching XMLType Columns

Data in XMLType columns can be longer than Oracle's 4000 byte string length limit. When data is fetched as a string, queries may fail depending on the data length. For example, the RES column in RESOURCE_VIEW (which is a way to access the Oracle XML DB repository from SQL) is an XMLType:

```
SQL> describe resource_view
      Name          Null?    Type
```

RES	SYS.XMLTYPE(XMLSchema "http://xmlns.oracle.com/xdb/XDBResource.xsd" Element "Resource")
ANY_PATH	VARCHAR2(4000)
RESID	RAW(16)

PHP code to query it is:

```
$s = oci_parse($c, 'select res from resource_view');
oci_execute($s);
while ($row = oci_fetch_array($s, OCI_ASSOC)) {
    var_dump($row);
}
```

This is likely to successfully fetch and display some rows before failing:

```
<Owner>SYS</Owner>
<Creator>SYS</Creator>
<LastModifier>SYS</LastModifier>
<SchemaElement>http://xmlns.oracle.com/xdb/XDBSchema.xsd#binary</SchemaElement>
<Contents>
    <binary>47494638396116001 . . . 003B</binary>
</Contents>
</Resource>
PHP Warning: oci_fetch_array(): ORA-19011: Character string buffer too small
```

The failure happens because the database does a conversion from XMLType to a string before returning results to PHP. When the rows are short there is no error. During testing you could be tricked into thinking your query will always return a complete set of rows.

Use the [XMLTYPE.GETCLOBVAL\(\)](#) function to force XMLType conversion to return a CLOB, avoiding the string size limit problem. Standard OCI8 CLOB methods can be used on the returned data:

Script 116: xmltype.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, 'select xmltype.getclobval(res) from resource_view');
oci_execute($s);
while ($row = oci_fetch_array($s, OCI_NUM)) {
    var_dump($row[0]->load());
    $row[0]->free();
}

?>
```

Using XML With Oracle and PHP

Inserting Into XMLType Columns

You can insert or update XMLType columns by binding as a CLOB.

This warehouse example updates a table without an XMLSchema, and which stores the XMLType column as a CLOB.

Script 117: xmlcreate.sql

```
create table xwarehouses (warehouse_id number, warehouse_spec xmltype)
    xmltype warehouse_spec store as clob;
```

PHP code to insert a warehouse loading dock is:

Script 118: xmlinsert.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

// XML data to be inserted
$xml = <<<EOF
<?xml version="1.0"?>
<Warehouse>
<WarehouseId>1</WarehouseId>
<WarehouseName>Southlake, Texas</WarehouseName>
<Building>Owned</Building>
<Area>25000</Area>
<Docks>2</Docks>
<DockType>Rear load</DockType>
<WaterAccess>true</WaterAccess>
<RailAccess>N</RailAccess>
<Parking>Street</Parking>
<VClearance>10</VClearance>
</Warehouse>
EOF;

// Insert new XML data using a temporary CLOB
$s = oci_parse($c,
    "insert into xwarehouses (warehouse_id, warehouse_spec)
        values (:id, XMLType(:clob))");
$id = 1;
oci_bind_by_name($s, ':id', $id);
$lob = oci_new_descriptor($c, OCI_D_LOB);
oci_bind_by_name($s, ':clob', $lob, -1, OCI_B_CLOB);
$lob->writeTemporary($xml);
oci_execute($s);
$lob->close();

?>
```

PHP code to update the number of available warehouse docks is:

Script 119: xmlupdate.php

```
<?php
```

```

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$s = oci_parse($c, 'select xmltype.getclobval(warehouse_spec)
                      from xwarehouses where warehouse_id = :id');
$id = 1;
$r = oci_bind_by_name($s, ':id', $id);
oci_execute($s);
$row = oci_fetch_array($s, OCI_NUM);

// Manipulate the data using SimpleXML
$sx = simplexml_load_string((binary)$row[0]->load());
$row[0]->free();

$sx->Docks -= 1; // change the data

// Insert changes using a temporary CLOB
$s = oci_parse($c, 'update xwarehouses
                      set warehouse_spec = XMLType(:clob)
                      where warehouse_id = :id');
oci_bind_by_name($s, ':id', $id);
$lob = oci_new_descriptor($c, OCI_D_LOB);
oci_bind_by_name($s, ':clob', $lob, -1, OCI_B_CLOB);
$lob->writeTemporary($sx->asXml());
oci_execute($s);
$lob->close();

?>

```

The `$sx->asXml()` method converts the SimpleXML object to the text representation used to update the table. A temporary LOB is created to pass the new XML value to the database.

After running the PHP script, querying the record in SQL*Plus shows the number of docks has been decremented from 2 to 1:

```

SQL> set long 1000 pagesize 100
SQL> select warehouse_spec from xwarehouses;

```

This gives:

```

WAREHOUSE_SPEC
-----
<?xml version="1.0"?>
<Warehouse>
  <WarehouseId>1</WarehouseId>
  <WarehouseName>Southlake, Texas</WarehouseName>
  <Building>Owned</Building>
  <Area>25000</Area>
  <Docks>1</Docks>
  <DockType>Rear load</DockType>
  <WaterAccess>true</WaterAccess>
  <RailAccess>N</RailAccess>
  <Parking>Street</Parking>
  <VClearance>10</VClearance>
</Warehouse>

```

Using XML With Oracle and PHP

See *Using XML in SQL Statements* in the *Oracle Database SQL Reference* for more discussion of XMLType.

Fetching an XMLType from a PL/SQL Function

The [GETCLOBVAL\(\)](#) function is also useful when trying to get an XMLType from a stored PL/SQL function. File *xmlfunc.sql* creates a simple PL/SQL function returning XMLType query data for a given identifier value.

Script 120: xmlfunc.sql

```
drop table mytab;
create table mytab (id number, data xmltype);
insert into mytab (id, data) values (1, '<something>mydata</something>');

create or replace function myf(p_id number) return xmltype as
  loc xmltype;
begin
  select data into loc from mytab where id = p_id;
  return loc;
end;
/
```

To access this function in PHP, use [GETCLOBVAL\(\)](#) and bind a LOB descriptor to the return value. OCI8 LOB methods like [load\(\)](#) can be used on the descriptor:

Script 121: xmlfunc.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$bd = oci_new_descriptor($c, OCI_D_LOB);
$s = oci_parse($c, "begin :bv := myf(1).getclobval(); end;");
oci_bind_by_name($s, ":bv", $bd, -1, OCI_B_CLOB);
oci_execute($s);

echo htmlentities($bd->load()); // Print output
$bd->close();

?>
```

The output is the expected:

```
<something>mydata</something>
```

XQuery XML Query Language

Oracle's support for XQuery was introduced in Oracle Database 10g Release 2. A basic XQuery to return the records from the EMPLOYEES table is:

```
for $i in ora:view("employees") return $i
```

In use, this XQuery syntax is embedded in a special `SELECT`:

```
select column_value from xmltable('for $i in ora:view("employees") return $i')
```

The different quoting styles used by SQL and XQuery need careful attention in PHP. It can be coded:

Script 122: xquery.php

```
<?php

$c = oci_connect('hr', 'welcome', 'localhost/XE');

$xq = 'for $i in ora:view("employees") return $i';
$query = "select column_value from xmltable('.$xq.')";

$s = oci_parse($c, $query);
oci_execute($s);
while ($row = oci_fetch_array($s, OCI_NUM)) {
    foreach ($row as $item) {
        echo htmlentities($item)." ";
    }
}
?>
```

The query could also be in a single PHP NOWDOC:

```
$query = <<<'END'
select column_value from xmltable('for $i in ora:view("employees") return $i')
END';
```

Note there cannot be whitespace on the last line before or after the token `END`;

Prior to PHP 5.3, use HEREDOC syntax and escape XQuery variables like `$i` with a backslash.

Whichever query format you choose, table rows are automatically wrapped in tags and returned:

```
<ROW>
    <EMPLOYEE_ID>100</EMPLOYEE_ID>
    <FIRST_NAME>Steven</FIRST_NAME>
    <LAST_NAME>King</LAST_NAME>
    <EMAIL>SKING</EMAIL>
    <PHONE_NUMBER>515.123.4567</PHONE_NUMBER>
    <HIRE_DATE>1987-06-17</HIRE_DATE>
    <JOB_ID>AD_PRES</JOB_ID>
    <SALARY>24000</SALARY>
    <DEPARTMENT_ID>90</DEPARTMENT_ID>
</ROW>
...
<ROW>
    <EMPLOYEE_ID>206</EMPLOYEE_ID>
    <FIRST_NAME>William</FIRST_NAME>
    <LAST_NAME>Gietz</LAST_NAME>
    <EMAIL>WGIETZ</EMAIL>
```

Using XML With Oracle and PHP

```
<PHONE_NUMBER>515.123.8181</PHONE_NUMBER>
<HIRE_DATE>1994-06-07</HIRE_DATE>
<JOB_ID>AC_ACCOUNT</JOB_ID>
<SALARY>8300</SALARY>
<MANAGER_ID>205</MANAGER_ID>
<DEPARTMENT_ID>110</DEPARTMENT_ID>
</ROW>
```

You can also use [RETURNING CONTENT](#) to return a single document node:

```
$query = <<<'END'
select xmlquery('for $i in ora:view("hr", "locations")/ROW
                  return $i/CITY'
                  returning content) from dual
END;
```

For both [XMLTABLE\(\)](#) and [XMLQUERY\(\)](#) PHP OCI8 will generate an *ORA-19011: Character string buffer too small* error for results sets bigger than a several thousand bytes. To prevent this, use the [XMLTYPE.GETCLOBVAL\(\)](#) function, for example:

```
$query = <<<'END'
select xmldocument.getclobval(column_value)
from xmldocument('for $i in ora:view("employees") return $i')
END;
```

and

```
$query = <<<'END'
select xmldocument.getclobval(xmlquery('for $i in ora:view("hr",
                                         "locations")/ROW return $i/CITY'
                                         returning content)) from dual
END;
```

The returned data is a LOB locator and so the fetch needs LOB methods, such as [load\(\)](#):

```
$s = oci_parse($c, $query);
oci_execute($s);
while ($row = oci_fetch_array($s, OCI_NUM)) {
    var_dump($row[0]->load());
    $row[0]->free();
}
```

Accessing Data Over HTTP With XML DB

XML DB allows you to access data directly via HTTP, FTP or WebDAV. The Oracle Network listener will handle all these requests. As an example of HTTP access, use the PL/SQL DBMS_XDB package to create a resource, which here is some simple text:

Script 123: *xdb.sql*

```
declare
  res boolean;
begin
  begin
    -- delete if it already exists
```

Accessing Data Over HTTP With XML DB

```
dbms_xdb.deleteResource('/public/test1.txt');
exception when others then
  null;
end;
-- create the file - don't forget to commit
res := dbms_xdb.createResource('/public/test1.txt', 'the text to store');
commit;
end;
/
```

For testing, remove access control on the public resource:

```
SQL> connect system/systempwd
SQL> alter user anonymous identified by anonymous account unlock;
```

The file can now be accessed from a browser (or PHP application) using:

<http://localhost:8080/public/test1.txt>

You may need to enter database credentials such as HR and HR's password when prompted in the browser.

There is extensive Oracle documentation on XML DB in the Oracle manuals, and on the Oracle Technology network at

<http://www.oracle.com/technetwork/database/features/xmlldb/>.

Using XML With Oracle and PHP

PHP CONNECTION POOLING AND HIGH AVAILABILITY

This chapter discusses two database features supported by PHP OCI8 that improve scalability and high availability:

- Database Resident Connection Pooling (DRCP): a feature introduced in Oracle Database 11g that addresses scalability requirements in environments requiring large numbers of connections while using minimal database resources. An Oracle benchmark showed 20,000 connections being supported on a small commodity machine with 2GB of RAM.
- Fast Application Notification (FAN): Web applications that run in high availability configurations such as with Oracle Real Application Clusters (RAC) or Data Guard Physical Stand-By can take advantage of FAN events in PHP to allow applications to respond quickly to database node failures.

The features are usable separately or together. PHP 5.3 onwards (OCI8 1.3 onwards) has immediate support for them.

Database Resident Connection Pooling

Oracle Database 11g DRCP addresses scalability requirements in environments requiring large numbers of connections while using minimal database resources. DRCP allows pooling of a set of dedicated database server processes (known as *pooled servers*), which can be shared across multiple applications running on the same or several hosts. A connection broker process manages the pooled servers at the database instance level. DRCP is a configurable feature chosen at program runtime, allowing traditional and DRCP-based connection architectures to be in concurrent use.

Without DRCP, each PHP process creates and destroys database servers when connections are opened and closed. This can be expensive and crippling to high scalability. Or alternatively, PHP processes use persistent connections, keeping connections open even when they are not processing any user scripts. This removes connection creation and destruction costs but incurs unnecessary memory overhead in the database, as shown in Figure 70.

PHP Connection Pooling and High Availability

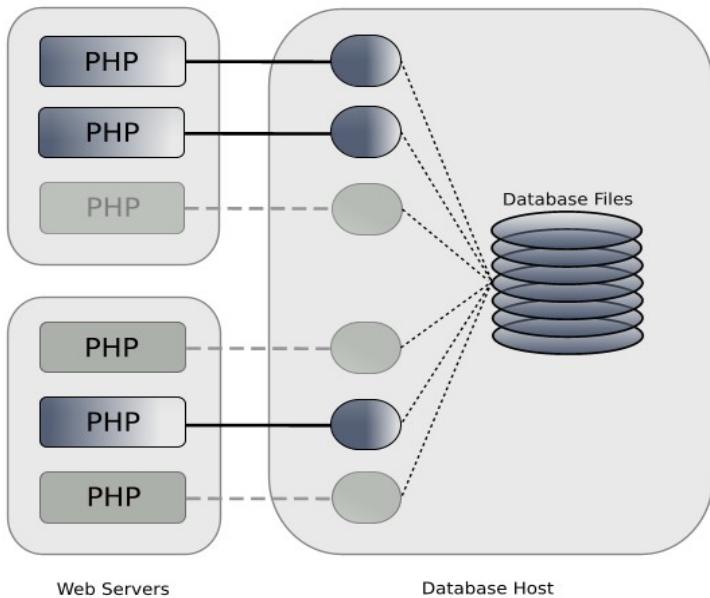


Figure 70: Without DRCP, idle persistent connections from PHP still consume database resources.

How DRCP Works

The architecture of DRCP is shown in Figure 71. A connection broker accepts incoming connection requests from PHP processes and assigns each a free server in the DRCP pool. Each PHP process that is executing a PHP script communicates with this Oracle server until the connection is released. This release can be explicit with `oci_close()` or it will happen automatically at the end of the script. When the connection is released, the server process is returned to the pool and the PHP process keeps a link only to the connection broker. Active pooled servers contain the Process Global Area (PGA) and the user session data. Idle servers in the pool can optionally retain the user session for reuse by subsequent persistent PHP connections.

When the number of persistent connections is less than the number of pooled servers, a “dedicated optimization” avoids unnecessarily returning servers to the pool when a PHP connection is closed. Instead, the dedicated association between the PHP process and the server is kept in anticipation that the PHP process will quickly become active again. If PHP scripts are executed by numerous web servers, the DRCP pool can grow to its maximum size (albeit typically a relatively small size), even if the rate of incoming user requests is low. Each PHP process, either busy or now idle, will be attached to its own pooled server. When the pool reaches its maximum size, another PHP process that needs a pooled server will cause an idle server in the pool to be made available for immediate reuse.

Database Resident Connection Pooling

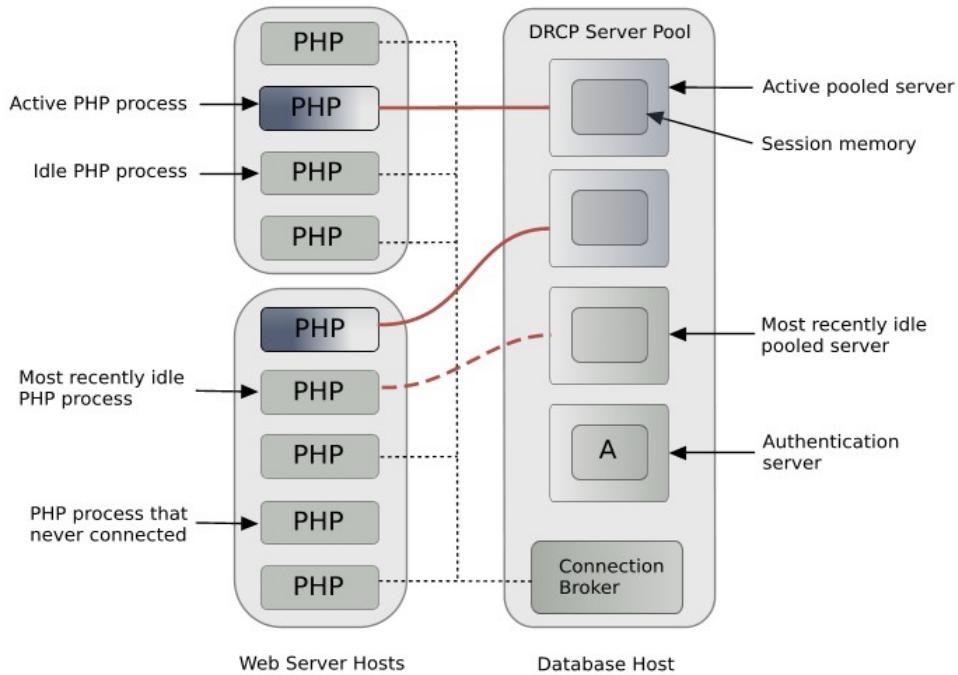


Figure 71: DRCP Architecture.

The pool size and number of connection brokers are configurable. There is always at least one connection broker per database instance when DRCP is enabled. Also, at any time, around 5% of the current pooled servers are reserved for authenticating new PHP connections.

Authentication is performed when a PHP process establishes a connection to the connection broker.

DRCP boosts the scalability of the database and the web server tier because connections to the database are held at minimal cost. Database memory is only used by the pooled servers, and scaling can be explicitly controlled by DRCP tuning options.

With the introduction of pooled servers used by DRCP, there are now three types of database server process models that Oracle applications can use: dedicated servers, shared servers and pooled servers.

Table 10: Differences between dedicated servers, shared servers, and pooled servers.

Dedicated Servers	Shared Servers	Pooled Servers
When the PHP connection is created, a network connection to a dedicated server process and associated session are created.	When the PHP connection is created, a network connection to the dispatcher process is established. A session is created in the SGA.	When the PHP connection is created, a network connection to the connection broker is established.

PHP Connection Pooling and High Availability

Dedicated Servers	Shared Servers	Pooled Servers
Activity on a connection is handled by the dedicated server.	Each action on a connection goes through the dispatcher, which hands the work to a shared server.	Activity on a connection wakes the broker, which hands the network connection to a pooled server. The server then handles subsequent requests directly, just like a dedicated server.
Scripts executing but with idle PHP connections hold a server process and session resources.	Scripts executing but with idle PHP connections hold session resources but not a server process.	Scripts executing but with idle PHP connections hold a server process and session resources.
Closing a PHP connection causes the session to be freed and the server process to be terminated.	Closing a PHP connection causes the session to be freed.	Closing a PHP connection optionally causes the session to be destroyed. The pooled server is released to the pool. A network connection to the connection broker is retained.
Memory usage is proportional to the number of server processes and sessions. There is one server and one session for each PHP connection.	Memory usage is proportional to the sum of the shared servers and sessions. There is one session for each PHP connection.	Memory usage is proportional to the number of pooled servers and their sessions. There is one session for each pooled server.

Pooled servers used by PHP are similar in behavior to dedicated servers. After connection, PHP directly communicates with the pooled server for all database operations.

PHP OCI8 Connections and DRCP

The implementation of the connection functions, [oci_connect\(\)](#), [oci_new_connect\(\)](#), and [oci_pconnect\(\)](#) was reworked in OCI8 1.3. All three benefit from using DRCP. Table 11 compares dedicated and pooled servers. Shared servers are not shown but behave similarly to dedicated servers with the exception that only the session and not the server is destroyed when a connection is closed.

Database Resident Connection Pooling

Table 11: Behavior of OCI8 connection functions for Dedicated and Pooled Servers.

OCI8 Function	Dedicated Servers	Pooled Servers
oci_connect()	Creates a PHP connection to the database using a dedicated server. The connection is cached in the PHP process for reuse by subsequent <code>oci_connect()</code> calls in the same script. At the end of the script or with <code>oci_close()</code> , the connection is closed and the server process and session are destroyed.	Gets a pooled server from the DRCP pool and creates a brand new session. Subsequent <code>oci_connect()</code> calls in the same script use the same connection. When the script completes, or <code>oci_close()</code> is called, the session is destroyed and the pooled server is available for other PHP connections to use.
oci_new_connect()	Similar to <code>oci_connect()</code> above, but an independent new PHP connection and server process is created every time this function is called, even within the same script. All PHP connections and the database servers are closed when the script ends or with <code>oci_close()</code> . Sessions are destroyed at that time.	Similar to <code>oci_connect()</code> above, but an independent server in the pool is used and a new session is created each time this function is called in the same script. All sessions are destroyed at the end of the script or with <code>oci_close()</code> . The pooled servers are made available for other connections to use.
oci_pconnect()	Creates a persistent PHP connection which is cached in the PHP process. The database connection is not closed at the end of the script. When no script is executing, an idle PHP process still holds the server process and session resource. The server and session are available for reuse by subsequent <code>oci_pconnect()</code> calls that pass the same credentials in any script handled by this PHP process.	Creates a persistent PHP connection. Calling <code>oci_close()</code> releases the connection. This makes the server and its intact session available in the pool for reuse by other PHP processes. If <code>oci_close()</code> is not called, then this connection release happens at the end of the script. When no script is executing, an idle PHP process retains only an authenticated network connection to the broker. Subsequent <code>oci_pconnect()</code> calls passing the same credentials in scripts handled by this PHP process reuse the existing network connection to quickly get a server and session from the pool

PHP Connection Pooling and High Availability

With DRCP, all three connection functions save on the cost of authentication and benefit from the network connection to the connection broker being maintained, even for connections that are “closed” from PHP’s point of view. They also benefit from having pre-spawned server processes in the DRCP pool.

The `oci_pconnect()` function reuses sessions, allowing even greater scalability. The non-persistent connection functions create and destroy new sessions each time they are used, allowing less sharing at the cost of reduced performance.

Overall, after a brief warm-up period for the pool, DRCP allows reduced connection times in addition to the reuse benefits of pooling.

When to use DRCP

DRCP is typically preferred for PHP applications with a large number of connections. Shared servers can be useful for a medium number of connections if DRCP is not available. Dedicated sessions are preferred for small numbers of connections. The threshold sizes are relative to the amount of memory available on the database host.

DRCP provides the following advantages:

- It enables resource sharing among multiple client applications and multiple middle-tier application servers.
- It improves scalability of databases and applications by reducing resource usage on the database host.

DRCP can be used if:

- PHP applications mostly use the same database credentials for all connections.
- The applications acquire a database connection, work on it for a relatively short duration, and then release it.
- Connections look identical in terms of session settings, for example date format settings and PL/SQL package state.

These are all typically true for PHP applications.

For persistent PHP connections, normal dedicated servers can be fastest. There is no broker or dispatcher overhead. The database server process is always connected and available whenever the PHP process needs it. But as the number of connections increases, the memory cost of keeping connections open quickly reduces efficiency of the database system.

For non-persistent PHP connections, DRCP can be fastest because the use of pooled server processes removes the need for PHP connections to create and destroy processes, and removes the need to re-authenticate for each connect call.

Consider an application in which the memory required for each session is 400 KB. On a 32 bit operating system the memory required for each server process could be, for example, 4 MB, and DRCP could use 35 KB per connection (mostly in the connection broker). If the number of pooled servers is configured at 100, the number of shared servers is configured at 100, and the deployed application creates 5000 PHP connections, then the memory used by each type of server is estimated in Table 10.

Database Resident Connection Pooling

Table 12: Example database host memory use for dedicated, shared and pooled servers.

	Dedicated Servers	Shared Servers	Pooled Servers
Database Server Memory	5000 * 4 MB	100 * 4 MB	100 * 4 MB
Session Memory	5000 * 400 KB	5000 * 400 KB Note: For Shared Servers, session memory is allocated from the SGA.	100 * 400 KB
DRCP Connection Broker Overhead			5000 * 35 KB
Total Memory	21 GB	2.3 GB	610 MB

There is a significant memory saving when using DRCP.

Even if sufficient memory is available to run in dedicated mode, DRCP can still be a viable option if the PHP application needs database connections for only short periods of time. In this case the memory saved by using DRCP can be used towards increasing the SGA, thereby improving overall performance.

Pooling is available when connecting over TCP/IP with userid/password based database authentication. It is not available using Oracle's "bequeath" connections.

With Oracle 11.2, pooled connections can take advantage of Oracle's Client Result Cache feature.

Sharing the Server Pool

DRCP guarantees that pooled servers and sessions initially used by one database user are only ever reusable by connections with that same user identifier. DRCP also further partitions the pool into logical groups or "connection classes". A connection class is a user chosen name set with `oci8.connection_class` in the `php.ini` configuration file.

Session-specific attributes, like the date format or an explicit role, may be re-usable by any connection in a particular application. Subsequent persistent connections will reuse the session and inherit those settings if the username and connection class are the same as the previous connection.

Applications that need different state in the session memory should use different usernames and/or connection classes.

PHP Connection Pooling and High Availability

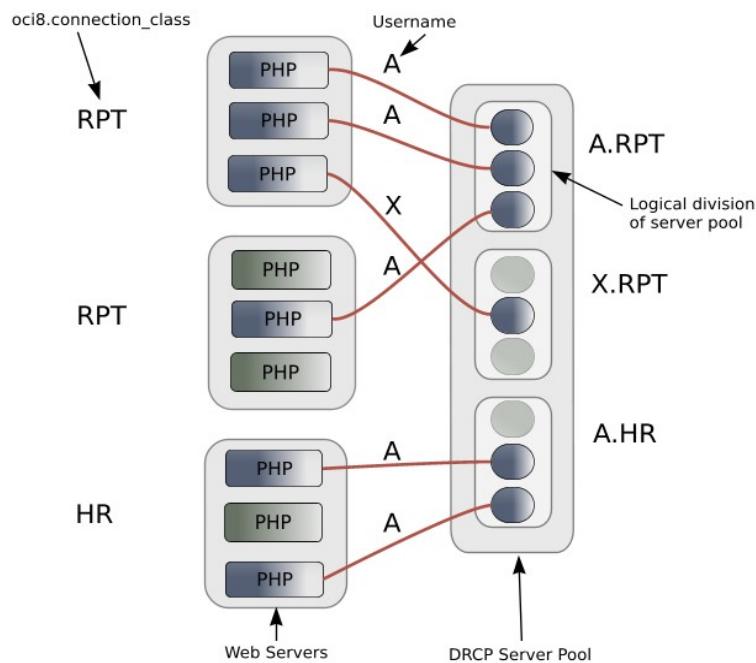


Figure 72: The DRCP pool is logically partitioned by username and connection class.

For example, applications in a suite called RPT may be willing to share pooled servers between themselves but not with applications in a suite called HR. An example of different connection classes and the resulting logical partitioning of the DRCP server pool is shown in the previous figure. Connections with the same username and connection class from any host will share the same sub-pool of servers. If there are no free pooled servers matching a request for a userid in the specified connection class, and if the pool is already at its maximum size, then an idle server in the pool will be used and a new session created for it. If the server originally belonged to a different connection class, the current session will be destroyed, the server will migrate to the new class, and a new session will be created. If there are no pooled servers available, the connection request waits for one to become available. This allows the database to continue without becoming overloaded.

The connection class should be set to the same value for each instance of PHP running the same application where sharing of pooled connections is desired. If no connection class is specified, each web server process will have a unique, system generated class name, limiting sharing of connections to each process, and affecting overall performance.

If DRCP is used but session sharing is not desirable under any condition, use `oci_connect()` or `oci_new_connect()` which recreate the session each time.

Although session data may be reused by subsequent persistent connections, transactions do not span connections across scripts. Uncommitted data will be rolled back at the end of a PHP script.

Using DRCP in PHP

Using DRCP with PHP applications involves the following steps:

1. Configuring and enabling the pool.
2. Configuring PHP.
3. Deploying the application.

PHP applications deployed as Apache modules, FastCGI, CGI and standalone applications can benefit from DRCP. PHP applications deployed as Apache modules or with FastCGI gain most, since the PHP processes are long running and remain connected to the connection broker over multiple script executions. This lets them take advantage of other optimizations, such as statement caching.

Configuring and Enabling the Pool

Every instance of Oracle Database 11g uses a single, default connection pool. User defined pools are currently not supported. The default pool can be configured and administered by a DBA using the DBMS_CONNECTION_POOL package:

```
SQL> execute dbms_connection_pool.configure_pool(
      pool_name          => 'SYS_DEFAULT_CONNECTION_POOL',
      minsize            => 4,
      maxsize            => 40,
      incrsize           => 2,
      session_cached_cursors => 20,
      inactivity_timeout    => 300,
      max_think_time       => 600,
      max_use_session      => 500000,
      max_lifetime_session  => 86400)
```

Alternatively the method `dbms_connection_pool.alter_param()` can be used to set a single parameter:

```
SQL> execute dbms_connection_pool.alter_param(
      pool_name    => 'SYS_DEFAULT_CONNECTION_POOL',
      param_name   => 'MAX_THINK_TIME',
      param_value  => '1200')
```

There is a `dbms_connection_pool.restore_defaults()` procedure to reset all values.

When DRCP is used with RAC, each database instance has its own connection broker and pool of servers. Each pool has the identical configuration. For example, all pools start with MINSIZE server processes. A single DBMS_CONNECTION_POOL command will alter the pool of each instance at the same time.

The pool needs to be started before connection requests begin. The command below does this by bringing up the broker, which registers itself with the database listener:

```
SQL> execute dbms_connection_pool.start_pool()
```

Once enabled this way, the pool automatically restarts when the instance restarts, unless explicitly stopped with the `dbms_connection_pool.stop_pool()` command:

```
SQL> execute dbms_connection_pool.stop_pool()
```

The pool can't be stopped while connections are open. If the PHP application uses persistent connections, then the web server should be stopped before shutting the pool down. Similarly users cannot be dropped unless the web server is first stopped.

PHP Connection Pooling and High Availability

The DRCP configuration options are described in the next table:

Table 13: DRCP Configuration Options.

DRCP Option	Description
pool_name	The pool to be configured. Currently the only supported name is the default value SYS_DEFAULT_CONNECTION_POOL.
minsize	Minimum number of pooled servers in the pool. The default is 4.
maxsize	Maximum number of pooled servers in the pool. If this limit is reached and all the pooled servers are busy, then connection requests wait until a server becomes free. The value should be less than the database <i>init.ora</i> values of sessions and processes. The default value is 40.
incrsize	The number of pooled servers is increased by this value when servers are unavailable for PHP connections and if the pool is not yet at its maximum size. The default is 2.
session_cached_cursors	Indicates to turn on SESSION_CACHED_CURSORS for all connections in the pool. This value is typically set to the size of the working set of frequently used statements. The cache uses cursor resources on the server. The default is 20. Note: there is also an <i>init.ora</i> parameter for setting the value for the whole database instance. The pool option allows a DRCP-based application to override the instance setting.
inactivity_timeout	Time to live for an idle server in the pool. If a server remains idle in the pool for this time, it is killed. This parameter helps to shrink the pool when it is not used to its maximum capacity. The default is 300 seconds.
max_think_time	Maximum time of inactivity the PHP script is allowed after connecting. If the script does not issue a database call for this amount of time, the pooled server may be returned to the pool for reuse. The PHP script will get an ORA error if it later tries to use the connection. The default is 120 seconds.
max_use_session	Maximum number of times a server can be taken and released to the pool before it is flagged for restarting. The default is 500000.
max_lifetime_session	Time to live for a pooled server before it is restarted. The default is 86400 seconds.
num_cbrok	The number of connection brokers that are created to handle connection requests. The default is 1. Note: this can only be set with alter_param() .

DRCP Option	Description
maxconn_cbrok	The maximum number of connections that each connection broker can handle. Set the per-process file descriptor limit of the operating system sufficiently high so that it supports the number of connections specified. The default is 40000. Note: this can only be set with alter_param() .

The parameters have been described here relative to their use in PHP but it is worth remembering that the DRCP pool is usable concurrently by other programs, including those using Perl's DBD::Oracle and Python's cx_Oracle extensions.

In general, if pool parameters are changed, the pool should be restarted, otherwise server processes will continue to use old settings.

The *inactivity_timeout* setting terminates idle pooled servers, helping optimize database resources. To avoid pooled servers permanently being held onto by a dead web server process or a selfish PHP script, the *max_think_time* parameter can be set. The parameters *num_cbrok* and *maxconn_cbrok* can be used to distribute the persistent connections from the clients across multiple brokers. This may be needed in cases where the operating system per-process descriptor limit is small.

Some customers have found that having several connection brokers improves performance.

The *max_use_session* and *max_lifetime_session* parameters help protect against any unforeseen problems affecting server processes. The default values will be suitable for most users.

Users of Oracle Database 11.1.0.6 must apply the database patch for bug 6474441 to avoid query errors. It also enables LOGON trigger support. This patch is not needed with 11.2 or 11.1.0.7 onwards.

Configuring PHP for DRCP

PHP must be built with the OCI8 1.3 or later extension. (PHP 5.3 contains OCI8 1.4).

DRCP functionality is only available when PHP OCI8 is linked with Oracle Database 11g client libraries and connected to Oracle Database 11g.

Before using DRCP, the new *php.ini* parameter *oci8.connection_class* should be set to specify the connection class used by all the requests for pooled servers by the PHP application.

```
oci8.connection_class = MYPHPAPP
```

The parameter can be set in *php.ini*, *.htaccess* or *httpd.conf* files. It can also be set and retrieved programmatically using the PHP functions [ini_set\(\)](#) and [ini_get\(\)](#).

The OCI8 extension has several legacy *php.ini* configuration parameters for tuning persistent connections. These were mainly used to limit idle resource usage. With DRCP, the parameters still have an effect but it may be easier to use the DRCP pool configuration options.

PHP Connection Pooling and High Availability

Table 14: Existing php.ini parameters for persistent connections.

php.ini Parameter	Behavior with DRCP
oci8.persistent_timeout	At the timeout of an idle PHP connection, PHP will close the Oracle connection to the broker. The default is no timeout.
oci8.max_persistent	The maximum number of unique persistent connections that each PHP process will maintain to the broker. When the limit is reached a new persistent connection behaves like <code>oci_connect()</code> and releases the connection at the end of the script. The default is no limit. Note: The DRCP <code>maxsize</code> setting will still be enforced by the database independently from <code>oci8.max_persistent</code> .
oci8.ping_interval	The <code>oci8.ping_interval</code> value is also used for non-persistent connections when DRCP is used. The default is 60 seconds.

With `oci8.ping_interval`, the non-DRCP recommendation to set it to -1 to disable pinging, and to use appropriate error checking still holds true with DRCP. Also, the use of FAN (see later) reduces the chance of idle connections becoming unusable.

Web servers and the network should benefit from `oci8.statement_cache_size` being set. For best performance it should generally be larger than the size of the working set of SQL statements. To tune it, monitor general web server load and the AWR “**bytes sent via SQL*Net to client**” values. The latter statistic should benefit from not needing to send statement meta-data to PHP. Adjust the statement cache size to your satisfaction.

Once you are happy with the statement cache size, then tune the DRCP pool `session_cached_cursors` value. Monitor AWR reports with the goal to make the “**session cursor cache hits**” close to the number of soft parses. Soft parses can be calculated from “**parse count (total)**” minus “**parse count (hard)**”.

Application Deployment for DRCP

PHP applications must specify the server type POOLED in the connect string. Using Oracle’s Easy Connect syntax, the PHP call to connect to the *sales* database on *myhost* would look like:

```
$c = oci_pconnect('myuser', 'mypassword', 'myhost/sales:POOLED');
```

Or if PHP uses an Oracle Network connect name that looks like:

```
$c = oci_pconnect('myuser', 'mypassword', 'salespool');
```

Then only the Oracle Network configuration file *tnsnames.ora* needs to be modified:

```
salespool=(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
    (HOST=myhost.dom.com)
    (PORT=1521))(CONNECT_DATA=(SERVICE_NAME=sales)
    (SERVER=POOLED)))
```

If these changes are made and the database is not actually configured for DRCP, or the pool is not started, then connections will not succeed and an error will be returned to PHP.

Although applications can choose whether or not to use pooled connections at runtime, care must be taken to configure the database appropriately for the number of expected connections, and also to stop inadvertent use of non-pooled connections leading to a resource shortage.

Closing Connections

PHP scripts that do not currently use `oci_close()` should be examined to see if they can use it to explicitly return connections to the pool, allowing maximum use of pooled servers:

```
// 1. Do some database operations
$c = oci_pconnect('myuser', 'mypassword', 'myhost/sales:POOLED');
. .
oci_commit($c);
oci_close($c); // Release the connection to the DRCP pool

// 2. Do lots of non-database work
. .

// 3. Do some more database operations
$c = oci_pconnect('myuser', 'mypassword', 'myhost/sales:POOLED');
. .
oci_commit($c);
oci_close($c);
```

Remember to free statement and other resources that internally increase the reference count on the PHP connection and will otherwise stop a database connection from closing.

Prior to OCI8 1.3, closing `oci_connect()` and `oci_new_connect()` connections had an effect but closing an `oci_pconnect()` connection was a no-op. From OCI8 1.3, `oci_close()` on a persistent connection rolls back any uncommitted transaction. Also the extension will do a rollback when all PHP variables referencing a persistent connection go out of scope, for example if the connection was opened in a function and the function has now finished. For DRCP, in addition to the rollback, the connection is also released; a subsequent `oci_pconnect()` may get a different connection. For DRCP, the benefit is that scripts taking advantage of persistent connections can explicitly return a server to the pool when non-database processing occurs, allowing other concurrent scripts to make use of the pooled server.

With pooled servers, the recommendation is to release the connection when the script does a significant amount of processing that is not database related. Explicitly control commits and rollbacks so there is no unexpectedly open transaction when the close or end-of-scope occurs. Scripts coded like this can use `oci_close()` to take advantage of DRCP but still be portable to older versions of the OCI8 extension.

If behavior where `oci_close()` is a no-op for all connection types is preferred, set the `php.ini` parameter `oci8.old_oci_close_semantics` to On.

Transactions Across Re-connection

Scripts should avoid re-opening connections if there are incomplete transactions:

```
// 1. Do some database operations
$c = oci_pconnect('myuser', 'mypassword', 'salespool');
// Start a transaction
$s = oci_parse($c, 'insert into mytab values (1)');
```

PHP Connection Pooling and High Availability

```
$r = oci_execute($s, OCI_NO_AUTO_COMMIT); // no commit
...
// BAD: no commit or rollback done

// 2. Continue database operations on same credentials
$c = oci_pconnect('myuser', 'mypassword', 'salespool');
$s = oci_parse($c, 'insert into mytab values (2)');
$r = oci_execute($s, OCI_NO_AUTO_COMMIT); // no commit
// Intend to commit both 1 & 2 but behavior could be random
oci_commit($c);
```

If there was a node or network failure just prior to point 2, the first transaction could be lost. The second connection command may return a new, valid connection if a ping (see `oci8.ping_interval`) occurs to validate the connection, and the script might not be aware that only the second part of the transaction is committed.

The script should do an explicit commit or rollback before the second connect, or simply continue to use the original connection. It should always do appropriate error handling.

LOGON and LOGOFF Triggers with DRCP

LOGON triggers are useful for setting session attributes needed by each PHP connection. For example a trigger could be used to execute an `ALTER SESSION` statement to set a date format. The LOGON trigger will execute when `oci_pconnect()` first creates the session, and the session will be reused by subsequent persistent connections. Scripts save time by no longer always executing code to set the date format.

The suggested practice with DRCP is to use LOGON triggers only for setting session attributes and not for executing per PHP-connection logic such as custom logon auditing. This recommendation is also true for persistent connections with dedicated or shared servers.

Database actions that must be performed exactly once per OCI8 connection call should be explicitly executed in the PHP script.

From Oracle 11gR2 onwards, LOGOFF triggers fire for pooled servers when sessions are terminated. For `oci_connect()` and `oci_new_connect()` connections, this is with `oci_close()` or at the end of the script. For `oci_pconnect()` connections, it can happen when the pooled server process naturally terminates or its session needs to be recreated.

It is not possible to depend on triggers for tracking PHP OCI8 connect calls. The caching, pooling, timing out and recreation of sessions and connections with or without DRCP can distort any record. With pooled servers, LOGON triggers can fire at authentication and when the session is created, in effect firing twice for the initial connection.

Changing Passwords with DRCP Connections

In general, PHP applications that change passwords should avoid using `oci_pconnect()`. This call will use the old password to match an open connection in PHP's persistent connection cache without requiring re-authentication to the database with the new password. This can cause confusion over which password to connect with since if there is no cached connection it is the new password that must be used. With DRCP, there is a further limitation - connections cannot be used to change passwords programmatically. PHP scripts that use `oci_password_change()` should continue to use dedicated or shared servers.

Monitoring DRCP

Data dictionary views are available to monitor the performance of DRCP. Database administrators can check statistics such as the number of busy and free servers, and the number of hits and misses in the pool against the total number of requests from clients. The views are:

- DBA_CPOOL_INFO
- V\$PROCESS
- V\$SESSION
- V\$CPOOL_STATS
- V\$CPOOL_CC_STATS
- V\$CPOOL_CONN_INFO

For Oracle RAC, there are GV\$CPOOL_STATS, GV\$CPOOL_CC_STATS and GV\$CPOOL_CONN_INFO views corresponding to the instance-level views. These record DRCP statistics across clustered instances. If a database instance in a cluster is shut down, the statistics for that instance are purged from the GV\$ views.

The DRCP statistics are reset each time the pool is started.

DBA_CPOOL_INFO View

DBA_CPOOL_INFO displays configuration information about the DRCP pool. The columns are equivalent to the [dbms_connection_pool.configure_pool\(\)](#) settings described in the table of DRCP configuration options, with the addition of a STATUS column. The status is ACTIVE if the pool has been started and INACTIVE otherwise. Note the pool name column is called CONNECTION_POOL. This example checks whether the pool has been started and finds the maximum number of pooled servers:

```
SQL> select connection_pool, status, maxsize
      from dba_cpool_info;

CONNECTION_POOL          STATUS        MAXSIZE
-----  -----
SYS_DEFAULT_CONNECTION_POOL  ACTIVE           40
```

In Oracle 11gR2, DBA_CPOOL_INFO gained NUM_CBROK and MAXCONN_CBROK columns, equivalent to the pool configuration options of the same names. In Oracle 11gR1 the number of configured brokers per instance can be found from the V\$PROCESS view, for example on Linux this query shows one broker has been enabled:

```
SQL> select program
      from v$process
     where program like 'oracle%(N%)';

PROGRAM
-----
oracle@localhost (N001)
```

PHP Connection Pooling and High Availability

V\$PROCESS and V\$SESSION Views

The V\$SESSION view will show information about the currently active DRCP sessions. It can also be joined with V\$PROCESS via `V$SESSION.PADDR = V$PROCESS.ADDR` to correlate the views.

V\$CPOOL_STATS View

V\$CPOOL_STATS displays information about the DRCP statistics for an instance.

Table 15: V\$CPOOL_STATS View.

Column	Description
POOL_NAME	Name of the Database Resident Connection Pool.
NUM_OPEN_SERVERS	Total number of busy and free servers in the pool (including the authentication servers).
NUM_BUSY_SERVERS	Total number of busy servers in the pool (not including the authentication servers).
NUM_AUTH_SERVERS	Number of authentication servers in the pool.
NUM_REQUESTS	Number of client requests.
NUM_HITS	Total number of times client requests found matching pooled servers and sessions in the pool.
NUM_MISSES	Total number of times client requests could not find a matching pooled server and session in the pool.
NUM_WAITS	Total number of client requests that had to wait due to non-availability of free pooled servers.
WAIT_TIME	Reserved for future use.
CLIENT_REQ_TIMEOUTS	Reserved for future use.
NUM_AUTHENTICATIONS	Total number of authentications of clients done by the pool.
NUM_PURGED	Total number of sessions purged by the pool.
HISTORIC_MAX	Maximum size that the pool has ever reached. With PHP this is likely to reach the maximum pool size value.

The V\$CPOOL_STATS view can be used to assess how efficient the pool settings are. This example query shows an application using the pool effectively. The low number of misses indicates that servers and sessions were reused. The wait count shows just over 1% of requests had to wait for a pooled server to become available:

```
SQL> select num_requests,num_hits,num_misses,num_waits
   from v$cpool_stats;
```

NUM_REQUESTS	NUM_HITS	NUM_MISSES	NUM_WAITS
100031	99993	38	1054

If `oci8.connection_class` is set (allowing pooled servers and sessions to be reused) then `NUM_MISSES` is low. If the pool `maxsize` is too small for the connection load then `NUM_WAITS` is high:

NUM_REQUESTS	NUM_HITS	NUM_MISSES	NUM_WAITS
50352	50348	4	50149

Tune the pool size by monitoring the `NUM_WAITS` trend. If the value is high then increase the number of pooled servers.

If the connection class is left unset, the sharing of pooled servers is restricted to within each web server process. Even if the pool size is large, session sharing is limited causing poor utilization of pooled servers and contention for them:

NUM_REQUESTS	NUM_HITS	NUM_MISSES	NUM_WAITS
64152	17941	46211	15118

If `NUM_HITS` is 0 or very low, one potential cause is the use of `oci_connect()` since this is designed to recreate the session each time it is called. Use `oci_pconnect()` instead.

V\$CPOOL_CC_STATS View

`V$CPOOL_CC_STATS` displays information about the connection class level statistics for the pool per instance. The columns are similar to those of `V$CPOOL_STATS` described in Table 15, with a `CCLASS_NAME` column giving the name of the connection sub-pool the results are for:

```
SQL> select cclass_name, num_requests, num_hits, num_misses
      from v$cpool_cc_stats;
```

CCLASS_NAME	NUM_REQUESTS	NUM_HITS	NUM_MISSES
HR.MYPHPAPP	100031	99993	38
SCOTT.SHARED	10	0	10
CJ.OCI:SP:wshbIFDtb7rgQwMyuYvodA	1	0	1

For PHP, the `CCLASS_NAME` value is composed of the value of the username and of the `oci8.connection_class` value used by the connecting PHP processes. The example above shows an application known as MYPHPAPP using the pool effectively.

For programs like SQL*Plus that were not built using Oracle's Session Pooling APIs, the class name will be SHARED. The example shows that ten such connections were made as the user SCOTT. Although these programs share the same connection class, new sessions are created for each connection, keeping each cleanly isolated from any unwanted session changes. This is similar to using PHP's `oci_connect()` with DRCP pooled servers.

The last line of the example output shows a system generated class name for an application that created a connection without explicitly setting `oci8.connection_class`, or had it set to an empty string. Pooling would not be effectively used if this application continued to be executed. Such an entry could be an indication that a `php.ini` file is mis-configured.

PHP Connection Pooling and High Availability

V\$CPOOL_CONN_INFO View

This view gives insight into client processes that are connected to the connection broker, making it easier to monitor and trace applications that are currently using pooled servers or are idle.

This view was introduced in Oracle 11gR2.

Table 16: V\$CPOOL_CONN_INFO View.

Column	Description
CMON_ADDR	Address of the connection broker
SESSION_ADDR	Address of the session associated with the connection. NULL if there is no active session. Can be joined with V\$SESSION.SADDR
CONNECTION_ADDR	Address of the connection
USERNAME	Name of the user associated with the connection
PROXY_USER	Name of the proxy user
CCLASS_NAME	Connection class associated with the connection
PURITY	Will be SELF for oci_pconnect() calls or NEW otherwise
TAG	Not set by PHP
SERVICE	TNS service name for the connection
PROCESS_ID	Process ID of the PHP or Apache process
PROGRAM	Program name of the PHP or Apache process
MACHINE	Machine name where PHP is running
TERMINAL	Terminal identifier of the PHP process that created the connection
CONNECTION_MODE	Reserved for internal use
CONNECTION_STATUS	Status of the connection: NONE, CONNECTING, ACTIVE, WAITING, IDLE

You can monitor V\$CPOOL_CONN_INFO to, for example, identify mis-configured machines that do not have the connection class set correctly. This view maps the machine name to the class name:

```
SQL> select cclass_name, machine from v$cpool_conn_info;
CCCLASS_NAME          MACHINE
-----
. . .
CJ.OCI:SP:wshbIFDtb7rgQwMyuYvodA      cjlinux
. . .
```

In Oracle 11gR1 or in highly dynamic situations you can find this same information by turning on OCI tracing in the database:

```
$ sqlplus / as sysdba
SQL> alter system set events '10524 trace name context forever, level 1';
```

There is no need to restart the database.

Now run the application to generate the trace files.

The trace file directory can be found by doing [show parameter background_dump_dest](#) in SQL*Plus. In the pooled server trace files, search for *kpplsPopulate*:

```
$ grep kpplsPopulate *.trc
. . .
orcl3_1005_28576.trc:kpplsPopulate: cso authenticated,
key=(CJ*CJ.OCI:SP:wshbIFDtb7rgQwMyuYvodA**1*orcl3*ORA$BASE*)
```

You can see the correspondence between the "key" fields and the class names shown earlier in V\$CPOOL_CC_STATS. Open the related trace file, for example *orcl3_1005_28576.trc*. Near the *CJ.OCI:SP:wshbIFDtb7rgQwMyuYvodA* entry it contains:

```
. . .
*** SESSION ID:(9.4985) 2012-06-18 16:16:29.973
*** SERVICE NAME:(orcl3) 2012-06-18 16:16:29.973
*** MODULE NAME:(httpd@cjlinux (TNS V1-V3)) 2012-06-18 16:16:29.973
*** ACTION NAME:() 2012-06-18 16:16:29.973

kpplsPostProcessState:(acquired=0)(call=0SESSKEY)
kpplsh_cso_release_hdrl: cso=(0xcbfdbdb88)
Release (FALSE): cso=0xcbfdbdb88
kpplsPreProcessState:(acquired=0)(call=0AUTH)
Authentication complete: cso=0xcbfdbdb88
kpplsPopulate: cso authenticated,
key=(CJ*CJ.OCI:SP:wshbIFDtb7rgQwMyuYvodA**1*orcl3*ORA$BASE*)
kpplsClearSession: svsoCtx = (0xaf578bc8)
kpplsClearSession: Destroying session= (0xcf640268)
. . .
```

Reading back up the trace file to the MODULE NAME line shows the host name was *cjlinux*. This is the host Apache was running on. Check the *php.ini* of this machine, ensuring that it contains a valid *oci8.connection_class* parameter.

Tracing can now be turned off:

```
alter system set events '10524 trace name context forever, level 0';
```

PHP Connection Pooling and High Availability

High Availability With FAN and RAC

Clients that run in high availability configurations such as with Oracle RAC or Data Guard Physical Stand-By can take advantage of Fast Application Notification (FAN) events to allow applications to respond quickly to database node failures. FAN support in PHP may be used with or without DRCP – the two features are independent.

Without FAN, when a database instance or machine node fails unexpectedly, PHP applications may be blocked waiting for a database response until a TCP timeout expires. Errors are therefore delayed, sometimes up to several minutes.

By leveraging FAN events, PHP applications are quickly notified of failures that affect their established database connections. Connections to a failed database instance are pro-actively terminated without waiting for a potentially lengthy TCP timeout. This allows PHP scripts to recover quickly from a node or network failure. The application can reconnect and continue processing without the user being aware of a problem.

Also, all inactive network connections cached in PHP to the connection broker in case of DRCP, and persistent connections to the server processes or dispatcher in case of dedicated or shared server connections on the failed instances, are automatically cleaned up.

A subsequent PHP connection call will create a new connection to a surviving RAC node, activated stand-by database, or even the restarted single-instance database.

Configuring FAN Events in the Database

To get the benefit of high availability, the database service to which the applications connect must be enabled to post FAN events. For example, to enable events on the service SALES in an Oracle 11gR2 database SALESDB:

```
$ srvctl modify service -d SALESDB -s SALES -q TRUE
```

The `-q` option indicates that AQ high availability events should be enabled.

Configuring PHP for FAN

With the OCI8 extension, a *php.ini* configuration parameter `oci8.events` allows PHP to be notified of FAN events:

```
oci8.events = On
```

FAN support is only available when PHP uses OCI8 1.3 onwards, and is linked with Oracle Database 10g Release 2 or 11g libraries. PHP must connect to Oracle Database 10g Release 2 or 11g.

With older versions of Oracle Database, review the patches for Oracle bugs 7143299 (fixed in Oracle 11.2.0.1) and 8670389 (fixed in 11.2.0.2) to improve login times in various conditions when using `oci8.events`.

Application Deployment for FAN

The error codes returned to PHP will generally be the same as without FAN enabled, so application error handling can remain unchanged.

Alternatively, applications can be enhanced to reconnect and retry actions, taking advantage of the higher level of service given by FAN.

As an example, the code below does some work (perhaps a series of update statements). If there is a connection failure, it reconnects, checks the transaction state and retries the work. The OCI8 extension will detect the connection failure and be able reconnect on request, but the user script must also determine that work failed, why it failed, and be able to continue that work. The example code detects connections errors so it can identify it needs to continue or retry work. It is generally important not to redo operations that already committed updated data.

Typical errors returned after an instance failure are *ORA-12153: TNS:not connected* or *ORA-03113: end-of-file on communication channel*. Some more connection related errors are shown in the example but others, and errors including standard database errors, may be returned, depending on timing.

```

function isConnectionError($err)
{
    switch($err) {
        case 22: /* session does not exist */
        case 28: /* session killed */
        case 378: /* buffer pool param incorrect */
        case 602: /* core dump */
        case 603: /* fatal error */
        case 604: /* recursive SQL error */
        case 609: /* attach failed */
        case 1012: /* not logged in */
        case 1033: /* init or shutdown in progress */
        case 1041: /* internal error */
        case 1043: /* Oracle not available */
        case 1089: /* immediate shutdown in progress */
        case 1090: /* shutdown in progress */
        case 1092: /* instance terminated */
        case 3113: /* disconnect */
        case 3114: /* not connected */
        case 3122: /* closing window */
        case 3135: /* lost contact */
        case 12153: /* TNS: not connected */
        case 27146: /* fatal or instance terminated */
        case 28511: /* Lost RPC */
            return true;
    }
    return false;
}

$c = doConnect();
$err = doSomeWork($c);
if (isConnectionError($err)) {
    // reconnect, find what was committed, and retry
    $c = doConnect();
    $err = checkApplicationStateAndContinueWork($c);
}
if ($err) {
    // end the application
    handleError($err);
}

```

PHP Connection Pooling and High Availability

The complexity is with identifying what work the application has completed and what needs to be redone after reconnection. Make sure the application uses transactions and does not auto-commit. One technique for identifying what work remains to be done is to add a status column to a table. Each part of the script updates the status column when it has completed.

RAC Connection Load Balancing With PHP

PHP OCI8 1.3 onwards will automatically balance new connections across RAC instances with Oracle's Connection Load Balancing (CLB) to use resources efficiently. The balancing happens at the first connect for each set of credentials in a PHP process. The same RAC instance will then be used for the life of the PHP process.

It is recommended to use FAN and CLB together.

No PHP script changes are needed to use CLB. The connection balancing is handled transparently by the Oracle Net listener. To enable CLB, the database service must be modified to send load events to the listener. In Oracle 11gR2 use the `-j SHORT` or `-j LONG` options to `srvctl`. For example:

```
$ srvctl modify service -d SALESDB -s SALES -j LONG
```

Table 17: CLB goal parameter values.

Parameter Value	Parameter Description
SHORT	Use for connection load balancing method for applications that have short-lived connections such as created by <code>oci_connect()</code> in quick scripts. This uses CPU-based statistics to distribute connections.
LONG	Use for applications that have long-lived connections such as created by <code>oci_pconnect()</code> . This uses a simple session-based metric to distribute connections

PHP AND TIMES TEN IN-MEMORY DATABASE

TimesTen is an in-memory database that can be used standalone or as a cache to Oracle database. The product marketing description gives this overview:

Oracle TimesTen In-Memory Database (TimesTen) is a full-featured, memory-optimized, relational database with persistence and recoverability. It provides applications with the instant responsiveness and very high throughput required by database-intensive applications. Deployed in the application tier, TimesTen operates on databases that fit entirely in physical memory (RAM). Applications access the TimesTen database using standard SQL interfaces. For customers with existing application data residing on the Oracle Database, TimesTen is deployed as an in-memory cache database with automatic data synchronization between TimesTen and the Oracle Database.

The native C interface to TimesTen is ODBC but there is an Oracle OCI layer on top of this that allows some standard PHP OCI8 features to work. You can build PHP OCI8 and connect to both Oracle Database and TimesTen In-Memory Database.

Many basic PHP scripts will work unchanged against either database. However, TimesTen is a different database than Oracle Database. Some PHP OCI8 features cannot be expected to work the same way, or don't work at all. In particular LOB support from PHP OCI8 is not available, even though TimesTen 11g Release 2 allows LOBS to be stored. Collection support isn't available and you might experience some edge cases with binding, particularly if your code is not well formed. Error messages might differ. If you are migrating existing applications to TimesTen is also important to test thoroughly because TimesTen does not support all SQL, PL/SQL or database features of Oracle Database. Problem resolution for using PHP with TimesTen is via OTN forums, which is the same as for PHP with Oracle Database.

Native ODBC applications will generally be much faster than PHP due to the overheads of PHP, some overheads of the Oracle OCI layer that sits on top of TimeTen's native ODBC API, and due to the scale-out architecture commonly used by PHP applications not being directly congruent with the in-memory high performance design of TimesTen.

Installing TimesTen on Linux

Install TimesTen following the *Oracle TimesTen In-Memory Database Installation Guide 11*. These steps are summarized below using 64-bit Oracle Linux 6.

1. Download "TimesTen 11.2.2.2.0 for Linux x86 (64-bit)" from the TimesTen downloads page, <http://www.oracle.com/technetwork/products/timesten/downloads/>
2. As root, create a new software owner and set a password:

```
# useradd -m -c "TimesTen Owner" -d /home/ttadmin -s /bin/bash ttadmin
# chmod 755 /home/ttadmin
# passwd ttadmin
```

PHP and TimesTen In-Memory Database

3. Create a new *timesten* group and add the TimesTen administrator and the Apache users to it:

```
# groupadd timesten  
# usermod -a -G timesten ttadmin  
# usermod -a -G timesten apache
```

4. Create the instance registry:

```
# mkdir /etc/TimesTen  
# chgrp ttadmin /etc/TimesTen  
# chmod 770 /etc/TimesTen/
```

5. Login as the *ttadmin* user.
6. In a terminal, extract the downloaded software archive:

```
$ tar -xf /tmp/timesten112220.linux8664.tar.gz
```

7. Install TimesTen:

```
$ cd linux8664  
$ ./setup.sh
```

Accept the default values *except* for the Group and the QuickStart questions. The prompts are answered:

- Instance name: *tt1122*
 - Install the "Client/Sever and Data manager"
 - Install into */home/ttadmin*
 - Create the daemon directory */home/ttadmin/TimesTen/tt1122/info*
 - Daemon logs are in */home/ttadmin/TimesTen/tt1122/info*
 - Default port for the TimesTen daemon: *53396*
 - Change the group to restrict access from *ttadmin* to *timesten*.
 - Enable PL/SQL
 - *TNS_ADMIN* is skipped and left unset
 - The TimesTen server listens on *53397*
 - Install the QuickStart and Documentation: **yes** (This is not the default.)
 - Create the DemoDataStore in */home/ttadmin/TimesTen/tt1122/info*
 - Do not use Replication with Oracle Clusterware
8. If you want the database to start when the machine boots, login as the root user and follow the steps given on the screen to install the startup scripts in */etc/init.d*:

```
# cd /home/ttadmin/TimesTen/tt1122/bin  
# ./setuproot -install
```

TimesTen will now be running.

Managing TimesTen

The */home/ttadmin/TimesTen/tt1122/quickstart/ttquickstartenv.sh* script is a convenient way to set environment variables before accessing the sample database.

As the *ttadmin* user, check the status of the instance by setting the environment and running the *ttstatus* command:

```
$ source /home/ttadmin/TimesTen/tt1122/quickstart/ttquickstartenv.sh
$ ttstatus
```

This gives:

```
TimesTen status report as of Mon Jun 18 12:44:50 2012

Daemon pid 23629 port 53396 instance tt1122
TimesTen server pid 23638 started on port 53397
-----
Data store /home/ttadmin/TimesTen/tt1122/info/DemoDataStore/sampledbs_1122
There are no connections to the data store
Replication policy : Manual
Cache Agent policy : Manual
PL/SQL enabled.
-----
Accessible by group timestamp
End of report
```

There is a command-line tool for running ad hoc queries against an instance:

```
$ ttisql "dsn=sampledbs_1122;uid=appuser"
```

Installing the sample database is shown in the next section. The default *appuser* password is *oracle*.

If you need to manually stop or start the instance of TimesTen, login as *ttadmin*, set the environment, and run:

```
$ ttdaemonadmin -start
```

or

```
$ ttdaemonadmin -stop
```

If you want to remove the instance of TimesTen, first stop it and then, as *root*, run:

```
# cd /home/ttadmin
# TimesTen/tt1122/bin/setup.sh -uninstall
```

Creating the TimesTen Sample Database

Create the sample database by logging in as the *ttadmin* operating system user and following these steps:

1. Run the creation script:

```
$ cd \ /home/ttadmin/TimesTen/tt1122/quickstart/sample_scripts/createdb
$ ./build_sampledb.sh
```

2. Choose and remember passwords when prompted for the *adm*, *appuser* and *xlauser* users.

PHP and TimesTen In-Memory Database

Installing Apache and PHP for TimesTen

The packaged Apache and PHP binaries are used below, but installation can be customized as needed using steps shown earlier in this book.

Install Apache and PHP with:

1. As root, use the bundled packages for the web server and for PHP:

```
# yum install httpd php php-devel
```

2. Edit */etc/php.ini* and add a timezone setting using an appropriate timezone, for example:

```
date.timezone = America/Los_Angeles
```

3. Download the latest OCI8 extension from PECL. Extract and install it with:

```
# tar -xf oci8-1.4.9.tgz
# cd oci8-1.4.9
# phpize
# ./configure --with-oci8=instantclient,
/home/ttadmin/TimesTen/tt1122/toracle_home/instantclient_11_2
# make
# make install
```

4. Edit */etc/php.ini* and add:

```
extension=oci8.so
```

5. Configure Apache. In */etc/sysconfig/httpd* set TNS_ADMIN and LD_LIBRARY_PATH:

```
TT=/home/ttadmin/TimesTen/tt1122
export TNS_ADMIN=$TT/network/admin/samples
export LD_LIBRARY_PATH=$TT/lib:$TT/toracle_home/instantclient_11_2
```

6. Make sure SELinux is in Permissive mode by editing */etc/sysconfig/selinux* and rebooting. Alternatively turn it off temporarily with:

```
# setenforce permissive
```

Checking the Installation

Use the *httpd* service if you manually need to stop and restart Apache. Since the configuration just changed when OCI8 was installed, restart it now:

```
# service httpd restart
```

Test PHP by creating a file */var/www/html/pi.php*:

```
<?php
phpinfo();
?>
```

Load this in a browser using *http://localhost/pi.php*. You will get the normal page of configuration information about PHP. Check there is a section for OCI8.

Connecting to TimesTen With PHP OCI8

The *ttisql* example given earlier shows the credentials and connect string for the sample database. These can be used in PHP scripts to connect to TimesTen. Test with a basic query script such as */var/www/html/oci8tt.php*:

Script 124: /var/www/html/oci8tt.php

```
<?php

$c = oci_connect("appuser", "oracle", "sampledb_1122");
if (!$c) {
    $m = oci_error();
    trigger_error('Could not connect to database: ' .
                  $m['message'], E_USER_ERROR);
}

$s = oci_parse($c, "select * from emp");
if (!$s) {
    $m = oci_error($c);
    trigger_error('Could not parse statement: ' .
                  $m['message'], E_USER_ERROR);
}
$r = oci_execute($s);
if (!$r) {
    $m = oci_error($s);
    trigger_error('Could not execute statement: ' .
                  $m['message'], E_USER_ERROR);
}

echo "<table border='1'>\n";
while (($row =
        oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_NULLS)) != false) {
    echo "<tr>\n";
    foreach ($row as $item) {
        echo "    <td>" . ($item !== null ? htmlentities($item,
                                                       ENT_QUOTES) : "&nbsp;") . "</td>\n";
    }
    echo "</tr>\n";
}
echo "</table>\n";

?>
```

Load this in a browser with *http://localhost/oci8tt.php*. The query results will be displayed in a table.

If you get a blank screen or have problems, check the Apache log in */var/logs/httpd/error_log*.

Configuring TimesTen

The default RAM policy of TimesTen is *inUse*, meaning that the database is loaded into memory at time of first connection, and unloaded when the last connection completes. This

PHP and TimesTen In-Memory Database

means that the first and last connections can be slow. This is most noticeable if you are using command line scripts or using non-persistent connections.

You can change the *inUse* grace period or alter the RAM policy with `ttRamPolicySet()` to overcome this. The changed value is persistent.

With an *inUse* policy, the slow unload-on-last-connect issue can be resolved by setting a grace period time the database is kept in RAM after the last application has disconnected. For example this PHP code sets it to five seconds, deferring the database shutdown and allowing the PHP application to close the last connection without delay:

```
$s = oci_parse($cc, "call ttRamPolicySet('inUse', 5)");
oci_execute($s);
```

However instead of *inUse*, a general recommendation is to change the RAM policy to *manual*:

```
$s = oci_parse($cc, "call ttRamPolicySet('manual', null)");
oci_execute($s);
```

With this done, before running applications, you must explicitly load the database into RAM by using the *ttAdmin* utility:

```
$ ttAdmin -ramLoad
```

When the database is no longer needed it can be unloaded with:

```
$ ttAdmin -ramUnLoad
```

The TimesTen manual discusses RAM policies and gives other options.

PHP AND ORACLE TUXEDO

Oracle Tuxedo is a transaction oriented application server which can be used for developing and deploying applications written in PHP, Python and Ruby, as well as in the traditional languages C, C++, and COBOL. Tuxedo is based on a routing and queuing system, which is highly configurable.

This chapter shows using Oracle Tuxedo 11.1 with PHP applications running under Apache. HTTP requests are forwarded from Apache to *mod_tuxedo* which then invokes the PHP script engine.

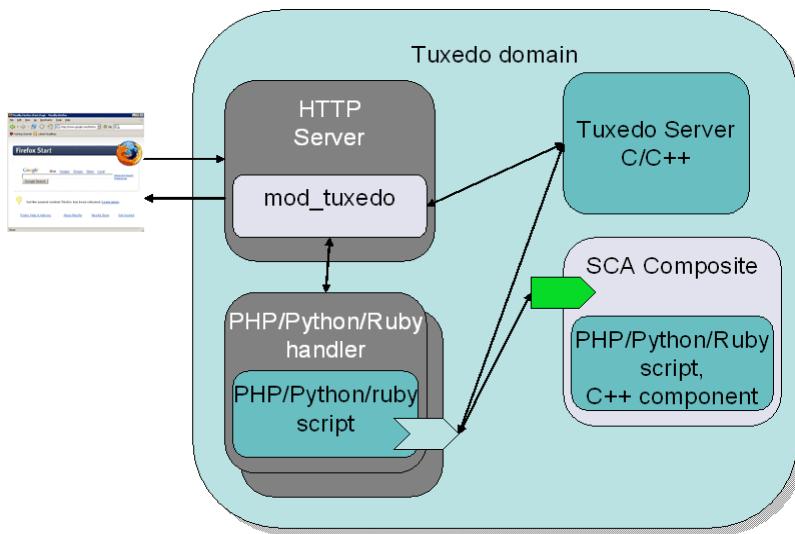


Figure 73: Oracle SALT Web Application Server Architecture

The *mod_tuxedo* component for handling PHP (and Python and Ruby) web requests is licensed with *Oracle Service Architecture Leveraging Tuxedo (SALT)*. It is available on Linux and Solaris. It is included in SALT from 11.1.1.2.2 onwards. In Tuxedo 12c, it is part of the base Tuxedo component.

There is a pre-configured demonstration *Tuxedo Web Application Server Demo VM* for VirtualBox available that shows similar examples to this chapter. It can be found on the Tuxedo download page:

<http://www.oracle.com/technetwork/middleware/tuxedo/downloads/index.html>.

With a service orientated architecture, Tuxedo also allows multiple languages, including PHP, to be used as service consumers and producers. This is not covered in this manual.

Installing Tuxedo 11.1 and SALT for PHP Web Applications

Begin by reviewing the prerequisites in the documentation at

<http://www.oracle.com/technetwork/middleware/tuxedo/documentation/index.html>.

PHP and Oracle Tuxedo

Remember to turn off SELinux before beginning the installation by editing `/etc/sysconfig/selinux` and rebooting.

Installing Oracle Tuxedo

To install Tuxedo, follow these steps:

1. Create a user to be the Tuxedo administrator and login as that user, for example `cjones`.
2. If you intend to run the operating system packaged Apache, make the new user directory readable to the webserver:

```
$ chmod 755 /home/cjones
```

3. Download *Oracle Tuxedo 11gR1 (11.1.1.2.0)* from
<http://www.oracle.com/technetwork/middleware/tuxedo/downloads/index.html>
4. Run the Tuxedo installer:

```
$ sh tuxedo111120_64_Linux_01_x86.bin
```

5. A GUI appears. Start the menu by clicking *Next*:

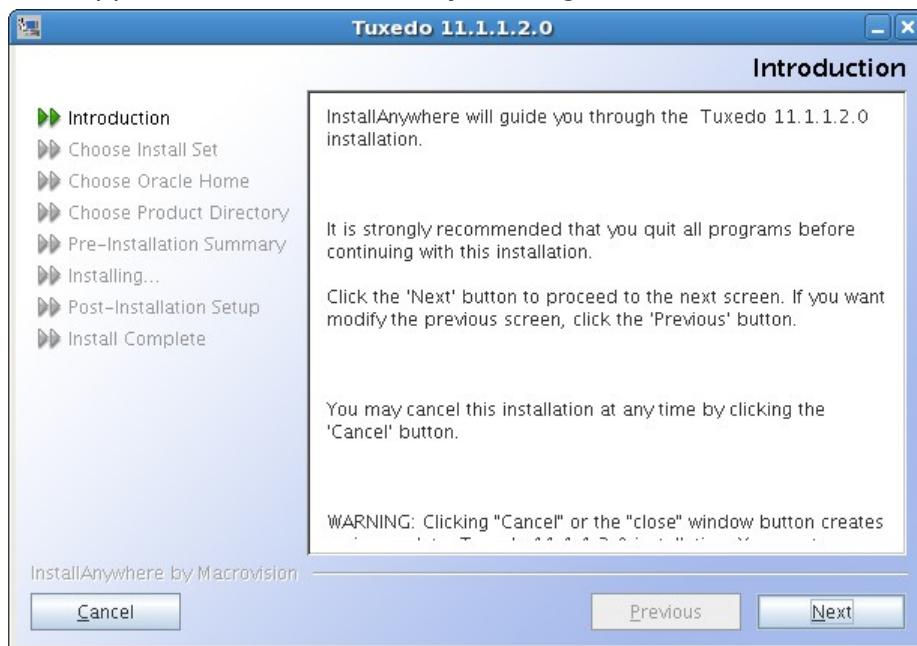


Figure 74: Oracle Tuxedo Installer Introduction

Installing Tuxedo 11.1 and SALT for PHP Web Applications

6. Do a *Full Install*:

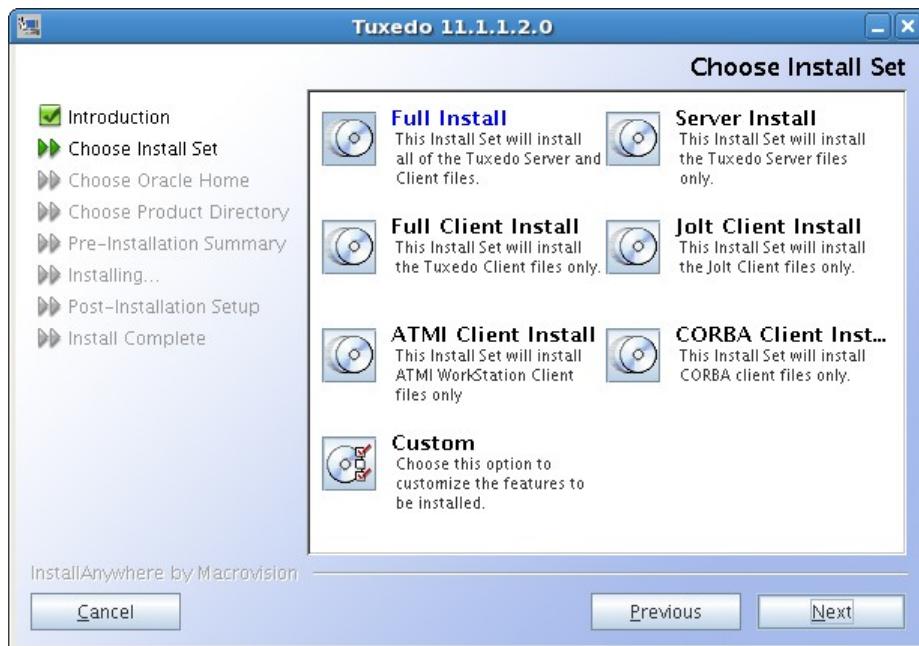


Figure 75: Oracle Tuxedo Install Set Menu

7. Create a new *Oracle Home*. Here, the directory `/home/cjones/oracle` was chosen:

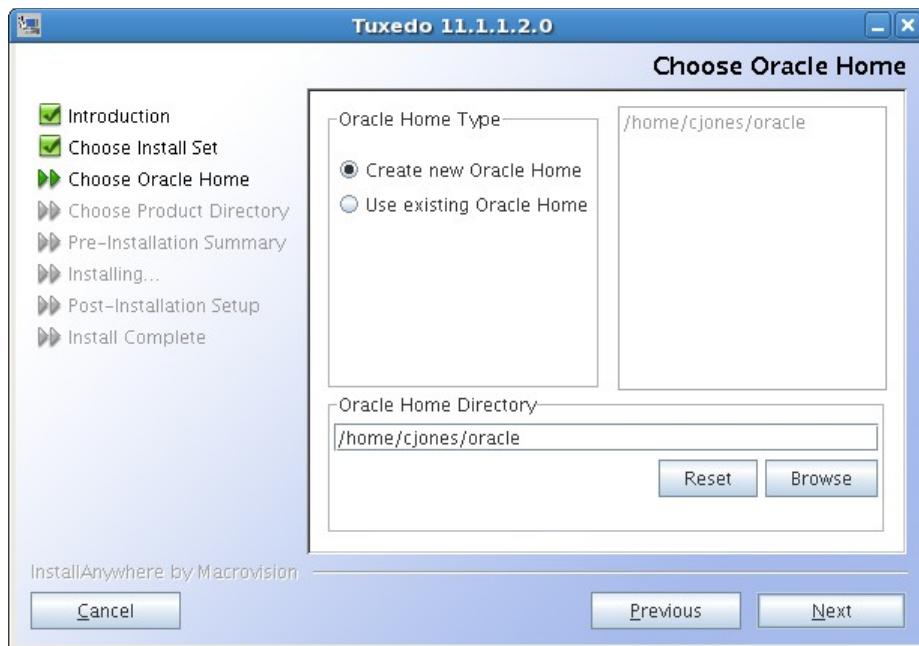


Figure 76: Oracle Tuxedo Choosing the Oracle Home

PHP and Oracle Tuxedo

8. Install the Samples by checking the box in the bottom right of the installer. Leave the Product Installation Directory as the default.

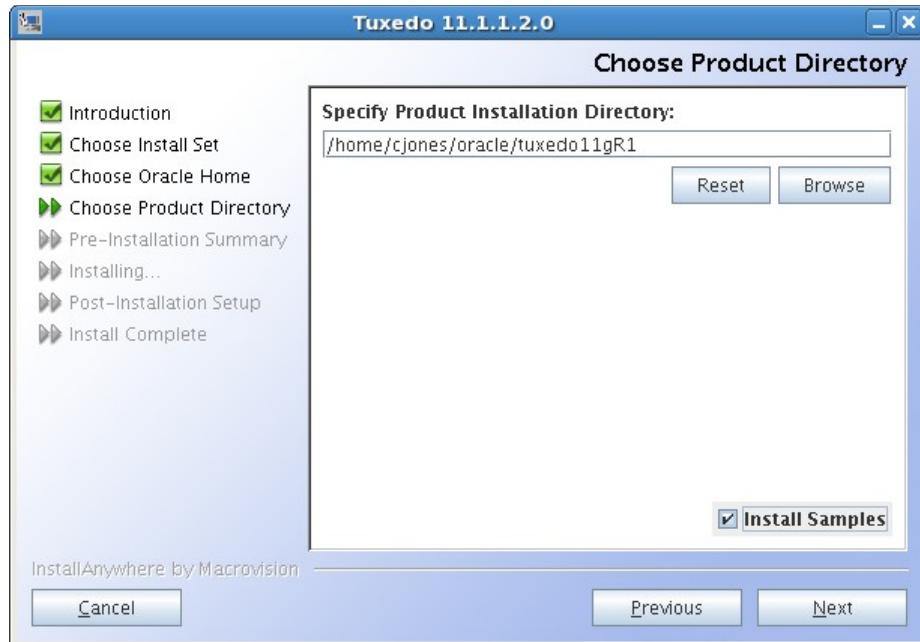


Figure 77: Oracle Tuxedo Choosing the Product Directory

9. Confirm the installation:

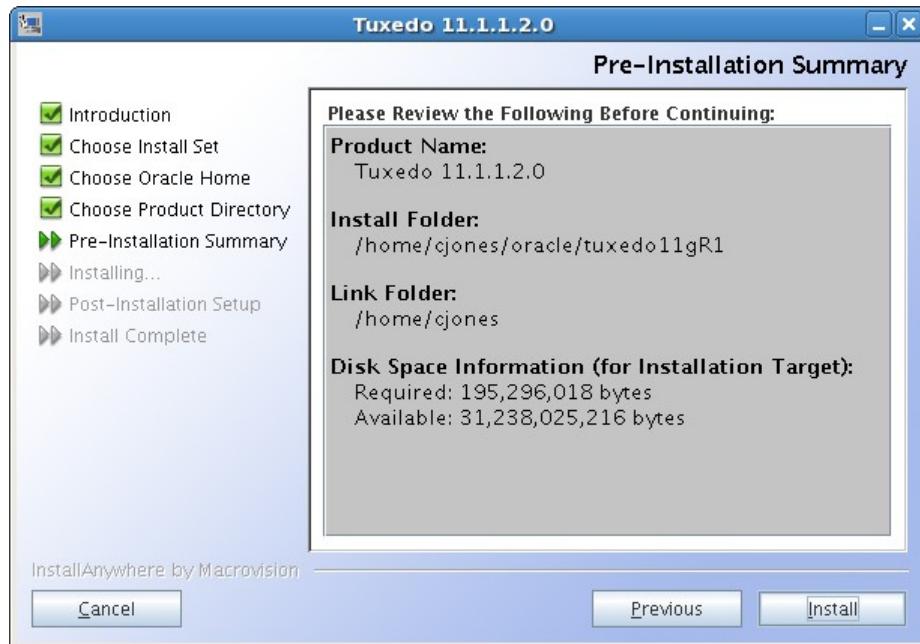


Figure 78: Oracle Tuxedo Pre-Installation Summary

Installing Tuxedo 11.1 and SALT for PHP Web Applications

10. In the post installation setup, Enter a password for *tlisten*. This won't be used for this example:

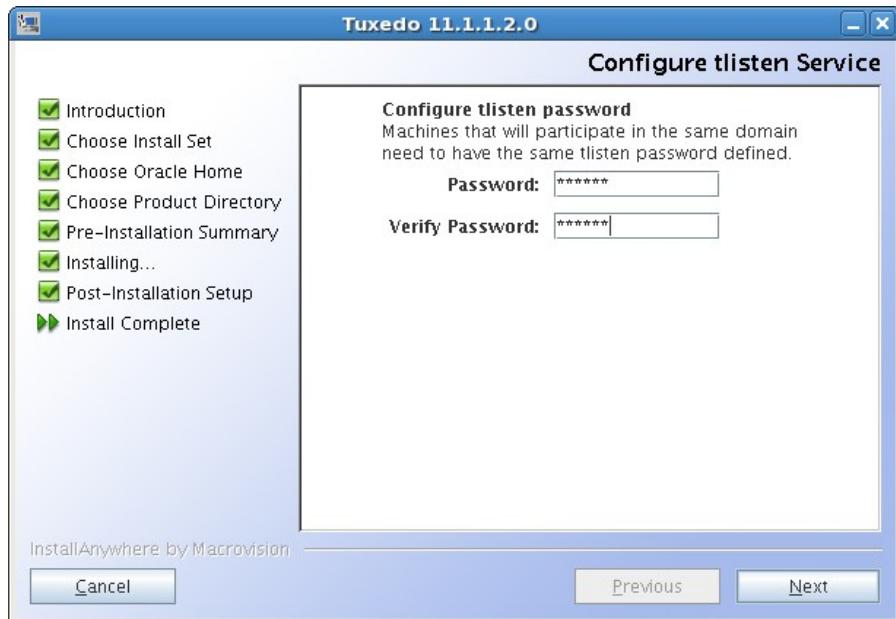


Figure 79: Oracle Tuxedo "tlisten" configuration

11. For this example, complete the installation by choosing *No* to configuring LDAP in the SSL Installation Choice menu:

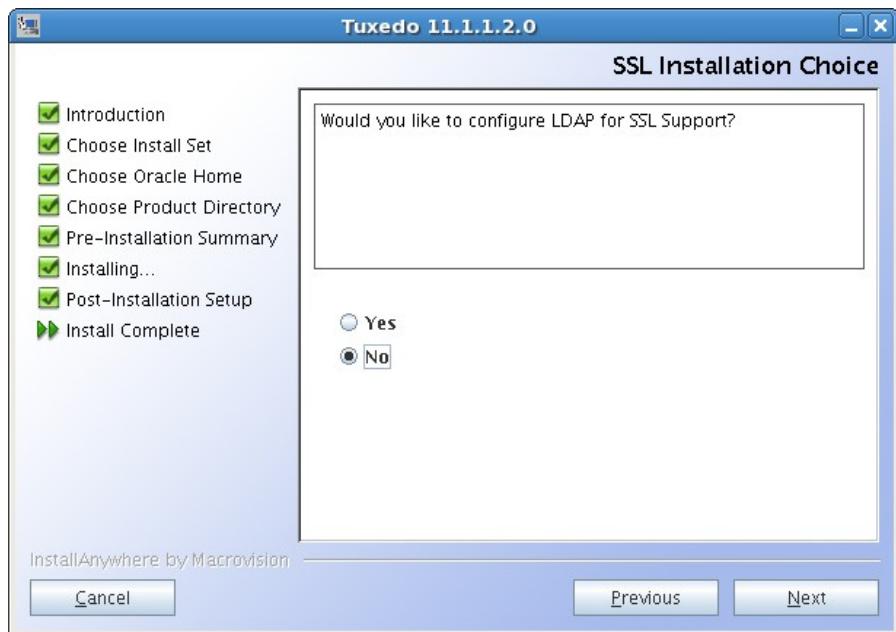


Figure 80: Oracle Tuxedo SSL Configuration

PHP and Oracle Tuxedo

Installing Oracle SALT

To install Oracle SALT follow these instructions:

1. Download *Oracle Service Architecture Leveraging Tuxedo (SALT) 11gR1 (11.1.1.2.2)* from <http://www.oracle.com/technetwork/middleware/tuxedo/downloads/index.html>
2. Run the SALT installer:

```
$ sh salt111122_64_Linux_01_x86.bin
```

3. Start the menu by clicking *Next*:

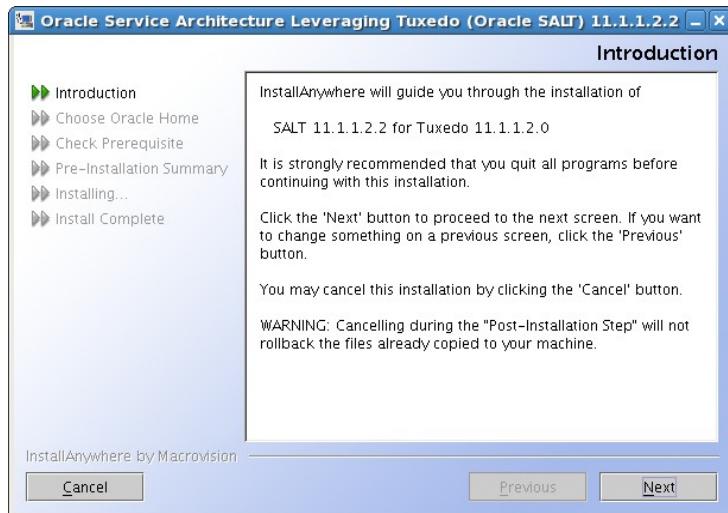


Figure 81: Oracle SALT Installation Introduction

4. Install into the same home as Tuxedo, /home/cjones/oracle:

Installing Tuxedo 11.1 and SALT for PHP Web Applications

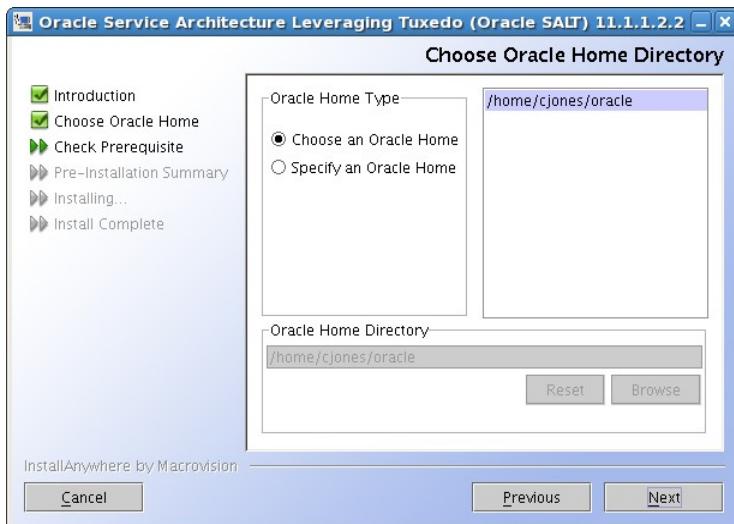


Figure 82: Oracle SALT Choosing the Home Directory

5. Choose to install the *SALT Client and Server*. The client is for the Service Component Architecture which makes components distributable:

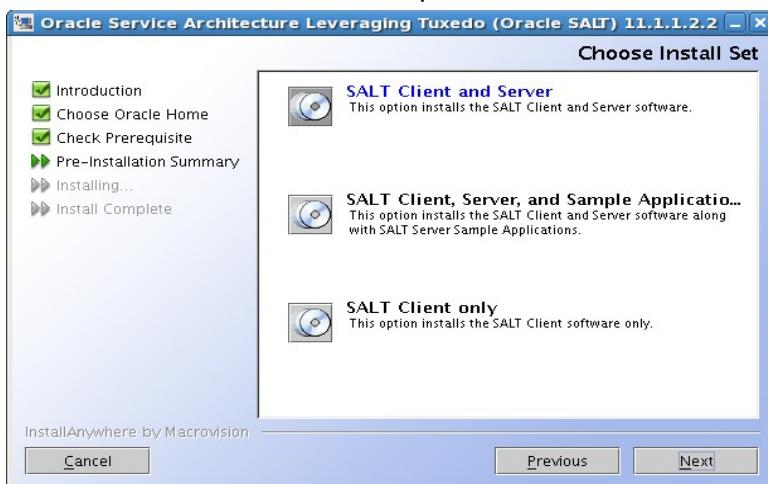


Figure 83: Oracle SALT Install Set Choice

6. Review the install options and click *Install*:

PHP and Oracle Tuxedo

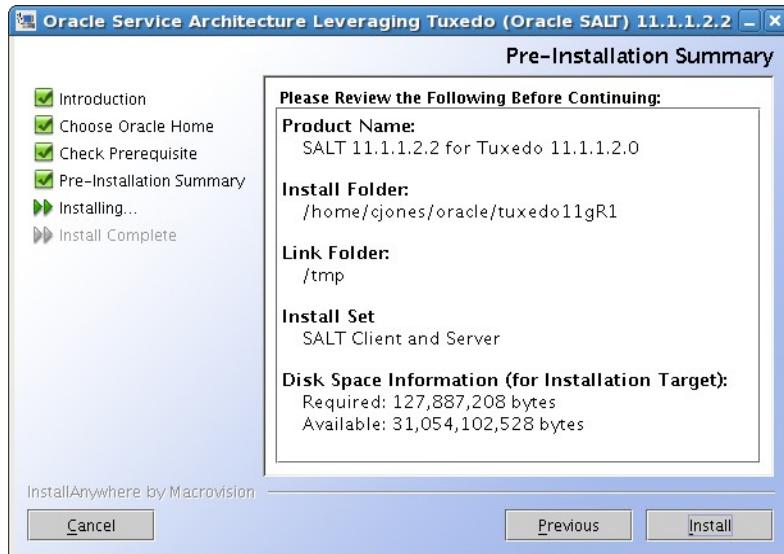


Figure 84: Oracle SALT Installation Review

7. The installation of SALT is complete:

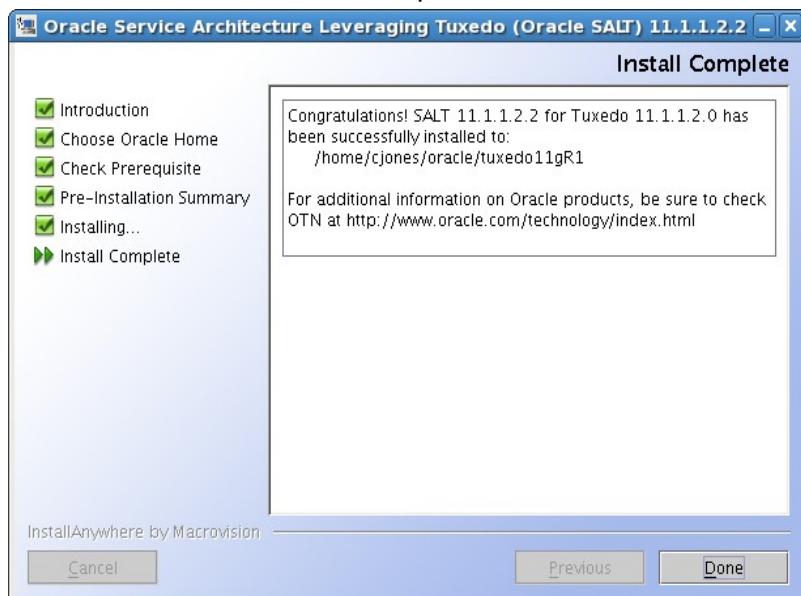


Figure 85: Oracle SALT Installation Completion

Installing PHP for Oracle Tuxedo

PHP must be built with the *embeded* option so it can be loaded by *mod_tuxedo*.

1. Install Oracle Instant Client for the PHP OCI8 Extension. Download the *basic* and *devel* packages from <http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html> and install with:

```
# rpm -i oracle-instantclient11.2-basic-11.2.0.3.0-1.x86_64.rpm
# rpm -i oracle-instantclient11.2-devel-11.2.0.3.0-1.x86_64.rpm
```

2. Download PHP from <http://php.net/downloads.php>. Use PHP 5.3.2 or higher.
3. Extract, configure and install PHP with the embedded option, for example with:

```
$ tar -jxf php-5.4.4.tar.bz2
$ cd php-5.4.4
$ ./configure --prefix=/home/cjones/php --with-oci8=instantclient \
              --enable-embed
$ make install
```

In this example, OCI8 will automatically locate the Instant Client RPMs. Adjust the configuration line if you use Oracle client libraries in a different location. Refer to the chapter *Installing and Configuring PHP*.

4. Copy a *php.ini* configuration file:

```
$ cp php8.ini-development $HOME/php/lib/php.ini
```

5. Edit the new *php.ini* and set the time zone, for example:

```
date.timezone = America/Los_Angeles
```

These steps have created the shared, embedded library */home/cjones/php/lib/libphp5.so*.

Installing Oracle Tuxedo into Apache

1. Typically Tuxedo is installed in Oracle HTTP Server as shown in the Tuxedo manuals. However for this example the packaged system Apache is used. Edit */etc/httpd/conf/httpd.conf* and, for the duration of this example, change the user and group so that Apache runs as the Tuxedo software owner, for example:

```
User cjones
Group cjones
```

You could alternatively install Apache 2.2 as the Tuxedo software owner, configuring it, for example, with *./configure --prefix=\$HOME/apache --enable-so --with-port=8888*. Modify any paths suggested below if you choose this option.

2. As the root users, copy the Tuxedo module into Apache's modules directory:

```
# cp /home/cjones/oracle/tuxedo11gR1/udataobj/mod_tuxedo.so \
      /etc/httpd/modules/
```

3. In */etc/httpd/conf/httpd.conf* load *mod_tuxedo.so*:

PHP and Oracle Tuxedo

```
LoadModule tuxedo_module modules/mod_tuxedo.so
```

4. Also in *httpd.conf*, locate the Directory directive for the document root */var/www/html*. Add an *<IfModule mod_tuxedo.c>* block at the bottom of the section:

```
<Directory "/var/www/html">
    .
    .
    <IfModule mod_tuxedo.c>
        AddHandler tuxedo-script .php
        TuxService myphpdemo
        TuxConfig /home/cjones/phpdemo/tuxconfig
    </IfModule>
</Directory>
```

This forwards requests for PHP files located in Apache's document root directory to the *myphpdemo* service of Tuxedo which is described in the */home/cjones/phpdemo/tuxconfig* configuration file. The next steps show how to create this service and how to create the configuration file. Change the user directory *cjones* to your Tuxedo software owner.

Because it is Tuxedo that invokes PHP, you could also configure Apache to run in the more efficient threaded mode than the standard Apache *mod_php* uses.

Configuring Oracle Tuxedo for PHP

The sample PHP application does a database query. Follow these steps to configure Tuxedo:

1. Make a directory for the sample application's configuration:

```
$ mkdir $HOME/phpdemo
```

2. Create a Tuxedo "Universal Bulletin Board" configuration file. The state of a Tuxedo system is kept in a "bulletin board", hence the configuration file name. A binary version of this configuration file will be used by Tuxedo administrative commands. For this example the configuration file *\$HOME/phpdemo/ubbphpdemo* contains:

```
*RESOURCES
IPCKEY          123456

DOMAINID        phpdemoapp
MASTER          phpdemo
MAXACCESSERS    50
MAXSERVERS      50
MAXSERVICES     100
MODEL           SHM
LDBAL           N

*MACHINES
DEFAULT:
APPDIR="/home/cjones/phpdemo"
```

```
UXCONFIG="/home/cjones/phpdemo/tuxconfig"
TUXDIR="/home/cjones/oracle/tuxedo11gR1"

"localhost.localdomain"    LMID=phpdemo

*GROUPS
GROUP1
        LMID=phpdemo GRPNO=1 OPENINFO=NONE

*Servers
DEFAULT:
        CLOPT="-A"

WEBHNDLR      SRVGRP=GROUP1 SRVID=1 CLOPT="-A -- -l PHP -S myphpdemo"
               MIN=5 MAX=50 RESTART=Y MAXGEN=255

*SERVICES
```

Change *cjones* in the directory paths to your user name. TUXCONFIG gives the name of the binary file that *ubbphpdemo* will be compiled to, as shown in the next section.

The IPCKEY value sets the address of shared memory, effectively “naming” the bulletin board.

The MAXACCESSERS, MAXSERVERS and MAXSERVICES values configure the size of Tuxedo data structures, determining the limits of the system. One PHP application is one service. Each HTTP request will be sent to a server for processing. MAXACCESSERS is the number of clients and servers that can connect to the bulletin board.

The MODEL is set to SHM for “shared memory”, which is suitable for this demonstration’s single machine setup.

The load balancer control, LDBAL, is set to *N* as recommended for a standalone server.

Other options exist, such as BLOCKTIME for limiting the request time. The general UBB configuration options are documented in the *UBBCONFIG(5)* section of the *File Formats, Data Descriptions, MIBs, and System Processes Reference* http://docs.oracle.com/cd/E18050_01/tuxedo/docs11gr1/rf5/rf5.html#wp3370051. There are other examples of configuration files in the example directories under *\$HOME/oracle/tuxedo11gR1/samples/atmi*.

The WEBHNDLR entry is a system process server used with *mod_tuxedo* for scripts written in PHP, Python or Ruby. In this example, it is defined to handle PHP requests. The name *myphpdemo* matches the TuxService name in *httpd.conf*. WEBHNDLR syntax is shown in the *Oracle SALT Command Reference* http://docs.oracle.com/cd/E18050_01/salt/docs11gr1/ref/comref.html#wp1185303.

PHP and Oracle Tuxedo

Your machine name must be given in the MACHINES section of *ubbphpdemo*. Without this, the *tmloadcf* command (used in the next section) will fail with *CMDTUX_CAT:868 ERROR: tmloadcf cannot run on a non-master node*. Use the name returned by your *hostname* command. If your machine has a long DHCP generated name, you can temporarily set a shorter name by executing this as root:

```
# hostname phptux
```

However if you do this command, connections to an Oracle database will fail with *ORA-24408: could not generate unique server group name* unless you also add the name to */etc/hosts*:

```
127.0.0.1 localhost.localdomain localhost phptux
```

You can also set the name in *httpd.conf* to avoid an Apache warning when the web server is started:

```
ServerName phptux:80
```

3. Build the binary configuration file from *ubbphpdemo*:

```
$ cd $HOME/phpdemo  
$ source env.sh  
$ tmloadcf -y ubbphpdemo
```

This creates the file *tuxconfig* specified by the *ubbphpdemo* TUXCONFIG directive. Tuxedo uses *tuxconfig* to set up the “bulletin board” on each server machine in the Tuxedo domain.

Starting and Managing Tuxedo

1. Create a file *\$HOME/phpdemo/env.sh* to set the environment for Tuxedo administration commands:

```
export TUXDIR=$HOME/oracle/tuxedo11gR1  
export PATH=$TUXDIR/bin:$PATH  
export LD_LIBRARY_PATH=$TUXDIR/lib:$HOME/php/lib:$LD_LIBRARY_PATH  
export APPDIR=$HOME/phpdemo  
export TUXCONFIG=$APPDIR/tuxconfig  
export FLDTBLDIR32=$TUXDIR/udataobj  
export FIELDTBL32=http.fml32
```

The “field table” *httpd.fml32* maps the HTTP header fields for *mod_tuxedo*. The library path includes the location of the PHP embedded library.

2. Run the file to set the environment:

```
$ source $HOME/phpdemo/env.sh
```

3. Apache needs the same environment. Edit */etc/sysconfig/httpd* and add these variables:

```
export TUXDIR=/home/youruser/oracle/tuxedo11gR1  
export FLDTBLDIR32=$TUXDIR/udataobj  
export FIELDTBL32=http.fml32
```

```
export LD_LIBRARY_PATH=$TUXDIR/lib:$LD_LIBRARY_PATH
```

If you installed your own Apache 2.2, add the variables to the `envvars` file, for example `/opt/apache/bin/envvars`, or otherwise make sure the shell that starts Apache has the values set.

Note that Apache does not invoke PHP and doesn't need the PHP library directory in `LD_LIBRARY_PATH`.

4. Use `tmboot` to start Tuxedo as the Tuxedo software owner:

```
$ tmboot -y
Booting all admin and server processes in /home/cjones/phpdemo/tuxconfig
INFO: Oracle Tuxedo, Version 11.1.1.2.0, 64-bit, Patch Level (none)

Booting admin processes ...

exec BBL -A :
process id=3625 ... Started.

Booting server processes ...

exec WEBHNDLR -A -- -l PHP -S myphpdemo :
process id=3626 ... Started.
exec WEBHNDLR -A -- -l PHP -S myphpdemo :
process id=3627 ... Started.
exec WEBHNDLR -A -- -l PHP -S myphpdemo :
process id=3628 ... Started.
exec WEBHNDLR -A -- -l PHP -S myphpdemo :
process id=3629 ... Started.
exec WEBHNDLR -A -- -l PHP -S myphpdemo :
process id=3630 ... Started.
6 processes started.
```

You can see the services running by using `tmadmin`, which gives information about the Tuxedo "Bulletin Boards":

```
$ tmadmin
> printservice
Service Name Routine Name Prog Name Grp Name ID Machine # Done Status
-----
myphpdemo SVCWEB WEBHNDLR GROUP1 1 simple 0 AVAIL
SVCWEB SVCWEB WEBHNDLR GROUP1 1 simple 0 AVAIL
myphpdemo SVCWEB WEBHNDLR GROUP1 2 simple 0 AVAIL
SVCWEB SVCWEB WEBHNDLR GROUP1 2 simple 0 AVAIL
myphpdemo SVCWEB WEBHNDLR GROUP1 3 simple 0 AVAIL
SVCWEB SVCWEB WEBHNDLR GROUP1 3 simple 0 AVAIL
myphpdemo SVCWEB WEBHNDLR GROUP1 4 simple 0 AVAIL
SVCWEB SVCWEB WEBHNDLR GROUP1 4 simple 0 AVAIL
myphpdemo SVCWEB WEBHNDLR GROUP1 5 simple 0 AVAIL
SVCWEB SVCWEB WEBHNDLR GROUP1 5 simple 0 AVAIL
```

For more information on `tmadmin` see *The Use of tmadmin(1)* in
http://docs.oracle.com/cd/E13203_01/tuxedo/tux64/tag/adminops.htm.

PHP and Oracle Tuxedo

There is a separately installable *Oracle Tuxedo System and Application Monitor (TSAM)* for Tuxedo that can be used to monitor and manage the Tuxedo and SALT. It includes graphing and alerting functionality. Data can be viewed in real time or historically. TSAM is available from the Tuxedo web pages,
<http://www.oracle.com/us/products/middleware/tuxedo/tsam/index.html>.

5. To handle requests for PHP scripts, start Apache as root:

```
# service httpd start
```

6. When finished with the example, shutdown Apache first:

```
# service httpd stop
```

Then shutdown Tuxedo as the Tuxedo software owner:

```
$ tmshutdown -y
```

Verifying PHP and Tuxedo

1. Create */var/www/html/phpinfo.php* containing:

```
<?php  
phpinfo();  
?>
```

2. Load the file in a browser *http://localhost/phpinfo.php*. The standard PHP configuration screen is shown.
3. Create a database query script, */var/www/html/ocitux.php*:

Script 1: ocitux.php

```
<?php  
  
$c = oci_connect("hr", "welcome", "localhost/XE");  
  
$s = oci_parse($c, "select city from locations");  
oci_execute($s);  
echo "<table border='1'>\n";  
while (($row = oci_fetch_array($s, OCI_ASSOC)) != false) {  
    echo " <tr>\n";  
    echo " <td>".htmlentities($row['CITY'], ENT_QUOTES)."</td>\n";  
    echo " </tr>\n";  
}  
echo "</table>\n";  
  
?>
```

4. This file can be run by loading the URL *http://localhost/ocitux.php*:

Beijing
Bern
Bombay
Geneva
Hiroshima
London
Mexico City
Munich
Oxford
Roma
Sao Paulo

Figure 86: Output from ocitux.php

Existing PHP applications can make use of Tuxedo via *mod_tuxedo* allowing them to be integrated into a highly managed environment.

PHP and Oracle Tuxedo

GLOBALIZATION

This chapter discusses global application development in a PHP and Oracle Database environment. It addresses the basic tasks associated with developing and deploying global Internet applications, including developing locale awareness, constructing HTML content in the user-preferred language, and presenting data following the cultural conventions of the locale of the user.

Building a global Internet application that supports different locales requires good development practices. A locale refers to a national language and the region in which the language is spoken. The application itself must be aware of the locale preference of the user and be able to present content following the cultural conventions expected by the user. It is important to present data with appropriate locale characteristics, such as the correct date and number formats. Oracle Database is fully internationalized to provide a global platform for developing and deploying global applications.

Establishing the Environment Between Oracle and PHP

Correctly setting up the connectivity between the PHP engine and the Oracle database is the first step in building a global application. It guarantees data integrity across all tiers. Most Internet based standards support Unicode as a character encoding. This chapter focuses on using Unicode as the character set for data exchange.

Setting the Language, Territory and Character Set With NLS_LANG

OCI8 is an Oracle OCI application, and rules that apply to OCI also apply to PHP. Oracle locale behavior (including the client character set used in OCI applications) is defined by Oracle's *national language support* NLS_LANG environment variable. This environment variable has the form:

```
<language>_<territory>.<character set>
```

For example, for a German user in Germany running an application in Unicode, NLS_LANG should be set to:

```
GERMAN_GERMANY.AL32UTF8
```

The language and territory settings control Oracle behaviors such as the Oracle date format, error message language, and the rules used for sort order. The character set AL32UTF8 is the Oracle name for UTF-8. You should use a character set compatible with the operating system, for example on Windows where there is no UTF-8, you could use WE8MSWIN1252 to match the English Windows code page 1252.

In some operating system environments you may also need to set the LC_ALL and LANG environment variables, for example these could be added to /etc/sysconfig/httpd:

```
export NLS_LANG=FRENCH_FRANCE.WE8IS08859P1
export LC_ALL=french
export LANG=french
```

Globalization

The NLS_LANG character set can also be passed as a parameter to the OCI8 connection functions. Doing this is recommended for connection performance reasons, even if NLS_LANG is also set.

If the character set used by PHP OCI8 does not match the character set used by the database, Oracle will try to convert when data is inserted and queried. This may reduce runtime performance. Also an accurate mapping is not always be possible, resulting in data being converted to question marks.

There are other environment variables that can be used to set particular aspects of globalization. For information on NLS_LANG and other Oracle language environment variables, see the Oracle documentation.

The section *Setting the Oracle Environment on Linux* in the *Installing and Configuring PHP* chapter discusses how environment variables can be set for Apache.

If the globalization settings are invalid, PHP may fail to connect to Oracle and give an error like *ORA-12705: Cannot access NLS data files or invalid environment specified*.

Some globalization values can be changed per connection:

```
$s = oci_parse($c, "alter session set nls_territory=germany nls_language=german");
oci_execute($s);
```

After executing this, Oracle error messages will be in German and some localization features such as the default date format will have changed.

Caution: When changing the session settings for a *persistent* PHP connection, the next time the connection is used, the old values will still be in effect.

If PHP is installed on Oracle HTTP Server, you must set NLS_LANG as an environment variable in *\$ORACLE_HOME/opmn/conf/opmn.xml*:

```
<ias-component id="HTTP_Server">
    <process-type id="HTTP_Server" module-id="OHS">
        <environment>
            <variable id="PERL5LIB"
value="D:\oracle\1012J2EE\Apache\Apache\mod_perl\site\5.6.1\lib"/>
            <variable id="PHPRC" value="D:\oracle\1012J2EE\Apache\Apache\conf"/>
            <variable id="NLS_LANG" value="german_germany.al32utf8"/>
        </environment>
        <module-data>
            <category id="start-parameters">
                <data id="start-mode" value="ssl-disabled"/>
            </category>
        </module-data>
        <process-set id="HTTP_Server" numprocs="1"/>
    </process-type>
</ias-component>
```

You must restart the Web listener to implement the change.

To find the language and territory currently used by PHP, and the character set with which the database stores data, execute:

```
$s = oci_parse($c,
    "select sys_context('userenv', 'language') as nls_lang from dual");
```

```
oci_execute($s);
$res = oci_fetch_array($s, OCI_ASSOC);
echo $res['NLS_LANG'] . "\n";
```

Output is of the form:

```
AMERICAN_AMERICA.WE8MSWIN1252
```

Setting the Oracle Number Format With NLS_NUMERIC_CHARACTERS

OCI8 fetches numbers as PHP strings. The conversion is done by Oracle before the values are returned to the application code. It is possible to lose precision or get errors in PHP when strings not using the US decimal and thousands separator conventions expected by PHP are later cast to numbers for arithmetic. To avoid problems it is recommended to explicitly set the number conversion format.

The following examples illustrate the differences in the decimal character and group separator between the United States and Germany when numbers are converted to strings by Oracle.

```
SQL> alter session set nls_territory = america;
Session altered.

SQL> select employee_id EmpID,
  2 substr(first_name,1,1)||'.'||last_name "EmpName",
  3 to_char(salary, '99G999D99') "Salary"
  4 from employees
  5 where employee_id < 105;

  EMPID EmpName          Salary
-----  -----
    100 S.King            24,000.00
    101 N.Kochhar         17,000.00
    102 L.De Haan         17,000.00
    103 A.Hunold           9,000.00
    104 B.Ernst            6,000.00

SQL> alter session set nls_territory = germany;
Session altered.

SQL> select employee_id EmpID,
  2 substr(first_name,1,1)||'.'||last_name "EmpName",
  3 to_char(salary, '99G999D99') "Salary"
  4 from employees
  5 where employee_id < 105;

  EMPID EmpName          Salary
-----  -----
    100 S.King            24.000,00
    101 N.Kochhar         17.000,00
    102 L.De Haan         17.000,00
    103 A.Hunold           9.000,00
```

Globalization

104 B.Ernst

6.000,00

The format '99G999D99' contains the 'G' thousands separator and 'D' decimal separator at the appropriate places in the desired output number. In the two territories, the actual character displayed is different.

The equivalent PHP example is:

Script 125: numformat.php

```
<?php  
  
$c = oci_connect('hr', 'welcome', 'localhost/XE');  
  
$s = oci_parse($c, "alter session set nls_territory = germany");  
oci_execute($s);  
  
$s = oci_parse($c, "select 123.567 as num from dual");  
oci_execute($s);  
$r = oci_fetch_array($s, OCI_ASSOC);  
  
$n1 = $r['NUM']; // value as fetched  
var_dump($n1);  
  
$n2 = (float)$n1; // now cast it to a number  
var_dump($n2);  
  
?>
```

The output is:

```
string(7) "123,567"  
float(123)
```

If NLS_TERRITORY had instead been set to [america](#) the output would have been correct:

```
string(7) "123.567"  
float(123.567)
```

The problem can also occur depending on the territory component of NLS_LANG, or the value of NLS_NUMERIC_CHARACTERS. The latter variable can be used to override the number format while other territory settings remain in effect. It can be set as an environment variable in /etc/sysconfig/httpd or in the shell that runs the PHP executable:

```
export NLS_LANG=AMERICAN_AMERICA.AL32UTF8  
export NLS_NUMERIC_CHARACTERS=".,"
```

The NLS_NUMERIC_CHARACTERS environment variable is only evaluated if NLS_LANG is also set.

The value can also be set with a LOGON trigger, or by using an ALTER SESSION command in PHP:

```
$s = oci_parse($c, "alter session set nls_numeric_characters = '.',,'");  
oci_execute($s);
```

Changing the setting in PHP is likely to be the slowest of the methods.

Establishing the Environment Between Oracle and PHP

The tip *Do Not Set the Date or Numeric Format Unnecessarily* in the chapter *Connecting to Oracle Using OCI8* shows how an ALTER SESSION command can be used in a database LOGON trigger.

Setting the Oracle Date Format With NLS_DATE_FORMAT

The basic date format used by Oracle depends on your Globalization settings, such as the value in NLS_LANG.

The three different date presentation formats in Oracle are standard, short, and long dates. The following examples illustrate the differences between the short date and long date formats for both the United States and Germany.

```
SQL> alter session set nls_territory = america nls_language = american;
```

```
SQL> select employee_id EmpID,
  2 substr(first_name,1,1)||'.'||last_name "EmpName",
  3 to_char(hire_date,'DS') "Hiredate",
  4 to_char(hire_date,'DL') "Long HireDate"
  5 from employees
 6* where employee_id <105;
```

EMPID	EmpName	Hiredate	Long HireDate
100	S.King	06/17/1987	Wednesday, June 17, 1987
101	N.Kochhar	09/21/1989	Thursday, September 21, 1989
102	L.De Haan	01/13/1993	Wednesday, January 13, 1993
103	A.Hunold	01/03/1990	Wednesday, January 3, 1990
104	B.Ernst	05/21/1991	Tuesday, May 21, 1991

```
SQL> alter session set nls_territory=germany nls_language=german;
```

```
SQL> select employee_id EmpID,
  2 substr(first_name,1,1)||'.'||last_name "EmpName",
  3 to_char(hire_date,'DS') "Hiredate",
  4 to_char(hire_date,'DL') "Long HireDate"
  5 from employees
 6* where employee_id <105;
```

EMPID	EmpName	Hiredate	Long HireDate
100	S.King	17.06.87	Mittwoch, 17. Juni 1987
101	N.Kochhar	21.09.89	Donnerstag, 21. September 1989
102	L.De Haan	13.01.93	Mittwoch, 13. Januar 1993
103	A.Hunold	03.01.90	Mittwoch, 3. Januar 1990
104	B.Ernst	21.05.91	Dienstag, 21. Mai 1991

In addition to these three format styles you can customize the format using many other date format specifiers. Search the Oracle SQL language documentation for “datetime format elements” to see a list.

If the date format derived from the NLS_LANG setting is not the one you want for your PHP session, you can override the format by setting the environment variable NLS_DATE_FORMAT in /etc/sysconfig/httpd or in the shell that runs the PHP executable:

```
export NLS_LANG=AMERICAN_AMERICA.AL32UTF8
```

Globalization

```
export NLS_DATE_FORMAT='YYYY-MM-DD HH24:MI:SS'
```

Note if you set NLS_DATE_FORMAT as an environment variable, you also need to set NLS_LANG otherwise the variable is ignored.

Alternatively you can set it in a LOGON trigger, or change it after connecting in PHP:

```
$s = oci_parse($c, "alter session set nls_date_format='YYYY-MM-DD HH24:MI:SS'");  
oci_execute($s);
```

Subsequent queries will return the new format:

```
$s = oci_parse($c, "select sysdate from dual");  
$r = oci_execute($s);  
$row = oci_fetch_array($s, OCI_ASSOC);  
echo "Date is " . $row["SYSDATE"] . "\n";
```

The output is:

```
Date is 2007-08-01 11:43:30
```

One advantage of setting the date format globally instead of using [TO_CHAR\(\)](#) is it allows PHP and Oracle to share a common format for inserts and queries:

Script 126: dateformat.php

```
<?php  
  
$c = oci_connect('hr', 'welcome', 'localhost/XE');  
  
// Set default Oracle date format  
$s = oci_parse($c, "alter session set nls_date_format='YYYY-MM-DD HH24:MI:SS'");  
oci_execute($s);  
  
// This PHP Date format matches the new Oracle format  
$d = date('Y-m-d H:i:s');  
echo "Inserting $d\n";  
$s = oci_parse($c, "insert into employees  
    (employee_id, last_name, email, hire_date, job_id)  
    values (1, 'Jones', 'cj@example.com', :dt, 'ST_CLERK')");  
oci_bind_by_name($s, ":dt", $d);  
oci_execute($s);  
  
$s = oci_parse($c, "select hire_date from employees where employee_id = 1");  
oci_execute($s);  
oci_fetch_all($s, $res);  
var_dump($res);  
  
?>
```

The output is:

```
Inserting 2008-10-23 04:01:17  
array(1) {  
  ["HIRE_DATE"]=>  
  array(1) {  
    [0]=>
```

```
    string(19) "2008-10-23 04:01:17"
}
```

Date formats can be explicitly altered by a language setting:

```
SQL> select to_char(systimestamp,
  'Dy, dd Mon yyyy hh24:mi:ss tzhtzm', 'NLS_DATE_LANGUAGE=American') from dual;
TO_CHAR(SYSTIMESTAMP, 'DY,DDMONYYYYHH24:MI:SSTZHTZ
-----
Thu, 07 Jun 2012 17:05:55 -0700

SQL> select to_char(systimestamp,
  'Dy, dd Mon yyyy hh24:mi:ss tzhtzm', 'NLS_DATE_LANGUAGE=German') from dual;
TO_CHAR(SYSTIMESTAMP, 'DY,DDMONYYYYHH24:MI:SST
-----
Do, 07 Jun 2012 17:05:56 -0700
```

Here, the day of the week has been translated.

Setting the Default Session Time Zone With ORA_SDTZ

The ORA_SDTZ environment variable sets the time zone used when querying TIMESTAMP WITH LOCAL TIME ZONE columns. It also affects TIMESTAMP values converted to TIMESTAMP WITH TIME ZONE or TIMESTAMP WITH LOCAL TIME ZONE data types. It does not affect SYSDATE and SYSTIMESTAMP values which use the database server time.

You can set ORA_SDTZ to a timezone, an offset or a region:

```
$ export ORA_SDTZ='-05:00'
```

Manipulating Strings

To handle data in PHP, there are several extensions that can be used. The common extensions are *mbstring*, *intl* and *iconv*. These can be installed and configured like any other PHP source code or PECL extension.

Once *mbstring* is installed, you can change the behavior of the standard PHP string functions by setting *mbstring.func_overload* in *php.ini* to one of the *Overload* settings. For more information, see the PHP *mbstring* reference manual at <http://php.net/mbstring>.

Your application code should use functions such as *mb_strlen()* to calculate the number of characters in strings. This may return different values than *strlen()*, which returns the number of bytes in a string.

The *intl* package implements some International Components for Unicode (ICU) functionality such as Collators and Transliterations.

Determining the Locale of the User

In a global environment, your application should accommodate users with different locale preferences. Once it has determined the preferred locale of the user, the application should

Globalization

construct HTML content in the language of the locale and follow the cultural conventions implied by the locale.

A common method to determine the locale of a user is from the default ISO locale setting of the browser. Usually a browser sends its locale preference setting to the HTTP server with the *Accept-Language* HTTP header. If the *Accept-Language* header is NULL, then there is no locale preference information available, and the application should fall back to a predefined default locale.

The following PHP code retrieves the ISO locale from the *Accept-Language* HTTP header through the `$_SERVER` Server variable.

```
$s = $_SERVER["HTTP_ACCEPT_LANGUAGE"]
```

The *intl* package's `Locale::acceptFromHttp` method could also be used.

Developing Locale Awareness

Once the locale preference of the user has been determined, the application can call locale-sensitive functions, such as date, time, and monetary formatting to format the HTML pages according to the cultural conventions of the locale.

When you write global applications implemented in different programming environments, you should enable the synchronization of user locale settings between the different environments. For example, PHP applications that call PL/SQL procedures should map the ISO locales to the corresponding NLS_LANGUAGE and NLS_TERRITORY values and change the parameter values to match the locale of the user before calling the PL/SQL procedures. The PL/SQL UTL_I18N package contains mapping functions that can map between ISO and Oracle locales.

Table 18 shows how some commonly used locales are defined in ISO and Oracle environments.

Table 18: Locale representations in ISO, SQL and PL/SQL programming environments.

Locale	Locale ID	NLS_LANGUAGE	NLS_TERRITORY
Chinese (P.R.C.)	zh-CN	SIMPLIFIED CHINESE	CHINA
Chinese (Taiwan)	zh-TW	TRADITIONAL CHINESE	TAIWAN
English (U.S.A)	en-US	AMERICAN	AMERICA
English (United Kingdom)	en-GB	ENGLISH	UNITED KINGDOM
French (Canada)	fr-CA	CANADIAN FRENCH	CANADA
French (France)	fr-FR	FRENCH	FRANCE
German	de	GERMAN	GERMANY
Italian	it	ITALIAN	ITALY
Japanese	ja	JAPANESE	JAPAN
Korean	ko	KOREAN	KOREA
Portuguese (Brazil)	pt-BR	BRAZILIAN PORTUGUESE	BRAZIL
Portuguese	pt	PORTUGUESE	PORUGAL

Locale	Locale ID	NLS_LANGUAGE	NLS_TERRITORY
Spanish	es	SPANISH	SPAIN

Encoding HTML Pages

The encoding of an HTML page is important information for a browser and an Internet application. You can think of the page encoding as the character set used for the locale that an Internet application is serving. The browser must know about the page encoding so that it can use the correct fonts and character set mapping tables to display the HTML pages. Internet applications must know about the HTML page encoding so they can process input data from an HTML form.

Instead of using different native encodings for the different locales, Oracle recommends that you use UTF-8 (Unicode encoding) for all page encodings. This encoding not only simplifies the coding for global applications, but it also enables multilingual content on a single page.

Specifying the Page Encoding for HTML Pages

You can specify the encoding of an HTML page either in the HTTP header, or in HTML page header.

Specifying the Encoding in the HTTP Header

To specify HTML page encoding in the HTTP header, include the *Content-Type* HTTP header in the HTTP specification. It specifies the content type and character set. The *Content-Type* HTTP header has the following form:

```
Content-Type: text/html; charset=utf-8
```

The *charset* parameter specifies the encoding for the HTML page. The possible values for the *charset* parameter are the IANA names for the character encodings that the browser supports.

It is important to be consistent using the page encoding. For example, the [htmlentities\(\)](#) encoding parameter should match the page encoding so that characters are recognized and escaped properly. Otherwise your application will be vulnerable to security attacks.

Specifying the Encoding in the HTML Page Header

Use this method primarily for static HTML pages. To specify HTML page encoding in the HTML page header, specify the character encoding in the HTML header as follows:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

The *charset* parameter specifies the encoding for the HTML page. As with the *Content-Type* HTTP Header, the possible values for the *charset* parameter are the IANA (Internet Assigned Numbers Authority) names for the character encodings that the browser supports.

Specifying the Page Encoding in PHP

You can specify the encoding of an HTML page in the *Content-Type* HTTP header in PHP by setting the *default_charset* configuration variable in *php.ini*:

Globalization

```
default_charset = UTF-8
```

Note the default value of this character set directive was changed to UTF-8 in PHP 5.4. You can alternatively use the PHP [header\(\)](#) function to set the content type:

```
header('Content-Type: text/html; charset=utf-8');
```

This setting does not imply any conversion of outgoing pages. Your application must ensure that the server-generated pages are encoded in UTF-8.

Organizing the Content of HTML Pages for Translation

Making the user interface available in the local language of the user is a fundamental task in globalizing an application. Translatable sources for the content of an HTML page belong to the following categories:

- Text strings included in the application code
- Static HTML files, images files, and template files such as CSS
- Dynamic data stored in the database

Strings in PHP

You should externalize translatable strings within your PHP application logic, so that the text is readily available for translation. These text messages can be stored in flat files or database tables depending on the type and the volume of the data being translated. PHP's *gettext* extension is often used for this purpose.

Static Files

Static files such as HTML and text stored as images are readily translatable. When these files are translated, they should be translated into the corresponding language with UTF-8 as the file encoding. To differentiate the languages of the translated files, stage the static files of different languages in different directories or with different file names.

Data from the Database

Dynamic information such as product names and product descriptions is typically stored in the database. To differentiate various translations, the database schema holding this information should include a column to indicate the language. To select the desired language, you must include a [WHERE](#) clause in your query.

Presenting Data Using Conventions Expected by the User

Data in the application must be presented in a way that conforms to the expectation of the user. Otherwise, the meaning of the data can be misinterpreted. For example, the date '12/11/09' implies '11th December 2009' in the United States, whereas in the United Kingdom it means '12th November 2009'. Similar confusion exists for number and monetary formats of the users. For example, the symbol '.' is a decimal separator in the United States; in Germany this symbol is a thousand separator.

Presenting Data Using Conventions Expected by the User

Different languages have their own sorting rules. Some languages are collated according to the letter sequence in the alphabet, some according to the number of stroke counts in the letter, and some languages are ordered by the pronunciation of the words. Presenting data not sorted in the linguistic sequence that your users are accustomed to can make searching for information difficult and time consuming.

Depending on the application logic and the volume of data retrieved from the database, it may be more appropriate to format the data at the database level rather than at the application level. Oracle offers many features that help to refine the presentation of data when the locale preference of the user is known. Earlier in this chapter, examples using date and numeric formats were shown. The next section provides another example of a locale-sensitive operation in SQL.

Oracle Linguistic Sorts

Spain traditionally treats *ch*, *ll* as well as *ñ* as unique letters, ordered after *c*, *l* and *n*, respectively. The following examples illustrate the effect of using a Spanish sort against the employee names *Chen* and *Chung*.

```
SQL> alter session set nls_sort = binary;

SQL> select employee_id EmpID,
  2      last_name "Last Name"
  3  from employees
  4 where last_name like 'C%'
  5 order by last_name;

  EMPID Last Name
-----
  187 Cabrio
  148 Cambrault
  154 Cambrault
  110 Chen
  188 Chung
  119 Colmenares

SQL> alter session set nls_sort = spanish_m;

SQL> select employee_id EmpID,
  2      last_name "Last Name"
  3  from employees
  4 where last_name like 'C%'
  5 order by last_name;

  EMPID Last Name
-----
  187 Cabrio
  148 Cambrault
  154 Cambrault
  119 Colmenares
  110 Chen
  188 Chung
```

Globalization

Oracle Error Messages

The NLS_LANGUAGE parameter also controls the language of the database error messages being returned from the database. Setting this parameter prior to submitting your SQL statement ensures that language-specific database error messages will be logged by the application.

Consider the following server message:

```
ORA-00942: table or view does not exist
```

When the NLS_LANGUAGE parameter is set to *French*, the server message appears as follows:

```
ORA-00942: table ou vue inexistante
```

For more discussion of globalization support features in Oracle Database 11g XE, see *Working in a Global Environment* in the *Oracle Database Express Edition 2 Day Developer's Guide*.

TESTING PHP AND THE OCI8 EXTENSION

This chapter discusses running the PHP test suite on Linux. The PHP source code includes command-line tests for all the core functionality and extensions. You should run these tests after building PHP on Linux.

It is a good idea to pro-actively test your applications with PHP “release candidates” and snapshots, <http://snaps.php.net>. Please report problems so they can be fixed prior to each final PHP release. This ensures PHP continues doing what you need it to do. Consider contributing new tests to the PHP community. Adding tests that are relevant to your application reduces the risks of PHP developers breaking PHP features important to you. Please send new tests or report issues with PHP’s test suite to php-qa@lists.php.net.

Running OCI8 Tests

The tests in the PHP source directory `ext/oci8/tests` verify the behavior of the OCI8 extension. For the tests to run successfully some configuration is needed:

1. Set the Oracle connection details by editing `ext/oci8/tests/details.inc` and updating the credentials section:

```
$user = "system";
$password = "systempwd";
$dbase = "localhost/XE";
```

The tests rely on being able to create tables, types, stored procedures, and so on. If you change `$user`, you may have to grant that database user some extra privileges.

With OCI8 1.4, if you are using Oracle Database 11g Connection Pooling, also set:

```
$test_drcp = TRUE
```

To use DRCP, the pool must also be enabled in the database and the PHP connection string must specify that a pooled database server should be used, for example:

```
$dbase = "localhost/XE:pooled";
```

Instead of editing `details.inc`, equivalent environment variables can be set in a terminal window:

```
$ export PHP_OCI8_TEST_USER=system
$ export PHP_OCI8_TEST_PASS=systempwd
$ export PHP_OCI8_TEST_DB=localhost/XE
$ export PHP_OCI8_TEST_DRCP=FALSE
```

2. Check that `variables_order` has 'E' in your `php.ini`, for example:

Testing PHP and the OCI8 Extension

```
variables_order = "EGPCS"
```

Without this flag, the Oracle environment variables are not propagated through the test system and OCI8 tests can fail to connect.

- Set any necessary Oracle environment variables in your shell. For example, for PHP linked with Oracle Database 11g XE enter:

```
$ source /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
```

For other database editions run *oraenv* and enter the identifier of your database:

```
$ source /usr/local/bin/oraenv
ORACLE_SID = [ ] ? orcl
```

If Oracle is on a different machine, you may manually need to set the environment variables that are otherwise set by these scripts.

- Some of the OCI8 tests take longer than the default test timeout period, particularly if the database is on a remote server. Either increase the test suite timeout:

```
$ export TEST_TIMEOUT=600
```

or skip the slow tests:

```
$ export SKIP_SLOW_TESTS=1
```

- Run PHP's test suite with:

```
$ cd php-5.4
$ make test
```

If you want to run just the OCI8 tests use:

```
$ make test TESTS=ext/oci8
```

Each test script is executed and its status reported.

```
=====
PHP      : /home/myhome/php-5.4/sapi/cli/php
PHP_SAPI  : cli
PHP_VERSION : 5.4.4
ZEND_VERSION: 2.4.0
PHP_OS   : Linux - Linux cj-ol62x64 2.6.39-100
INI actual : /usr/local/apache/conf/php.ini
More .INIs :
CWD      : /home/myhome/php-5.4
Extra dirs :
VALGRIND  : not used
=====

Running selected tests.
PASS oci_bind_array_by_name() and invalid values 1 [array_bind_001.phpt]
PASS oci_bind_array_by_name() and invalid values 2 [array_bind_002.phpt]
PASS oci_bind_array_by_name() and invalid values 3 [array_bind_003.phpt]
...
```

Successful tests begin with PASS. Tests that are to be skipped in the current configuration are marked SKIP. Failing tests are marked FAIL. A summary of the test results is given at the completion of the tests.

Running a Single Test

To run only one or two tests, execute *run-tests.php* directly and pass the test names as parameters. For example, to run a script called *demotest.phpt*, do the following:

```
$ cd php-5.4
$ export TEST_PHP_EXECUTABLE=/home/myhome/php-5.4/sapi/cli/php
$ $TEST_PHP_EXECUTABLE run-tests.php ext/oci8/tests/demotest.phpt
```

The variable *TEST_PHP_EXECUTABLE* is used inside the controlling *run-tests.php* script when it invokes *demotest.phpt*. In the example, the same PHP binary is also used to run the *run-tests.php* script itself, but they could be different executables. The test output is similar to the previous output.

Tests that Fail

The output of failing tests is kept for analysis. For example, if *ext/oci8/tests/demotest.phpt* fails, the following files will be in *ext/oci8/tests*:

Table 19: Test files and their contents.

File name	File Contents
<i>demotest.phpt</i>	Test framework script
<i>demotest.php</i>	PHP file executed
<i>demotest.out</i>	Test output
<i>demotest.exp</i>	Expected output as coded in the <i>.phpt</i> file
<i>demotest.diff</i>	Difference between actual and expected output
<i>demotest.log</i>	Actual and expected output in a single file

It is common for the full PHP test suite to report some test failures. This is because many PHP extensions are wrappers around system libraries with slightly different behaviors on different platforms or versions. Also some system configurations may restrict access, for example to network ports. You can review commonly seen differences at <http://qa.php.net/reports>.

The PHP test infrastructure doesn't practically allow tests to accept differences across all versions of Oracle Database. Also if you use the latest OCI8 extension with an older version of PHP, differences in PHP's `var_dump()` output can make tests appear to fail.

Creating OCI8 Tests

To add a new OCI8 test, create a *phpt* file in *ext/oci8/tests* using the test file format. When you run *make test* the new file is automatically run, or you can call it explicitly as shown above. For example, create *demotest.phpt*:

Testing PHP and the OCI8 Extension

Script 127: *demotest.phpt*

```
--TEST--
Demo to test the Test system
--SKIPIF--
<?php
if (!extension_loaded('oci8')) die("skip no oci8 extension");
?>
--FILE--
<?php
require (__DIR__ . '/connect.inc');
$s = oci_parse($c, "select user from dual");
oci_execute($s);
oci_fetch_all($s, $res);
var_dump($res);
echo "Done\n";
?>
====DONE====
<?php exit(0); ?>
--EXPECT--
array(1) {
    ["USER"]=>
    array(1) {
        [0]=>
        string(6) "SYSTEM"
    }
}
====DONE====
```

The test begins with a comment that is displayed when the test runs. The **SKIPIF** section causes the test to be skipped when the OCI8 extension is not enabled in PHP. The **FILE** section is the PHP code to be executed. The **EXPECT** section has the expected output. The file *connect.inc* is found in *ext/oci8/tests/connect.inc*. It includes *details.inc*, connects to Oracle, and returns the connection resource in **\$c**.

The line **====DONE====** is outside the executed script and is echoed verbatim, verifying that the script completed. The extra PHP block containing **exit(0)** makes calling the test directly in PHP without using *run-tests.php* a little cleaner:

```
$ php demotest.phpt
--TEST--
Demo to test the Test system
--SKIPIF--
--FILE--
array(1) {
    ["USER"]=>
    array(1) {
        [0]=>
        string(6) "SYSTEM"
    }
}
Done
====DONE====
```

Running like this, some of the *demotest.php* test framework content is shown, but code executes and only the actual, and not the expected, output is displayed. This can make it easier to quickly validate tests, without having to set TEST_PHP_EXECUTABLE and invoke *run-tests.php*.

The page [Writing Tests at http://qa.php.net/write-test.php](http://qa.php.net/write-test.php) shows other sections a test file can have, including ways to set arguments and *php.ini* parameters. This page also has generic examples and helpful information on writing tests. The sections of a *.phpt* file are documented at http://qa.php.net/phpt_details.php.

Writing good units tests is an art. Making a test portable, accurate, simple and self-diagnosing requires fine judgment and tidy programming.

OCI8 Test Helper Scripts

Along with *connect.inc* and *details.inc*, there are several useful scripts in *ext/oci8/tests* for creating and dropping basic tables and types, these include:

- *create_table.inc*
- *create_type.inc*
- *drop_table.inc*
- *drop_type.inc*

You can include these in any new tests you write.

Make sure new tests create any needed objects and drop them at the end of the test. Use unique names for each object, for example if the test *demotest.php* needs to create (and drop) a table, a good table name would be DEMOTEST_TAB.

Configuring the Database For Testing

Sometimes it is possible for rapidly executing OCI8 test scripts to flood the database with connections. This may be noticeable with Oracle Database XE, which has smaller defaults. Random tests fail with errors like the following:

```
ORA-12516 TNS:listener could not find available handler with matching protocol stack
```

or:

```
ORA-12520: TNS:listener could not find available handler for requested type of server
```

The best general solution is to use DRCP connecting pooling solution as discussed in the chapter *PHP Connection Pooling and High Availability*. An alternative method suitable for the specific case of testing is to increase the number of “processes” that Oracle can handle. This method is also applicable when connection pooling is not desired or available, for example when using Oracle Database 10.2.

To increase the number of processes in Oracle:

1. You may need to [su](#) as the *oracle* user so you have operating system privileges when starting SQL*Plus:

```
$ su - oracle  
Password:
```

Testing PHP and the OCI8 Extension

2. Set the Oracle environment variables needed by SQL*Plus, for example:

```
$ source /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
```

3. Use SQL*Plus to connect as a privileged database user:

```
$ sqlplus / as sysdba
```

4. Check the current value of processes using the `SHOW PARAMETER PROCESSES` command:

```
SQL> show parameter processes
NAME          TYPE        VALUE
-----
...
processes      integer     40
```

5. Increase the value to, say, 100:

```
SQL> alter system set processes=100 scope=spfile;
System altered.
```

6. Restart the database using the `SHUTDOWN IMMEDIATE`, followed by the `STARTUP` command:

```
SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup
ORACLE instance started.
Total System Global Area 289406976 bytes
Fixed Size 1258488 bytes
Variable Size 96472072 bytes
Database Buffers 188743680 bytes
Redo Buffers2932736 bytes
Database mounted.
Database opened.
```

7. Use the `SHOW PARAMETER PROCESSES` command to confirm the new value is in effect:

```
SQL> show parameter processes
NAME          TYPE        VALUE
-----
...
processes      integer     100
```

8. Exit SQL*Plus using the `EXIT` command:

```
SQL> exit
```

9. Now the tests can be run again:

```
$ make test TESTS=ext/oci8
```

Testing PHP Applications

You should verify your applications work correctly with any new PHP binary before putting it into production. This gives load and real-life testing not possible with PHP's command-line test suite. Application level testing brings even more challenges. There are several PHP test frameworks with sophisticated features that might be suited to create application test suites. They include *PHPUnit* and *SimpleTest*. These are not covered in this manual.

Testing PHP and the OCI8 Extension

TRACING OCI8 INTERNALS

This appendix discusses tracing the OCI8 internals. To see exactly what calls to the Oracle database the OCI8 extension makes, you can turn on debugging output. This is mostly useful for the maintainers of the OCI8 extension.

Enabling OCI8 Debugging output

Tracing can be turned on in your script with `oci_internal_debug()`. For a script that connects and does an insert:

Script 128: trace.php

```
<?php

oci_internal_debug(1);          // turn on tracing

$c = oci_connect("hr", "welcome", "localhost/XE");
$s = oci_parse($c, "insert into testtable values ('my data')");
oci_execute($s, OCI_NO_AUTO_COMMIT);

?>
```

You get output like:

```
OCI8 DEBUG: OCINlsEnvironmentVariableGet at (phpsrc/php-5.4/ext/oci8/oci8.c:1873)
OCI8 DEBUG L1: Got NO cached connection at (phpsrc/php-5.4/ext/oci8/oci8.c:1918)
OCI8 DEBUG: OCIEnvNlsCreate at (phpsrc/php-5.4/ext/oci8/oci8.c:2916)
OCI8 DEBUG: OCIHandleAlloc at (phpsrc/php-5.4/ext/oci8/oci8.c:2737)
OCI8 DEBUG: OCIHandleAlloc at (phpsrc/php-5.4/ext/oci8/oci8.c:2749)
OCI8 DEBUG: OCIHandleAlloc at (phpsrc/php-5.4/ext/oci8/oci8.c:2766)
OCI8 DEBUG: OCIAttrSet at (phpsrc/php-5.4/ext/oci8/oci8.c:2786)
OCI8 DEBUG: OCIAttrSet at (phpsrc/php-5.4/ext/oci8/oci8.c:2795)
OCI8 DEBUG: OCISessionPoolCreate at (phpsrc/php-5.4/ext/oci8/oci8.c:2807)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8.c:2821)
OCI8 DEBUG L1: create_spool: (0xa881548) at (phpsrc/php-5.4/ext/oci8/oci8.c:2825)
OCI8 DEBUG L1: using shared pool: (0xa881548) at (phpsrc/php-5.4/ext/oci8/oci8.c:3143)
OCI8 DEBUG: OCIHandleAlloc at (phpsrc/php-5.4/ext/oci8/oci8.c:3154)
OCI8 DEBUG: OCIHandleAlloc at (phpsrc/php-5.4/ext/oci8/oci8.c:3164)
OCI8 DEBUG: OCIAttrSet at (phpsrc/php-5.4/ext/oci8/oci8.c:3173)
OCI8 DEBUG: OCIAttrSet at (phpsrc/php-5.4/ext/oci8/oci8.c:3185)
OCI8 DEBUG: OCIAttrGet at (phpsrc/php-5.4/ext/oci8/oci8.c:3197)
OCI8 DEBUG: OCIAttrGet at (phpsrc/php-5.4/ext/oci8/oci8.c:3198)
OCI8 DEBUG L1: (numopen=0)(numbusy=0)(numfree=0) at (phpsrc/php-5.4/ext/oci8/oci8.c:3200)
OCI8 DEBUG: OCISessionGet at (phpsrc/php-5.4/ext/oci8/oci8.c:3211)
OCI8 DEBUG: OCIAttrGet at (phpsrc/php-5.4/ext/oci8/oci8.c:3226)
OCI8 DEBUG: OCIAttrGet at (phpsrc/php-5.4/ext/oci8/oci8.c:3228)
OCI8 DEBUG: OCIContextGetValue at (phpsrc/php-5.4/ext/oci8/oci8.c:3230)
```

Tracing OCI8 Internals

```
OCI8 DEBUG: OCIContextGetValue at (phpsrc/php-5.4/ext/oci8/oci8.c:3325)
OCI8 DEBUG: OCIMemoryAlloc at (phpsrc/php-5.4/ext/oci8/oci8.c:3332)
OCI8 DEBUG: OCIContextSetValue at (phpsrc/php-5.4/ext/oci8/oci8.c:3346)
OCI8 DEBUG: OCIAttrSet at (phpsrc/php-5.4/ext/oci8/oci8.c:3256)
OCI8 DEBUG L1: New Non-Persistent Connection address: (0xb452d160) at
(phpsrc/php-5.4/ext/oci8/oci8.c:2167)
OCI8 DEBUG L1: num_persistent=(0), num_links=(1) at (phpsrc/php-
5.4/ext/oci8/oci8.c:2169)
OCI8 DEBUG: OCIHandleAlloc at (phpsrc/php-5.4/ext/oci8/oci8_statement.c:57)
OCI8 DEBUG: OCISstmtPrepare2 at (phpsrc/php-5.4/ext/oci8/oci8_statement.c:72)
OCI8 DEBUG: OCIAttrSet at (phpsrc/php-5.4/ext/oci8/oci8_statement.c:123)
OCI8 DEBUG: OCIAttrGet at (phpsrc/php-5.4/ext/oci8/oci8_statement.c:433)
OCI8 DEBUG: OCISstmtExecute at (phpsrc/php-5.4/ext/oci8/oci8_statement.c:461)
OCI8 DEBUG: OCISstmtRelease at (phpsrc/php-5.4/ext/oci8/oci8_statement.c:766)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8_statement.c:774)
OCI8 DEBUG: OCITransRollback at (phpsrc/php-5.4/ext/oci8/oci8.c:2241)
OCI8 DEBUG: OCISessionRelease at (phpsrc/php-5.4/ext/oci8/oci8.c:2402)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8.c:2288)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8.c:2291)
OCI8 DEBUG: OCISessionPoolDestroy at (phpsrc/php-5.4/ext/oci8/oci8.c:3293)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8.c:3297)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8.c:3301)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8.c:3305)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8.c:1144)
OCI8 DEBUG: OCIHandleFree at (phpsrc/php-5.4/ext/oci8/oci8.c:1149)
```

Many of these calls just allocate local resources (*handles*) and set local state (*attributes*), but some require a *round trip* to the database.

One of these is the [OCITransRollback\(\)](#) call near the end of the script. The `OCI_NO_AUTO_COMMIT` flag said not to auto-commit and there was no explicit [oci_commit\(\)](#) call. As part of PHP's end of HTTP request shutdown at the conclusion of the script, the rollback was issued.

Note: if you change to auto-commit mode you will not see a call to [OCITransCommit\(\)](#) because the commit message is piggy-backed with Oracle's statement execution call, thus saving a round-trip. If a script only inserts one row it is fine to auto-commit. Otherwise, do the transaction management yourself.

OCI8 PHP.INI PARAMETERS

This appendix gives an overview of the *php.ini* parameters for the OCI8 extension. Detailed discussion of their use is covered in previous chapters.

The parameters can be changed in the PHP configuration file *php.ini*, for example:

```
oci8.default_prefetch = 75
```

Variables can also be set in Apache's *httpd.conf*:

```
<IfModule mod_php5.c>
    php_admin_flag oci8.old_oci_close_semantics On
    php_admin_value oci8.connection_class MYPHPAPP
</IfModule>
```

The web server must be restarted for any changes to take effect.

The location of *php.ini* can be found by running the command line PHP executable `php --ini`. The current values of the OCI8 parameters can be found by running `php --ri oci8`. Loading *phpinfo.php* in a browser will show all parameters and is equivalent to `php -i`:

Script 129: phpinfo.php

```
<?php
phpinfo();
?>
```

If you might have multiple versions of PHP installed, check the settings via both command line and a browser.

If you are using Windows, remember to edit *php.ini* using administrative privileges.

Enabling PHP OCI8 in *php.ini*

On Windows, you will need to enable OCI8 in *php.ini*. If you are using OCI8 with Oracle 11gR2 client libraries than add:

```
extension=php_oci8_11g.dll
```

If you are using OCI8 with Oracle 10gR2 client libraries than add:

```
extension=php_oci8.dll
```

Only one of these extensions can be enabled at a time.

If PHP OCI8 was compiled as a shared extension on Linux, you will need to enable it in *php.ini* with:

```
extension=oci8.so
```

If OCI8 was built statically into the PHP executable you do not need to set this.

On both platforms, make sure *extension_dir* is set to the directory containing the shared library.

OCI8 php.ini Parameters

PHP OCI8 *php.ini* Parameters

The *php.ini* directives for OCI8 are shown in the next table:

Table 20: OCI8 *php.ini* parameters

Name	Default	Valid Range	Description
oci8.connection_class	null	A short string	A user-chosen name for Oracle Database 11g Connection Pooling (DRCP). In general, use the same name for all web servers running the same application. Can also be set with ini_set() . Introduced in OCI8 1.3.
oci8.default_prefetch	100 Prior to OCI8 1.3.4 the default was 10	>= 0 Prior to OCI8 1.4 it had to be >= 1	The number of extra rows that Oracle fetches and internally buffers whenever a query row is physically retrieved from the database. This improves query performance by reducing the number of network accesses to the database. From Oracle 11gR2 onwards this parameter also affects fetching from REF CURSORS. Also see oci_set_prefetch() .
oci8.events	Off	Off or On	Allows PHP to receive Fast Application Notification (FAN) events from Oracle to give immediate notification of a database node or network failure. The database must also be configured to post events. Introduced in OCI8 1.3.
oci8.max_persistent	-1 -1 means no limit	>= -1	Maximum number of persistent connections each PHP process caches. Note this is not a system-wide total.
oci8.old_oci_close_semantics	Off	Off or On	Toggles whether oci_close() uses the old behavior, which was a "no-op".

PHP OCI8 php.ini Parameters

Name	Default	Valid Range	Description
oci8.persistent_timeout	-1	> -1 -1 means no timeout	How many seconds a persistent connection is allowed to remain idle before being terminated by its PHP process. PHP processes will check this each time the process is reused.
oci8.ping_interval	60	>= 0 -1 means no extra check occurs	How many seconds a persistent connection can be unused before an extra check during oci_pconnect() verifies the database connection is still valid.
oci8.privileged_connect	Off	Off or On	Toggles whether SYSDBA and SYSOPER connections are permitted.
oci8.statement_cache_size	20	>= 0	Improves database performance by caching the given number of SQL statements in PHP. Setting it to 0 disables statement caching.

OCI8 php.ini Parameters

OCI8 FUNCTION NAMES IN PHP 4 AND PHP 5

In PHP 5 several extensions including OCI8 underwent function name standardization. PHP 4 functions like `OCILogin()` became `oci_connect()`, the function `OCIParse()` became `oci_parse()`, and so on. Although deprecated, the old OCI8 names still exist as aliases. PHP 4 scripts do not necessarily need to be changed when migrating. PECL OCI8 releases from 1.1 onwards have the new function names.

Note: The OCI8 1.4 extension builds and runs with PHP 4 and PHP 5. If you are still using PHP 4 and cannot upgrade to PHP 5, you should replace the OCI8 code with the new version to get improved stability and performance optimizations. Steps to install from PECL are given in this book.

Function names in PHP are case insensitive. There seems to be a common acceptance for PHP 4 names to be written with a capitalized prefix but the PHP 5 names to be all lowercase.

Table 21 shows the PHP 4 OCI8 functions and gives their PHP 5 replacements. Note several functions such as `ocifreedesc()` and `ocilobclose()` were only usable as methods in PHP 4.

Table 21: Relationship between OCI8's PHP 4 and PHP 5 function names.

Operation	Action	PHP 4 Name	PHP 5 Name
Connection	Open connection	<code>ociologon()</code>	<code>oci_connect()</code>
	Open new connection	<code>ocinlogon()</code>	<code>oci_new_connect()</code>
	Persistent connection	<code>ociplogon()</code>	<code>oci_pconnect()</code> (and new <i>php.ini</i> parameters)
	Close connection	<code>ocilogoff()</code>	<code>oci_close()</code>
Cursor	Open cursor	<code>ocinewcursor()</code>	<code>oci_new_cursor()</code>
	Close cursor	<code>ocifreecursor()</code> <code>ocifreestatement()</code>	<code>oci_free_statement()</code>
Parsing	Parse statement	<code>ociparse()</code>	<code>oci_parse()</code>
Binding	Bind variable	<code>ocibindbyname()</code>	<code>oci_bind_by_name()</code>

OCI8 Function Names in PHP 4 and PHP 5

Operation	Action	PHP 4 Name	PHP 5 Name
	Bind array	Not available	oci_bind_array_by_name()
Defining	Define output variables	ocidefinebyname()	oci_define_by_name()
Execution	Execute statement	ociexecute()	oci_execute()
Fetching	Fetch row	ocifetch()	oci_fetch()
	Fetch row	ocifetchinto()	oci_fetch_array() oci_fetch_row() oci_fetch_assoc() oci_fetch_object()
	Fetch all rows	ocifetchstatement()	oci_fetch_all()
	Fetch column	ociresult()	oci_result()
	Is the column NULL?	ocicolumnisnull()	oci_field_is_null()
	Cancel Fetch	ocicancel()	oci_cancel()
Transaction Management	Commit	ocicommit()	oci_commit()
	Rollback	ocirollback()	oci_rollback()
Descriptors		ocinewdescriptor()	oci_new_descriptor()
		ocifreedesc()	oci_free_descriptor()
Error Handling		ocierror()	oci_error()
Long Objects (LOBs)		ocisavelob()	OCI-Lob::save()
		ocisavelobfile()	OCI-LOB::import()
		ociwritelobtofile()	OCI-LOB::export()
		ociwritetemporarylob()	OCI-Lob::writeTemporary()
		ociloadlob()	OCI-Lob::load()
		ocicloselob()	OCI-Lob::close()
Collections		ocinewcollection()	oci_new_collection()
		ocifreecollection()	OCI-Collection::free()

OCI8 Function Names in PHP 4 and PHP 5

Operation	Action	PHP 4 Name	PHP 5 Name
		ocicollappend()	OCI-Collection::append()
		ocicollgetelem()	OCI-Collection::getElem()
		ocicollassign()	OCI-Collection::assign()
		ocicollassignelem()	OCI-Collection::assignElem()
		ocicollsize()	OCI-Collection::size()
		ocicollmax()	OCI-Collection::max()
		ocicolltrim()	OCI-Collection::trim()
Metadata	Statement type	ocistatementtype()	oci_statement_type()
	Name of result column	ocicolumnname()	oci_field_name()
	Size of result column	ocicolumnsize()	oci_field_size()
	Data type of result column	ocicolumntype()	oci_field_type()
	Data type of result column	ocicolumntyperaw()	oci_field_type_raw()
	Precision of result column	ocicolumnprecision()	oci_field_precision()
	Scale of result column	ocicolumnscale()	oci_field_scale()
	Number of rows affected	ocirowcount()	oci_num_rows()
	Number of columns returned	ocinumcols()	oci_num_fields()
Changing Password		ocipasswordchange()	oci_password_change()

OCI8 Function Names in PHP 4 and PHP 5

Operation	Action	PHP 4 Name	PHP 5 Name
Monitoring, Tuning and Auditing		Not available	oci_set_module_name() oci_set_action() oci_set_client_info() oci_set_client_identifier()
Tracing		ociinternaldebug()	oci_internal_debug()
Upgrading	Allow multiple versions of PL/SQL objects to be used	Not available	oci_set_edition()
Oracle Client Version		Not available	oci_client_version()
Server Version		ociserverversion()	oci_server_version()
Tuning		ociisetprefetch()	oci_set_prefetch() (and new <i>php.ini</i> parameter)

THE OBSOLETE ORACLE EXTENSION

This appendix compares the long obsolete Oracle PHP extension to the current OCI8 extension. Very rarely you might come across PHP scripts that use this early Oracle API. It is no longer included with PHP. The functionality it offered was limited. Upgrading to the new OCI8 extension might be as simple as enabling the newer OCI8 extension in the PHP binary and renaming the old function calls in your scripts. Pay some attention to transaction management and connection handling changes. It is wise to test your application and make sure it behaves as you expect it to.

Oracle and OCI8 Comparison

The following table shows the general relationship between the obsolete and current extensions:

Table 22: Relationship between the OCI8 and the obsolete Oracle extensions.

Operation	Action	ORA function (obsolete)	OCI8 function
Connection	Open connection	ora_logon()	oci_connect()
	Open new connection	Not available	oci_new_connect()
	Persistent connection	ora_plogon()	oci_pconnect() (and new <i>php.ini</i> parameters)
	Close connection	ora_logoff()	oci_close()
	Used for monitoring, auditing and VPD	Not available	oci_set_client_identifier()
Cursor	Open cursor	ora_open()	oci_new_cursor()
	Close cursor	ora_close()	oci_free_statement()
Parsing	Parse statement	ora_parse()	oci_parse()
Binding	Bind variable	ora_bind()	oci_bind_by_name()
	Bind array	Not available	oci_bind_array_by_name()
Execution	Execute statement	ora_exec()	oci_execute()

The Obsolete Oracle Extension

Operation	Action	ORA function (obsolete)	OCI8 function
	Prepare, execute and fetch	ora_do()	oci_parse() Followed by oci_execute() and one of: oci_fetch_all() oci_fetch_array() oci_fetch_assoc() oci_fetch_object() oci_fetch_row() oci_fetch()
Fetching	Fetch row	ora_fetch()	oci_fetch()
	Fetch row	ora_fetch_into	oci_fetch_array() oci_fetch_row() oci_fetch_assoc() oci_fetch_object()
	Fetch all rows	Not available	oci_fetch_all()
	Fetch column	ora_getcolumn()	oci_result()
	Is the column NULL?	Not available	oci_field_is_null()
	Cancel Fetch	Not available	oci_cancel()
Transaction Management	Commit	ora_commit()	oci_commit()
	Commit mode	ora_commiton() ora_commitoff()	Pass OCI_NO_AUTO_COMMIT flag to oci_execute()
	Rollback	ora_rollback()	oci_rollback()
Error Handling		ora_error() ora_errorcode()	oci_error()

Oracle and OCI8 Comparison

Operation	Action	ORA function (obsolete)	OCI8 function
Long Objects (LOBS)		Not available	OCI-Lob->append OCI-Lob->close OCI-Lob->eof OCI-Lob->erase OCI-Lob->export OCI-Lob->flush OCI-Lob->free OCI-Lob->getBuffering OCI-Lob->import OCI-Lob->load OCI-Lob->read OCI-Lob->rewind OCI-Lob->save OCI-Lob->saveFile OCI-Lob->seek OCI-Lob->setBuffering OCI-Lob->size OCI-Lob->tell OCI-Lob->truncate OCI-Lob->write OCI-Lob->writeTemporary OCI-Lob->writeToFile
Collections		Not available	OCI-Collection->append OCI-Collection->assign OCI-Collection->assignElem OCI-Collection->free OCI-Collection->getElem OCI-Collection->max OCI-Collection->size OCI-Collection->trim
Metadata	Statement type	Not available	oci_statement_type()
	Name of result column	ora_columnname()	oci_field_name()

The Obsolete Oracle Extension

Operation	Action	ORA function (obsolete)	OCI8 function
	Size of result column	ora_columnsize()	oci_field_size()
	Data type of result column	ora_columntype()	oci_field_type() oci_field_type_raw()
	Precision of result column	Not available	oci_field_precision()
	Scale of result column	Not available	oci_field_scale()
	Number of rows effected	ora_numrows()	oci_num_rows()
	Number of columns returned	ora_numcols()	oci_num_fields()
SQL Monitoring	Send metadata to Oracle for tracing and monitoring	Not available	oci_set_module_name() oci_set_action() oci_set_client_info()
Changing Password		Not available	oci_password_change()
Tracing		Not available	oci_internal_debug()
Oracle Client and Server Versions		Not available	oci_client_version() oci_server_version()
Tuning		Not available	oci_set_prefetch() (and new <i>php.ini</i> parameter)
Upgrading	Allow multiple versions of PL/SQL objects to be used	Not available	oci_set_edition()

REOURCES

This appendix gives links to documentation, resources and articles discussed in this book, and to other web sites of interest. This book itself can be found free online at:

<http://www.oracle.com/technetwork/topics/php/underground-php-oracle-manual-098250.html>

General Information and Forums

PHP Developer Center on Oracle Technology Network (OTN)

<http://www.oracle.com/technetwork/topics/php/whatsnew/>

OTN PHP Discussion Forum

<http://www.oracle.com/technetwork/forums/php/>

Blog: Christopher Jones on OPAL

<http://blogs.oracle.com/opal/>

AskTom

General Oracle language and application design help

<http://asktom.oracle.com/>

Oracle Linux

<https://linux.oracle.com/>

Oracle Support

<http://support.oracle.com/>

Oracle's Free and Open Source Software

<http://oss.oracle.com/>

Oracle Documentation and Whitepapers

Oracle 11g Release 2 Documentation Library

<http://www.oracle.com/pls/db112/homepage>

Resources

Oracle TimesTen In-Memory Database Documentation

<http://www.oracle.com/technetwork/products/timesten/documentation/>

Oracle Database Express Edition Documentation

<http://www.oracle.com/technetwork/products/express-edition/documentation/>

Oracle Call Interface Programmer's Guide 11g Release 2 (11.2)

http://docs.oracle.com/cd/E11882_01/appdev.112/e10646/toc.htm

Oracle Database Express Edition 2 Day + PHP Developer's Guide 11g Release 2 (11.2)

http://docs.oracle.com/cd/E17781_01/appdev.112/e18555/toc.htm

Oracle Database Net Services Administrator's Guide 11g Release 2 (11.2)

http://docs.oracle.com/cd/E11882_01/network.112/e10836/toc.htm

Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)

http://docs.oracle.com/cd/E11882_01/appdev.112/e25519/toc.htm

Oracle Database SQL Language Reference 11g Release 2 (11.2)

http://docs.oracle.com/cd/E11882_01/server.112/e26088/toc.htm

Oracle Tuxedo Documentation

<http://www.oracle.com/technetwork/middleware/tuxedo/documentation/index.html>

Whitepaper: *Oracle Tuxedo: An Enterprise Platform for Dynamic Languages*

<http://www.oracle.com/technetwork/middleware/tuxedo/tuxedo-dynamic-langs-twp-401471.pdf>

Selected PHP and Oracle Books

Oracle Database 11g PL/SQL Programming

Michael McLaughlin, Oracle Press, 2008.

Oracle Database AJAX & PHP Web Application Development

Lee Barney and Michael McLaughlin, Oracle Press, 2008

PHP Oracle Web Development

Yuli Vasiliev, Packt Publishing, 2007

Beginning PHP and Oracle: From Novice to Professional

W. Jason Gilmore and Bob Bryla, Apress, 2007

Application Development with Oracle & PHP on Linux for Beginners

Ivan Bayross and Sharanam Shah, Shroff Publishers & Distributors, 2nd Edition 2007

Oracle Database 10g Express Edition PHP Web Programming

Michael McLaughlin, Osbourne Oracle Press, 2006

Easy Oracle PHP: Create Dynamic Web Pages with Oracle Data

Mladen Gogala, Rampant TechPress, 2006

Software and Source Code

PHP Distribution Releases

Source and Windows binaries

<http://www.php.net/downloads.php>

PHP Snapshots (includes OCI8)

Snapshots of PHP's source code and Windows binaries

<http://snaps.php.net/>

PHP Source (includes OCI8)

<http://git.php.net/?p=php-src.git>

<https://github.com/php/php-src>

PECL OCI8 Source package (standalone)

<http://pecl.php.net/package/oci8>

Zend Server

<http://www.oracle.com/technetwork/topics/php/zend-server-096314.html>

Oracle Instant Client

<http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html>

Oracle NetBeans IDE

<http://netbeans.org/>

Oracle SQL Developer

<http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/>

ADOdb Database Abstraction Library for PHP (and Python)

<http://adodb.sourceforge.net/>

Resources

PHPUnit PHP Unit Tester

<http://www.phpunit.de/>

SimpleTest PHP Unit Tester

<http://www.simpletest.org/>

Xdebug - Debugger and Profiler Tool for PHP

<http://www.xdebug.org/>

PHP Links

PHP Home Page

<http://php.net/>

PHP Documentation

<http://php.net/docs.php>

PHP Oracle OCI8 Documentation

<http://php.net/oci8>

PHP Quality Assurance Site

<http://qa.php.net/>

PHP Bug System

<http://bugs.php.net/>

PHP Wiki

<http://wiki.php.net/>

GLOSSARY

Anonymous Block

A PL/SQL block that appears in your application and is not named or stored in the database. In many applications, PL/SQL blocks can appear wherever SQL statements can appear. A PL/SQL block groups related declarations and statements. Because these blocks are not stored in the database, they are generally for one-time use.

AWR

Automatic Workload Repository. Used to store and report statistics on database performance.

Binding

A method of including data in SQL statements that allows SQL statements to be efficiently reused with different data.

BFILE

The BFILE data type stores unstructured binary data in operating-system files outside the database. A BFILE column or attribute stores a file locator that points to an external file containing the data.

BLOB

The BLOB data type stores unstructured binary data in the database.

CHAR

The CHAR data type stores fixed-length character strings in the database.

CLOB and NCLOB

The CLOB and NCLOB data types store up to eight terabytes of character data in the database. CLOBs store database character set data, and NCLOBs store Unicode national character set data.

Collection Type

A collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. PL/SQL data types TABLE and VARRAY enable collection types such as arrays, bags, lists, nested tables, sets and trees.

Connection Identifier

The string used to identify which database to connect to, for example, [localhost/XE](#).

Connection String

The full string used to identify which database to connect to commonly used for SQL*Plus. It contains the username, password and connect identifier, for example, hr/welcome@localhost/XE.

Data Dictionary

A set of tables and views that are used as a read-only reference about the database.

Database

A database stores and retrieves data. Each database consists of one or more data files. Although you may have more than one database per machine, typically a single Oracle database contains multiple schemas. A schema is often equated with a user. Multiple applications can use the same database without any conflict by using different schemas.

Database Link

A pointer that defines a one-way communication path from an Oracle Database server to another database server. A database link connection allows local users to access data on a remote database.

Database Name

The name of the database. In PHP, this is the text used in [oci_connect\(\)](#) calls. Also see *Easy Connect*.

Data type

Each column value and constant in a SQL statement has a data type, which is associated with a specific storage format, constraints, and a valid range of values. When you create a table, you must specify a data type for each of its columns. For example, NUMBER, or DATE.

DATE

The DATE data type stores point-in-time values (dates and times) in a database table.

DATETIME

An extension of the date type with greater precision than DATE and with an associated timezone.

DBA

Database Administrator. A person who administers the Oracle database. This person is a specialist in Oracle databases, and would usually have SYSDBA access to the database.

DDL

SQL's Data Definition Language. SQL statements that define the database structure or schema, like [CREATE](#), [ALTER](#), and [DROP](#).

DML

SQL's Data Manipulation Language. SQL statements that define or manage the data in the database, like [SELECT](#), [INSERT](#), [UPDATE](#) and [DELETE](#).

DRCP

Database Resident Connection Pooling. Introduced in Oracle Database 11g, this connection pooling allows multiple client processes and machines to share connection data structures on the database server, reducing the memory requirements of large numbers of users.

Easy Connect

A simple hostname and database connect identifier that is used to identify which database to connect to.

Git

An open source version control system used for development of PHP.

HR

The sample user created by default with an Oracle seed database installation. The HR user has access to the Human Resources demonstration tables in the HR schema.

Index

Indexes are optional structures associated with database tables. Indexes can be created to increase the performance of data retrieval.

Instance

The Oracle Instance is the running component of an Oracle database server. When an Oracle database is started, a system global area (SGA) is allocated and Oracle background processes are started. The combination of the background processes and memory buffers is called an Oracle instance.

Instant Client

The Oracle Instant Client is a small set of libraries, which allows applications to connect to an Oracle Database. A subset of the full Oracle Client, it requires minimal installation but has full functionality. Instant Client is downloadable from OTN and is usable and distributable for free.

LOB

A large object. LOBS may be persistent (stored in the database) or temporary. See *CLOB*, *BLOB*, and *BFILE*.

LOB Locator

A “pointer” to LOB data.

Materialized View

A materialized view provides access to table data by storing the results of a query in a separate database schema object. Unlike an ordinary view, which does not take up any storage space or contain any data, a materialized view contains the rows resulting from a query against one or more base tables or views.

NCHAR and NVARCHAR2

NCHAR and NVARCHAR2 are Unicode data types that store Unicode character data in the database.

NUMBER

The NUMBER data type stores fixed and floating-point numbers in the database.

Object Privilege

A right to perform a particular action on a specific database schema object. Different object privileges are available for different types of schema objects. The privilege to delete rows from the DEPARTMENTS table is an example of an object privilege.

OCI

Oracle Call Interface. The main C API for accessing the Oracle Database. First introduced in version eight of Oracle Database it is commonly referred to as OCI8.

OCI8

PHP's main extension for accessing Oracle Database It is implemented using calls to Oracle's OCI API.

ORACLE_HOME install

An Oracle Client or Oracle Database install. These installs contain all software required by PHP in a directory hierarchy. This set of directories includes binaries, utilities, configuration scripts, demonstration scripts and error message files for each component of Oracle. Any program using Oracle typically requires the ORACLE_HOME environment variable to be set to the top level installation directory.

Oracle Net

The networking component of Oracle that connects client tools such as PHP to local or remote databases. The Oracle Net listener is a process that handles connection requests from clients and passes them to the target database.

OTN

The Oracle Technology Network is Oracle's free repository of articles on Oracle technologies. It also hosts software downloads and many discussion forums, including one on PHP.

Package

A group of PL/SQL procedures, functions, and variable definitions stored in the Oracle database. Procedures, functions, and variables in packages can be called from other packages, procedures, or functions.

PDO

PHP Data Objects. A data abstraction layer for PHP. It consists of two parts, a generic layer and database vendor specific drivers, each of which may expose database specific features.

PEAR

The PHP Extension and Application Repository is a repository for reusable packages written in PHP.

PECL

The PHP Extension Community Library is a repository of PHP extensions that can be linked into the PHP binary.

PHP

A popular, interpreted scripting language commonly used for web applications. PHP is a recursive acronym for “PHP: Hypertext Preprocessor”.

php.ini

The configuration file used by PHP. Many (but not all) options that are set in *php.ini* can also be set at runtime using [ini_set\(\)](#). Systems can be configured to read multiple initialization files.

PL/SQL

Oracle's procedural language extension to SQL. It is a server-side, stored, procedural language that enables you to mix SQL statements with procedural constructs. With PL/SQL, you can create and run PL/SQL program units such as procedures, functions, and packages. PL/SQL program units generally are categorized as anonymous blocks, stored functions, stored procedures, and packages.

Prepared Statement

A SQL statement that has been parsed by the database. In Oracle, it is generally called a parsed statement.

Procedures and Functions

A PL/SQL procedure or function is a schema object that consists of a set of SQL statements and other PL/SQL programming constructs, grouped together, stored in the database, and run as a unit to solve a specific problem or perform a set of related tasks.

Regular Expression

A pattern used to match data. Oracle has several functions that accept regular expressions.

Round Trip

A call and return sequence from PHP OCI8 to the Database performed by the underlying driver libraries. Each round trip takes network time and machine CPU resources. The fewer round trips performed, the more scalable a system is likely to be. PHP OCI8 functions may initiate zero or many round trips.

Schema

A schema is a collection of database objects. A schema is owned by a database user and has the same name as that user. Schema objects are the logical structures that directly refer to the database's data. Schema objects include structures like tables, views, and indexes.

SDK

Software Development Kit. Oracle Instant Client has an SDK for building programs that use the Instant Client libraries.

Sequence

A sequential series of Oracle integers of up to 38 digits defined in the database.

Service Name

A service name is a string that is the global database name, comprised of the database name and domain name. You can obtain it from the SERVICE_NAMES parameter in the database initialization parameter file or by using [SHOW PARAMETERS](#) in SQL*Plus. It is used during connection to identify which database to connect to.

SID (System Identifier)

The system identifier is commonly used to mean the database name alias in the connection string.

SID (Session Identifier)

A session identifier is a unique number assigned to each database user session when a user connects to the database.

SQL*Plus

The traditional command line tool for executing SQL statements available with all Oracle databases. Although recently superseded by GUI tools like Oracle's free SQL Developer, SQL*Plus remains hugely popular. It is also convenient to show examples using SQL*Plus.

Stored Procedures and Functions

A PL/SQL block that Oracle stores in the database and can be called by name from an application. Functions are different than procedures in that functions return a value when executed. When you create a stored procedure or function, Oracle parses the procedure or function, and stores its parsed representation in the database.

Synonym

A synonym is an alias for any database table, view, materialized view, sequence, procedure, function, package, type, Java class schema object, user-defined object type, or another synonym.

SYS

An Oracle database administrative user account name. SYS has access to all base tables and views for the database data dictionary.

SYSDBA

An Oracle database system privilege that, by default, is assigned only to the SYS user. It enables SYS to perform high-level administrative tasks such as starting up and shutting down the database.

SYSOPER

Similar to SYSDBA, but with a limited set of privileges that allows basic administrative tasks without having access to user data.

SYSTEM

An Oracle database administrative user account name that is used to perform all administrative functions other than starting up and shutting down the database.

System privilege

The right to perform a particular action, or to perform an action on any database schema objects of a particular type. For example, the privileges to create tables and to delete the rows of any table in a database.

Table

Tables are the basic unit of data storage. Database tables hold all user-accessible data. Each table has columns and rows.

Tablespace

Tablespaces are the logical units of Oracle data storage made up of one or more datafiles. Tablespaces are often created for individual applications because tablespaces can be conveniently managed. Users are assigned a default tablespace that holds all the data the users creates. A database is made up of default and DBA-created tablespaces.

Temporary Table

A Global Temporary Table is a special table that holds session-private data that exists only for the duration of a transaction or session. The table is created before the application runs.

TimesTen

An in-memory database that can be standalone with file-backed storage, or can act as a cache for Oracle Database.

Tnsnames.ora

The Oracle Net configuration file used for connecting to a database. The file maps an alias to a local or remote database and allows various configuration options for connections. The alias is used in the PHP connection string. TNS stands for Transparent Network Substrate.

Transaction

A sequence of SQL statements whose changes are either all committed, or all rolled back.

Trigger

A stored procedure associated with a database table, view, or event. The trigger can be called after the event, to record it, or take some follow-up action. The trigger can be called before the event, to prevent erroneous operations or fix new data so that it conforms to business rules.

ULN

Unbreakable Linux Network. It provides Oracle Linux software patches, updates and fixes for support subscribers.

User

A database user is often equated to a schema. Each user connects to the database with a username and secret password, and has access to tables, and so on, in the database.

VARCHAR and VARCHAR2

These data types store variable-length character strings in the database. The names are currently synonyms but VARCHAR2 is recommended to ensure maximum compatibility of applications in future.

View

Views are customized presentations of data in one or more tables or other views. A view can also be considered a stored query. Views do not actually contain data. Rather, they derive their data from the tables on which they are based, referred to as the base tables of the views.

VPD

Virtual Private Database. An Oracle Database features that allows you to create security policies restricting access to data at the row and column level.

XMLType

XMLType is a database data type that can be used to store XML data in table columns.

The Underground PHP and Oracle Manual

About this Book

This book is for PHP programmers developing applications for Oracle Database. It bridges the gap between the many PHP and Oracle books available and shows how to use the PHP scripting language with Oracle Database. You may be starting out with PHP for your Oracle Database. You may be a PHP programmer wanting to learn Oracle. You may be unsure how to install PHP or Oracle. Or you may just want to know the latest best practices. This book gives you the fundamental building blocks needed to create high-performance PHP Oracle Web applications.

About the Authors

Christopher Jones works for Oracle on dynamic scripting languages with a strong focus on PHP. He is a lead maintainer of PHP's open source OCI8 extension and works closely with the PHP community. He also helps ensure that future versions of Oracle Database are compatible with PHP. He is the author of various technical articles on PHP and Oracle technology, and has presented at conferences including PHP|Tek, the International PHP Conference, the O'Reilly Open Source Convention, and ZendCon. He also helps present Oracle PHP tutorials and PHPFests worldwide.

Alison Holloway is a Consulting Technical Writer at Oracle with a number of years experience in advanced technology. She has presented at various PHP conferences. Most recently she has been working with Oracle VM.