

THE UNDERGROUND PHP AND ORACLE[®] MANUAL



CHRISTOPHER JONES AND ALISON HOLLOWAY

The Underground PHP and Oracle® Manual, Release 1.5, December 2008.
Copyright © 2008, Oracle. All rights reserved.

Authors: Christopher Jones and Alison Holloway

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

The latest edition of this book is available online at:

<http://www.oracle.com/technology/tech/php/pdf/underground-php-oracle-manual.pdf>

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Oracle, JD Edwards, and PeopleSoft are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

CONTENTS

Chapter 1 Introduction.....	1
Who Should Read This Book?.....	1
Introduction to Oracle.....	1
Databases and Instances.....	2
Tablespaces.....	2
Schemas and Users.....	2
Introduction to PHP.....	2
Chapter 2 Getting Started With PHP.....	5
Creating and Editing PHP Scripts.....	5
PHP Syntax Overview.....	5
Running PHP Scripts.....	8
Running PHP Scripts in a Browser.....	8
Running Scripts with Command Line PHP.....	8
Debugging PHP Scripts.....	8
Chapter 3 PHP Oracle Extensions.....	11
PHP Oracle Extensions.....	11
Oracle Extension.....	11
OCI8 Extension.....	11
PDO Extension.....	12
PHP Database Abstraction Libraries.....	13
ADODB.....	13
PEAR DB.....	14
PEAR MDB2.....	14
Getting the OCI8 Extension.....	14
OCI8 and Oracle Installation Options.....	15
Getting the PDO Extension.....	16
Zend Core for Oracle.....	17
The PHP Release Cycle.....	17
Chapter 4 Installing Oracle Database 10g Express Edition.....	19
Oracle Database Editions.....	19
Oracle Database XE.....	19
Installing Oracle Database XE on Linux.....	20
Installing Oracle Database XE on Debian, Ubuntu, and Kubuntu.....	21
Installing Oracle Database XE on Windows.....	22
Testing the Oracle Database XE Installation.....	24
Configuring Oracle Database XE.....	25
Setting the Oracle Database XE Environment Variables on Linux.....	25

Enabling Database Startup and Shutdown from Menus on Linux.....	26
Starting and Stopping the Listener and Database.....	26
Enabling Remote Client Connection.....	29
Chapter 5 Using Oracle Database.....	31
Oracle Application Express.....	31
Logging In To Oracle Application Express.....	31
Unlocking the HR User.....	32
Creating Database Objects.....	33
Working with SQL Scripts.....	37
Creating a PL/SQL Procedure.....	38
Creating a Database User.....	40
Monitoring Database Sessions.....	42
Database Backup and Recovery.....	44
Oracle SQL*Plus.....	47
Starting SQL*Plus.....	48
Executing SQL and PL/SQL Statements in SQL*Plus.....	49
Controlling Query Output in SQL*Plus.....	49
Running Scripts in SQL*Plus.....	50
Information On Tables in SQL*Plus.....	50
Accessing the Demonstration Tables in SQL*Plus.....	51
Oracle SQL Developer.....	51
Creating a Database Connection.....	51
Creating a Table.....	54
Executing a SQL Query.....	55
Editing, Compiling and Running PL/SQL.....	57
Running Reports.....	59
Creating Reports.....	61
Chapter 6 Installing Apache HTTP Server.....	63
Installing Apache HTTP Server on Linux.....	63
Starting and Stopping Apache HTTP Server.....	64
Configuring Apache HTTP Server on Linux.....	64
Installing Apache HTTP Server on Windows.....	64
Starting and Stopping Apache HTTP Server.....	65
Chapter 7 Installing PHP.....	67
Installing PHP with OCI8 on Linux.....	67
Installing OCI8 Using a Local Database.....	67
Installing OCI8 Using Oracle Instant Client.....	69
Upgrading PHP with PECL OCI8 on Linux.....	70
Upgrading OCI8 as a Static Library on Linux	70
Upgrading OCI8 on Linux Using the PECL Channel.....	71
Upgrading OCI8 as a Shared Library on Linux.....	72

Installing PHP With OCI8 on Windows.....	73
Installing OCI8 Using a Local Database on Windows.....	73
Installing OCI8 with Instant Client on Windows.....	74
Upgrading OCI8 on Windows.....	75
Installing OCI8 with Oracle Application Server on Linux.....	76
Installing PHP With PDO.....	78
Installing PDO on Linux.....	79
Installing PDO on Windows.....	80
Checking OCI8 and PDO_OCI Installation.....	80
Chapter 8 Installing Zend Core for Oracle.....	83
Installing Zend Core for Oracle.....	83
Installing Zend Core for Oracle on Linux.....	83
Testing the Zend Core for Oracle Installation on Linux.....	89
Installing Zend Core for Oracle on Windows.....	90
Testing the Zend Core for Oracle Installation on Windows.....	97
Configuring Zend Core for Oracle.....	97
Chapter 9 Connecting to Oracle Using OCI8.....	101
Oracle Connection Types.....	101
Standard Connections.....	101
Multiple Unique Connections.....	101
Persistent Connections.....	101
Oracle Database Name Connection Identifiers.....	102
Easy Connect String.....	103
Database Connect Descriptor String.....	104
Database Connect Name.....	104
Common Connection Errors.....	105
Setting Oracle Environment Variables for Apache.....	106
Closing Oracle Connections.....	108
Close Statement Resources Before Closing Connections.....	109
Transactions and Connections.....	110
Session State with Persistent Connections.....	110
Optional Connection Parameters.....	111
Connection Character Set.....	111
Connection Session Mode.....	112
Changing the Database Password.....	114
Changing Passwords On Demand.....	114
Changing Expired Passwords.....	115
Tuning Oracle Connections in PHP.....	117
Use the Best Connection Function.....	117
Pass the Character Set.....	117
Do Not Set the Date Format Unnecessarily.....	117

Managing Persistent Connections.....	119
Maximum Number of Persistent Connections Allowed.....	119
Timeout for Unused Persistent Connections.....	119
Pinging for Closed Persistent Connections.....	119
Apache Configuration Parameters.....	120
Reducing Database Server Memory Used By Persistent Connections.....	120
Oracle Net and PHP.....	121
Connection Rate Limiting.....	121
Setting Connection Timeouts.....	122
Configuring Authentication Methods.....	122
Detecting Dead PHP Apache Sessions.....	123
Other Oracle Net Optimizations.....	123
Tracing Oracle Net.....	123
Connection Management in Scalable Systems.....	124
Chapter 10 Executing SQL Statements With OCI8.....	125
SQL Statement Execution Steps.....	125
Query Example.....	125
Oracle Datatypes.....	127
Fetch Functions.....	127
Fetching as a Numeric Array.....	128
Fetching as an Associative Array.....	129
Fetching as an Object.....	130
Defining Output Variables.....	131
Fetching and Working with Numbers.....	131
Fetching and Working with Dates.....	132
Insert, Update, Delete, Create and Drop.....	134
Transactions.....	134
Autonomous Transactions.....	136
The Transactional Behavior of Connections.....	137
PHP Error Handling.....	138
Handling OCI8 Errors.....	138
Tuning SQL Statements in PHP Applications.....	141
Using Bind Variables.....	141
Tuning the Prefetch Size.....	149
Tuning the Statement Cache Size.....	150
Using the Server and Client Query Result Caches.....	151
Limiting Rows and Creating Paged Datasets.....	153
Auto-Increment Columns.....	155
Getting the Last Insert ID.....	157
Exploring Oracle.....	157
Case Insensitive Queries.....	157

Analytic Functions in SQL.....	158
Chapter 11 Using PL/SQL With OCI8.....	159
PL/SQL Overview.....	159
Blocks, Procedures, Packages and Triggers.....	160
Anonymous Blocks.....	160
Stored or Standalone Procedures and Functions.....	160
Packages.....	161
Triggers.....	162
Creating PL/SQL Stored Procedures in PHP.....	162
End of Line Terminators in PL/SQL with Windows PHP.....	162
Calling PL/SQL Code.....	163
Calling PL/SQL Procedures.....	163
Calling PL/SQL Functions.....	164
Binding Parameters to Procedures and Functions.....	164
Array Binding and PL/SQL Bulk Processing.....	165
PL/SQL Success With Information Warnings.....	167
Using REF CURSORS for Result Sets.....	168
Closing Cursors.....	170
Converting from REF CURSOR to PIPELINED Results.....	172
Oracle Collections in PHP.....	173
Using PL/SQL and SQL Object Types in PHP.....	175
Using OCI8 Collection Functions.....	176
Using a REF CURSOR.....	177
Binding an Array.....	179
Using a PIPELINED Function.....	180
Getting Output with DBMS_OUTPUT.....	181
PL/SQL Function Result Cache.....	183
Using Oracle Locator for Spatial Mapping.....	184
Inserting Locator Data.....	184
Queries Returning Scalar Values.....	184
Selecting Vertices Using SDO_UTIL.GETVERTICES.....	186
Using a Custom Function.....	186
Scheduling Background or Long Running Operations.....	188
Reusing Procedures Written for MOD_PLSQL.....	191
Chapter 12 Using Large Objects in OCI8.....	193
Working with LOBs.....	193
Inserting and Updating LOBs.....	193
Fetching LOBs.....	194
Temporary LOBs.....	195
LOBs and PL/SQL procedures.....	196
Other LOB Methods.....	197

Working with BFILES.....	198
Chapter 13 Using XML with Oracle and PHP.....	203
Fetching Relational Rows as XML.....	203
Fetching Rows as Fully Formed XML.....	204
Using the SimpleXML Extension in PHP.....	205
Fetching XMLType Columns.....	206
Inserting into XMLType Columns.....	207
Fetching an XMLType from a PL/SQL Function.....	209
XQuery XML Query Language.....	210
Accessing Data over HTTP with XML DB.....	212
Chapter 14 PHP Scalability and High Availability.....	213
Database Resident Connection Pooling.....	213
How DRCP Works.....	214
PHP OCI8 Connections and DRCP.....	216
When to use DRCP.....	218
Sharing the Server Pool.....	219
Using DRCP in PHP.....	220
Configuring and Enabling the Pool.....	221
Configuring PHP for DRCP.....	223
Application Deployment for DRCP.....	224
Monitoring DRCP.....	226
V\$PROCESS and V\$SESSION Views.....	227
DBA_CPOOL_INFO View.....	227
V\$CPOOL_STATS View.....	227
V\$CPOOL_CC_STATS View.....	229
High Availability with FAN and RAC.....	229
Configuring FAN Events in the Database.....	230
Configuring PHP for FAN.....	230
Application Deployment for FAN.....	230
RAC Connection Load Balancing with PHP.....	231
Chapter 15 Globalization.....	233
Establishing the Environment Between Oracle and PHP.....	233
Manipulating Strings.....	235
Determining the Locale of the User.....	235
Encoding HTML Pages.....	236
Specifying the Page Encoding for HTML Pages.....	236
Specifying the Encoding in the HTTP Header.....	237
Specifying the Encoding in the HTML Page Header.....	237
Specifying the Page Encoding in PHP.....	237
Organizing the Content of HTML Pages for Translation.....	237
Strings in PHP.....	238

Static Files.....	238
Data from the Database.....	238
Presenting Data Using Conventions Expected by the User.....	238
Oracle Number Formats.....	239
Oracle Date Formats.....	240
Oracle Linguistic Sorts.....	242
Oracle Error Messages.....	243
Chapter 16 Testing PHP and the OCI8 Extension.....	245
Running OCI8 Tests.....	245
Running a Single Test.....	247
Tests that Fail.....	247
Creating OCI8 Tests	248
OCI8 Test Helper Scripts.....	249
Configuring the Database For Testing.....	249
Appendix A Tracing OCI8 Internals.....	253
Enabling OCI8 Debugging output.....	253
Appendix B OCI8 php.ini Parameters.....	255
Appendix C OCI8 Function Names in PHP 4 and PHP 5.....	257
Appendix D The Obsolete Oracle Extension.....	261
Oracle and OCI8 Comparison.....	261
Appendix E Resources.....	267
General Information and Forums.....	267
Oracle Documentation.....	267
Selected PHP and Oracle Books.....	268
Software and Source Code.....	269
PHP Links.....	271
Glossary.....	273

INTRODUCTION

This book is designed to bridge the gap between the many PHP and the many Oracle texts available. It shows how to use the PHP scripting language with the Oracle database, from installation to using them efficiently.

The installation and database discussion in this book highlights the Oracle Database 10g Express Edition, but everything covered in this book also applies to the other editions of the Oracle database, including Oracle Database 11g. The PHP you write for Oracle Database 10g Express Edition can be run, without change, against all editions of the Oracle database as well.

The book contains much unique material on PHP's Oracle OCI8 and PDO_OCI extensions. It also incorporates several updated installation guides previously published on the Oracle Technology Network web site. The chapter on globalization is derived from the *Oracle Database Express Edition 2 Day Plus PHP Developer Guide*. Sue Harper contributed the chapter on SQL Developer. The chapter on PHP Scalability and High Availability is derived from the Oracle whitepaper, *PHP Scalability and High Availability*, April 2008.

We gratefully acknowledge all the Oracle staff that contributed to this book.

Who Should Read This Book?

This book is aimed at PHP programmers who are developing applications for an Oracle database. It bridges the gap between the many PHP and the many Oracle books available. It shows how to use the PHP scripting language with the Oracle database, from installation to using them together efficiently.

You may already be using another database and have a requirement or a preference to move to Oracle. You may be starting out with PHP database development. You may be unsure how to install PHP and Oracle. You may be unclear about best practices. This book aims to remove any confusion.

This book is not a complete PHP syntax or Oracle SQL guide. It does not describe overall application architecture. It is assumed that you already have basic PHP and SQL knowledge and want best practices in using PHP against an Oracle database.

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

Oracle's own extensive set of documentation is freely available online. For newcomers we suggest reading the *Oracle Database Express Edition 2 Day Plus PHP Developer Guide* which walks through building a PHP application against an Oracle database. Comprehensive PHP documentation and resources are also online.

URLs to the most useful online resources and books are listed in the *Resources* appendix.

Introduction to Oracle

The Oracle Database is well known for its scalability, reliability and features. It is the leading database and is available on many platforms.

Introduction

There are some subtle differences between the terminology used when describing an Oracle database and a database from other software vendors. The following overview of the main Oracle terms might help you to understand the Oracle terminology. Check the Glossary for more descriptions.

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 creates. 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 a single Oracle database contains 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, use the `CREATE USER` command to create a new schema in the database. In Oracle Database 10g Express Edition (known as "Oracle Database XE") there is a wizard to create new users in the Oracle Application Express management console.

Introduction to PHP

PHP is a hugely popular, interpreted scripting language commonly used for web applications. PHP is open source and free, and has a BSD-style license, making it corporation-friendly. PHP is perfect for rapidly developing applications both big and small, and is great for creating Web 2.0 applications. It powers over twenty million 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 comes with many extensions offering all kinds of functionality such as database access. PHP 5 introduced strong object orientated capabilities.

PHP is typically installed as an Apache module, or run by the web server using FastCGI. It includes the PHP OCI8 extension and is linked with the Oracle Client Libraries, enabling Oracle Database access. When a user enters the URL of a PHP script `hello.php` (see step 1 in Figure 1) in their browser, Apache invokes PHP to process the file. The PHP code is executed (2), with calls to the database (3). Finally, the HTML output is returned to the user's browser (4), which formats and displays the page.

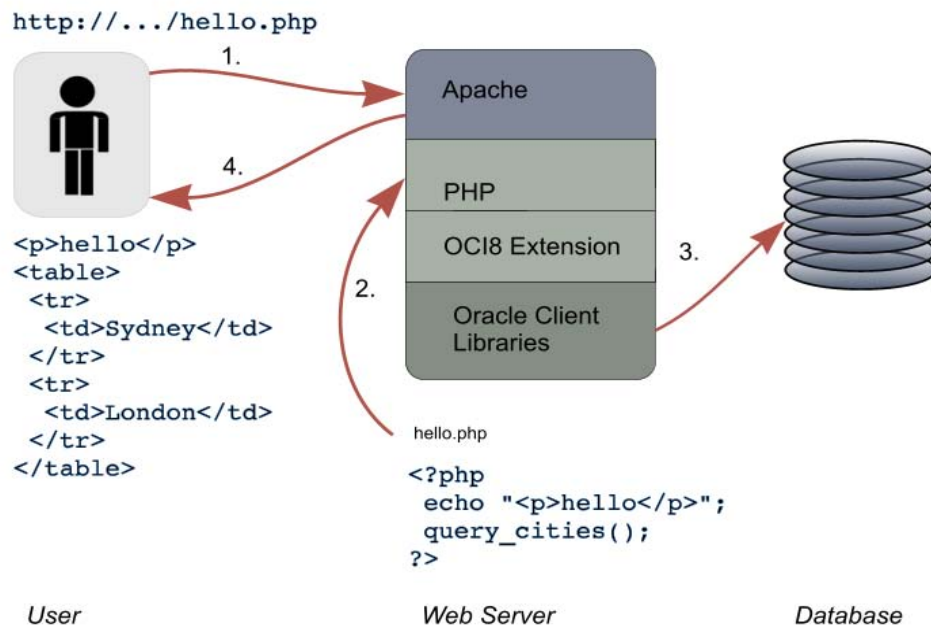


Figure 1: The four stages of processing a PHP script.

The PHP command line interface (CLI) can also be used to run PHP scripts from an operating system shell window.

Introduction

GETTING STARTED WITH PHP

This Chapter gives you a very brief overview of the PHP language. Basic PHP syntax is simple to learn. It has familiar loops, tests and assignment constructs.

Creating and Editing PHP Scripts

There are a number of specialized PHP editors available, including Oracle's JDeveloper which can be configured with a PHP extension. Many developers still prefer text editors, or editors with modes that highlight code syntax and aid development. This manual does not assume any particular editor or debugger is being used.

PHP scripts often have the file extension *.php*, but sometimes *.phtml* or *.inc* are also used. The web server can be configured to recognize the extension(s) that you choose.

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>
```

Getting Started With PHP

```
</tr>
</table>
```

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/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 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.

A function may be defined:

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

Functions may have variable numbers of arguments. This function could be called using:

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

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

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:

```
// 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 compress white space including new-lines, script output can differ between command-line and browser invocation of the same script.

Many aspects of PHP are controlled by settings in the *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();
?>
```

Values can be changed by editing *php.ini* or using the Zend Core for Oracle console, and restarting the web server. Some values can also be changed within scripts by using the `ini_set()` function.

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

Running PHP Scripts in a Browser

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

`http://localhost/myphpinfo.php`

When a web server is configured to run PHP files through the PHP interpreter, requesting the script in a browser will cause the PHP code to be executed and all its output to be streamed to the browser.

Running Scripts with Command Line PHP

If your PHP code is in a file, 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` options gives the help text:

```
$ php -h
```

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 a specialized PHP editor, debugging will be an old-fashioned matter of using `echo` to print variables and check code flow.

The `var_dump()` function is 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 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.

PHP ORACLE EXTENSIONS

PHP has several extensions that let applications use Oracle. There are also database abstraction libraries written in PHP which are popular. Although this book concentrates on the OCI8 extension for PHP, it is worth knowing the alternatives.

Database access in each extension and abstraction library is fundamentally similar. The differences are in their 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 want database independence, consider using the PHP Data Object (PDO) extension or the ADOdb abstraction library.

The PHP world can move at a fast pace, so examine your requirements and the latest information available before starting a project.

PHP Oracle Extensions

The PHP Oracle extensions are written in C and linked into the PHP binary. The extensions are:

- Oracle
- OCI8
- PDO

You can also use the ODBC extension.

Oracle Extension

The extension called "Oracle" was included in PHP 3, 4 and 5.0. It had limited functionality, is no longer in PHP, and is not maintained. The extension accessed the database using Oracle's obsolete "OCI7" C language API. New development using this extension is not recommended.

OCI8 Extension

OCI8 is the recommended extension to use. It is included in PHP 3, 4, and 5. It is also in PHP 6, which is in the early stage of development. 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 various versions of PHP. It is recommended to upgrade the default OCI8 code in PHP 4 – PHP 5.2 to the latest version of the extension.

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

Script 1: intro.php

```
<?php
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');
```

PHP Oracle Extensions

```
$s = oci_parse($c, 'select city from locations');
oci_execute($s);
while ($res = oci_fetch_array($s, OCI_ASSOC)) {
    echo $res['CITY'] . "<br>";
}
?>
```

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

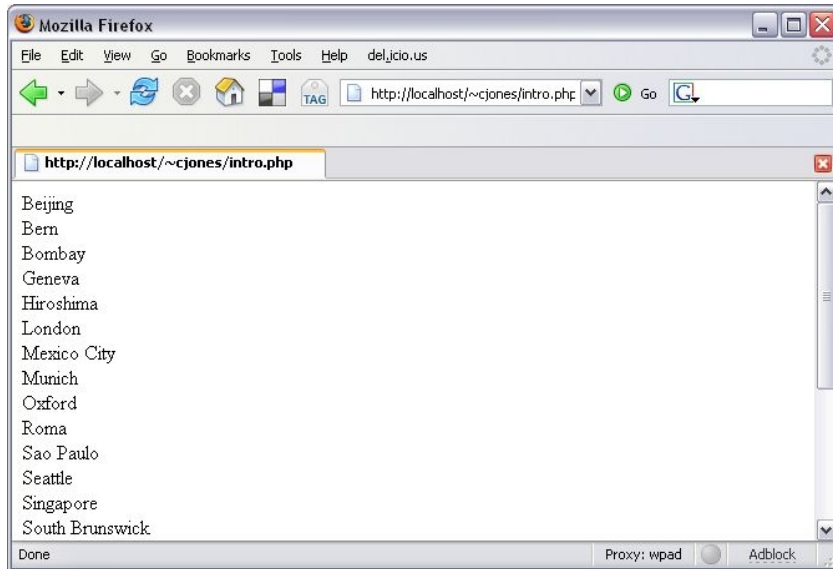


Figure 3: PHP Output in a web browser.

In PHP 5, some extension function names were standardized. PHP 4 functions like `OCILogin()` became `oci_connect()`, `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. All unqualified references to OCI8 in this book refer to the PHP extension.

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.

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', 'hrpwd');
$s = $dbh->prepare("select city from locations");
```

```
$s->execute();
while ($r = $s->fetch(PDO::FETCH_ASSOC)) {
    echo $r['CITY'] . "<br>";
}
?>
```

The output is the same as the OCI8 example in Figure 3.

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

PHP Database Abstraction Libraries

Like PDO, the abstraction libraries allow simple PHP applications to work with different brands of database.

There are three main database abstraction libraries for PHP. They are written in PHP and, when configured for Oracle, they use functionality provided by the OCI8 extension. The abstraction libraries are:

- ADOdb
- PEAR DB
- PEAR MDB2

Other abstractions such as Creole have dedicated fan-base, but the support for Oracle features varies.

You can freely download and use the PHP code for these libraries.

ADODB

The popular 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", "hrpwd");
$s = $db->Execute("select city from locations");
while ($r = $s->FetchRow()) {
    echo $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 widely 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:hrpwd@//localhost/XE');
$res = $mdb2->query("select city from locations");
while ($row = $res->fetchRow(MDB2_FETCHMODE_ASSOC)) {
    echo $row['city'] . "<br>";
}
?>
```

Getting the OCI8 Extension

The OCI8 extension is included in various PHP bundles. There are three main distribution channels: the PHP releases, Zend Core for Oracle, and the PHP Extension Community Library (PECL) site which contains PHP extensions as individual downloads.

The OCI8 extension is available in several forms because of the differing needs of the community. Many PHP users install the full PHP source and do their own custom configuration. If they need a specific bug fix they use PECL or PHP's latest development source code to get it. Windows users commonly install PHP's pre-built Windows binaries. At time of writing, the site <http://pecl4win.php.net/> that was useful for obtaining Windows PHP fixes is no longer being maintained. A new site at <http://windows.php.net/> is under construction.

If you do not want to compile PHP, or this is your first time with PHP and Oracle, or you want a supported stack, install Zend Core for Oracle.

Table 1 shows where OCI8 can be downloaded.

Table 1: OCI8 Availability.

Bundle Containing OCI8	Location and Current Release
PHP Source Code	http://www.php.net/downloads.php <i>php-5.2.7.tar.bz2</i> Compiles and runs on many platforms

Bundle Containing OCI8	Location and Current Release
PHP Windows Binaries	http://www.php.net/downloads.php <i>php-5.2.7-Win32.zip</i>
PECL Source Code	http://pecl.php.net/package/oci8 <i>oci8-1.3.4.tgz</i> Used to add or upgrade OCI8 for an existing PHP installation
Zend Core for Oracle 2.5	http://www.oracle.com/technology/tech/php/zendcore/ <i>ZendCoreForOracle-v2.5.0-Linux-x86.tar.gz</i> <i>ZendCoreforOracle-v.2.5.0-Windows-x86.exe</i> Other platforms are also available

OCI8 and Oracle Installation Options

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

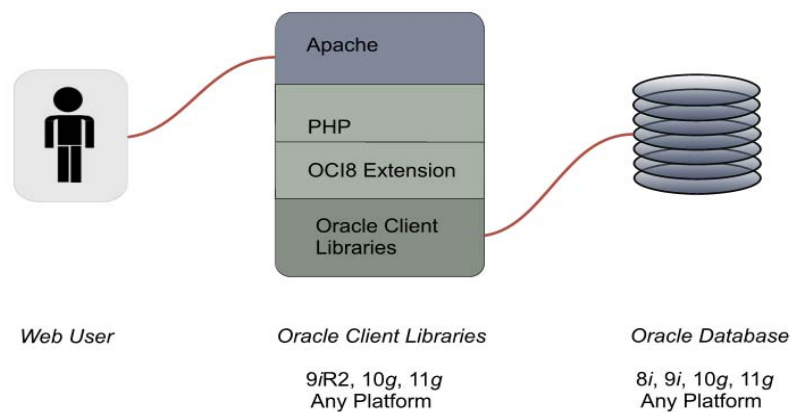


Figure 4: PHP links with Oracle client libraries.

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.

PHP Oracle Extensions

Full OCI8 functionality may not be 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.

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 5.2.7	OCI8 1.2.5	8i, 9i, 10g, 11g
PHP Release Windows Binaries	Current release is 5.2.7	OCI8 1.2.5	10g, 11g
PECL OCI8 Source Code	Builds with PHP 4.3.9 onwards	Latest release is OCI8 1.3.4	9iR2, 10g, 11g
Zend Core for Oracle 2.5	Includes PHP 5.2.5	OCI8 1.2.3	<i>Inbuilt Oracle Database 10g client</i>

If OCI8 is being used with PHP 4, 5.0, 5.1 or 5.2, consider replacing the default OCI8 code with the latest version from PECL to get improved stability, behavior and performance optimizations. This is important for PHP 4 and 5.0 because their versions of OCI8 are notoriously unstable. Instructions for updating OCI8 are shown later in this book.

Getting the PDO Extension

The PDO_OCI driver for PDO is included with PHP source code and, like OCI8, is also available on PECL.

The PHP community has let the PDO project languish and Oracle recommends using OCI8 instead whenever possible because of its better feature set, performance and reliability. Only a few minor changes have been made to PDO_OCI in PHP releases since its introduction. The version of PDO_OCI on PECL has not been updated with these fixes and is still at version 1.0.

PDO_OCI is independent of OCI8 and has no code in common with it. The extensions can be enabled separately or at the same time.

Table 3 shows the compatibility of the PDO_OCI driver with PHP and Oracle.

Table 3: PDO_OCI Availability and Compatibility.

Bundle Containing PDO_OCI	Location and Current Release	Oracle Client Libraries Usable with PDO_OCI
PHP Source Code	http://www.php.net/downloads.php <i>php-5.2.7.tar.bz2</i> Compiles and runs on many platforms	8i, 9i, 10g, 11g
PHP Windows Binaries	http://www.php.net/downloads.php <i>php-5.2.7-Win32.zip</i>	8i, 9i, 10g, 11g

Bundle Containing PDO_OCI	Location and Current Release	Oracle Client Libraries Usable with PDO_OCI
PECL Source Code	http://pecl.php.net/package/PDO_OCI <i>PDO_OCI-1.0.tgz</i>	8i, 9i, 10g (no 11g support)
Zend Core for Oracle 2.5	http://www.oracle.com/technology/tech/php/zendcore/ <i>ZendCoreForOracle-v2.5.0-Linux-x86.tar.gz</i> <i>ZendCoreforOracle-v.2.5.0-Windows-x86.exe</i> Other platforms are also available	Inbuilt Oracle Database 10g client

Zend Core for Oracle

Zend Core for Oracle (ZCO) is a pre-built release of PHP from Zend that comes enabled with the OCI8 and PDO_OCI extensions. It is available for several platforms, has a simple installer, and includes a convenient browser-based management console for configuration and getting updates. It comes with Oracle Instant Client 10g and includes an optional Apache web server. ZCO can connect to local and remote Oracle databases.

Although ZCO 2.5 includes PHP 5.2.5, Zend decided to use a tested OCI8 extension from an older PHP release, so not all recent bug fixes are included.

ZCO is free to download and use. A support package is available from Zend.

The PHP Release Cycle

PHP's source code is under continual development in a source code control system viewable at <http://cvs.php.net/>. (Migration from CVS to SVN is planned). This is the only place bug fixes are merged. The code is open source and anyone can read the code in CVS or seek approval to contribute.

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

- Two-hourly snap-shots are created containing a complete set of all PHP's source in CVS 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://snaps.php.net/>.
- After several months of development the PHP release manager will release a new stable version of PHP. It uses the most current CVS code at the time of release.
- PECL OCI8 source code snapshots are taken from CVS at infrequent intervals. Recently, snapshots have been made concurrently at the time of a PHP release (when OCI8 has changed).
- Zend Core for Oracle also takes snapshots of PHP from CVS.
- Various operating systems bundle the version of PHP current at the time the OS is released and provide critical patch updates.

PHP Oracle Extensions

The schedules of PHP releases, the PECL source snapshots, and Zend Core for Oracle are not fully synchronized.

As a result of a recent PHP version numbering overhaul, the OCI8 extension included in the PHP source code is now labeled with the same version number as the equivalent PECL release.

Table 4 shows the major features in each revision of PECL OCI8.

Table 4: Major Revisions of OCI8.

PECL 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. Available in PHP 5.1 – 5.2.
OCI8 1.3	Refactored connection management gives better handling of restarted databases and adds support for Oracle's external authentication. Also some of Oracle recent scalability and high availability features can be now be used. These features are discussed in a later chapter. OCI8 1.3 will be included in PHP 5.3.

INSTALLING ORACLE DATABASE 10G EXPRESS EDITION

This Chapter contains an overview of, and installation instructions for, Oracle Database 10g Express Edition (Oracle Database XE). The installation instructions are given for Linux, Windows, Debian, Ubuntu and Kubuntu.

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 implemented in each edition. Enterprise Edition has all the bells and whistles, whereas the Express Edition has a limited feature set, but still has all the reliability and performance of the 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/database/product_editions.html.

This book discusses working with Oracle Database XE. This is the free edition. Free to download. Free to develop against. Free to distribute with your applications. Yes, that is free, free, free!

Oracle Database XE

Oracle Database XE is available on 32-bit Windows and Linux platforms. Oracle Database XE is a good choice for development of PHP applications that require a free, small footprint database.

Oracle Database XE is available on the Oracle Technology Network (<http://otn.oracle.com/x>) for the following operating systems:

- Windows 2000 Service Pack 4 or later
- Windows Server 2003

Installing Oracle Database 10g Express Edition

- Windows XP Professional Service Pack 1 or later
- Oracle Enterprise Linux 4 and 5
- Red Hat Enterprise Linux RHEL 3, 4, and 5
- Suse SLES-9
- Fedora Core 4
- Red Flag DC Server 5.0/MIRACLE LINUX V4.0/Haansoft Linux 2006 Server (Asianux 2.0 Inside)
- Debian 3.1

The following libraries are required for the Linux-based operating systems:

- *glibc* release 2.3.2
- *libaio* release 0.3.96

There are some limitations with Oracle Database XE:

- 4GB of data
- Single database instance
- Single CPU used, even if multiple CPUs exist
- 1GB RAM used, even if more RAM is installed

Oracle Database XE has a browser-based management interface, Oracle Application Express. Support for Oracle Database XE is through an Oracle Technology Network (<http://otn.oracle.com/>) discussion forum, which is populated by peers and product experts. You cannot buy support from Oracle for Oracle Database 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, and use these for application development and testing, but when you go production, you will need to pay Oracle for the license costs.

Installing Oracle Database XE on Linux

If you do not have a version of *libaio* over release 0.3.96, you need to install this library before you can install Oracle Database XE. To install Oracle Database XE:

1. Download the Oracle Database XE from <http://otn.oracle.com/xe>.
2. Log in or *su* as *root*:

```
# su
Password:
```

3. Install the RPM:

```
# rpm -ivh oracle-xe-univ-10.2.0.1-1.0.i386.rpm
```

Oracle Database XE installs.

4. Configure the database

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

5. Accept the default ports of **8080** for Application Express, and **1521** for the Database Listener.
6. Enter and confirm the password for the default users.
7. 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 can install Oracle Database XE with:

```
# up2date oracle-xe
```

After this download completes, follow the configuration step 4 onwards.

Installing Oracle Database XE on Debian, Ubuntu, and Kubuntu

There is an Advanced Package Tool (*apt-get*) repository available on the Oracle Open Source web site for Oracle Database XE. To include this repository, add the following to the file */etc/apt/sources.list*:

```
deb http://oss.oracle.com/debian unstable main non-free
```

libaio and *bc* are included in the repository, and will be installed from the repository if you do not already have them installed.

If you download Oracle Database XE from the Oracle Technology Network (<http://otn.oracle.com/xe>), you need to make sure that you have already installed the *libaio* and *bc* packages. If you are using Ubuntu or Kubuntu, the *bc* package is installed by default on the desktop version, but not on the server version.

To install Oracle Database XE on Debian, Ubuntu and Kubuntu, follow these steps:

1. Log in or su as *root*

```
# su
Password:
```

2. Install Oracle Database XE

```
# apt-get update
# apt-get install oracle-xe
```

If you have not added the *apt-get* repository, you can download Oracle Database XE from <http://otn.oracle.com/xe>, and run the following command to begin the install:

```
# dpkg -i downloads/oracle-xe-universal_10.2.0.1-1.0_i386.deb
```

Oracle Database XE installs.

3. Configure the database

Installing Oracle Database 10g Express Edition

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

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

Installing Oracle Database XE on Windows

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

1. Log on to Windows as a user with Administrative privileges.
2. If an `ORACLE_HOME` environment variable has been set, delete it using the **Control Panel > System** dialog.
3. Download the Oracle Database XE from <http://otn.oracle.com/xe>.
4. Double click on the *OracleXEUniv.exe* file.
5. In the Oracle Database XE - Install Wizard welcome window, click **Next**.

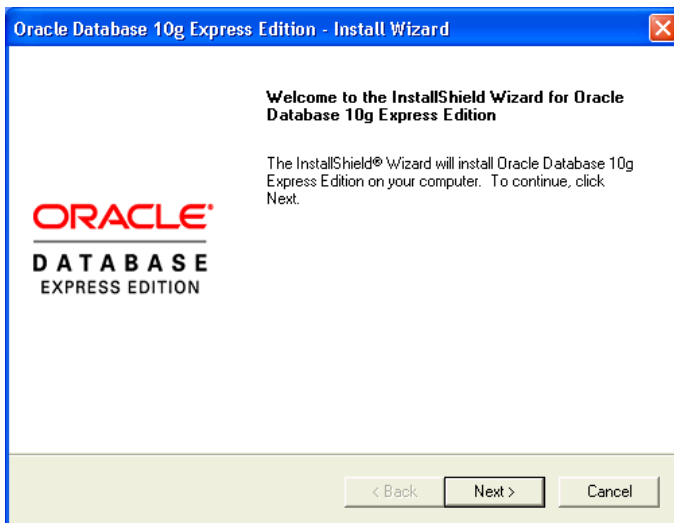


Figure 5: Oracle Database XE install welcome dialog.

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

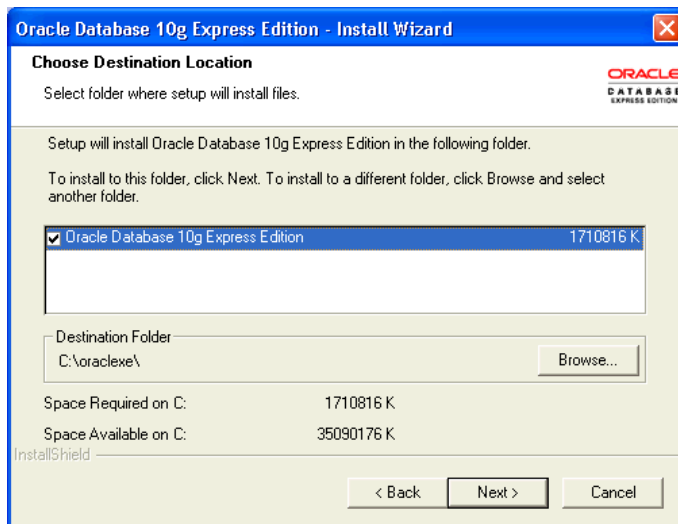


Figure 6: Oracle Database XE install location dialog.

8. Oracle Database XE requires a number of ports and selects a number of default ports. If these ports are already being used, you are prompted to enter another port number.
9. In the Specify Database Passwords window, enter and confirm the password to use for the *sys* and *system* database accounts. Click **Next**.

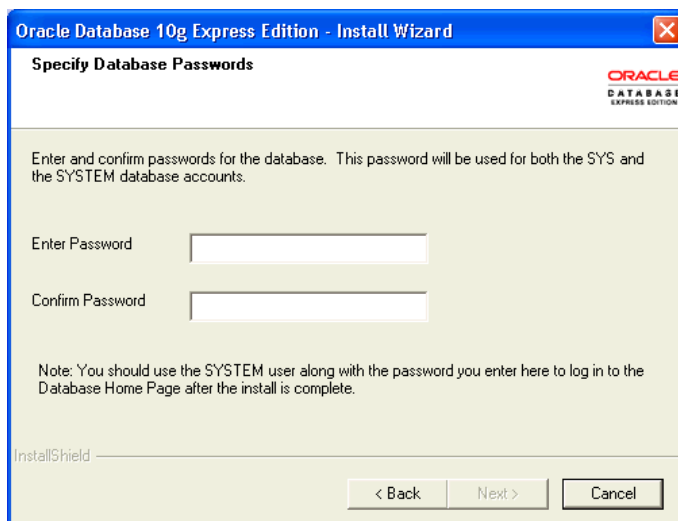


Figure 7: Oracle Database XE database password dialog.

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

Installing Oracle Database 10g Express Edition

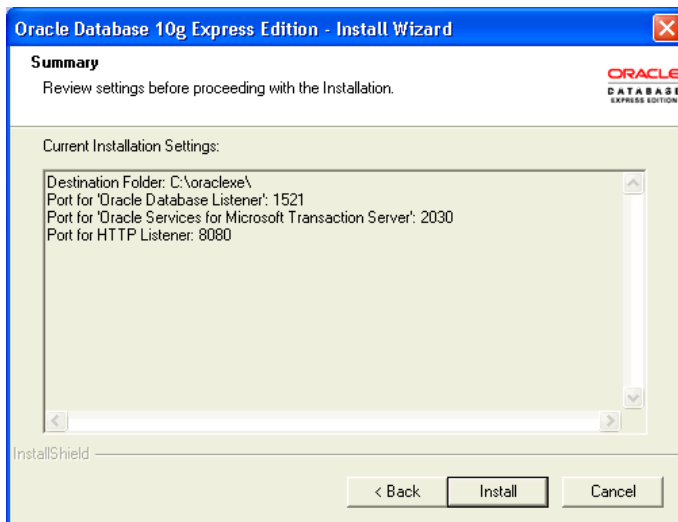


Figure 8: Oracle Database XE install summary dialog.

11. In the InstallShield Wizard Complete window, click **Launch the Database homepage** to display the Database Home Page. Click **Finish**.

Testing the Oracle Database XE Installation

To test the installation of Oracle Database XE:

1. If you do not already have the Database homepage displayed in your web browser, open a web browser and enter:
`http://127.0.0.1:8080/apex`
2. The Database homepage is displayed.

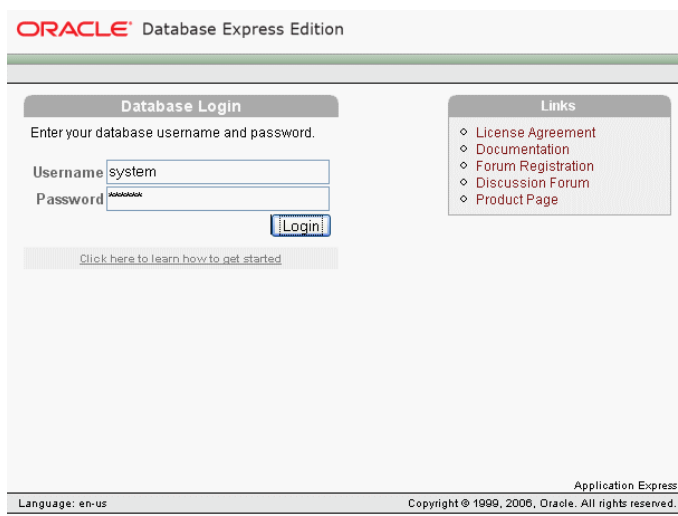


Figure 9: Oracle Database XE home page login screen.

- Log in as user *system* with the password you entered during the installation. You should now be logged into the Oracle Database homepage.

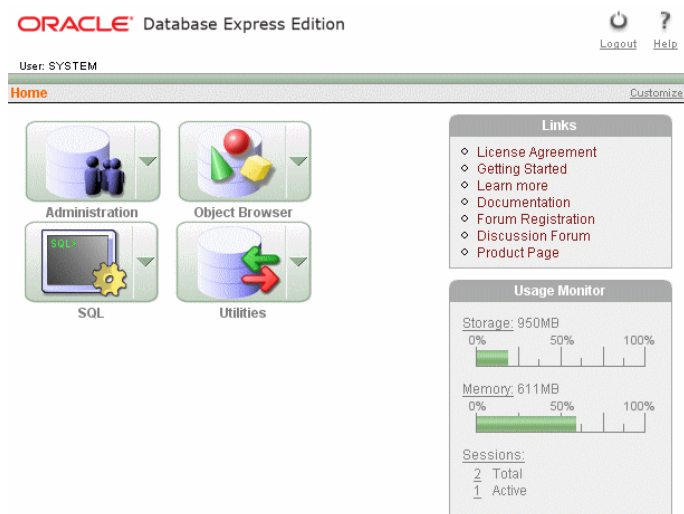


Figure 10: Oracle Database XE home page.

Configuring Oracle Database XE

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

Setting the Oracle Database XE Environment Variables on Linux

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:

```
/usr/lib/oracle/xe/app/oracle/product/10.2.0/server/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 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:

```
. /usr/lib/oracle/xe/app/oracle/product/10.2.0/server/bin/oracle_env.sh
```

(Note the space after the period). To add the script to your login profile for C and tcsh shells, add the following lines to your *.login* or *.cshrc* file:

```
source /usr/lib/oracle/xe/app/oracle/product/10.2.0/server/bin/oracle_env.csh
```

Enabling Database Startup and Shutdown from Menus on Linux

You may not be able to start and stop the database using the menu on Linux platforms. This is because your user is not a member of the operating system *dba* group. To enable this functionality, add the user name to the *dba* group using the System Settings.

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 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.

To start the database, you must log in as a user who is a member of the operating system *dba* user group. This applies to all the methods of starting and stopping the database.

Starting and Stopping the Listener and Database on Linux

To start up the listener and database on Linux platforms using the desktop, do one of the following:

- On Linux with Gnome: Select **Applications > Oracle Database 10g Express Edition > Start Database**.
- On Linux with KDE: Select **K Menu > Oracle Database 10g Express Edition > Start Database**.

To shut down the database on Linux platforms using the desktop, do one of the following:

- On Linux with Gnome: Select **Applications > Oracle Database 10g Express Edition > Stop Database**.
- On Linux with KDE: Select **K Menu > Oracle Database 10g Express Edition > Stop Database**.

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

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

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

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

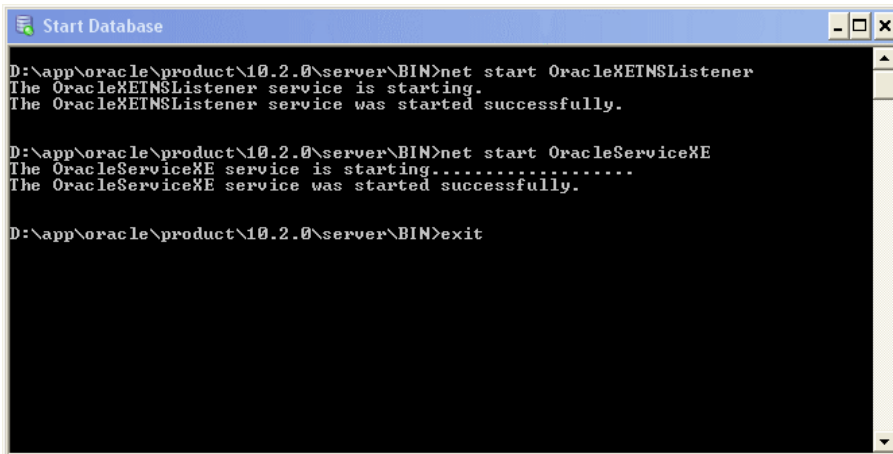
You can also use the Services dialog from the Desktop to start and stop the listener and database.

To start the listener and database from the Desktop Services dialog, select **Applications > System Settings > Server Settings > Services**. Select **oracle-xe** from the list of services and select **Start**.

To stop the listener and database from the Desktop Services dialog, select **Applications > System Settings > Server Settings > Services**. Select **oracle-xe** from the list of services and select **Stop**.

Starting and Stopping the Listener and Database on Windows

To start the listener and database on Windows platforms, select **Start > Oracle Database 10g Express Edition > Start Database**. A Window is displayed showing the status of the listener and database startup process.



```
D:\app\oracle\product\10.2.0\server\BIN>net start OracleXETNSListener
The OracleXETNSListener service is starting.
The OracleXETNSListener service was started successfully.

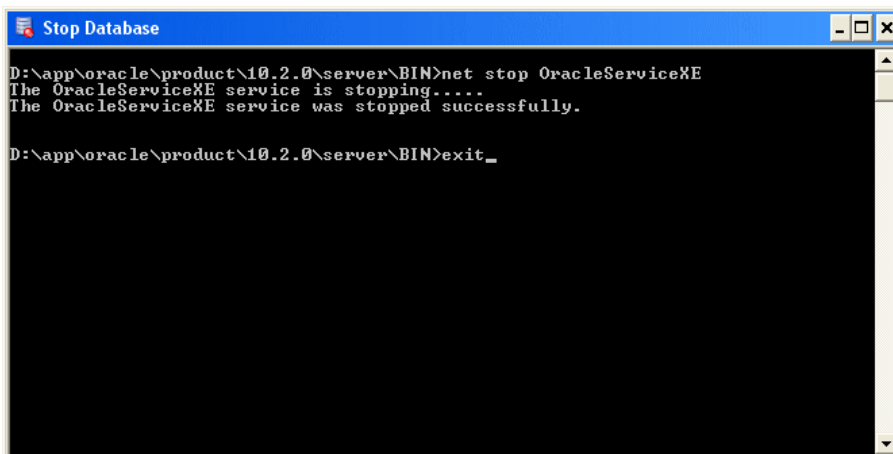
D:\app\oracle\product\10.2.0\server\BIN>net start OracleServiceXE
The OracleServiceXE service is starting.....
The OracleServiceXE service was started successfully.

D:\app\oracle\product\10.2.0\server\BIN>exit
```

Figure 11: 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 **Start > Oracle Database 10g Express Edition > Stop Database**. A Window is displayed showing the status of the listener and database shutdown process.



```
D:\app\oracle\product\10.2.0\server\BIN>net stop OracleServiceXE
The OracleServiceXE service is stopping.....
The OracleServiceXE service was stopped successfully.

D:\app\oracle\product\10.2.0\server\BIN>exit_
```

Figure 12: 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.

To start the listener on Windows platforms, open the Services dialog using **Start > Settings > Control Panel > Administrative Tools > Services**, and select the **OracleXETNSListener** service. Right click on the **Listener** service, and select **Start**.

Installing Oracle Database 10g Express Edition

To stop the listener on Windows platforms, open the Services dialog using **Start > Settings > Control Panel > Administrative Tools > Services**, and select the **OracleXETNSListener** service. Right click on the **Listener** service, and select **Stop**.

You can also start and stop the database separately on Windows platforms using the Services dialog.

To start the database using the Services dialog on Windows platforms, open the Services dialog using **Start > Settings > Control Panel > Administrative Tools > Services**, and select the **OracleServiceXE** service. Right click on the database service, and select **Start**.

To stop the database using the Services dialog on Windows platforms, open the Services dialog using **Start > Settings > Control Panel > Administrative Tools > Services**, and select the **OracleServiceXE** service. Right click on the database service, and select **Stop**.

Starting and Stopping the Listener and Database Using SQL*Plus

You can also use the command line shell and SQL*Plus command line to start and stop the database. Make sure that you are logged in as a privileged user and have your operating system environment set up correctly as discussed in an earlier section of this chapter.

On Windows, to control the listener, use the Services dialog as discussed above.

To start up the listener on Linux, open a terminal window and run the following command:

```
# lsnrctl start
```

Oracle Net starts the listener and it is ready to take database requests. If you want to shut down the listener manually, you use the similar command from the operating system command prompt:

```
# lsnrctl stop
```

After starting the listener, you also need to start the database using SQL*Plus. For this, you must log in as a database user with the *sysdba* role. This is the *sys* user in default installations, or you can use operating system authentication if you are on the local machine in the operating system *dba* group. To start up a database using SQL*Plus, enter the following at the command line prompt:

```
# sqlplus /nolog
```

The SQL*Plus command line starts. You can also start SQL*Plus from the **Applications > Oracle Database 10g Express Edition > Run SQL Command Line** on Linux, or **Start > Programs > Oracle Database 10g 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 it this happens, connections to the database will fail.

To shut down the database, you need to log in as *sysdba*, and issue the **SHUTDOWN IMMEDIATE** command. Log into SQL*Plus as before and issue the following command:

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

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

Enabling Remote Client Connection

The Oracle Database XE home page is only available from the local machine, not remotely. If you want to enable access from a remote client, you should be aware that HTTPS cannot be used (only HTTP), so your login credentials are sent in clear text, and are not encrypted, so if you don't need to set this up, it is more secure to leave it as the default setup.

To enable connection to the Oracle Database XE home page from remote machines, follow these steps:

1. Open a web browser and load the Oracle Database XE home page:
`http://127.0.0.1:8080/apex`
2. Log in as the *system* user.
3. Select **Administration** from the home page.
4. Select **Manage HTTP Access** from the **Tasks** option.
5. Check the **Available from local server and remote clients** radio button. Click **Apply Changes**.

You can also use SQL*Plus command line to enable access from remote clients. To use SQL*Plus command line to change this setting, log into SQL*Plus as *system*, and run the following command:

```
SQL> EXEC DBMS_XDB.SETLISTENERLOCALACCESS (FALSE) ;
```


USING ORACLE DATABASE

This Chapter contains an overview of the Oracle Application Express, SQL*Plus and Oracle SQL Developer applications that you can use to perform database development and administration.

Oracle Application Express

Oracle Application Express is a browser-based application builder for the Oracle database. It is installed with Oracle Database XE and is also available for download from Oracle Technology Network (<http://otn.oracle.com>) as a standalone product for other versions and editions of the database. It also contains a database development tool, not covered in this book. The release of Oracle Application Express installed with Oracle Database XE has an additional module for performing database administration, monitoring and maintenance.

Logging In To Oracle Application Express

To start and log in to Oracle Application Express:

1. Open a web browser and enter the URL:
`http://127.0.0.1:8080/apex`
2. The Oracle Application Express login screen is displayed.

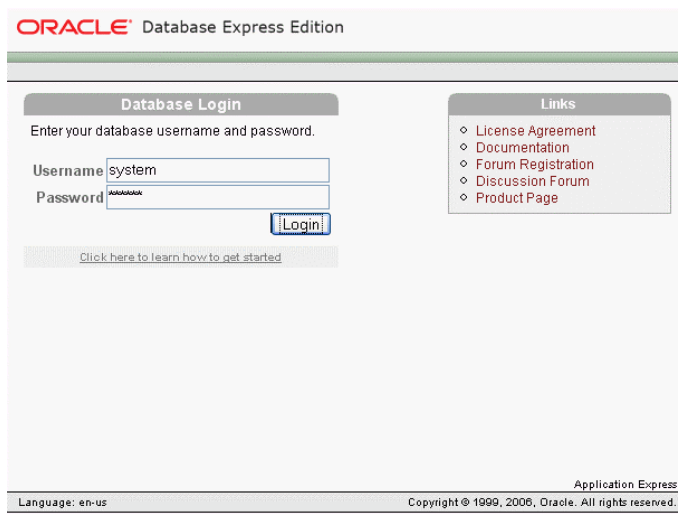


Figure 13: Oracle Application Express login screen.

3. Log in as *system*. The password you enter here is the password you entered when you installed Oracle. There is no default *system* password set by the Oracle Installer.

Using Oracle Database

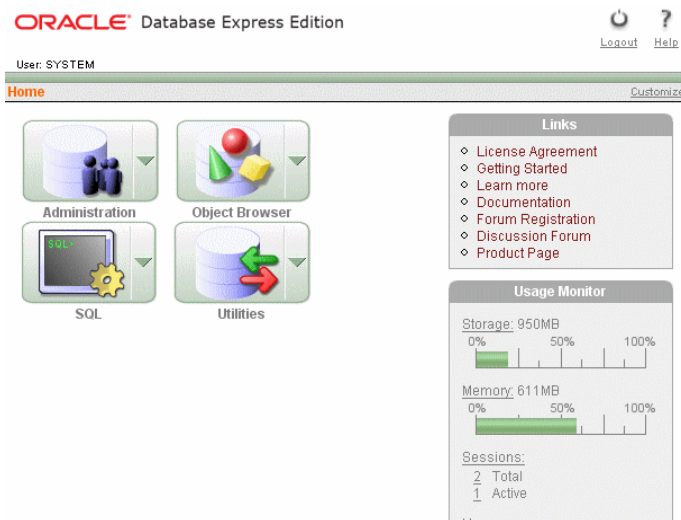


Figure 14: Oracle Application Express interface.

Unlocking the HR User

Many of the examples in this book, and other Oracle books use the *hr* user, which is automatically installed and configured during an Oracle database installation. You may want to unlock this user now so that many of the examples in this book work correctly.

To unlock the *hr* user:

1. Log in to Oracle Application Express as shown in the previous section.
2. Select **Administration > Database Users > Manage Users**.

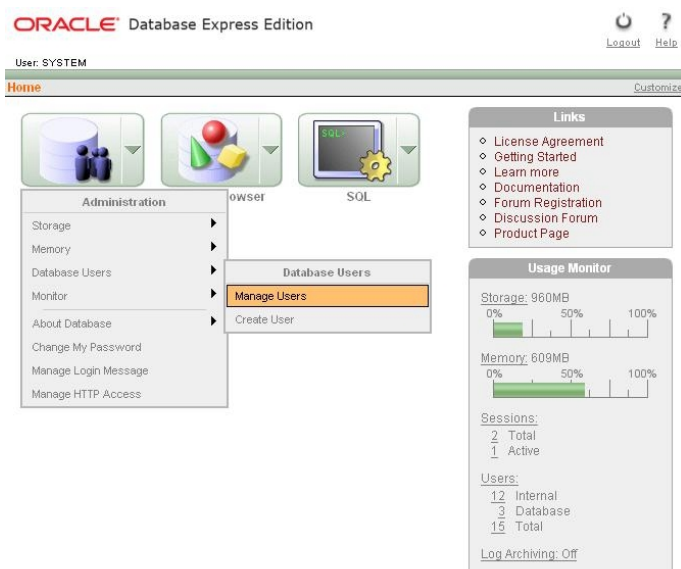
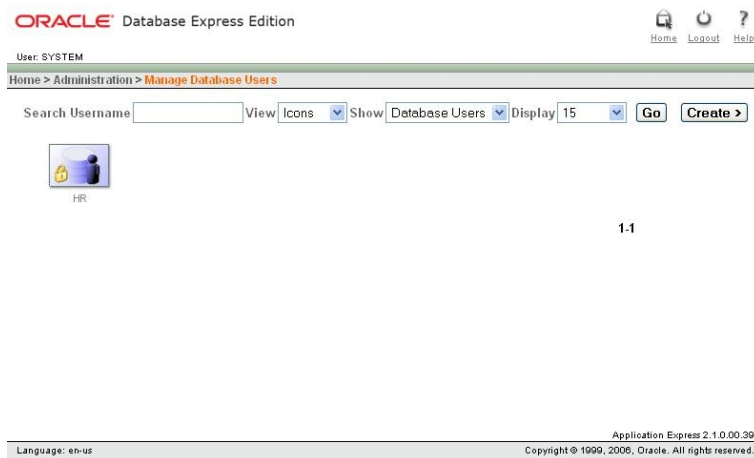


Figure 15: Manage Database Users menu.

3. Select the **HR** icon.



1-1

Figure 16: Manage Database Users screen.

4. To unlock the *hr* user, enter a password in the **Password** and **Confirm Password** fields. The examples in later chapters use the password *hrpwd*. Change Account Status to **Unlocked**, and click the **Alter User** button.

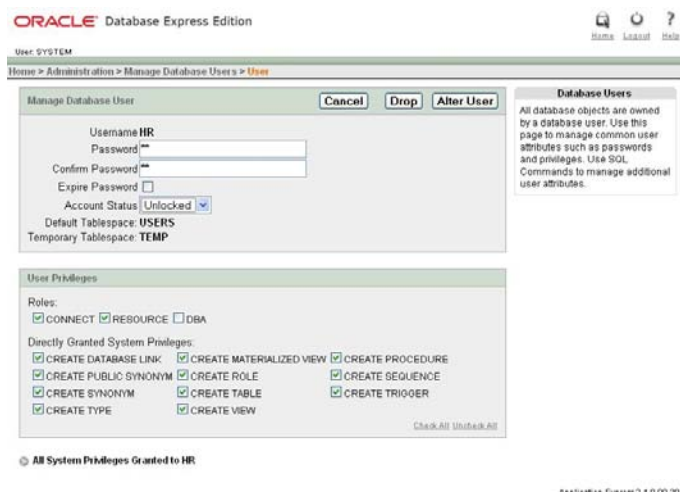


Figure 17: User screen.

The *hr* account is now unlocked.

Creating Database Objects

The Oracle Application Express Object Browser can be used to create or edit the following objects:

- Table
- View
- Index

Using Oracle Database

- Sequence
- Collection Type
- Package
- Procedure
- Function
- Trigger
- Database Link
- Materialized View
- Synonym

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 can all be performed through the Object Browser.

To create a new table:

1. On the Database home page, click the **Object Browser** icon. The Object Browser page is displayed.

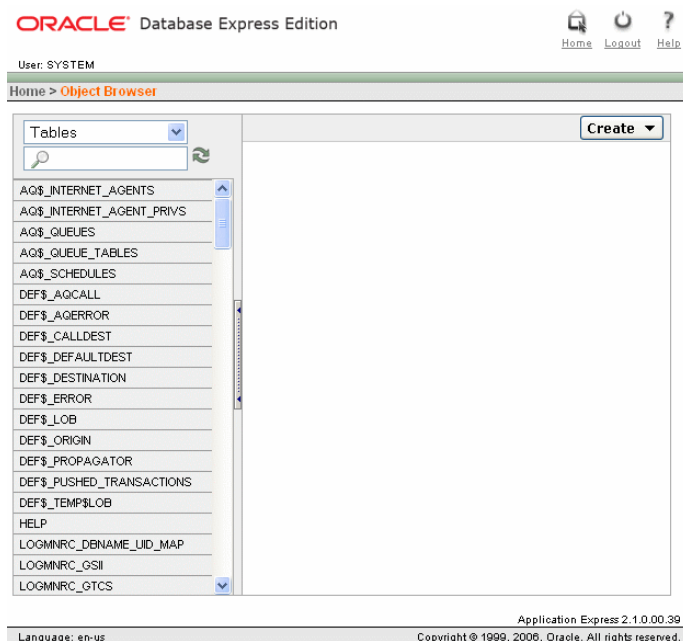


Figure 18: Oracle Application Express create object screen.

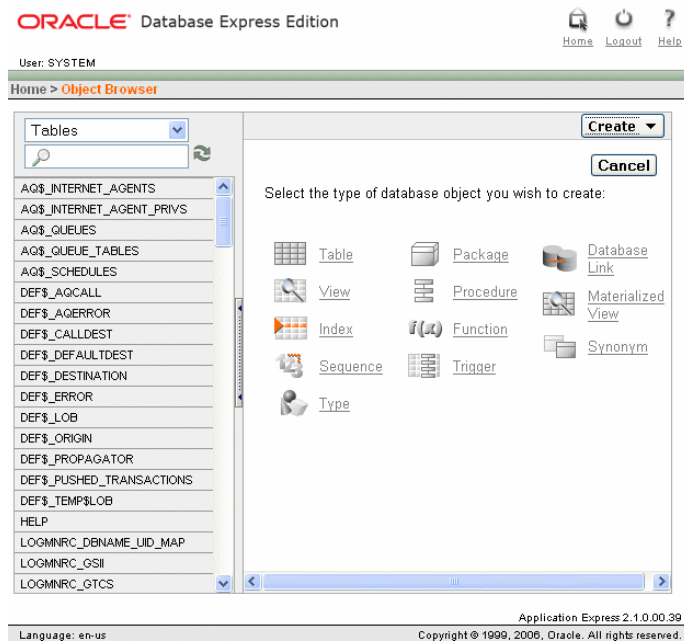
2. Click **Create**.

Figure 19: Oracle Application Express create table object screen.

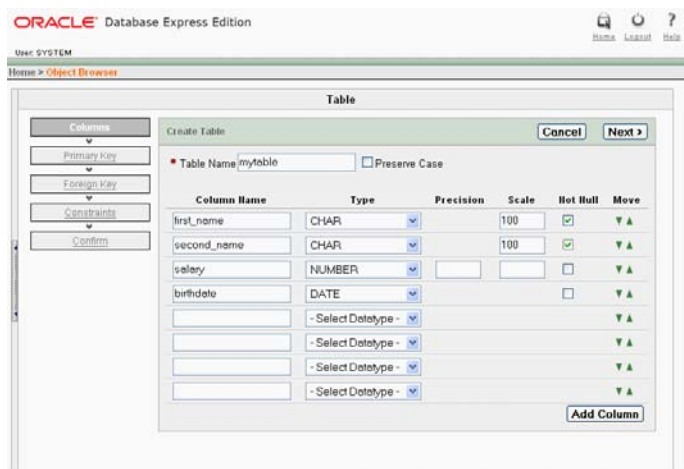
3. From the list of object types, select **Table**.

Figure 20: Oracle Application Express table definition screen.

4. Enter a table name in the Table Name field, and details for each column. Click **Next**.
5. Define the primary key for this table (optional).
6. Add foreign keys (optional).

Using Oracle Database

7. Add a constraint (optional).
8. Click **Finish**. To view the SQL used to create the table, click **SQL Syntax**.

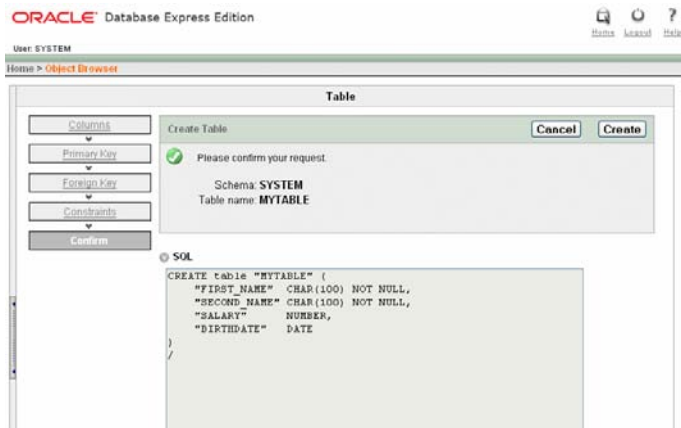


Figure 21: Oracle Application Express confirm create table screen.

The SQL that is generated to create the table is:

```
CREATE table "MYTABLE" (  
    "FIRST_NAME" CHAR(100) NOT NULL,  
    "SECOND_NAME" CHAR(100) NOT NULL,  
    "SALARY" NUMBER,  
    "BIRTHDATE" DATE  
)  
/
```

You could also run this command in SQL*Plus command line to create the table.

9. Click **Create**. The table is created and a description of the table is displayed.

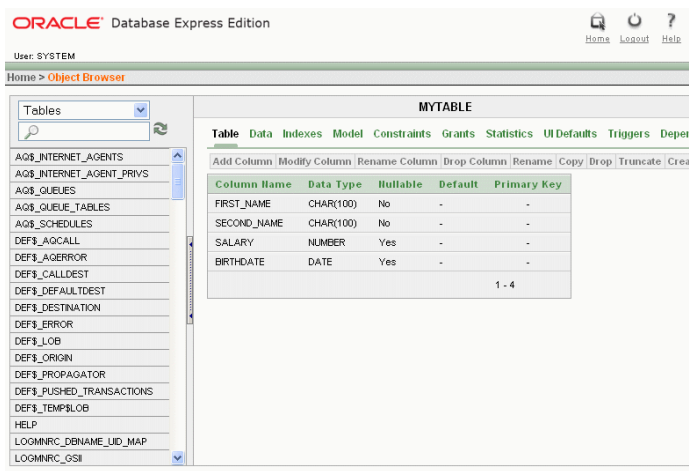


Figure 22: Oracle Application Express table created confirmation screen.

Working with SQL Scripts

The SQL Console 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. To access the tables used, log in as the *hr* user. To access Query Builder:

1. On the **Database** home page, click the **SQL** icon.
2. Click the **Query Builder** icon.



Figure 23: Oracle Application Express SQL options screen.

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

Using Oracle Database

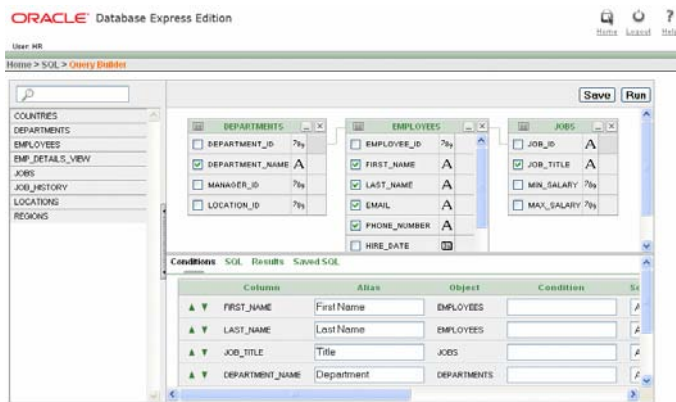


Figure 24: Oracle Application Express SQL query builder screen.

4. Select columns from the objects in the Design pane to select which columns to include in the query results.
5. Establish relationships between objects by clicking on the right-hand column of each table (optional).
6. Create query conditions (optional).
7. Click **Run** to execute the query and view the results.

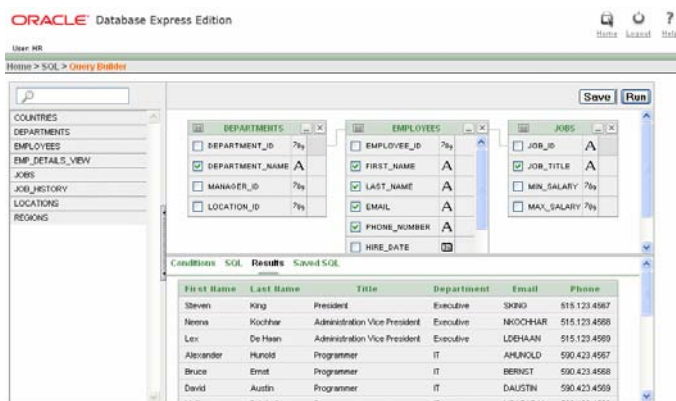


Figure 25: Oracle Application Express SQL query results screen.

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

Creating a PL/SQL Procedure

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

1. On the Database home page, click the **SQL** icon to display the SQL page.
2. Click the **SQL Commands** icon to display the SQL Commands page.

- On the SQL Commands page, enter some PL/SQL code. Here is some PL/SQL code to try if you aren't familiar with PL/SQL. This procedure, `emp_stat`, averages the salary for departments in the `hr` schema, and is encapsulated in a PL/SQL package called `emp_sal`. You need to enter each block separately, and click **Run**.

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

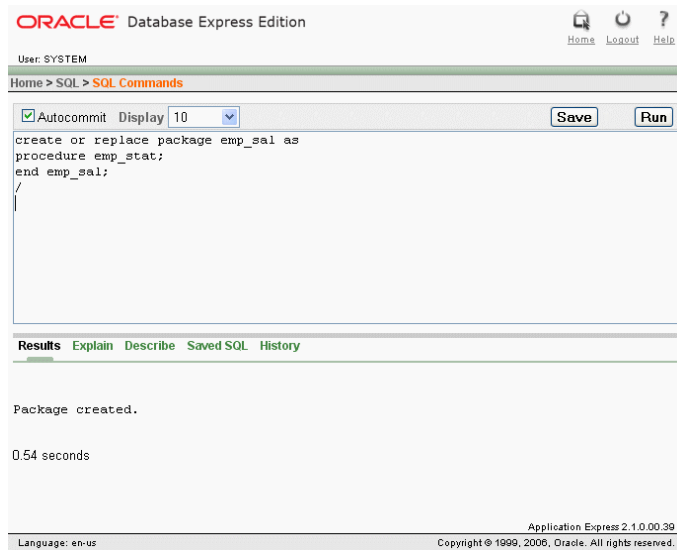


Figure 26: Oracle Application Express PL/SQL and SQL command screen.

Click **Run** to execute the PL/SQL. Then enter the following code:

```
create or replace package body emp_sal as
Procedure emp_stat is
TYPE EmpStatType is record (Dept_name varchar2(20),
Dept_avg number);
EmpStatVar EmpStatType;
BEGIN
DBMS_OUTPUT.PUT_LINE('Department          Avg Salary');
DBMS_OUTPUT.PUT_LINE('-----          -----');
For EmpStatVar in (select round(avg(e.salary),2)
                    a,d.department_name b
                    from departments d, employees e
                    where d.department_id=e.department_id
                    group by d.department_name)
LOOP
DBMS_OUTPUT.PUT_LINE(RPAD(EmpStatVar.b,16,' ')||
TO_CHAR(EmpStatVar.a,'999,999,999.99'));
END LOOP;
END;
```

Using Oracle Database

```
end emp_sal;
```

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

4. You can execute the stored procedure by entering the following PL/SQL code and clicking **Run**.

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

5. If you want to save the PL/SQL code for future use, click **Save**.

Creating a Database User

The Administration console is used to manage database:

- Storage
- Memory
- Users
- Activity of sessions and operations

The installation process creates an account named *system*. This account is considered an administrator account because it has DBA privileges (*SYSDBA*). To perform database administration such as creating new users, log into Oracle Application Express as the *system* user.

To create a database user:

1. On the Database home page, click the **Administration** icon. Click the **Database Users** icon.

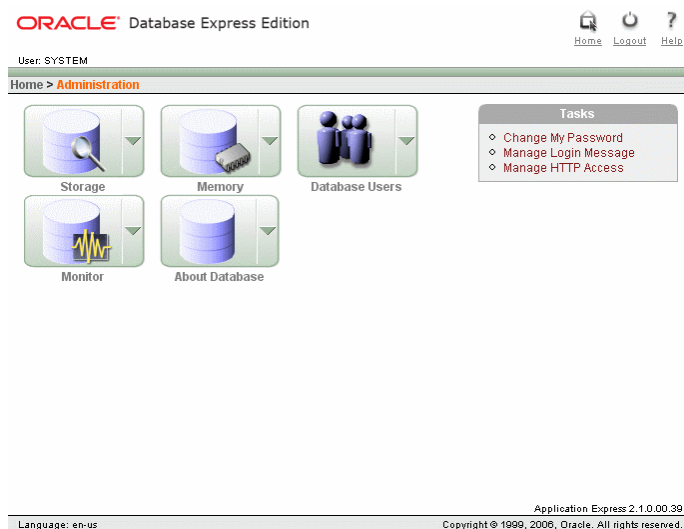


Figure 27: Oracle Application Express Administration screen.

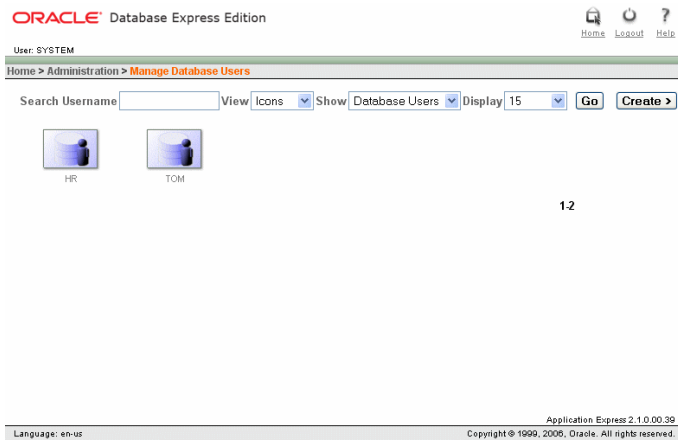


Figure 28: Oracle Application Express Manage Database Users screen.

2. On the Manage Database Users page, click **Create**. The Create Database User page is displayed.

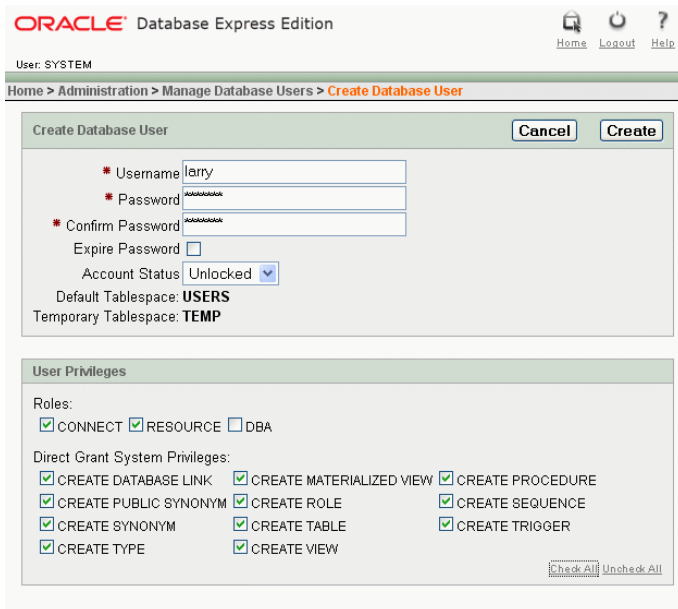
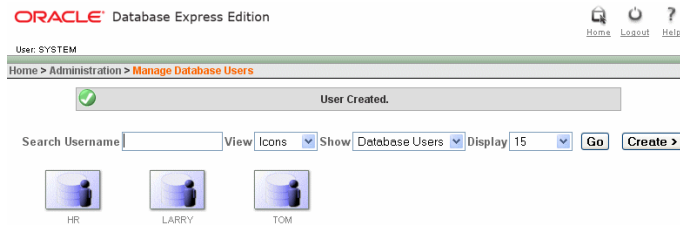


Figure 29: Oracle Application Express Create Database User screen.

3. Enter user information into text fields:
4. In the Username field, enter a new username.
5. In the Password and Confirm Password fields, enter a password.
6. Grant all create object system privileges by clicking Check All at the lower right-hand corner of the User Privileges box.

Using Oracle Database

7. The DBA role is by default not selected. The DBA privilege, gives the user the ability to create schema objects in other users' schemas, and to create other users.
8. Click **Create**. The Manage Database Users page displays a confirmation that the user was created.



13

Figure 30: Oracle Application Express Manage Database Users screen.

Monitoring Database Sessions

You can use the Oracle Database XE graphical user interface to monitor the current database sessions. This enables you to determine the users who are currently logged in to the database and what applications they are running.

You can also use the Oracle Database XE graphical user interface to kill a session—to cause it to be disconnected and its resources to be relinquished.

When you view sessions, you can view:

- All sessions
- Active sessions only
- Sessions that match a search string

You should be logged in as the *system* user to perform any database administration. To view the current sessions:

1. On the Database home page, click the **Administration** icon. Click **Monitor**.

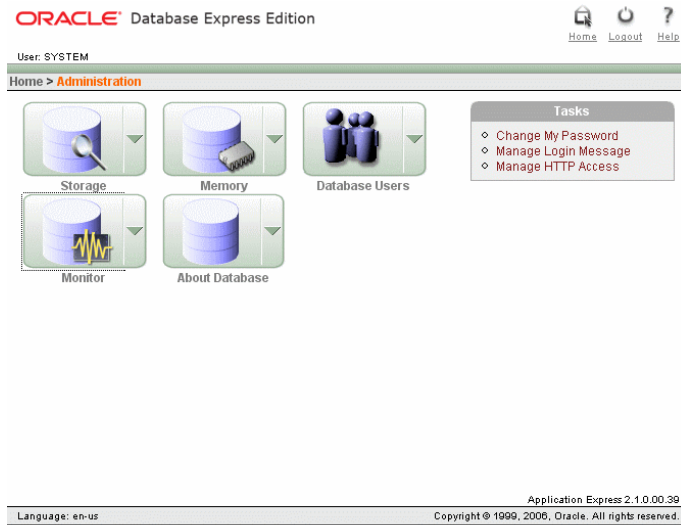


Figure 31: Oracle Application Express Administration screen.

2. On the Database Monitor page, click **Sessions**.

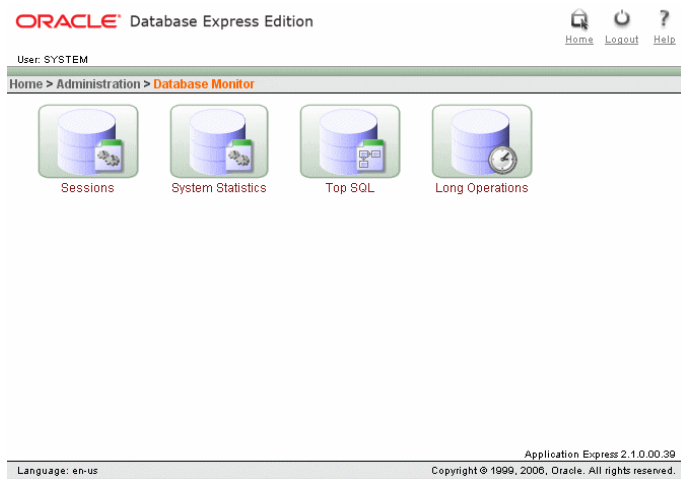


Figure 32: Oracle Application Express Administration Monitor screen.

Using Oracle Database

3. The Sessions page is displayed and shows the current active sessions.

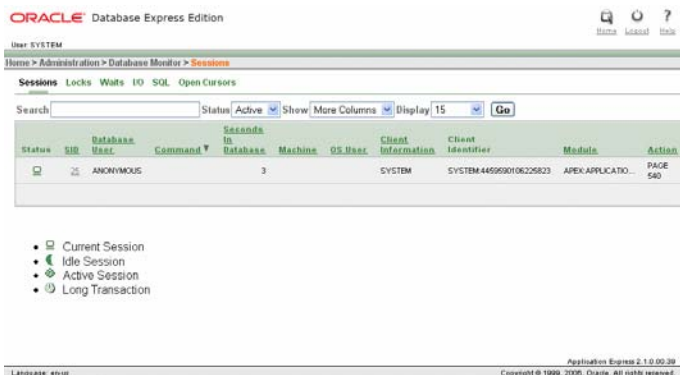


Figure 33: Oracle Application Express Sessions screen.

4. (Optional) In the Status list, select **All**, and click **Go**. The page displays all sessions, including idle sessions. (An example of an idle session is a SQL*Plus command line session that is not currently running a command.)
5. (Optional) Narrow down the sessions list by entering search text into the **Search** field and clicking **Go**. A session is shown if any of the following fields contain the search text: SID, Database User, Machine, OS User, Client Information, Client Identifier, and Module.
6. (Optional) Click any of the hyperlinks above the Search field to view the following information for all sessions: Locks, Waits, Input/Output (I/O), running SQL statements, and open cursors.
7. (Optional) Under the SID column, click a session ID to view the Session Details page for that session. The Session Details page enables you to kill the session.

Database Backup and Recovery

Backing up and restoring Oracle Database XE is based on protecting the physical files that make up the database: the datafiles, the control file, the server parameter file (SPFILE), and, if in ARCHIVELOG mode, the redo log files.

In Oracle Database XE, database backup and recovery is handled by Recovery Manager (RMAN). Oracle Database XE includes backup and restore scripts that you access using menu on your desktop. These scripts perform a full backup and restore of the entire database, and store backup files in the *flash recovery area*.

Oracle Database XE implements a backup retention policy that dictates that two complete backups of the database must be retained. In ARCHIVELOG mode, all archived logs required for media recovery from either backup are also retained. The database automatically manages backups and archived logs in the flash recovery area, and deletes obsolete backups and archived logs at the end of each backup job.

The backup script performs online backups of databases in ARCHIVELOG mode and offline backups of databases in NOARCHIVELOG mode. Online backups are backups that can run while the database is running. Offline backups are backups that run when the database is shut down.

The restore script restores the database differently depending on whether log archiving is on or off.

Log archiving on (ARCHIVELOG mode) restores the backed up database files, and then uses the online and archived redo log files to recover the database to the state it was in before the software or media failure occurred. All committed transactions that took place after the last backup are recovered, and any uncommitted transactions that were under way when the failure took place are rolled back (using undo data from the restored undo tablespace).

Log archiving off (NOARCHIVELOG mode) restores the database to its state at the last backup. Any transactions that took place after the last backup are lost.

Backing Up The Database

To back up the database:

1. Log in to the Oracle Database XE host computer as a user who is a member of the *dba* user group.
2. Do one of the following:
 - On Linux with Gnome, select **Applications > Oracle Database 10g Express Edition > Backup Database**.
 - On Linux with KDE, select **K Menu > Oracle Database 10g Express Edition > Backup Database**.
 - On Windows, select **Start > Programs > Oracle Database 10g Express Edition > Backup Database**.
3. A console window opens so that you can interact with the backup script. If running in ARCHIVELOG mode, the script displays the following output:

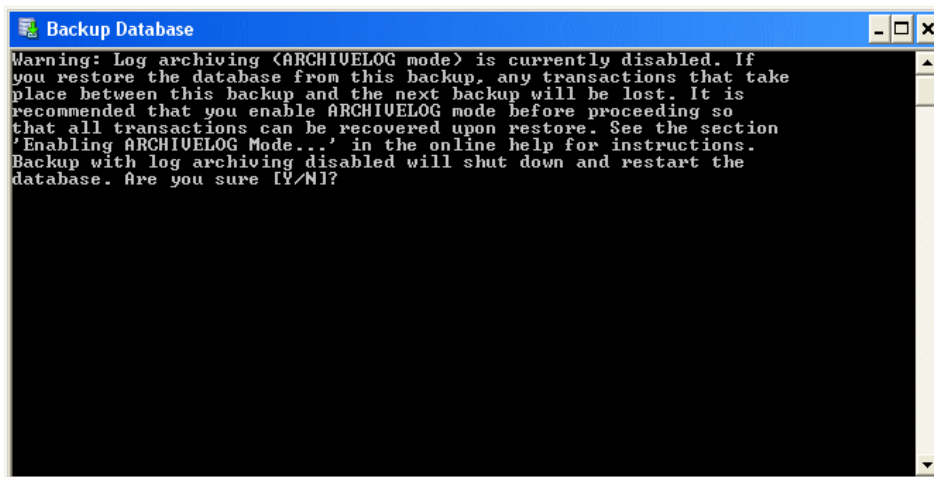
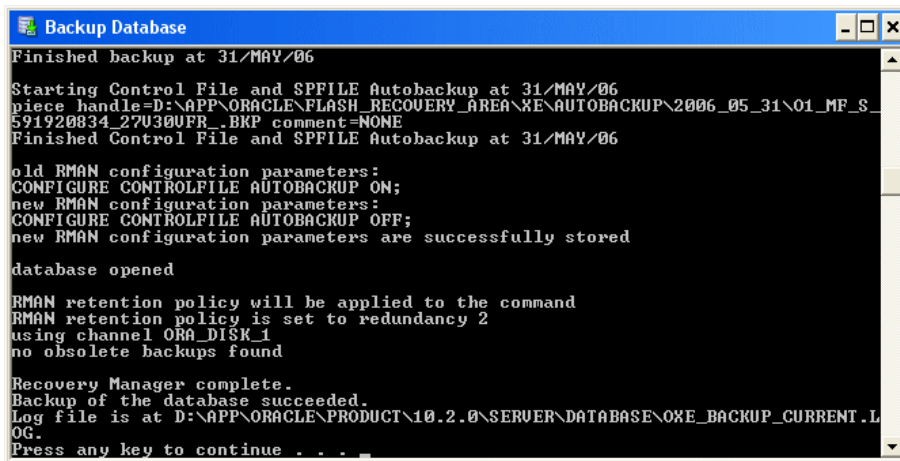


Figure 34: Oracle Application Express backup dialog.

If prompted, answer *y* to confirm the database shutdown and begin the backup. After the backup is complete, the script displays the following output:

Using Oracle Database

A screenshot of a Windows-style window titled "Backup Database". The window contains a text area with the following text: "Finished backup at 31/MAY/06", "Starting Control File and SPFILE Autobackup at 31/MAY/06", "piece handle=D:\APP\ORACLE\FLASH_RECOVERY_AREA\XE\AUTOBACKUP\2006_05_31\01_MF_S_591920834_27030UFR_.BKP comment=NONE", "Finished Control File and SPFILE Autobackup at 31/MAY/06", "old RMAN configuration parameters:", "CONFIGURE CONTROLFILE AUTOBACKUP ON;", "new RMAN configuration parameters:", "CONFIGURE CONTROLFILE AUTOBACKUP OFF;", "new RMAN configuration parameters are successfully stored", "database opened", "RMAN retention policy will be applied to the command", "RMAN retention policy is set to redundancy 2", "using channel ORA_DISK_1", "no obsolete backups found", "Recovery Manager complete.", "Backup of the database succeeded.", "Log file is at D:\APP\ORACLE\PRODUCT\10.2.0\SERVER\DATABASE\XE_BACKUP_CURRENT.LOG.", and "Press any key to continue . . .".

```
Backup Database
Finished backup at 31/MAY/06
Starting Control File and SPFILE Autobackup at 31/MAY/06
piece handle=D:\APP\ORACLE\FLASH_RECOVERY_AREA\XE\AUTOBACKUP\2006_05_31\01_MF_S_
591920834_27030UFR_.BKP comment=NONE
Finished Control File and SPFILE Autobackup at 31/MAY/06
old RMAN configuration parameters:
CONFIGURE CONTROLFILE AUTOBACKUP ON;
new RMAN configuration parameters:
CONFIGURE CONTROLFILE AUTOBACKUP OFF;
new RMAN configuration parameters are successfully stored
database opened
RMAN retention policy will be applied to the command
RMAN retention policy is set to redundancy 2
using channel ORA_DISK_1
no obsolete backups found
Recovery Manager complete.
Backup of the database succeeded.
Log file is at D:\APP\ORACLE\PRODUCT\10.2.0\SERVER\DATABASE\XE_BACKUP_CURRENT.L
OG.
Press any key to continue . . .
```

Figure 35: Oracle Application Express backup successful message.

4. Press any key to close the Backup Database window.

Logs containing the output from the backups are stored in the following locations:

```
$ORACLE_HOME/oxe_backup_current.log
$ORACLE_HOME/oxe_backup_previous.log
```

Restoring the Database

To restore a database from a backup:

1. Log in to the Oracle Database XE host computer as a user who is a member of the *dba* user group.
2. Do one of the following:
 - On Linux with Gnome, select **Applications > Oracle Database 10g Express Edition > Restore Database**.
 - On Linux with KDE, select **K Menu > Oracle Database 10g Express Edition > Restore Database**.
 - On Windows, select **Start > Programs > Oracle Database 10g Express Edition > Restore Database**.
3. A console window opens so that you can interact with the restore script. The script displays the following output:

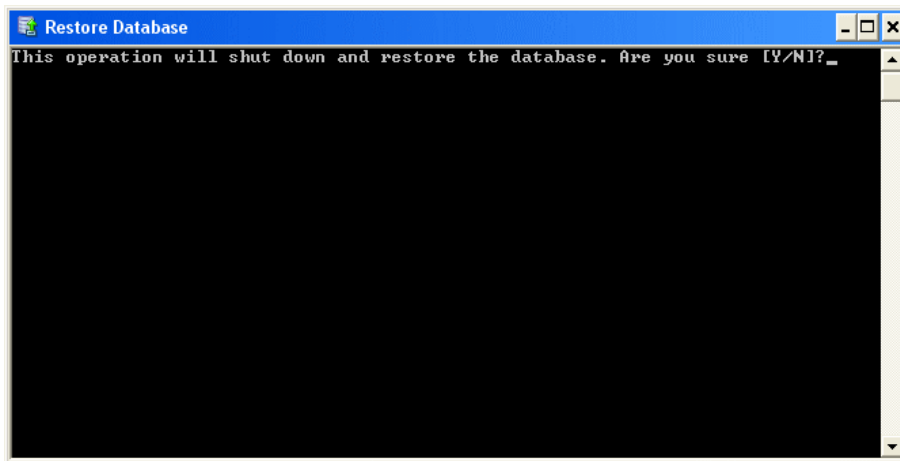


Figure 36: Oracle Application Express restore dialog.

4. Answer **y** to confirm the database shutdown and begin the restore. After the restore is complete, the script displays the following output:

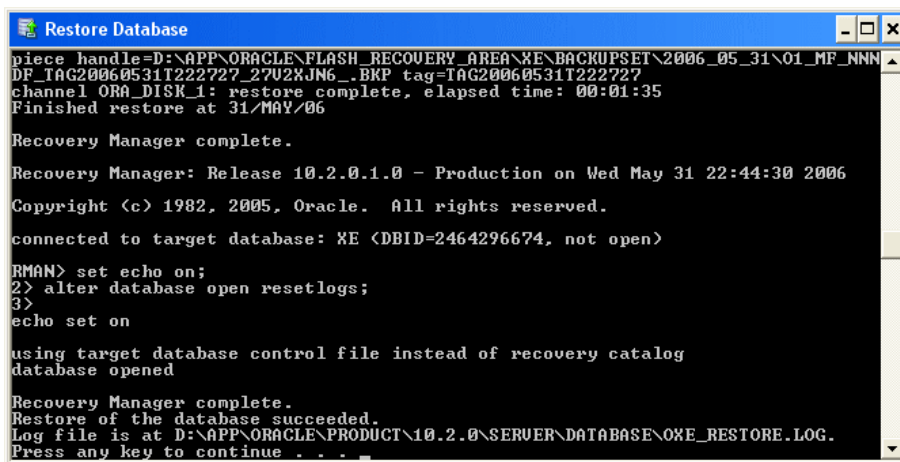


Figure 37: Oracle Application Express restore successful message.

5. Press any key to close the Restore Database window.

A log containing the output from the restore is stored in:

```
$ORACLE_HOME/oxe_restore.log
```

Oracle SQL*Plus

SQL*Plus is Oracle's traditional command line tool. It is available whenever the database is installed. Its command set is limited but it allows ad-hoc queries, scripting and fundamental database administration. Many books, including this one, use SQL*Plus to show SQL examples. For easy development, you may prefer to use SQL Developer, which is described in the next section.

Using Oracle Database

Starting SQL*Plus

Oracle Database 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.

On Linux, for Oracle Database XE, set the environment with:

```
$ . /usr/lib/oracle/xe/app/oracle/product/10.2.0/server/bin/oracle_env.sh
```

Note the space after the period.

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:

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

You will be prompted for the system identifier ("SID") of the database on this machine that you intend to connect to. 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.

Once the environment is set, SQL*Plus can be started with the [sqlplus](#) command:

```
$ sqlplus

SQL*Plus: Release 10.2.0.1.0 - Production on Thu Nov 6 11:17:09 2008
Copyright (c) 1982, 2005, Oracle. All rights reserved.

Enter user-name: hr
Enter password:

Connected to:
Oracle Database 10g Express Edition Release 10.2.0.1.0 - Production

SQL>
```

The password you enter is not displayed. The prompt [SQL>](#) is shown. Type [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 hr/hrpwd@localhost/XE
```

A better practice is to run:

```
sqlplus hr@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 hr/hrpwd@localhost/XE
Connected.
```

```
SQL>
```

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

```
$ sqlplus / as sysdba
```

or

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

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>
```

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 (with SQL*Plus 11g) from BLOB columns, increase the maximum number of characters that will display (the default is just 80):

Using Oracle Database

```
SQL> set long 1000
```

Note these are local commands to SQL*Plus and do not need a semi-colon.

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 with the [SPOOL](#) command:

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

Running Scripts in SQL*Plus

If multiple SQL statements are stored in a script *myscript.sql*, they can be executed with the [@](#) command 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, writing statements in scripts using an external editor and running them this way is recommended.

Information On Tables 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                           V          VARCHAR2(40)
REGION_ID                               V          NUMBER

```

Accessing the Demonstration Tables in SQL*Plus

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 hrpwd account unlock;

```

This sets the password to `hrpwd`.

Oracle SQL Developer

In addition to Oracle Application Express or SQL*Plus, you can also use Oracle SQL Developer for database development and maintenance. Oracle SQL Developer is a free, thick-client graphical tool for database development. You can use SQL Developer to execute SQL statements, execute and debug PL/SQL statements, and to run a few SQL*Plus commands (like `DESCRIBE`). SQL Developer includes some prebuilt reports, and you can create and save your own reports.

SQL Developer can connect to Oracle databases from version 9.2.0.1 onwards, and is available on Linux, Windows and Mac OS X.

You can download SQL Developer from the Oracle Technology Network at http://www.oracle.com/technology/products/database/sql_developer. You can also download patches, 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

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 38.

Using Oracle Database

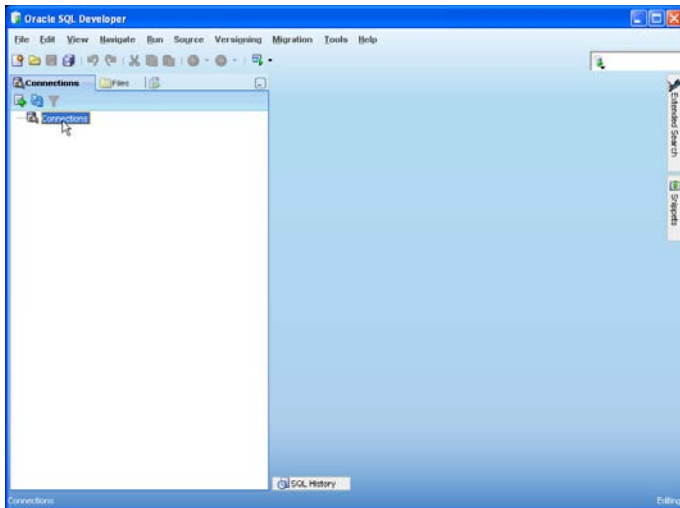


Figure 38: Oracle SQL Developer login screen.

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

1. Select **Connections** in the left pane, right click and select **New Connection**. Enter the login credentials for the Oracle Database XE with the username `hr`, and password you created for the `hr` user, the hostname `localhost`, and the SID `xe`.

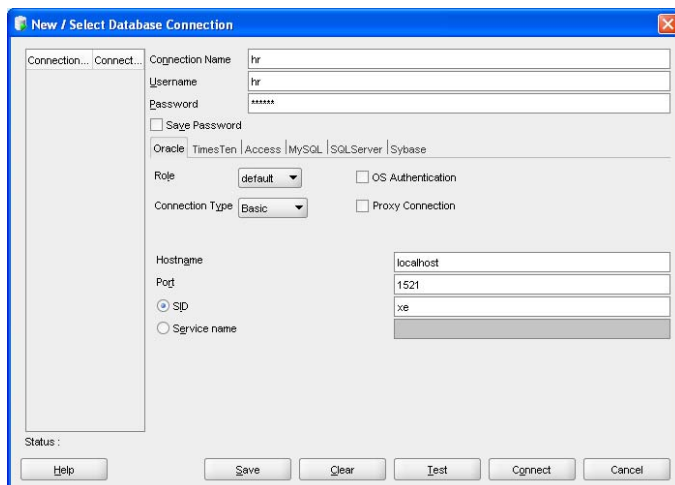


Figure 39: Oracle SQL Developer Connection screen.

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. Click the **Connect** button to save the connection and connect to it.

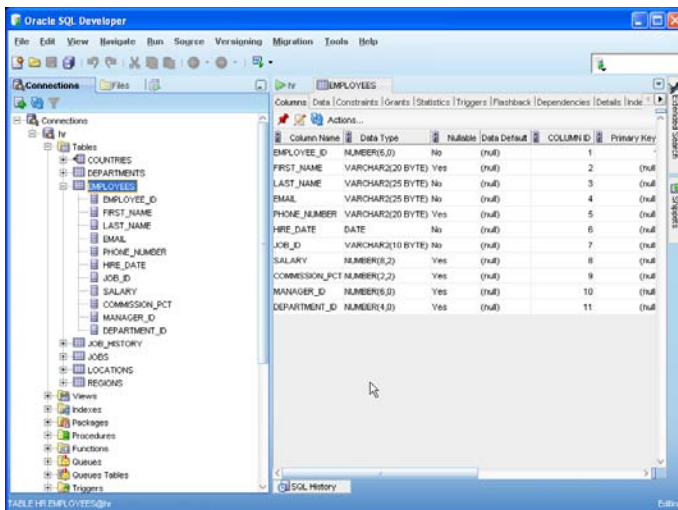


Figure 40: Oracle SQL Developer main screen.

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 40, the *employees* table is displayed.

Editing Data

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 asterisk. Throughout SQL Developer, there are context sensitive menus. Figure 41 shows the choice of context menus available in the data grid.

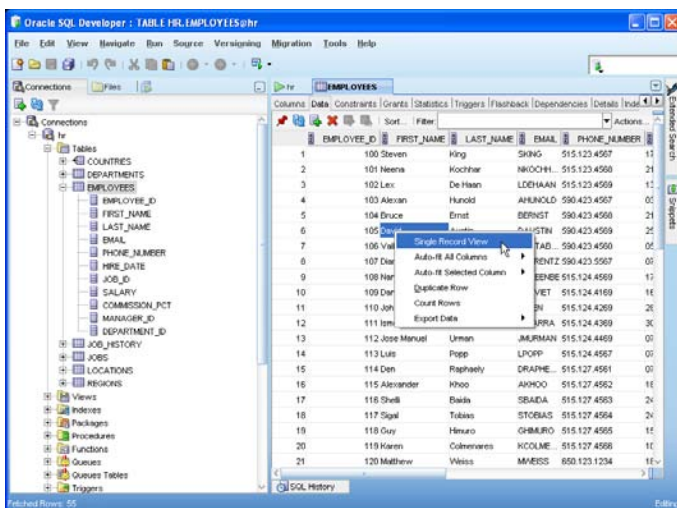


Figure 41: Oracle SQL Developer Data Grid.

Creating a Table

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 **Create Table**. The Create Table dialog is displayed. Enter the column names, types and other parameters as required.

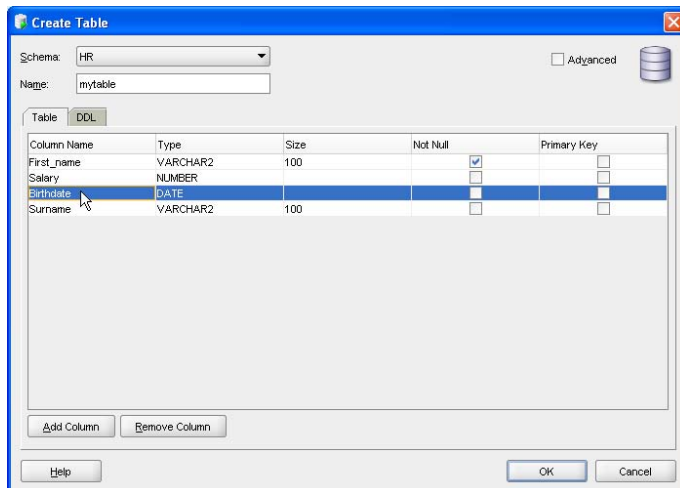


Figure 42: Oracle SQL Developer Create Table screen.

Click the **Advanced** check box to see more advanced options like constraints, indexes, foreign keys, and partitions.

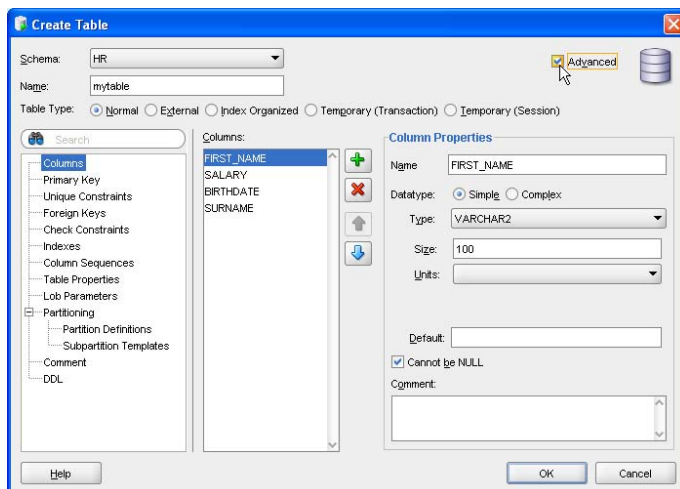


Figure 43: Oracle SQL Developer Advanced Create Table screen.

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

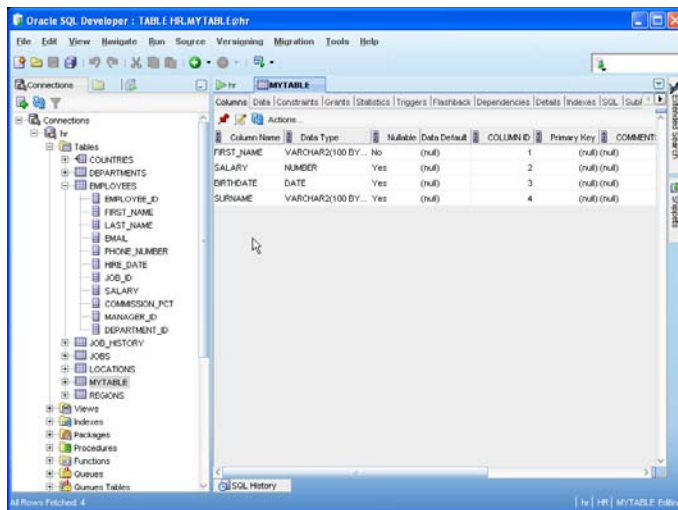


Figure 44: Oracle SQL Developer screen showing the new table, mytable.

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

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 **Database Connection** tab in the right hand pane. This is the connection created earlier in *Creating a Database Connection*. 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.

Using Oracle Database

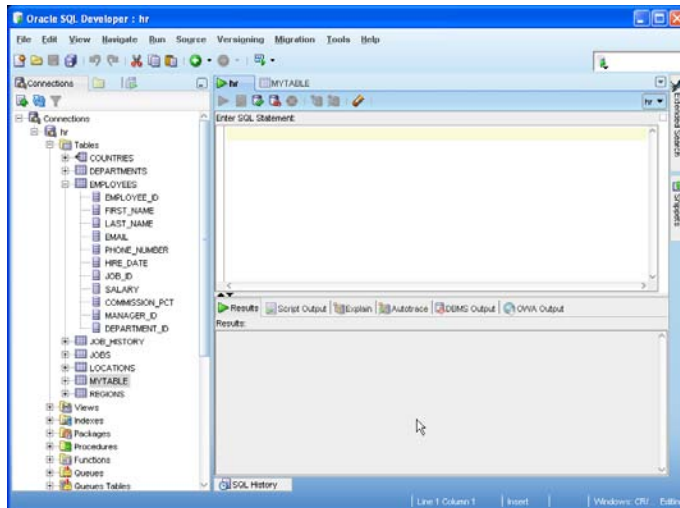


Figure 45: Oracle SQL Developer screen showing SQL Worksheet.

2. Enter the following two statements in the SQL Worksheet:

```
DESCRIBE HR.EMPLOYEES  
SELECT * FROM HR.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**. If you have used Oracle's command line tool SQL*Plus, the output in the Script Output window is likely to be familiar to you.

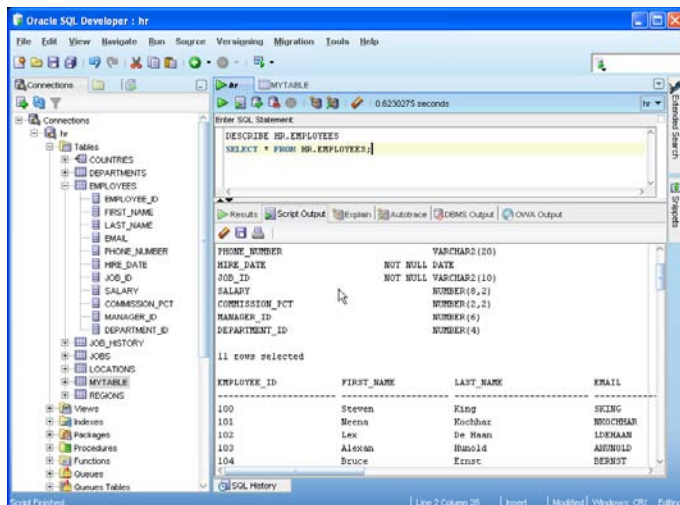


Figure 46: Oracle SQL Developer screen showing SQL Worksheet with output.

- If you want to execute a single line of the two-line statement, 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.

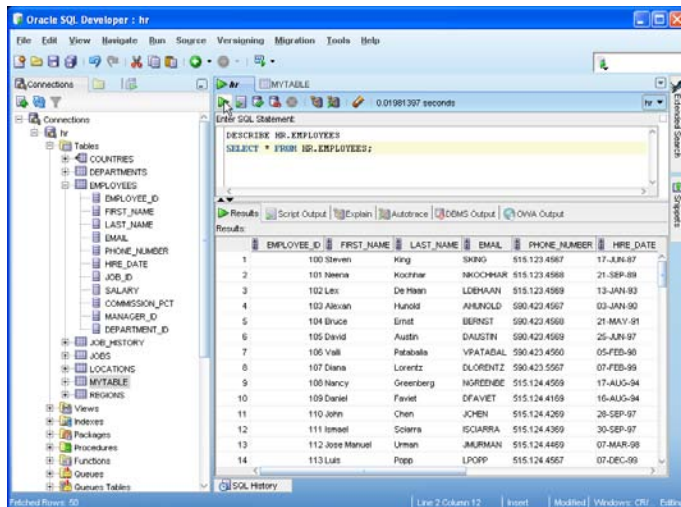


Figure 47: Oracle SQL Developer screen showing SQL Worksheet with output.

You should also take a moment to click on the **Snippets** on the top right hand of the interface. Snippets are a handy set of SQL and PL/SQL code snippets that you can drag into the SQL Worksheet component.

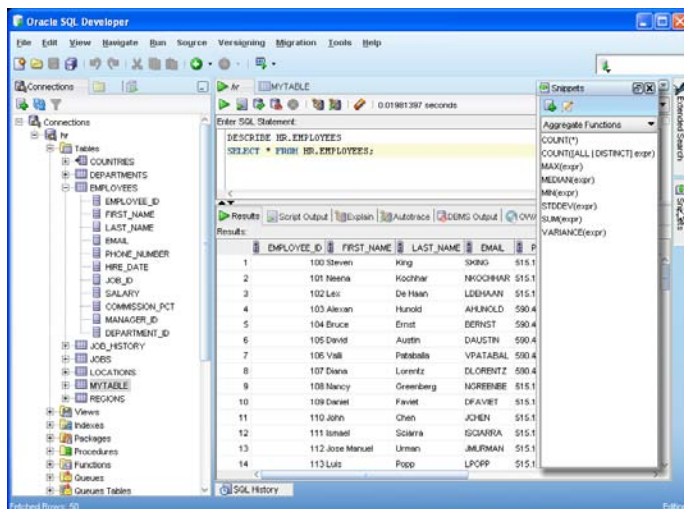


Figure 48: Oracle SQL Developer screen showing code snippets.

Editing, Compiling and Running PL/SQL

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

- Right click the **Procedures** node and select **New Procedure** from the context menu. Add the name

Using Oracle Database

ADD_DEPT and click OK.

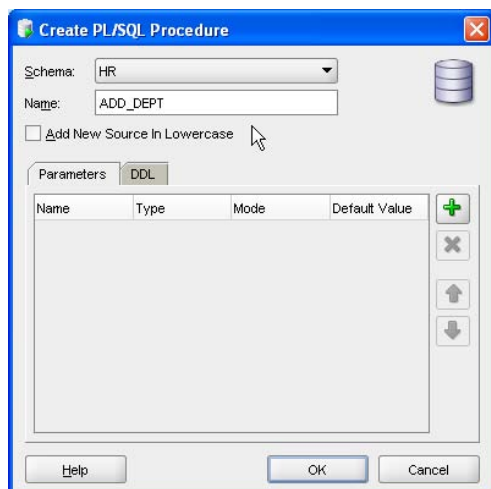


Figure 49: SQL Developer new procedure dialog.

2. The PL/SQL Editor is opened. Replace or add the text so that your 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 the Message-Log window below

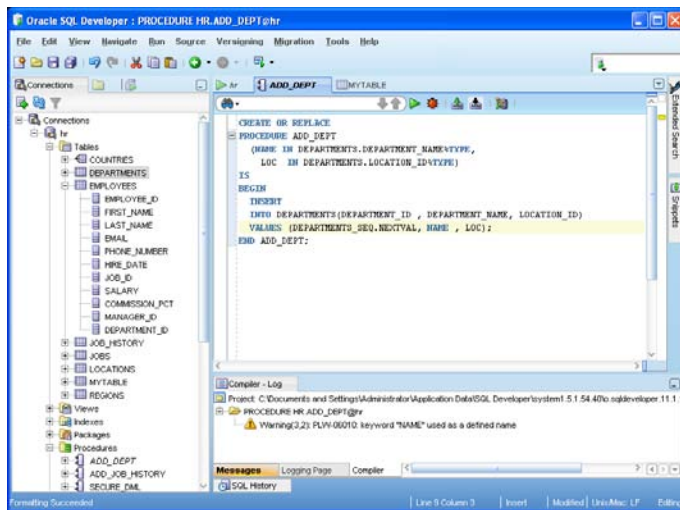


Figure 50: SQL Developer PL/SQL Code Editor.

4. Run the procedure. The green arrow icon runs the code.
5. SQL Developer provides 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**. Query the Departments table to see the results added.

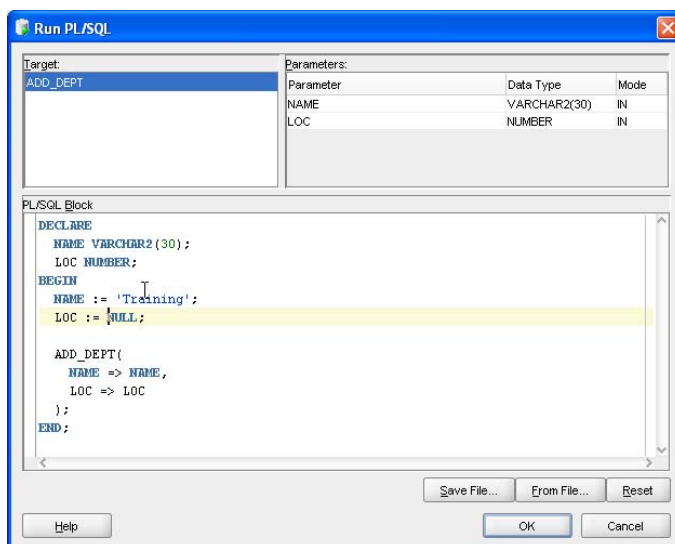


Figure 51: SQL Developer Run PL/SQL dialog.

Running Reports

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.

Using Oracle Database

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**. The report is run and the results are displayed in the right hand pane.

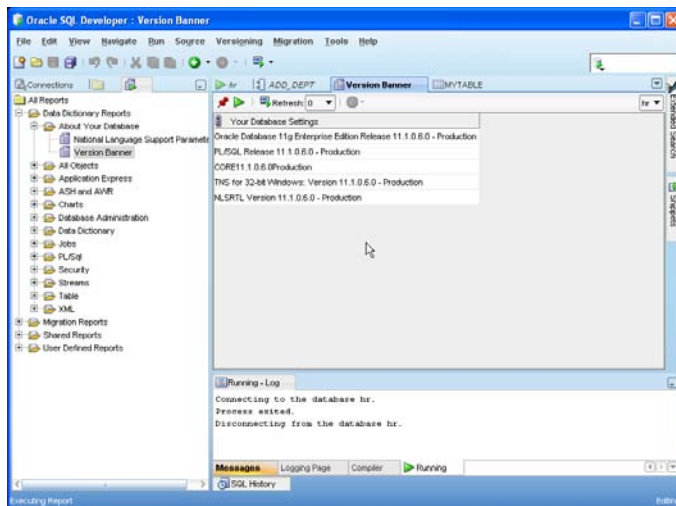


Figure 52: Oracle SQL Developer screen showing output from a report.

2. Click the **Run Report in SQL Worksheet** icon (next to Refresh). The source code is written to the SQL Worksheet. You can also export the supplied reports to edit and add back as user defined reports.

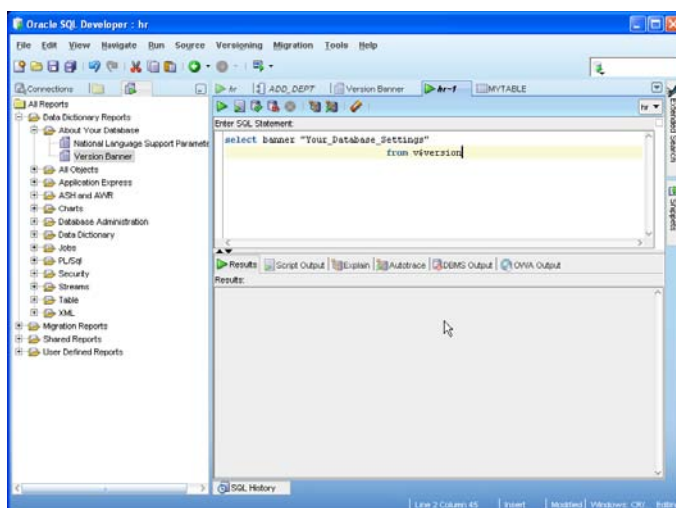


Figure 53: Oracle SQL Developer screen showing the source code of a report.

Creating Reports

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

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

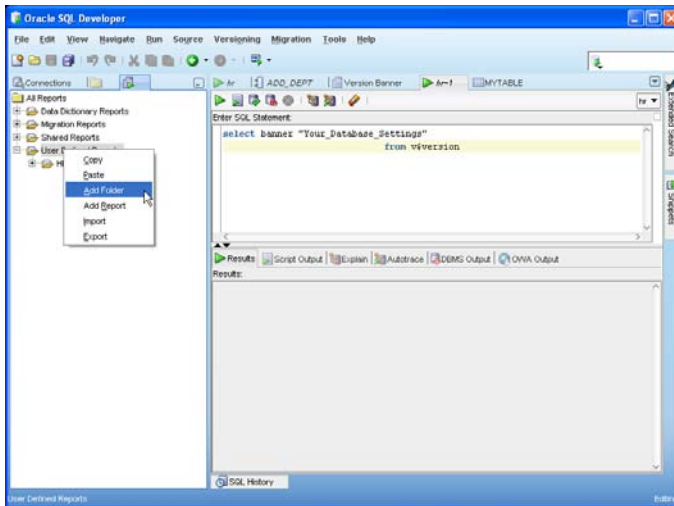


Figure 54: SQL Developer User Defined Reports

2. Select the folder you created and right click to select **Add Report**.
3. 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
```

4. 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** button on the **Details** tab.

Using Oracle Database

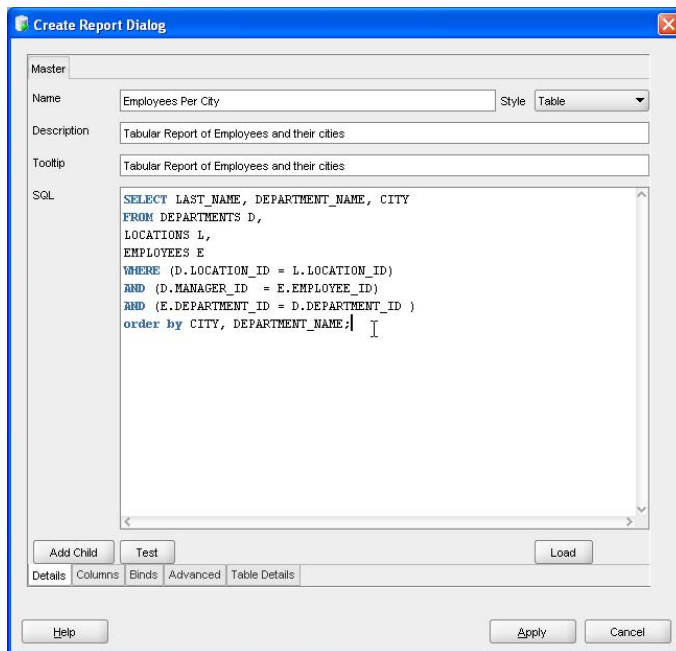


Figure 55: SQL Developer Create Report Dialog

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

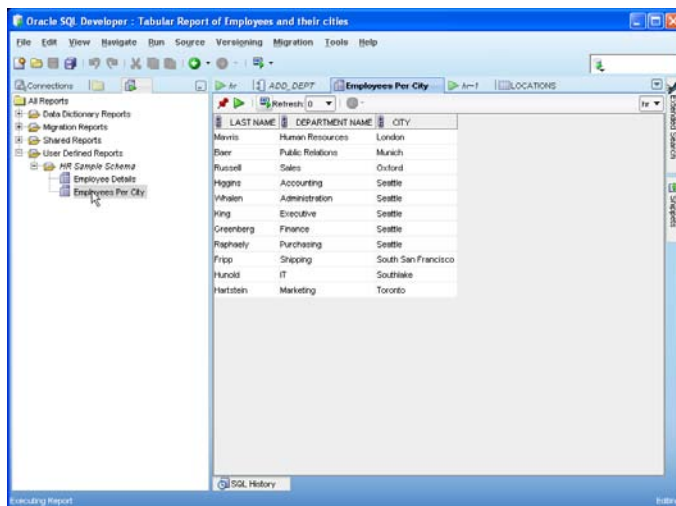


Figure 56: SQL Developer User Defined, Tabular Report

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.

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, you can skip this chapter.

The instructions use Apache HTTP Server Release 2.0.59 from <http://httpd.apache.org/download.cgi>. Steps are given for Oracle Enterprise Linux and Windows XP Professional Edition. The procedure to install on other Linux platforms is the same as for Oracle Enterprise Linux.

Note: Apache release 2.2.9 is the latest version available. The installation shown in this Chapter may need some variation if this latest version is used.

Installing Apache HTTP Server on Linux

To install the Apache HTTP Server for use with PHP on Oracle Enterprise Linux.

1. Download the Apache HTTP Server from <http://httpd.apache.org/download.cgi>. The version used in this installation is *httpd-2.0.59.tar.bz2*.
2. Log in as the *root* user and extract the files:

```
# tar -jxvf httpd-2.0.59.tar.bz2
```

If you are familiar with the `tar` command on UNIX systems, you may be wondering why we did not need to invoke `bunzip2` to extract the *.tar* file. Linux includes the GNU version of `tar`, which has a new `-j` flag to automatically uncompress a bzipipped *.tar* file. If you downloaded the *.gz* gzipped file, you could have used the `-z` flag instead.

3. Configure and build the web server:

```
# cd httpd-2.0.59
# ./configure --prefix=/usr/local/apache --enable-module=so
# make
# make install
```

When configuring the web server, the option `--enable-module=so` allows PHP to be compiled as a Dynamic Shared Object (DSO). Also, the `--prefix=` option sets where Apache HTTP Server will be installed during the command `make install`.

Installing Apache HTTP Server

Note: With Apache 2, you should use the default pre-fork MPM (“Multi-Processing Module”), because thread-safety of many of the PHP libraries is not known.

Starting and Stopping Apache HTTP Server

The *apachectl* script is installed in the Apache *bin* directory. Use this script to start and stop Apache HTTP Server. To start Apache HTTP Server:

```
# /usr/local/apache/bin/apachectl start
```

You should test that Apache has been installed properly and is started on your machine by opening your web browser to <http://localhost/> or <http://127.0.0.1/> to display the Apache home page.

Now stop Apache so it can be configured for PHP:

```
# /usr/local/apache/bin/apachectl stop
```

Note: If you are using Oracle Database10g Release 10.2, but not Oracle Database XE, you must give the *nobody* user access to files in the Oracle directory. With Oracle Database 10g Release 10.2.0.2 onwards, there is a script located in `$ORACLE_HOME/install/changePerm.sh` to do this.

If there are errors, they are displayed on your screen. Errors may also be recorded in `/usr/local/apache/logs/error_log`. If you have problems, check your *httpd.conf* and *php.ini* configuration files for any incorrect settings, and make corrections.

Configuring Apache HTTP Server on 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, and what optional settings are desired. Information about setting the environment is in *Setting Oracle Environment Variables for Apache* in the chapter *Connecting to Oracle Using OCI8*.

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. 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 on Windows

The following procedure describes how to install the Apache HTTP Server on Windows.

1. Download the Apache HTTP Server Windows binaries from <http://www.apache.org/dist/httpd/binaries/win32/>. The release used in this installation is Apache HTTP Server 2.0.59, and the downloaded file is named *apache_2.0.59-win32-x86-no_ssl.msi*.
2. Double click the downloaded file *apache_2.0.59-win32-x86-no_ssl.msi*.
3. Follow the Apache HTTP Server installation wizards. You should select to install Apache HTTP Server **for All Users, on Port 80**, because the **only for the Current User** alternative will clash with Oracle Database Express Edition's default port 8080.

Starting and Stopping Apache HTTP Server

As part of installation, the Apache HTTP Server is started. You should now test that Apache HTTP Server has been installed properly and started on your machine by opening your web browser to <http://localhost/> or <http://127.0.0.1/> to display the Apache home page.

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.

Installing Apache HTTP Server

INSTALLING PHP

This Chapter discusses ways of installing PHP on Linux and Windows. Installation on other systems such as Solaris and Mac OS X is similar to the Linux steps.

The main installation scenarios are covered. You might come across other situations that require a combination of steps to be followed.

The Apache HTTP Server need to be installed before installing PHP, and a database must be accessible in your network.

Installing PHP with OCI8 on Linux

These steps assume the default Apache *httpd* RPM package is used on Oracle Enterprise Linux 5. However, if you manually installed Apache in */usr/local/apache*, then the command to start and stop Apache is */usr/local/apache/bin/apachectl*, the APXS utility is in */usr/local/apache/bin/apxs*, the configuration file is in */usr/local/apache/conf/httpd.conf* and the module directory is */usr/local/apache/modules*.

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

Installing OCI8 Using a Local Database

To install OCI8 on Oracle Enterprise Linux which has a database installed, login as the *root* user:

1. Shutdown Apache:

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

2. Download PHP 5.2.7 from <http://www.php.net/downloads.php>.
3. Log in as the *root* user and extract the PHP source code:

```
# tar -jxf php-5.2.7.tar.bz2
# cd php-5.2.7
```

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

4. Configure PHP with OCI8:

```
# export ORACLE_HOME=/usr/lib/oracle/xe/app/oracle/product/10.2.0/server
# ./configure \
>   --with-apxs2=/usr/sbin/apxs \
>   --with-oci8=$ORACLE_HOME \
>   --enable-sigchild
```

Use the appropriate path to your Oracle software installation. The path can be found in */etc/oratab*. If you installed your own Apache, use the appropriate path to the Apache extension tool, for example */usr/local/apache/bin/apxs*. If you have Apache 1.3 instead of Apache 2, change the `--with-apxs2` option to `--with-apxs`.

Installing PHP

Note: Although `configure` fails if it cannot find the Oracle software, prior to PHP 5.2.4 it silently ignores a misspelled option name. Before continuing, review the output and make sure the “*checking for Oracle (OCI8) support*” line was marked as “yes”.

Other desired options and extensions can be used in the `configure` command. To list all the options, use the command:

```
configure --help
```

5. Make and install PHP:

```
# make
# make install
```

6. Copy PHP’s supplied initialization file *php.ini-recommended*. To find the destination directory, use the `--ini` option to command line PHP:

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

This shows the path is */usr/local/lib*. Copy the file to that directory:

```
# cp php.ini-recommended /usr/local/lib/php.ini
```

7. For testing it is helpful to edit *php.ini* and set `display_errors` to `On` so you see any problems in your code. Make sure you change this configuration option back to `Off` before making your application available to users.
8. Edit Apache’s configuration file */etc/httpd/conf/httpd.conf* and add the following lines:

```
#
# This section will call PHP for .php, .phtml, and .phps files
#
AddType application/x-httpd-php .php
AddType application/x-httpd-php .phtml
AddType application/x-httpd-php-source .phps
```

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

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

10. Set any required Oracle environment variables, such as `ORACLE_HOME`, `LD_LIBRARY_PATH` and `NLS_LANG`. See *Setting Oracle Environment Variables for Apache* in the chapter *Connecting to Oracle Using OCI8*.

- Restart Apache:

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

Installing OCI8 Using Oracle Instant Client

PHP links with Oracle libraries. If PHP is not installed on the database host machine, then a small set of libraries called Oracle Instant Client can be used.

To install with Oracle Instant Client:

- Shutdown Apache:

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

- Download PHP 5.2.7 from <http://www.php.net/downloads.php>.
- Extract the PHP source code:

```
# tar -jxf php-5.2.7.tar.bz2
```

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

- Download the RPM or ZIP files for the **Basic** and **SDK** Instant Client packages from the Instant Client page on the Oracle Technology Network:

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

Collectively, the two packages are about 41 MB in size. 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.

- If you are using the RPMs, install the RPMs as the *root* user:

```
# rpm -Uvh oracle-instantclient11.1-basic-11.1.0.7.0-1.i386.rpm
# rpm -Uvh oracle-instantclient11.1-devel-11.1.0.7.0-1.i386.rpm
```

The first RPM puts the Oracle libraries in `/usr/lib/oracle/11.1/client/lib` and the second creates headers in `/usr/include/oracle/11.1/client`.

If you are using the Instant Client ZIP files, unzip the Basic and the SDK packages to a directory of your choice, for example `$HOME/instantclient_11_1`. The files should be unzipped together so the SDK is in `$HOME/instantclient_11_1/sdk`.

- If Instant Client was installed from the ZIP files, create a symbolic link:

```
# cd $HOME/instantclient_11_1
# ln -s libclntsh.so.11.1 libclntsh.so
```

- Configure PHP:

```
# cd php-5.2.7
# ./configure \
```

Installing PHP

```
> --with-apxs2=/usr/sbin/apxs \  
> --with-oci8=instantclient,/usr/lib/oracle/11.1/client/lib \  
> --enable-sigchild
```

If you are using the ZIP files then change the `--with-oci8` option to the unzipped directory:

```
--with-oci8=instantclient,$HOME/instantclient_11_1
```

If you are using Instant Client RPMs on 64 bit Linux, use:

```
--with-oci8=instantclient,/usr/lib/oracle/11.1/client64/lib
```

8. Rebuild PHP:

```
# make  
# make install
```

9. Copy PHP's supplied initialization file:

```
# cp php.ini-recommended /usr/local/lib/php.ini
```

10. Set any required Oracle environment variables such as `LD_LIBRARY_PATH` and `NLS_LANG`. See *Setting Oracle Environment Variables for Apache* in the chapter *Connecting to Oracle Using OCI8*.

11. Restart the Apache HTTP Server:

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

Upgrading PHP with PECL OCI8 on Linux

The latest OCI8 package from PECL can be used to upgrade OCI8 in a full PHP source bundle – if the PECL version is more recent than that included in PHP. Upgrading OCI8 is **strongly** recommended if you must use PHP 4. Note PHP 4 is no longer maintained by the PHP community and should not be used for new projects. OCI8 1.3 is also more recent than the OCI8 extension in PHP 5.2.

This section uses PHP 4 as the example target PHP distribution (OCI8 1.3 will build with PHP 4.3.9 onwards). The instructions are equivalent for other versions of PHP.

Upgrading OCI8 as a Static Library on Linux

If OCI8 is (or will be) statically linked with PHP, PHP must be rebuilt using the new extension source code. To build and install PHP 4 on Oracle Enterprise Linux:

1. Download `php-4.4.9.tar.bz2` from <http://www.php.net/downloads.php>.
2. Extract the file:

```
# tar -jxf php-4.4.9.tar.bz2
```

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

3. Delete the default oci8 extension:


```
# rm -rf php-4.4.9/ext/oci8
```

If you only rename the *oci8* directory the *configure* script will not be properly created.

4. Download the latest OCI8 extension from <http://pecl.php.net/package/oci8>.
5. Extract the files and move the directory to *ext/oci8*:

```
# tar -zxf oci8-1.3.4.tgz
# mv oci8-1.3.4 php-4.4.9/ext/oci8
```

6. Rebuild the configuration script with the updated extension's options:

```
# cd php-4.4.9
# rm -rf configure config.cache autom4te.conf
# ./buildconf --force
```

PHP prefers older versions of the operating system build tools. Before running *buildconf*, you might need to install the *autoconf213* package. If you have both old and new packages installed, later versions of PHP can be forced to use the appropriate version, for example with:

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

7. Configure and build PHP following step 4 onward in the previous section *Installing OCI8 Using a Local Database* or, if you intend to use Oracle Instant Client, by following the step 4 onwards in the section *Installing OCI8 Using Oracle Instant Client*.

Upgrading OCI8 on Linux Using the PECL Channel

PHP's PECL packaging system can also be used to install or upgrade OCI8 as a shared library.

These steps can also be used to add OCI8 to the default PHP that Oracle Enterprise Linux comes with. The *php-pear* and *php-devel* RPM packages are required to get the default *pear*, *pecl* and *phpize* executables on Enterprise Linux. To install:

1. Shutdown Apache:

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

2. Remove any existing OCI8 extension:

```
# pecl uninstall oci8
```

3. If you are behind a firewall, set the proxy, for example:

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

4. Download and install OCI8:

```
# pecl install oci8
```

Installing PHP

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 software location:

```
/usr/lib/oracle/xe/app/oracle/product/10.2.0/server
```

Otherwise if you have Oracle Instant Client 11.1.0.7 RPMs, type:

```
instantclient,/usr/lib/oracle/11.1/client/lib
```

On 64 bit Linux with Instant Client RPMs the line would be

```
instantclient,/usr/lib/oracle/11.1/client64/lib
```

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/lib/php/modules
```

6. Set any required Oracle environment variables such as `LD_LIBRARY_PATH` and `NLS_LANG`. See *Setting Oracle Environment Variables for Apache* in the chapter *Connecting to Oracle Using OCI8*.
7. Restart Apache:

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

Upgrading OCI8 as a Shared Library on Linux

These steps are the manual equivalent to the previous `pecl install oci8` command.

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

1. Shutdown Apache:

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

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

```
# rm /usr/lib/php/modules/oci8.so
```

3. Download the OCI8 1.3.4 extension from PECL, <http://pecl.php.net/package/oci8>
4. Extract and prepare the new code:

```
# tar -zxf oci8-1.3.4.tgz
# cd oci8-1.3.4
# phpize
```

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

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

Otherwise if you have Oracle Instant Client 11.1.0.7 use:

```
# ./configure --with-oci8=\
> shared,instantclient,/usr/lib/oracle/11.1/client/lib
```

6. Build the shared library:

```
# make
```

7. Install the library:

```
# cd oci8-1.3.4
# make install
```

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

```
extension = oci8.so
```

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

```
extension_dir = /usr/lib/php/modules
```

10. Set any required Oracle environment variables such as `LD_LIBRARY_PATH` and `NLS_LANG`. See *Setting Oracle Environment Variables for Apache* in the chapter *Connecting to Oracle Using OCI8*.

11. Restart Apache:

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

Installing PHP With OCI8 on Windows

This section discusses the PHP Windows MSI installer. PHP is also installable from ZIP files. Refer to the PHP documentation for those steps.

To install OCI8 on Windows, the Oracle 10g or 11g client libraries are required. These can be from a local database, or from Oracle Instant Client. Apache is also required.

PHP binaries for Windows come in thread-safe and non- thread-safe bundles. The non- thread-safe version requires the web server to be in a non-threaded mode.

Installing OCI8 Using a Local Database on Windows

To install OCI8 on Windows XP Professional Edition:

1. Download the PHP 5.2.7 installer package from <http://www.php.net/downloads.php>.
2. Double click on the downloaded file *php-5.2.7-win32-installer.msi*. The PHP Windows installer starts.
3. Click **Next** on the Welcome screen. The license agreement is displayed.
4. Accept the license agreement by checking the box and click **Next**.
5. Select the installation location of the PHP files and click **Next**.

Installing PHP

6. In the Web Server Setup screen, select the web server you want to use. This installation uses the Apache 2.0 module option. Select **Apache 2.0.x Module** and click **Next**.
7. In the Apache Configuration Directory screen, select the location of the Apache *conf* directory and click **Next**. For Apache 2.0, the standard configuration directory is *C:\Program Files\ApacheGroup\Apache2\conf*.
8. In the Choose Items to Install screen, scroll down to the **Extensions** branch and add the **Oracle 8** extension to the installation. Click **Next**.
9. In the Ready to Install PHP 5.2.7 screen, click **Install**. The installer installs PHP.
10. A dialog is displayed asking whether you want to configure Apache. Click **Yes**.
11. A confirmation message is displayed to confirm that the *httpd.conf* file has been updated. Click **OK**.
12. A confirmation message is displayed to confirm that the *mime.types* file has been updated. Click **OK**.
13. A final confirmation message is displayed to confirm that the PHP installation has completed. Click **Finish**.
14. Restart Apache with **Start > Programs > Apache HTTP Server 2.0.59 > Control Apache Server > Restart**. This opens a console window showing any error messages. Errors may also be recorded in *C:\Program Files\Apache Group\Apache2\logs\error.log*. If you have errors, double check your *httpd.conf* and *php.ini* files, and correct any problems.

Installing OCI8 with Instant Client on Windows

After installing the Apache HTTP Server, you can install Oracle Instant Client and configure PHP to connect to a remote Oracle database.

To install:

1. Download the Instant Client Basic package for Windows from the Instant Client page on the Oracle Technology Network:
<http://www.oracle.com/technology/tech/oci/instantclient/instantclient.html>
The Windows 32 bit ZIP file is called *instantclient-basic-win32-11.1.0.6.0.zip* and is around 42 MB in size.
If you need to connect to Oracle 8i, then install the Oracle 10g Instant Client.
On some Windows installations, to use Oracle 10g Instant Client you may need to locate a copy of *msvcr71.dll* and put it in your [PATH](#).
2. Create a new directory (for example, *C:\instantclient_11_1*). Unzip the downloaded file into the new directory.
3. Edit the Windows environment and add the location of the Oracle Instant Client files, *C:\instantclient_11_1*, 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.

If you are using a *tnsnames.ora* file to define Oracle Net connect names (database aliases), copy the *tnsnames.ora* file to *C:\instantclient_11_1*, and set the user environment variable `TNS_ADMIN` to *C:\instantclient_11_1*.

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, such as `ORACLE_HOME` and `ORACLE_SID`.

4. Install PHP following the steps in the previous section, *Installing OCI8 Using a Local Database on Windows*.
5. Restart the Apache HTTP Server.

Upgrading OCI8 on Windows

The pecl4win site mentioned here is no longer maintained and the steps no longer apply to recent versions of PHP. Current alternatives are to upgrade your complete PHP distribution, or build PHP yourself. The Windows build team in the PHP community is working on a replacement for pecl4win. This section has been retained for reference.

To install PHP 4 on Windows XP Professional Edition:

1. Download the PHP 4.4.9 ZIP file from <http://www.php.net/downloads.php>.
 2. Uncompress *php-4.4.9-Win32.zip* to a directory called *C:\php-4.4.9-Win32*.
 3. Delete the old OCI8 extension file *C:\php-4.4.9-Win32\extensions\php_oci8.dll*.
 4. Download *php_oci8.dll* for the 4.4 branch from http://pecl4win.php.net/ext.php/php_oci8.dll and move it to *C:\php-4.4.9-Win32\extensions\php_oci8.dll*.
-

Note: The OCI8 DLLs for Windows require the Oracle 10g or 11g Client libraries.

5. Copy *C:\php-4.4.9-Win32\php4ts.dll* into the *C:\php-4.4.9-Win32\sapi* directory. If you run the PHP command line interface copy it to the *C:\php-4.4.9-Win32\cli* directory too.
6. Copy *php.ini-recommended* to *C:\Program Files\Apache Group\Apache2\conf\php.ini*.
7. Edit *php.ini* and perform the following:
 - Change `extension_dir` to *C:\php-4.4.9-Win32\extensions*, which is the directory containing *php_oci8.dll* and the other PHP extensions.
 - Uncomment (remove the semicolon from the beginning of the line) the option `extension=php_oci8.dll`.

Installing PHP

- For testing, it is helpful to set `display_errors` to `On`, so you see any problems in your code. Make sure to unset it when your application is released.
- Edit the file `httpd.conf` and add the following lines. Make sure to use forward slashes `/` instead of back slashes `\`:

```
#
# This will load the PHP module into Apache
#
LoadModule php4_module C:/php-4.4.9-Win32/sapi/php4apache2.dll
#
# This section will call PHP for .php, .phtml, and .phps files
#
AddType application/x-httpd-php .php
AddType application/x-httpd-php .phtml
AddType application/x-httpd-php-source .phps
#
# This is the directory containing php.ini
#
PHPIniDir "C:/Program Files/Apache Group/Apache2/conf"
```

- Restart Apache.

Installing OCI8 with Oracle Application Server on Linux

Oracle includes PHP with its mid-tier Application Server 10g Release 3 allowing you to use the same web server for PHP and for J2EE applications.

PHP is enabled by default. The Oracle HTTP Server document root is

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

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

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. If you have a strong need to use a different version of PHP without installing a new web server, you may be able to compile your own PHP release using these steps.

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

The technical problem faced with building PHP is that the Oracle libraries for AS 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 of AS.

A previous installation of AS 10.1.3 is assumed. To install a new version of PHP:

1. Log on as the *oracle* user and change to the home directory:

```
$ cd $HOME
```

2. 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>

3. Extract the ZIP files:

```
$ unzip instantclient-basic-linux32-10.2.0.3-20061115.zip
$ unzip instantclient-sdk-linux32-10.2.0.3-20061115.zip
```

4. 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.

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!!**.

5. Download PHP 5.2.7 from <http://www.php.net/downloads.php> and extract the file:

```
$ tar -jxf php-5.2.7.tar.bz2
```

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

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

7. Shut down the Apache HTTP Server:

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

8. Edit `$ORACLE_HOME/Apache/Apache/conf/httpd.conf` and comment out the PHP 4 *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 will be:

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

9. Back up `$ORACLE_HOME/Apache/Apache/libexec/libphp5.so` 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
```

Installing PHP

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-5.2.7
$ ./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 \
> --enable-sigchild
```

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

12. Make and install PHP

```
$ make
$ 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
```

13. Back up and update *\$ORACLE_HOME/Apache/Apache/conf/php.ini* with options for PHP 5.2.7, for example all the new oci8 directives. Refer to *\$HOME/php-5.2.7/php.ini-recommended* for new options.
14. 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.

Installing PHP With PDO

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

The installation shown uses PHP 5.2.7, Apache 2.0.59, Oracle Enterprise Linux, and Windows XP Professional Edition. This procedure works for all versions of PHP after release 5.1. You can install PDO_OCI and OCI8 at the same time by combining the appropriate options to `configure` or by selecting both extensions during installation on Windows.

Installing PDO on Linux

To install PDO on Oracle Enterprise Linux:

1. Download PHP 5.2.7 from <http://www.php.net/downloads.php>.
2. Log in as the *root* user and extract the PHP source code using the following commands:

```
# tar -jxf php-5.2.7.tar.bz2
# cd php-5.2.7
```

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

3. Configure PHP:

```
# export ORACLE_HOME=/usr/lib/oracle/xe/app/oracle/product/10.2.0/server
# ./configure \
> --with-apxs2=/usr/sbin/apxs \
> --enable-pdo \
> --with-pdo-oci=$ORACLE_HOME \
> --enable-sigchild
```

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,11.1
```

This indicates to use the Instant Client in `/usr/lib/oracle/11.1`. Note the Oracle 11.1.0.6 Instant Client RPMs have 11.1.0.1 in the package name and installation path.

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

```
--with-pdo-oci=instantclient,$HOME/instantclient_11_1,11.1
```

The trailing version number in this command for ZIP installs 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 must be manually created unless the RPM files are used.

4. Make and install PHP.

```
# make
# make install
```

5. Continue from step 6 of the previous Linux section *Installing OCI8 Using a Local Database* to create the

Installing PHP

php.ini script and configure Apache.

Installing PDO on Windows

To install PDO on Windows XP Professional Edition:

1. Install an Oracle Client or Oracle Database. See the section *Installing OCI8 With Oracle Instant Client* for steps on installing Oracle Instant Client.
2. Follow the steps in *Installing OCI8 on Windows*.
3. At step 8, add the PDO extension by also selecting to install **PHP > Extensions > PDO > Oracle10g Client and above**

In PHP 5.2.6 and earlier, the option is called **Oracle 8i +**.

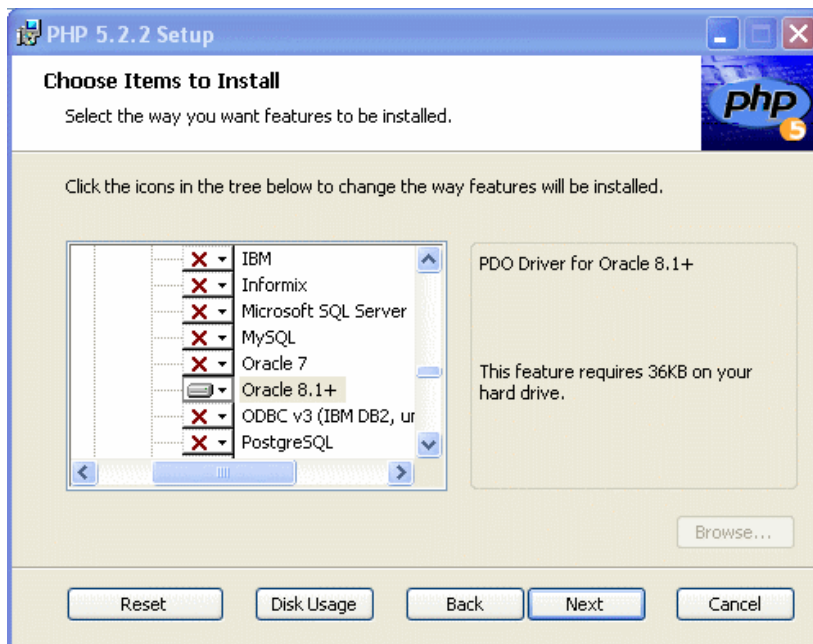


Figure 57: PDO Installation on Windows.

In PHP 5.2.6 and earlier you can select the Oracle 7 option if you have Oracle 8i or 9i client libraries. This choice is not available in PHP 5.2.7 onwards.

4. Continue with the rest of the PHP installation.
5. Restart the Apache HTTP Server.

The Oracle PDO_OCI extension is now installed and configured with PHP.

Checking OCI8 and PDO_OCI Installation

To confirm PHP was installed correctly, create a script that shows the PHP configuration settings. The file should be in a directory Apache can read, such as the document root, specified by the [DocumentRoot](#) setting in the *httpd.conf* file. On Oracle Enterprise Linux the directory is */var/www/html*.

Script 4: *phpinfo.php*

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

Load the script in your browser:

<http://localhost/phpinfo.php>

Alternatively, the script can be run in a terminal window with command-line PHP:

```
$ php phpinfo.php
```

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

The Environment section should show the Oracle environment variables. See *Setting Oracle Environment Variables for Apache* in the chapter *Connecting to Oracle Using OCI8*.

If OCI8 was installed, there will be a section for it. The parameter values are discussed in later chapters:

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	no value	no value
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 58: *phpinfo()* output when OCI8 is enabled.

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

Installing PHP

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

Figure 59: `phpinfo()` when `PDO_OCI` is enabled.

INSTALLING ZEND CORE FOR ORACLE

This Chapter shows you how to install Zend Core for Oracle Release 2.5, a fully tested and supported PHP distribution that includes integration with Oracle Database 10g client libraries. It is a pre-built stack that makes it easier to get started with PHP and Oracle, as all the hard work of installation and configuration has been done for you. Zend Core for Oracle release 2.5 includes PHP 5.2.5, the refactored OCI8 driver, Oracle Instant Client, and an optional Apache HTTP Server 2.2.4.

The collaboration between Oracle and Zend reinforces Oracle's commitment to the open source PHP community.

Zend Core for Oracle is supported by Zend on the following operating systems:

- Oracle Enterprise Linux
- Linux SLES V9 & V10 on x86, x86-64
- Linux RHEL V4 & V5 on x86, x86-64
- Windows XP & 2003 (32 bit), Vista (32 & 64 bit)
- Solaris V8, V9 & V10 on Sparc

The web servers that are supported are:

- Apache 1.3.x
- Apache 2.0.x (compiled in prefork mode only)
- Apache 2.2.x (compiled in prefork mode only)
- IIS 5, 6, 7
- Oracle HTTP Server

Zend Core for Oracle is supported against Oracle Database 11g, 10g and 9i. That means that you can have a fully supported stack of database, web server and PHP.

Installing Zend Core for Oracle

This procedure installs Zend Core for Oracle on Linux and Windows platforms. The procedure to install on other operating systems is similar as the installer is the same on all operating systems. There are some slight differences, like the names and locations of the web server, but the installation is similar on all platforms. So you can use this procedure on all supported operating systems.

Installing Zend Core for Oracle on Linux

To install Zend Core for Oracle on Linux:

1. Download the Zend Core for Oracle file from the Oracle Technology Network

Installing Zend Core for Oracle

(<http://otn.oracle.com/php>).

2. Log in or `su` as `root` if you are not already.

```
$ su
Password:
```

3. Extract the contents of the downloaded Zend Core for Oracle software file. This example uses the Linux 64-bit install. If you are using the 32-bit install, the file name may vary slightly.

```
# tar -zxvf ZendCoreForOracle-2.5.0-linux-glibc23-amd64.tar.gz
```

Files are extracted to a subdirectory called `ZendCoreForOracle-2.5.0-linux-glibc23-amd64`.

4. Change directory to `ZendCoreForOracle-2.5.0-linux-glibc23-amd64` and start the Zend Core for Oracle installation:

```
# cd ZendCoreForOracle-2.5.0-linux-glibc23-amd64
# ./install -g
```

The Zend Core for Oracle Installation screen is displayed.

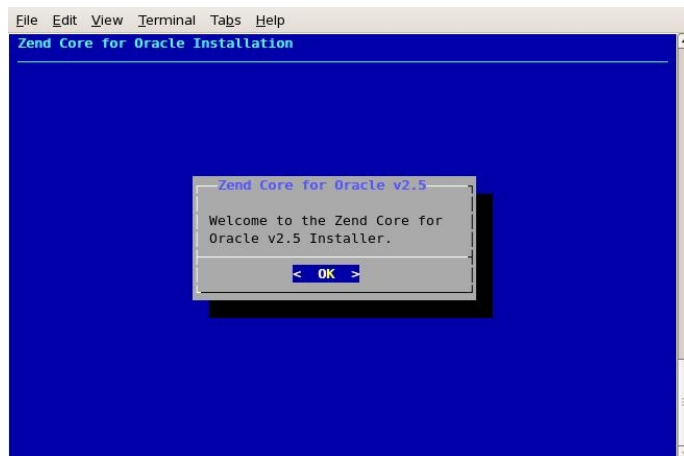


Figure 60: Zend Core for Oracle Installation screen.

5. Click **OK**. The Zend Core for Oracle license agreement is displayed.

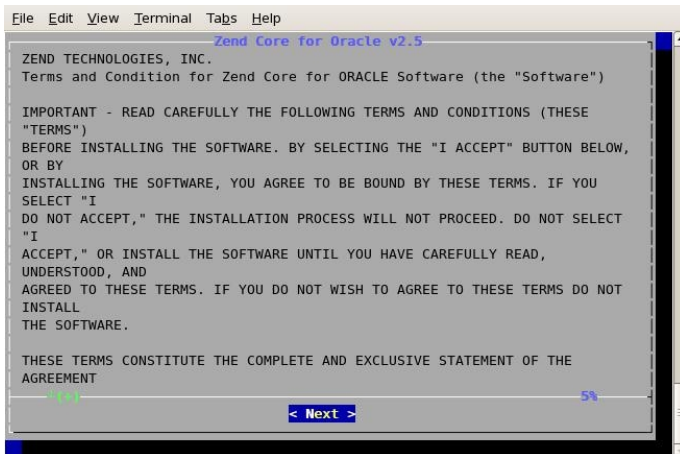


Figure 61: Zend Core for Oracle License Agreement screen.

6. Click **Yes**. The Zend Core for Oracle installation location screen is displayed.

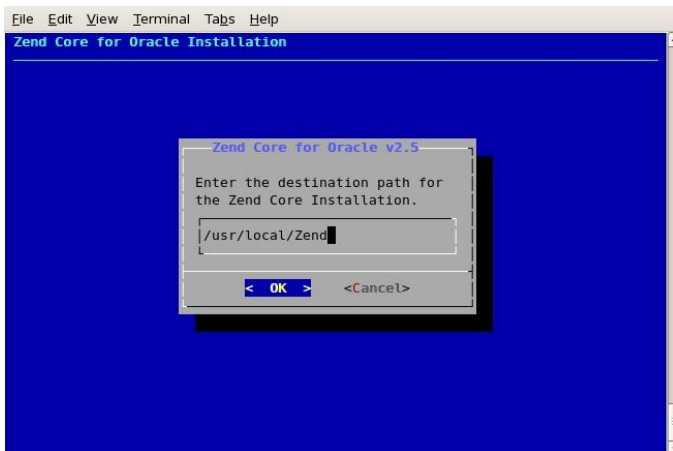


Figure 62: Zend Core for Oracle installation location screen.

7. Specify the location for installing Zend Core for Oracle: accept the default; or enter your preferred location. Click **OK**. The Zend Core for Oracle password screen is displayed.

Installing Zend Core for Oracle

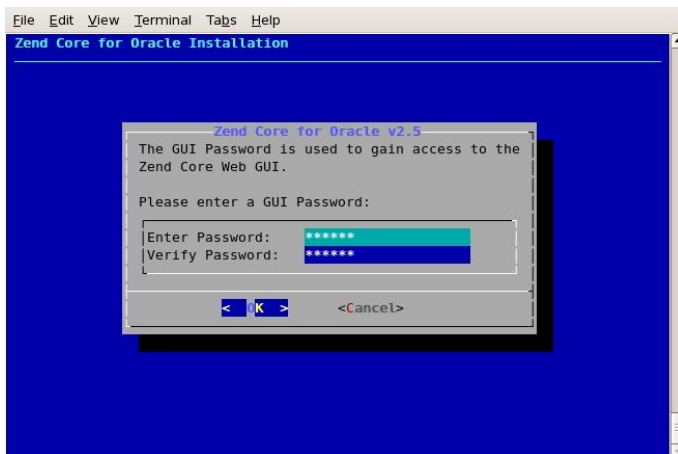


Figure 63: Zend Core for Oracle password screen.

8. Enter a password for the Zend Core for Oracle Administration Console. This is the password for the web-based interface to Zend Core for Oracle. Confirm the password and click **OK**. The Zend Core for Oracle installation options screen is displayed.



Figure 64: Zend Core for Oracle installation options screen.

9. Select the Zend Core for Oracle components you want to install. We suggest you select **Configure Apache Webserver**, and **Install Oracle Client**. You may optionally enter your Zend network user ID and password to be able to use the Zend Core Console to track when updates to Zend Core and PHP components are available. If you have not registered, or do not want to track updates, make sure the **Configure Zend Network** is not selected. Click **OK**. The Zend Core for Oracle select web server screen is displayed.

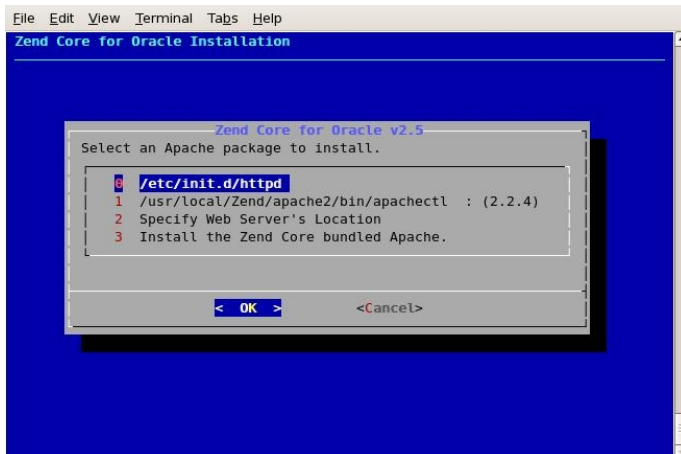


Figure 65: Zend Core for Oracle select web server screen.

10. Zend Core for Oracle can install Apache if you don't have it on your system. If you want to install and configure Apache, select **Install the Zend Core bundled Apache**, or if you want to use an existing web server, select the web server from the list. This installation assumes that you have selected to use an existing Apache server already installed on your machine. Click **OK**. The Zend Core for Oracle virtual host selection screen is displayed.

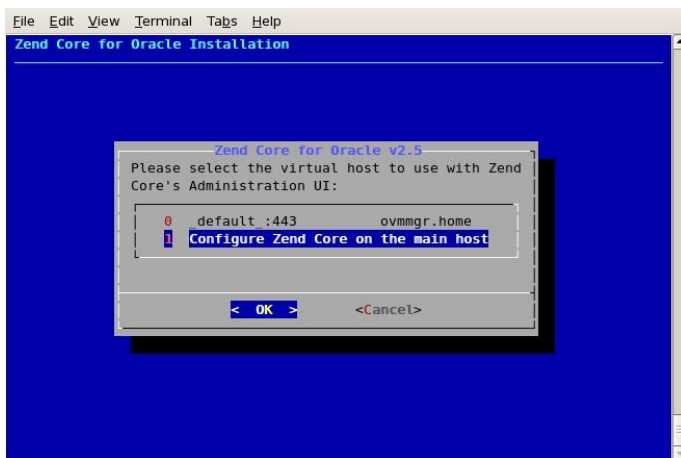


Figure 66: Zend Core for Oracle virtual host selection screen.

11. Select **Configure Zend Core on the main host**, and click **OK**. The Zend Core for Oracle Apache configuration details screen is displayed.

Installing Zend Core for Oracle

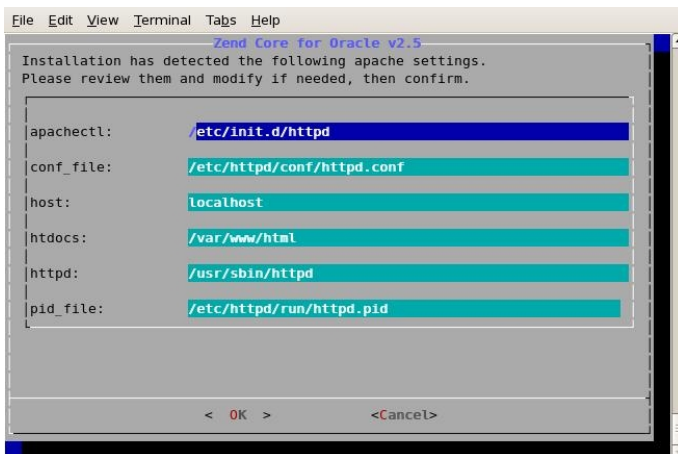


Figure 67: Zend Core for Oracle Apache configuration details screen.

12. Confirm the configuration details for your Apache installation. Click **OK**. The Zend Core for Oracle Apache installation method screen is displayed.

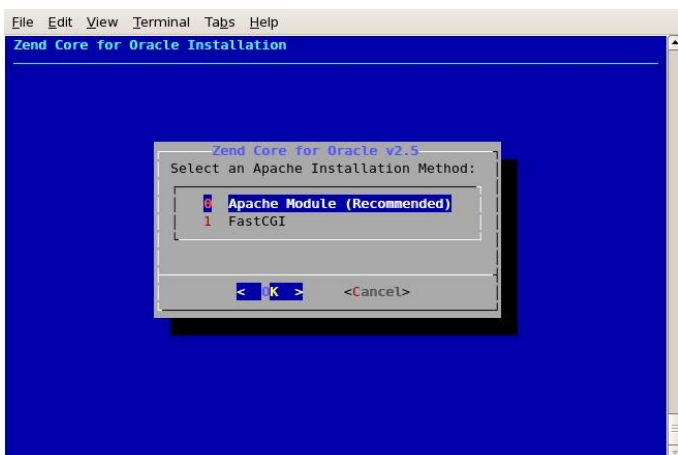


Figure 68: Zend Core for Oracle Apache installation method screen.

13. Select **Apache Module** as the installation method, and click **OK**. You can also use the **FastCGI** option if you prefer. The installer unpacks the installation files and installs Zend Core for Oracle. The Zend Core for Oracle further installation options screen is displayed.

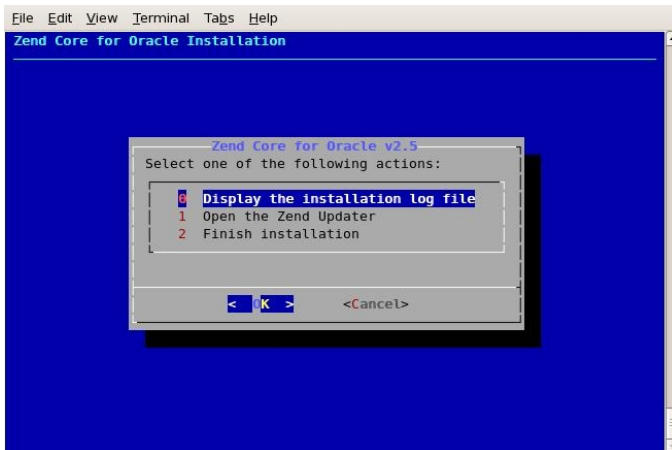


Figure 69: Zend Core for Oracle further installation options screen.

14. Select **Finish installation** and click **OK**. The Zend Core for Oracle installation successful screen is displayed.

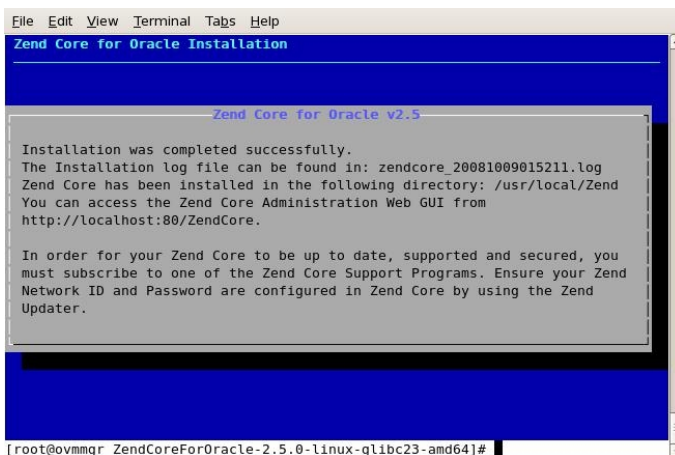


Figure 70: Zend Core for Oracle installation successful screen.

15. This screen contains useful configuration commands and the URL for the Zend Core for Oracle Administration Console. Take note of the information.

The Zend Core for Oracle installation is now complete.

Testing the Zend Core for Oracle Installation on Linux

To test the Zend Core for Oracle installation on Linux platforms:

1. Configure Apache to use a public virtual directory. As *root*, edit *APACHE_HOME/conf/httpd.conf* and add a comment to the following line:

```
#UserDir "disable"
```

Installing Zend Core for Oracle

2. Then remove the `#` from the following line:

```
UserDir public_html
```

3. As your normal user (not *root*), create a directory called *public_html* in your home directory, and change directory to the newly created directory, enter the following commands in a command window:

```
# cd $HOME
# mkdir public_html
# cd public_html
```

4. Create a file called *hello.php* that contains the following PHP code:

```
<?php
echo "Hello world!";
?>
```

5. Make sure the permissions on any files in the *public_html* directory, and the directories above it are set to `755` so the web server can read and execute them.
6. Open a web browser and enter the following URL in your browser:
`http://127.0.0.1/~username/hello.php`
The line `Hello world!` appears in the browser. Any errors in your PHP code are displayed as you have configured PHP to display errors.

Installing Zend Core for Oracle on Windows

To install Zend Core for Oracle on Windows platforms:

1. Download the Zend Core for Oracle. There is a link to the Zend site available on the Oracle Technology Network at <http://otn.oracle.com/php>.
2. Log in as a system administrator, or a user with system administrator privileges.
3. Double click on the file named *ZendCoreforOracle-2.5.0-Windows-x86.exe* to start the Zend Core for Oracle installation. The Zend Core for Oracle Welcome screen is displayed.

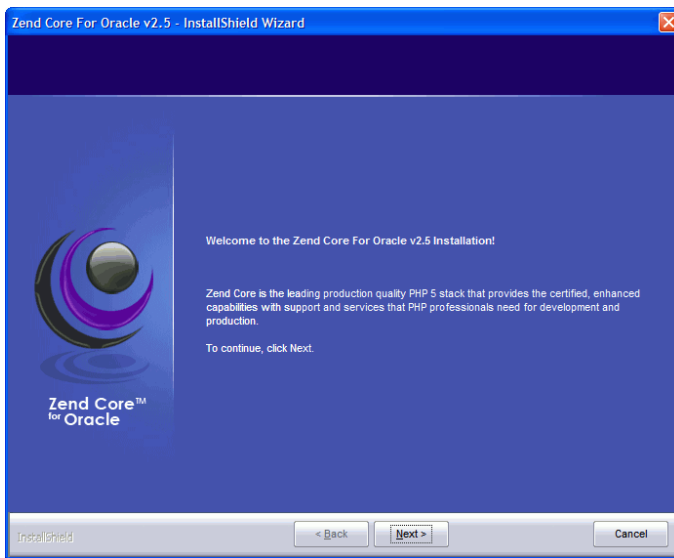


Figure 71: Zend Core for Oracle Welcome screen.

4. In the Zend Core for Oracle Welcome screen, click **Next**. The Zend Core for Oracle License Agreement screen is displayed.

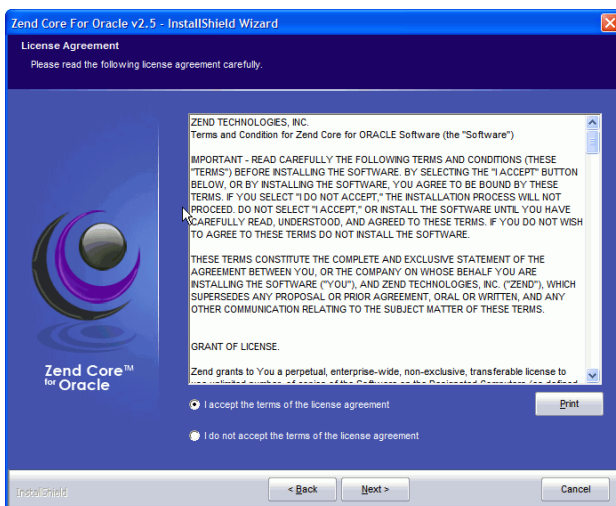


Figure 72: Zend Core for Oracle License Agreement screen.

5. Read and accept the terms of the license, and click **Next**. The Zend Core for Oracle Setup Type screen is displayed.

Installing Zend Core for Oracle

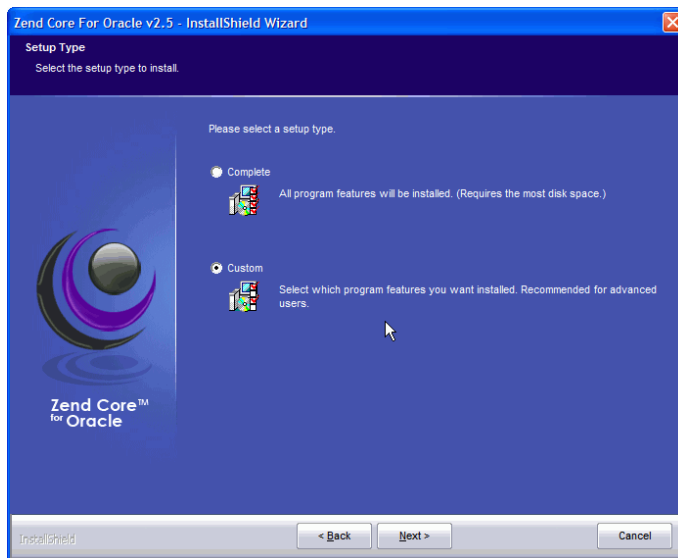


Figure 73: Zend Core for Oracle Setup Type screen.

6. On Windows platforms, you can choose to perform a **Complete** installation, or a **Custom** installation. This installation procedure assumes you select the **Custom** installation. If you prefer, select the **Complete** installation for the default install options. Click **Next**. The Zend Core for Oracle Choose Destination Location screen is displayed.

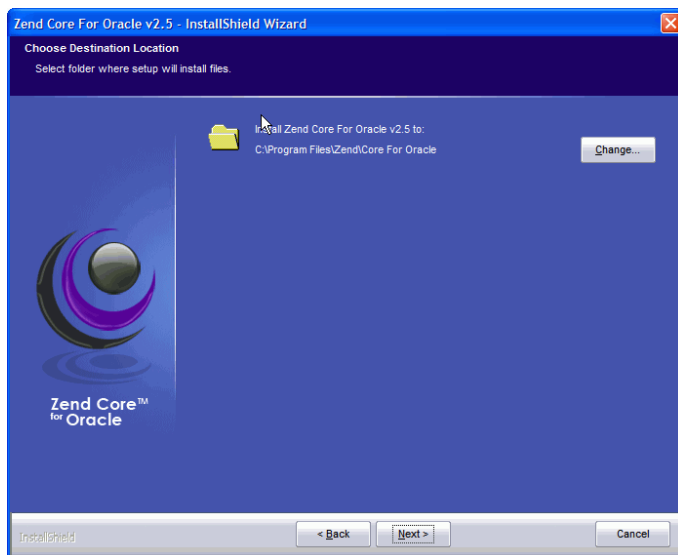


Figure 74: Zend Core for Oracle Choose Destination Location screen.

7. Specify the location for installing Zend Core for Oracle; accept the default; or enter your preferred location. Click **Next**. The Zend Core for Oracle Select Feature screen is displayed.

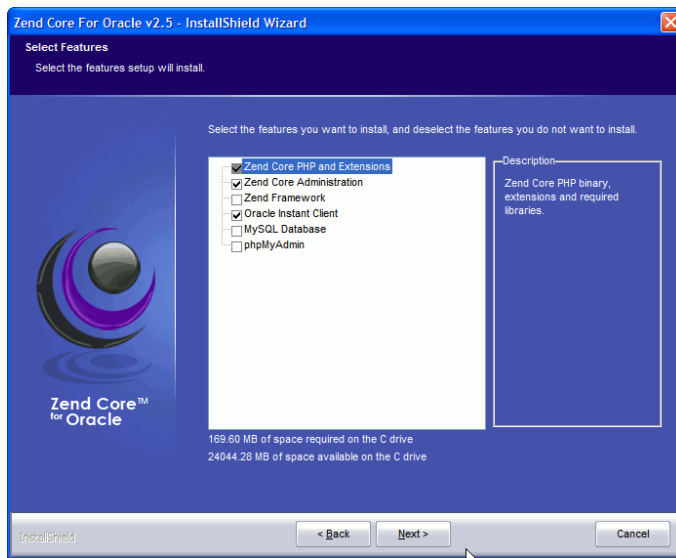


Figure 75: Zend Core for Oracle Select Features screen.

8. Select **Zend Core PHP and Extensions**, **Zend Core Administration**, and optionally **Oracle Instant Client** and **Zend Framework**. Click **Next**. The Zend Core for Oracle Web Server Selection screen is displayed.

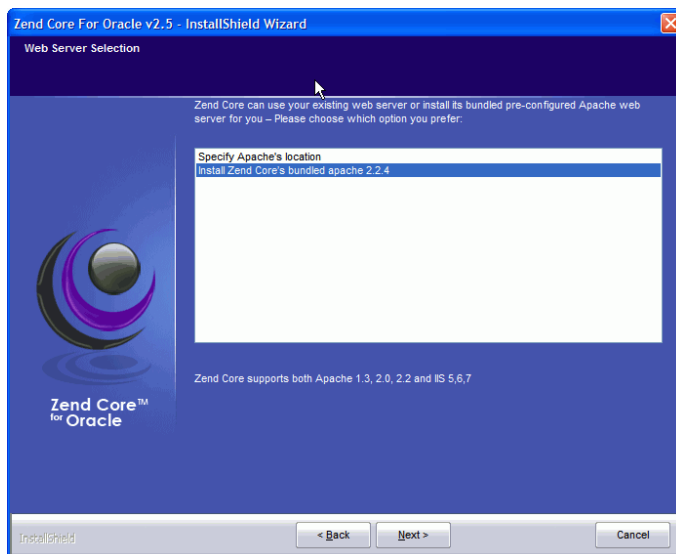


Figure 76: Zend Core for Oracle Web Server Selection screen.

9. If you have an Apache install already, select **Specify Apache's location**, or if you do not have Apache installed, select **Install Zend Core's bundled Apache 2.2.4**. Click **Next**. If you selected to install Zend Core's bundled Apache server, the Zend Core for Oracle Apache port number screen is displayed.

Installing Zend Core for Oracle

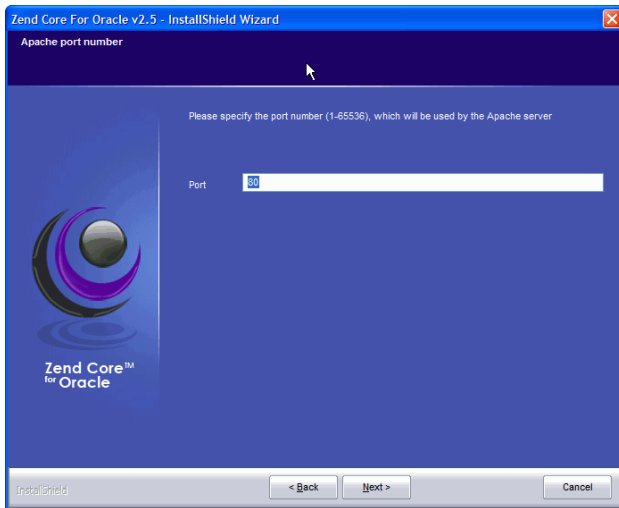


Figure 77: Zend Core for Oracle Apache port number screen.

10. If you have chosen to use an existing Apache install, select the Apache root folder. If you have chosen to install a new Apache server, select the port number you want Apache to listen on. Click **Next**. The Zend Core for Oracle Extension Association screen is displayed.

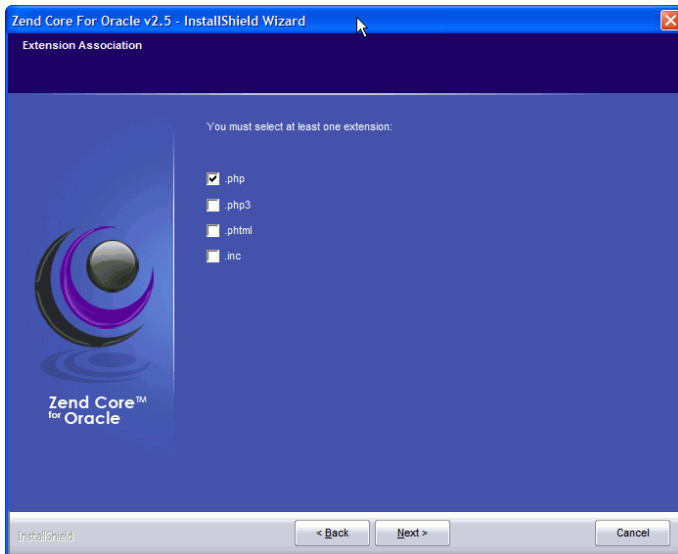


Figure 78: Zend Core for Oracle Extension Association screen.

11. Select the extensions you want to associate with PHP from the check boxes and click **Next**. The Zend Core for Oracle Administration Password screen is displayed.

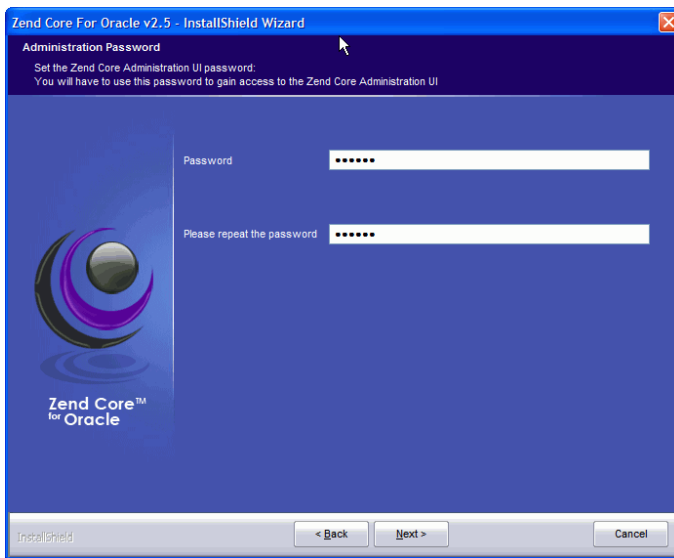


Figure 79: Zend Core for Oracle Administration Password screen.

12. Enter a password for the Administration Console. Confirm the password and click **Next**. The Zend Core for Oracle Zend Network Subscription screen is displayed.

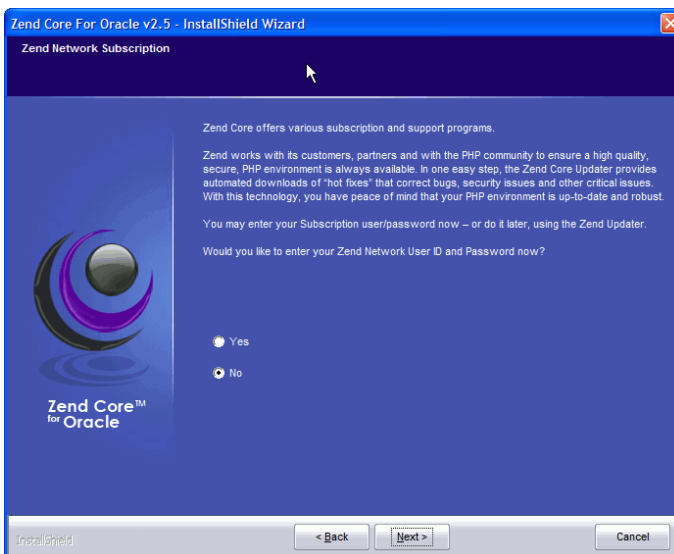


Figure 80: Zend Core for Oracle Zend Network Subscription screen.

13. On the Zend Network Subscription screen you may optionally enter your Zend network user ID and password to be able to use the Zend Core Console to track when updates to Zend Core and PHP components are available. If you have not registered, or do not want to track updates, select **No**. Click **Next**. The Zend Core for Oracle Proxy Configuration screen is displayed.

Installing Zend Core for Oracle

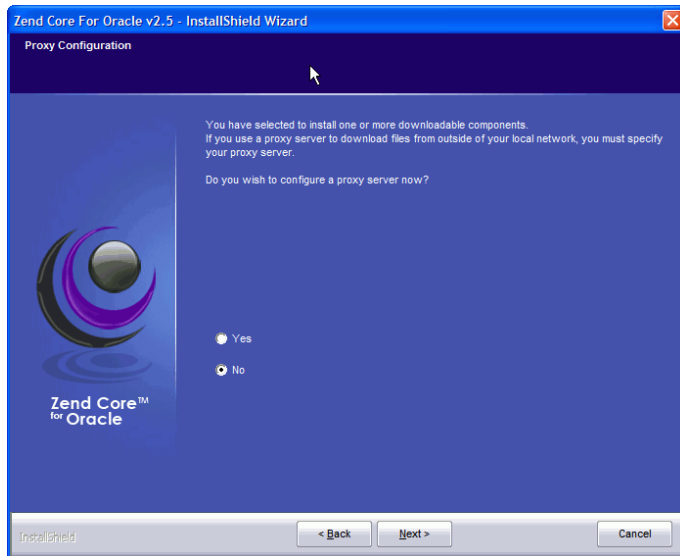


Figure 81: Zend Core for Oracle Proxy Configuration screen.

14. If you use a proxy server to connect to the Internet, select Yes to enter the the proxy server information. Otherwise, select No. Click Next. The Zend Core for Oracle Ready to Install screen is displayed.

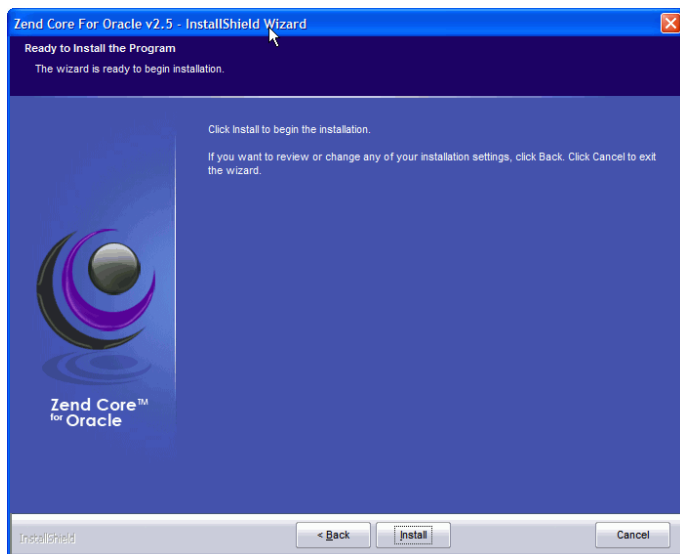


Figure 82: Zend Core for Oracle Ready to Install screen.

15. Click the **Install** button to begin the installation. The installer runs through the installation. The Zend Core for Oracle Installation Complete screen is displayed.

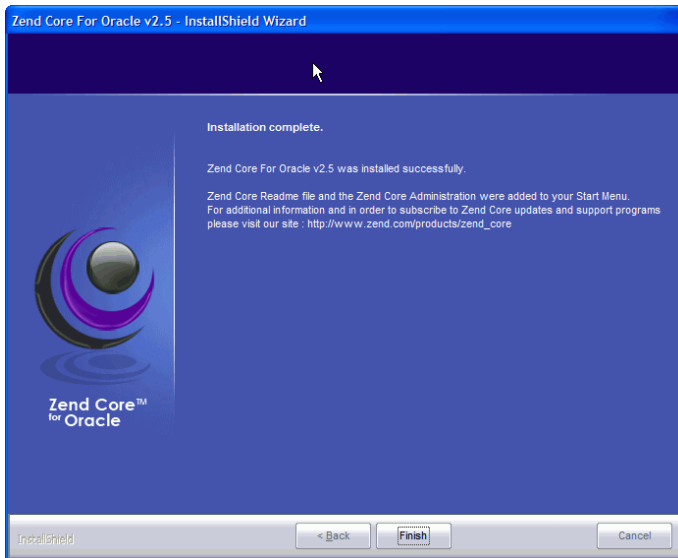


Figure 83: Zend Core for Oracle Installation Complete screen.

16. The final installation page is displayed confirming the installation is complete. Click **Finish**.
Zend Core for Oracle is installed. You may need to restart Windows if you see that some of the PHP extensions aren't loaded correctly.

Testing the Zend Core for Oracle Installation on Windows

To test the Zend Core for Oracle installation on Windows platforms:

1. Create a file called *hello.php* in the Apache *C:\Program Files\Apache Group\Apache2\htdocs* directory that contains the following PHP code:

```
<?php
echo "Hello world!";
?>
```

2. Open a web browser and enter the following URL in your browser:

`http://127.0.0.1/hello.php`

The line `Hello world!` appears in the browser. Any errors in your PHP code are displayed if you have configured PHP to display errors.

Configuring Zend Core for Oracle

In this section, you configure environment variables and Zend Core directives that control default error reporting in web pages.

1. Enter the following URL in a web browser to access the Zend Core for Oracle Administration console:

`http://127.0.0.1/ZendCore`

Installing Zend Core for Oracle



Figure 84: Zend Core for Oracle login screen.

2. Enter the GUI password that you provided during Zend Core for Oracle installation. Click the **login** >>> icon.
3. Click the **Configuration** tab to display the configuration options.
4. Click the + icon to expand the Error Handling and Logging configuration entry.
5. Set the **display_errors** directive to **On** to enable the display of errors in the HTML script output during development. Make sure you set this back to **Off** before you release any applications you build as users will see the errors and gain information about your system that you otherwise don't want them to see or know.

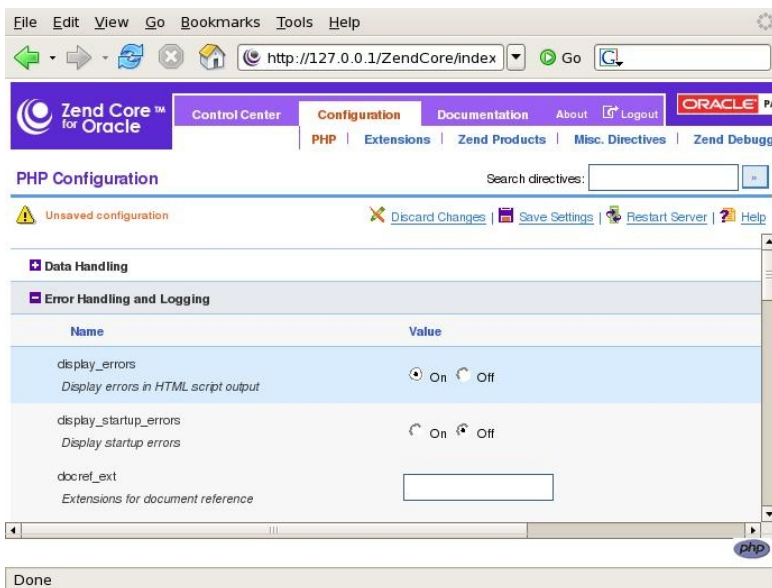


Figure 85: Zend Core for Oracle configuration screen.

6. Because there are unsaved changes, the "Unsaved configuration" message appears under the page header. Click **Save Settings** to save the configuration change.

7. Because you have made configuration changes, you must restart the Apache web server. Under the page header notice the "Please Restart Apache" message reminding you to do so. Click **Restart Server** to restart the Apache server. If you are using a Windows operating system, you should restart the Apache server using the Services dialog in Control Panel, or the Apache Monitor in the system tray.
8. Click **Logout** to exit the Zend Core for Oracle Administration page.

CONNECTING TO ORACLE USING OCI8

This Chapter covers connecting to an Oracle database from your PHP application, including the types of Oracle connections, environment variables that may affect connections, and tuning your connections. A later chapter *PHP Scalability and High Availability* discusses connection pooling and how it applies to connection management.

Before attempting to connect, review the section *Setting Oracle Environment Variables for Apache* part way through this chapter and make sure that your environment is configured appropriately.

The examples use the HR schema, a demonstration user account installed with the database. Use Application Express or SQL*Plus to unlock the account and set a password, as described in the chapter *Using Oracle Database*.

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.

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 username and database name, then you get a pointer to the original connection.

Multiple 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. This can be 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);
```

Connecting to Oracle Using OCI8

Persistent connections are not automatically closed at the end of a PHP script and remain open in PHP's persistent connection cache for reuse in other 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 username, the database name, the character set and the connection privilege to ensure reconnections are the same as the original.

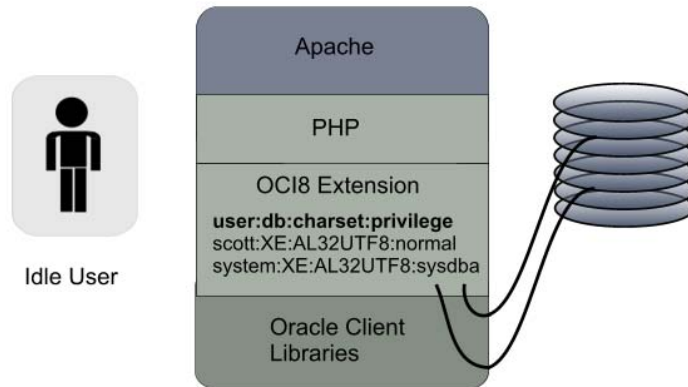


Figure 86: Persistent connections are cached in PHP and held open to the database.

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.

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*, and the PHP-enabled web server is on the same machine, you could connect to the HR schema with:

```
$c = oci_connect('hr', 'hrpwd', 'mymachine/XE');
```

In this guide, we assume the database is on the same machine as Apache and PHP so we use [localhost](#):

```
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');
```

Depending on your network configuration, you may need to use the equivalent IP address:

```
$c = oci_connect('hr', 'hrpwd', '127.0.0.1/XE');
```

The Easy Connect string is JDBC-like. The Oracle 10g syntax is:

```
[//]host_name[:port][//service_name]
```

If PHP links with Oracle 11g 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 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 on 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 almost always need to specify the connection identifier as at least [host_name/service_name](#).

The `lsnrctl` command on the database server shows the service names that the Oracle Net listener accepts requests for.

```
$ lsnrctl services

LSNRCTL for Linux: Version 10.2.0.1.0 - Production on 01-OCT-2008 18:17:10

Copyright (c) 1991, 2005, 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:55 refused:0 state:ready
        LOCAL SERVER
. . .
```

Connecting to Oracle Using OCI8

This shows the service [XE](#) is available.

You can use the Easy Connect syntax to connect to Oracle8i, Oracle9i, Oracle10g, and Oracle11g databases as long as PHP is linked with Oracle 10g or greater libraries. This syntax is usable in Zend Core for Oracle.

More information on the syntax can be found in the *Oracle® Database Net Services Administrator's Guide 11g Release 1 (11.1)*.

Database Connect Descriptor String

The full Oracle Net *connect descriptor* string gives total flexibility over the connection.

```
$db = '(DESCRIPTION =
    (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 database and Oracle Net features used. For example, by using the full syntax, 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 file called *tnsnames.ora* and refer to it in PHP using a *connect name*:

```
# tnsnames.ora
MYA = (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* [MYA](#) to connect to the database:

```
$c = oci_connect($username, $password, 'MYA');
```

PHP needs to be able to find the *tnsnames.ora* file to resolve the [MYA](#) 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 `$ORACLE_HOME/network/admin/tnsnames.ora` will then automatically be found. In Oracle Database XE, [\\$ORACLE_HOME](#) is:

```
/usr/lib/oracle/xe/app/oracle/product/10.2.0/server
```

If PHP was built with Oracle Instant Client, or Zend Core for Oracle is used, 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.

Common Connection Errors

If Zend Core for Oracle is not used, some environment variables need to be set before starting the web server. You may also optionally want to set the environment to control Oracle's globalization settings.

Note: Do not set Oracle environment variables in PHP scripts with `putenv()` because Oracle libraries may be loaded and initialized before the scripts run.

The OCI8 extension always needs to find Oracle libraries, error message data and optionally needs the `tnsnames.ora` file. Not finding the libraries can lead to Apache startup errors (see the Apache error log file) about OCI8 not being initialized, and to script runtime errors like:

```
PHP Fatal error: Call to undefined function oci_connect()
```

This error means that the OCI8 extension is not available. Check that `ORACLE_HOME` and/or `LD_LIBRARY_PATH` on Linux, or `PATH` on Windows are valid.

Another potential source of problems 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
```

This environment initialization problem is common on Windows when multiple Oracle environments are installed and PHP is not using the correct one. (Some users move the Instant Client DLLs to the Apache or PHP directory as a quick solution).

If you are using an Oracle Database 10g Release 2 database other than the Express Edition (Oracle Database XE), you may need to give the Apache process access to Oracle's libraries and globalization data. Refer to the `$ORACLE_HOME/install/changePerm.sh` script in later Oracle patchsets.

If an error like this occurs:

Connecting to Oracle Using OCI8

```
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
```

indicating 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 XE is currently running on your computer and you try to connect to `localhost/abc` you will get this error.

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 such as `$ORACLE_HOME` must be set in the shell that starts Apache.

Setting Oracle Environment Variables for Apache

When you use OCI8, you must set some Oracle environment variables before starting the web server.

If you have environment related problems – unexpected connection errors like those above are typical – then check the output from the PHP `phpinfo()` function:

Script 5: `phpinfo.php`

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

Look at the *Environment* section (not the *Apache Environment* section) and make sure the Oracle variables are set to the values you expect.

The variables needed are determined by how PHP is installed, how you connect, and what optional settings are desired.

Table 5: Common Oracle environment variables on Linux.

Oracle Environment Variable	Purpose
<code>ORACLE_HOME</code>	The directory containing the Oracle software. This directory must be accessible by the Apache process. The variable is not needed if PHP uses Oracle Instant Client or if Zend Core for Oracle is installed.

Oracle Environment Variable	Purpose
ORACLE_SID	The Oracle Net connect name of the database. Only used when PHP is on same machine as the database and the connection identifier is not specified in the PHP connect function. Not used for Oracle Instant Client or Zend Core for Oracle. Not commonly set for PHP.
LD_LIBRARY_PATH	Set this to include the Oracle libraries, for example <code>\$ORACLE_HOME/lib</code> or <code>\$HOME/instantclient_11_1</code> . Not needed if the libraries are located by an alternative method, such as with the <code>/etc/ld.so.conf</code> linker path file. Zend Core for Oracle sets this variable if necessary.
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.
TNS_ADMIN	The location of the <code>tnsnames.ora</code> and other Oracle Net configuration files. Only needed if a database connect name from a <code>tnsnames.ora</code> file is used in the OCI8 connect functions and the <code>tnsnames.ora</code> file is not in <code>\$ORACLE_HOME/network/admin</code> . Not needed if Easy Connect syntax is being used in the connect functions (unless a <code>sqlnet.ora</code> file is used).

With Oracle Database XE, you can set the shell's environment by using the `oracle_env.sh` script:

```
# . /usr/lib/oracle/xep/app/oracle/product/10.2.0/server/bin/oracle_env.sh
```

Note the space after the period. This command allows the script to set the environment of the shell itself. 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:

```
# . /usr/local/bin/oraenv
ORACLE_SID = [] ? orcl
```

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

To simplify things, you may create a script to set the environment and start Apache, for example:

Script 6: start_apache

```
#!/bin/sh

ORACLE_HOME=/usr/lib/oracle/xep/app/oracle/product/10.2.0/server
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
```

Connecting to Oracle Using OCI8

```
/usr/sbin/apachectl start
```

On Oracle Enterprise Linux, instead of creating a shell script to call *apachectl*, you can add environment variables to the end of */etc/sysconfig/httpd*:

```
...
export ORACLE_HOME=/usr/lib/oracle/xe/app/oracle/product/10.2.0/server
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ORACLE_HOME/lib
export NLS_LANG=AMERICAN_AMERICA.WE8MSWIN1252
```

Some Apache 2 installations use an “envvars” script in the Apache bin directory to set variables. Using Apache's [SetEnv](#) directive is sometimes insufficient. The important thing is to set the environment before Apache loads any Oracle library.

If Apache is started automatically when your machines boots, you will need to make sure Apache has the environment set at boot time.

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. 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.

Zend Core for Oracle is automatically configured to use Oracle Instant Client. As part of the install, it modifies */usr/local/Zend/apache2/bin/envvars* to add [LD_LIBRARY_PATH](#). The *envvars* file is called by *apachectl*. Make sure you preserve this file inclusion if you later modify the *apachectl* script. The Oracle environment variables you may want to set explicitly for Zend Core for Oracle are [NLS_LANG](#) and [TNS_ADMIN](#).

If PHP was built with Oracle Instant Client, it can be convenient to create a file */etc/ld.so.conf.d/instantclient.conf* containing the path to the Instant Client libraries:

```
/usr/lib/oracle/11.1.0.1/client/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. This removes the need to set [LD_LIBRARY_PATH](#) everywhere, but upgrading requires remembering to change the path.

If you have multiple versions of Oracle installed, you might be able to use [LD_PRELOAD](#) or equivalent to force Apache to load the desired Oracle *libclntsh.so* file. PHP should be run using the same version of the Oracle libraries as were used to build the PHP library and executables.

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 them by calling:

```
oci_close($c);
```

Any uncommitted data is rolled back. The function has no effect on persistent connections. (See the chapter on connection pooling 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. 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
```

Close Statement Resources Before Closing Connections

The `oci_close()` function works 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 7: close.php

```
<?php
$c = oci_connect("hr", "hrpwd", "localhost/XE");
$s = oci_parse($c, "select * from locations");
oci_execute($s);
oci_fetch_all($s, $res);

// oci_free_statement($s);
oci_close($c);

echo "Sleeping . . .";
sleep(10);
echo "Done";

?>
```

While *close.php* is sleeping, if you query 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`.

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.

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 go out of scope or are assigned new values. A common idiom is to assign null to the statement resource:

```
$s = null;
```

Connecting to Oracle Using OCI8

In the next example, `$c1` and `$c2` are the one database connection (because `oci_connect()` returns the same connection resource when called more than once in a script). 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 8: *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', 'hrpwd', 'localhost/XE');
$c2 = oci_connect('hr', 'hrpwd', '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. 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 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:


```
$c = oci_pconnect("hr", "hrpwd", "localhost/XE");
do_query($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 system default format. The second is different because the `ALTER SESSION` command changed the date 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`:

```
$c = oci_connect("hr", "hrpwd", "localhost/XE", 'AL32UTF8');
```

When not specified or `NULL`, the `NLS_LANG` environment variable setting is used. This setting 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.

Oracle Database XE is available in two distributions, one with a database character set of `WE8MSWIN1252` and the other of `AL32UTF8` (Oracle's name for UTF-8).

Connecting to Oracle Using OCI8

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.

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 user that is running the web server system process. An operating system authenticated privileged connection 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 run as a member of the OS *dba* group
- PHP is linked with the `ORACLE_HOME` software (that is, not Oracle Instant Client, or Zend Core for Oracle) that the database is using
- The database is your default local database, for example, specified by the `ORACLE_SID` environment variable

This would be typically be done by compiling and running PHP with the Oracle libraries used by the database.

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. 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

With Zend Core for Oracle and any other PHP based on Oracle Instant Client, a username and password must be given when connecting. 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, and 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 with the SQL*Plus and enter a command similar to the following command:

```
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
```

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 cl identified by clpw;
```

Connecting to Oracle Using OCI8

```
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`.

External Authentication

OCI8 1.3 supports Oracle External Authentication. Instead of storing a username and password in PHP scripts and authenticating against a username and password stored in the database, credentials can be authenticated by an outside system such as Oracle Wallet. The operating system user running the Apache process could be granted read access to the wallet using Access Control Lists.

To use external authentication, first configure the database to use an external authentication method. Refer to Oracle documentation for details.

In OCI8, pass the flag `OCI_CRED_EXT` as the `session_mode` parameter to `oci_connect()`, `oci_new_connect()` or `oci_pconnect()`:

```
$c = oci_connect("/", "", $db, null, OCI_CRED_EXT);
```

`OCI_CRED_EXT` can only be used with username of `"/"` and a empty password. The `php.ini` parameter `oci8.privileged_connection` may be `On` or `Off`.

The flag may be combined with the existing `OCI_SYSOPER` or `OCI_SYSDBA` modes. For example:

```
$c = oci_connect("/", "", $db, null, OCI_CRED_EXT+OCI_SYSOPER);
```

Note: `oci8.privileged_connection` needs to be `On` for `OCI_SYSDBA` and `OCI_SYSOPER` use.

The external authentication feature is not available in OCI8 on Windows for security reasons.

Changing the Database Password

The OCI8 extension allows Oracle database passwords to be changed.

Changing Passwords On Demand

After connecting, a password can be changed with `oci_password_change()`:

```
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');
oci_password_change($c, 'hr', 'hrpwd', 'newhrpwd');
```

Subsequent scripts may now connect using:

```
$c = oci_connect('hr', 'newhrpwd', '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 to expire passwords at a certain time by using `CREATE PROFILE`, or may have expired a password immediately with `ALTER USER` forcing the user to choose a new password the very first 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 9: connectexpired.sql

```
drop user peregrine cascade;
create user peregrine identified by abc;
grant create session to peregrine;
alter user peregrine password expire;
```

Script 10: 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 {
        $m = oci_error();
        if ($m['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: ", $m["message"], "<br>\n";
            exit;
        }
    }
}
```

Connecting to Oracle Using OCI8

```
    return($c);
}

function change_and_connect($un, $pw1, $pw2, $db)
{
    echo "Calling oci_password_change() to connect<br>\n";
    $c = oci_password_change($db, $un, $pw1, $pw2);
    if (!$c) {
        $m = oci_error();
        echo "Error: ", $m["message"], "<br>\n";
    }
    else {
        echo "Connected and changed password to $pw2<br>\n";
    }
    return($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/systempwd@localhost/XE @connectexpired.sql
```

When `oci_connect()` in `do_connect()` fails with an ORA-28001 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, $pw1, $pw2);` differs from the example in the previous section *Changing Passwords On Demand* in that it passes a database connection identifier identifier, *localhost/XE*, as the first parameter instead of passing the connection resource of an already opened connection. This new usage connects to the database and changes the password to `$pw2` 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.

Tuning Oracle Connections in PHP

Connections to Oracle can be tuned by changing the way OCI8 calls are made, by changing the network configuration, or by tuning the database.

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. However if currently unused, open persistent connections consume too much memory on the database server, consider tuning the timeout parameters or using connection pooling.

Pass the Character Set

Explicitly passing the client character set name as the fourth parameter to the connection functions improves performance:

```
$c = oci_connect("hr", "hrpwd", "localhost/XE", 'WE8DEC');
```

If you do not enter a character set, PHP has to determine a client character set to use. This may involve a potentially expensive environment lookup. Use the appropriate character set for your requirements.

Do Not Set the Date Format Unnecessarily

You can often remove `ALTER SESSION` statements used after connecting. For example, if your connection routine 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 HH24:MI:SS'");
    oci_execute($s);
    return $c;
}
```

One way to optimize this is to simply 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.

Sometimes different database users should have different session values so setting `NLS_DATE_FORMAT` globally is not possible. When `oci_connect()` is called multiple times in the one script or when persistent connections are used, the `ALTER SESSION` can be moved to a logon trigger. This is because session settings are retained in cached connections. Using a trigger means the date format is only set when the physical database connection is created the first time. The trigger does not fire when subsequent connect calls return a cached connection.

A logon trigger can be created using SQL*Plus by connecting as a privileged database user:

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

Connecting to Oracle Using OCI8

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 HH24:MI:SS'' ';
    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.

In PHP, the connection function can simply become:

```
function my_connect($un, $pw, $db)
{
    $c = oci_pconnect($un, $pw, $db);
    return $c;
}
```

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.

```
$c = my_connect('hr', 'hrpwd', '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";
```

This connects as *hr* and queries the current date. It shows the new format set by the trigger:

```
2007-05-04 13:50:35
```

If the connection is any user other than *hr* the standard default date format will be displayed:

```
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.

If you cannot use a trigger because each PHP invocation needs different settings, and you need more than one session value changed, 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 `ALTER SESSION` statements.

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.

Managing 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 extent with *php.ini* directives and Apache configuration settings.

The chapter *PHP Scalability and High Availability* discusses connection pooling in Oracle Database 11g which provides an advanced solution to connection management.

Maximum Number of Persistent Connections Allowed

```
oci8.max_persistent
```

This parameter limits the number of persistent connections cached by each Apache-PHP process. It is not a system-wide restriction on database usage. When the limit is reached by a PHP process, all `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.

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. 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 connection pooling 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.

Connecting to Oracle Using OCI8

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`, `oci_pconnect()` 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 also tune Apache to kill idle processes, which will also free up Oracle resources used by persistent connections. Table 6 lists the Apache configuration parameters that can be used to tune PHP.

Table 6: Apache 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.

John Coggeshall’s article *Improving Performance Through Persistent Connections* discusses Apache configuration in more depth.

Reducing Database Server Memory Used By Persistent Connections

There are several techniques that can be used if database server memory is limited but persistent connections are required for performance.

- Use Oracle Database 11g connection pooling. See the chapter on *PHP Scalability and High Availability*.
- Set `oci8.persistent_timeout`.
- Expire Apache processes with Apache configuration parameters.
- Reduce the number of database user credentials used by the application.

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, 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:

```
select * from olduser.mytable;
```

Or, if the application is large, the first statement executed can set the default schema. Subsequent queries will return:

```
alter session set current_schema = olduser;
select * from mytable;
```

Setting the default schema this way requires an extra database operation but, depending on the application, it may be bundled in a PL/SQL block that does other operations.

Oracle Net and PHP

Oracle Net has sophisticated control over network connection management for basic connectivity, performance and features such as encryption of network traffic between PHP and the database. 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))
```

Connecting to Oracle Using OCI8

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 network problem. This lets PHP 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", "hrpwd", "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/hrpwd@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.

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. This is configured by `SQLNET.EXPIRE_TIME` in the database `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.

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 tips are mentioned below.

The best session data unit 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` 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 `sqlnet.ora` in `$ORACLE_HOME/network/admin/sample` has some notes on the parameters that help. For example, with a `USER` level trace in `sqlnet.ora`:

```
trace_level_client = USER
trace_directory_client = /tmp
```

And the PHP code:

```
$c = oci_connect("hr", "hrpwd", "#c"); // invalid db name
```

The trace file, for example `/tmp/cli_3232.trc`, shows:

```
...
[10-MAY-2006 09:54:58:100] nnftmlf_make_system_addrfile: system names file is ...
[10-MAY-2006 09:55:00:854] snlinGetAddrInfo: Name resolution failed for #c
...
```

Connecting to Oracle Using OCI8

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 tracefiles 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.

Connection Management in Scalable Systems

Oracle achieved its well-known scalability in part through a multi-threaded architecture. PHP instead has a multi-process architecture. This difference means care is required when designing scalable applications.

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. Large sites should benchmark Oracle Shared Servers also known as “Multi Threaded Servers” (MTS), and Oracle 11g connection pooling. See the chapter *PHP Scalability and High Availability*.

Make sure that you understand the lifetime of your applications connections. Reuse connections where possible, but do not be afraid to create new connections. Close connections when no longer needed. Each connection will take some Oracle memory, so overall load can be reduced if idle connections are closed with Apache process timeouts or with the *php.ini* parameters to expire persistent connections.

For sites with hundreds of connections a second, tune the cache size of an 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”). For both RAC and non-RAC database, the `DBMS_SERVICE` package lets you specify workload management goals. This is a detailed topic; refer to Oracle’s manuals for more information.

Finally, make sure that your applications are as efficient as possible. This minimizes the length of time connections are held.

EXECUTING SQL STATEMENTS WITH OCI8

This Chapter discusses using SQL statements with the PHP OCI8 extension, including how statements are executed, the functions available, 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. Parsing is really just a preparatory step, since Oracle's actual text parse can occur at the execution stage. You can optionally *bind* local values into a statement similar to the way you use `%s` print format specifications in strings. This improves performance and security. You can also *define* where you want the results to be stored, but almost all scripts let the OCI8 fetch functions take care of this.

The possible steps are:

1. **Parse:** Prepares a statement for execution.
2. **Bind:** Optionally lets you bind data values, for example, in the `WHERE` clause, for better performance and security.
3. **Define:** Optional step allowing you to specify which PHP variables will hold the results. This is not commonly used.
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, PHP will terminate if a script takes longer than 30 seconds. If your SQL statements take longer, change the `php.ini` parameter `max_execution_time` or use the `set_time_limit()` function. You may also need to alter the configuration of your web server.

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 12: query.php

```
<?php
```

Executing SQL Statements With OCI8

```
$c = oci_connect("hr", "hrpwd", "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>';

oci_free_statement($s);

?>
```

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	YSW 9T2
Beijing	190518
Bombay	490231
Sydney	2901

Figure 87: Output from *query.php*.

In PHP, single and double quotes are used for strings. Strings with embedded quotes can be made by escaping the nested quotes with a backslash, or by using both quoting styles. The next example shows single quotes around the city name. To make the query a PHP string it, therefore, must be enclosed in double quotes:

```
$s = oci_parse($c, "select * from locations where city = 'Sydney'");
```

PHP 5.3 introduces 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);
. . .
```

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, most code snippets in this book do not follow this practice.

Oracle Datatypes

Each column has a datatype, which is associated with a specific storage format. The common built-in Oracle datatypes are:

- CHAR
- VARCHAR2
- NUMBER
- DATE
- TIMESTAMP
- INTERVAL
- BLOB
- CLOB
- BFILE
- XMLType

The CHAR, VARCHAR2, NUMBER, DATE, TIMESTAMP and INTERVAL datatypes are stored directly in PHP variables. BLOB, CLOB, and BFILE datatypes 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.

Fetch Functions

There are a number of OCI8 fetch functions, all documented in the *PHP Oracle OCI8 Manual*. Table 7 lists the functions.

Executing SQL Statements With OCI8

Table 7: OCI8 fetch functions.

OCI8 Function	Purpose
<code>oci_fetch_all()</code>	Gets all the results at once.
<code>oci_fetch_array()</code>	Gets the next row as an array of your choice.
<code>oci_fetch_assoc()</code>	Gets the next row as an associative array.
<code>oci_fetch_object()</code>	Gets a new row as an object.
<code>oci_fetch_row()</code>	Gets the next row as an integer indexed array.
<code>oci_fetch()</code>	Used with <code>oci_result()</code> , which returns the result of a given field or 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 8 lists the available modes.

Table 8: `oci_fetch_array()` options.

Parameter	Purpose
<code>OCI_ASSOC</code>	Return results as an associative array.
<code>OCI_NUM</code>	Return results as a numerically indexed array.
<code>OCI_BOTH</code>	Return results as both associative and numeric arrays. This is the default.
<code>OCI_RETURN_NULLS</code>	Return PHP NULL value for NULL data.
<code>OCI_RETURN_LOBS</code>	Return the actual LOB data instead of an OCI- LOB resource.

Modes can be used together by adding them:

```
$rowarray = oci_fetch_array($s, OCI_NUM + OCI_RETURN_NULLS);
```

The `oci_fetch_assoc()` and `oci_fetch_row()` functions are special cases of `oci_fetch_array()`.

Fetching as a Numeric Array

A basic example to fetch results in a numerically indexed PHP array is:

```
$s = oci_parse($c, "select city, postal_code from locations");
oci_execute($s);
while ($res = oci_fetch_array($s, OCI_NUM)) {
    echo $res[0] . " - " . $res[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:

```
$res = oci_fetch_array($s, OCI_NUM+OCI_RETURN_NULLS);
```

Fetching as an Associative Array

Associative arrays are keyed by the uppercase column name.

```
$s = oci_parse($c, "select postal_code from locations");
oci_execute($s);
while ($res = oci_fetch_array($s, OCI_ASSOC)) {
    echo $res["POSTAL_CODE"] . "<br>\n";
}
```

This displays:

```
00989
10934
1689
6823
26192
. . .
```

In an associative array there is no table prefix for the column name. If you 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 the `region_id` column twice:

```
$s = oci_parse($c, "select region_name,
                      regions.region_id as myreg,
                      country_name,
                      countries.region_id
                    from countries
                    inner join regions
                    on countries.region_id = regions.region_id");

oci_execute($s);

while ($res = oci_fetch_array($s, OCI_ASSOC)) {
    echo $res["REGION_NAME"] . " " . $res["MYREG"] . " - " .
```

Executing SQL Statements With OCI8

```
$res["COUNTRY_NAME"] . " " . $res["REGION_ID"] . " " .  
"<br>\n";  
}
```

The query column alias `MYREG` is used as the index to the result array for one of the `region_id` columns. The script output is:

```
Americas 2 - Argentina 2  
Asia 3 - Australia 3  
Europe 1 - Belgium 1  
Americas 2 - Brazil 2  
Americas 2 - Canada 2  
. . .
```

Fetching as 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`. This function is commonly used for debugging PHP scripts. 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. The result is:

```
City is Roma
City is Venice
City is Tokyo
City is Hiroshima
City is Southlake
. . .
```

The `oci_define_by_name()` function has an optional type parameter that is useful, for example, to specify that the PHP variable should be an integer instead of a string.

Fetching and Working with Numbers

Numbers are fetched as strings by OCI8 which can have implications for subsequent PHP arithmetic. Also PHP and Oracle differ in their 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 datatype for numeric columns is `string`:

Executing SQL Statements With OCI8

```
array(3) {  
    ["CN1"]=>  
        string(2) "71"  
    ["CN2"]=>  
        string(4) "70.6"  
    ["DIFF"]=>  
        string(2) ".4"  
}
```

The conversion from number to string is done by Oracle and 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 PHP later casts the string to a number for an arithmetic operation. Oracle's default formats can be changed easily and it is recommended to explicitly set them so your scripts are portable. See *Oracle Number Formats* in the chapter *Globalization*.

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);  
$diff = $row['CN1'] - $row['CN2'];  
echo "PHP difference is " . $diff . "\n";
```

The output shows:

```
PHP difference is 0.4000000000000001
```

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 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 shows the output from 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";
```

In this example the default format was the Oracle American standard of DD-MON-YY so the output is:

```
Hire date is 17-SEP-87
```

Dates inserted are expected to be in the default 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 settings 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 the `SYSDATE` function. 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 user 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

Executing Data Definition Language (DDL) and Data Manipulation Language (DML) statements, like `CREATE` and `INSERT`, simply requires a parse and execute:

```
$s = oci_parse($conn, "create table iltest (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 but one caveat is that temporary tables have no statistics for the Oracle optimizer to evaluate.

Transactions

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.

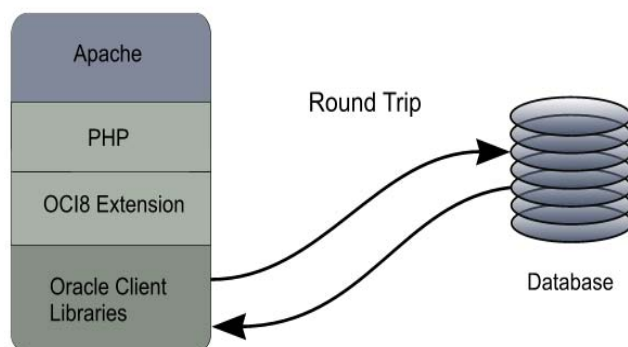


Figure 88: Each round trip between PHP and the Database reduces scalability.

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 and different from Oracle'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
```

Other users of the table will immediately be able to see the new record. Auto-committing can be handy for single `INSERTs` and `UPDATES`, but transactional and performance requirements should be thought about before using the default mode everywhere.

You specify not to auto-commit but to begin a transaction with:

```
$s = oci_parse($c, "insert into testtest values ('my data 2')");
oci_execute($s, OCI_DEFAULT); // not committed
```

The PHP parameter name `OCI_DEFAULT` is borrowed from Oracle's Call Interface, where the value of the `C` macro with the same name is actually the default value when nothing else is specified. In PHP a better name would have been `NO_AUTO_COMMIT`, but we are stuck with the awkward name.

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.

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, executing a query will commit an outstanding transaction if `OCI_DEFAULT` 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 the 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_DEFAULT`, PHP will send an explicit rollback to the database at the end of every script, even though it is unnecessary.

Autonomous Transactions

Oracle's procedural language for SQL, PL/SQL, 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 13: 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 14: logger.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$s = oci_parse($c, "insert into mytable values ('abc')");
oci_execute($s, OCI_DEFAULT); // don't commit

$s = oci_parse($c, "begin updatelog('INSERT attempted'); end;");
oci_execute($s, OCI_DEFAULT); // don't commit

oci_rollback($c);

?>
```

Even though `OCI_DEFAULT` is used, the autonomous transaction commits to the log table. This commit does not commit the 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 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 (col1 date);
```

Rerun *transactions.php* a few times, changing `oci_connect()` to `oci_new_connect()` and `oci_pconnect()`:

Script 15: transactions.php

```
<?php

function do_query($c, $query)
{
    $s = oci_parse($c, $query);
    oci_execute($s, OCI_DEFAULT);
    oci_fetch_all($s, $res);
    echo "<pre>";
    var_dump($res); // PHP debugging function for displaying output
    echo "</pre>";
}

$c1 = oci_connect("hr", "hrpwd", "localhost/XE"); // first connection
$s = oci_parse($c1, "insert into mytable values ('" . date('j:M:y') . "')");
oci_execute($s, OCI_DEFAULT); // does not commit
do_query($c1, "select * from mytable");

$c2 = oci_connect("hr", "hrpwd", "localhost/XE"); // second connection
do_query($c2, "select * from mytable");

?>
```

The script inserts (but does not commit) using one connection and queries back the results with 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 is:

```
array(1) {
  ["COL1"]=>
    array(1) {
      [0]=>
        string(9) "16-JUN-07"
    }
}
array(1) {
  ["COL1"]=>
    array(1) {
      [0]=>
        string(9) "16-JUN-07"
    }
}
```

```
}
```

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:

```
array(1) {  
  ["COL1"]=>  
  array(1) {  
    [0]=>  
    string(9) "16-JUN-07"  
  }  
}  
array(1) {  
  ["COL1"]=>  
  array(0) {  
  }  
}
```

PHP Error Handling

The PHP installation chapter recommended setting `display_errors` to `On` in `php.ini` to aid debugging. You might consider also enabling the `E_STRICT` level for the `error_reporting` parameter so you can catch any potential problems that will cause upgrade issues. 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.

Error handling can also be controlled at runtime. For example, to see all errors displayed, scripts can set:

```
error_reporting(E_ALL);  
ini_set('display_errors', true);
```

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', 'hrpwd', '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 being output, the partially complete page contents can be discarded and an error page created instead.

Handling OCI8 Errors

The error handling 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 display 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.

Table 9: Error array after `$e = oci_error()`.

Variable	Description
<code>\$e["code"]</code>	Oracle error number
<code>\$e["message"]</code>	Oracle error message
<code>\$e["offset"]</code>	Column position in the SQL statement of the error
<code>\$e["sqltext"]</code>	The text of the SQL statement

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

For connection errors, no argument to `oci_error()` is needed:

```
$c = oci_connect("hr", "not_hrpwd", "localhost/XE");
if (!$c) {
    $e = oci_error(); // No parameter passed
    var_dump($e);
}
```

With the invalid password, the output is:

```
array(4) {
  ["code"]=>
  int(1017)
  ["message"]=>
  string(50) "ORA-01017: invalid username/password; logon denied"
  ["offset"]=>
  int(0)
  ["sqltext"]=>
  string(0) ""
}
```

The output shows that `$e` is an array. The `code` entry `1017` matches the error number in the error message. Since there was no SQL statement involved, the `sqltext` is empty and the `offset` is `0`.

The internal reimplementation of OCI8 1.3 connection management makes it better at automatically recovering from database unavailability errors. Even so, with persistent connections PHP can return a cached connection without knowing if the database is still available. If the database has restarted since the time of first connection, or the DBA had enabled resource management that limited the maximum connection time for a user, or even if the DBA issued an `ALTER SESSION KILL` command to close the user's database session, an OCI8 call might return an Oracle error when it tries to reuse a persistent connection. However OCI8 will then mark the connection as invalid and the next time the Apache/PHP process tries to connect, a new connection will be successfully created and usable.

For example, consider a script that uses a persistent connection:

```
<?php
```

Executing SQL Statements With OCI8

```
$c = oci_pconnect('hr', 'hrpwd', 'localhost/XE');
if (!$c) {
    $e = oci_error();
    echo $e['message'] . "<br>\n";
    exit;
}

...

?>
```

After the script completes, if the DBA issues an `ALTER SESSION KILL` command for the database connection created, the next time the script is run it will display the error:

```
ORA-00028: your session has been killed
```

However if the script is run a third time it will connect and run to completion normally.

OCI8 Parse Errors

For parse errors, pass `oci_error()` the connection resource:

```
$s = oci_parse($c, "select city from locations");
if (!$s) {
    $e = oci_error($c); // Connection resource passed
    echo $e["message"] . "<br>\n";
}
```

OCI8 Execution and Fetching Errors

For execution and fetching errors, pass `oci_error()` the statement resource:

```
$rc = oci_execute($s);
if (!$rc) {
    $e = oci_error($s); // Statement resource passed
    var_dump($e);
}

$rc = oci_fetch_all($s, $results);
if (!$rc) {
    $e = oci_error($s); // Statement resource passed
    var_dump($e);
}
```

An example of an execution error is when the table being queried does not exist:

```
$s = oci_parse($c, "select city from not_locations");
$rc = oci_execute($s);
if (!$rc) {
    $e = oci_error($s);
    var_dump($e);
}
```

The output shows the message number and text. It also contains the text of the statement and the column offset position of the error in that statement. Column 17 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"
}
```

Tuning SQL Statements in PHP Applications

Tuning is an art and a science. The database-centric approach to tuning is to first tune the application, next tune the SQL, and finally tune the database.

SQL tuning and Database tuning are covered in the Oracle documentation. SQL tuning will help make efficient use of table design, indexes and the Oracle optimizer. Database tuning maximizes throughput and allows efficient use of buffering and I/O strategies.

On the OCI8 side, transaction management can help reduce round trips from PHP to the database and reduce processing overheads. Make sure to commit only when necessary.

It is almost always more efficient to select the minimum amount of data necessary and only return rows and columns that will be used by PHP. Consider using PL/SQL packages for complex data operations to minimize data transfers. Efficient caching of statements by having reusable code and using bind variables are fundamental in improving database performance.

Using Bind Variables

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.

Executing SQL Statements With OCI8

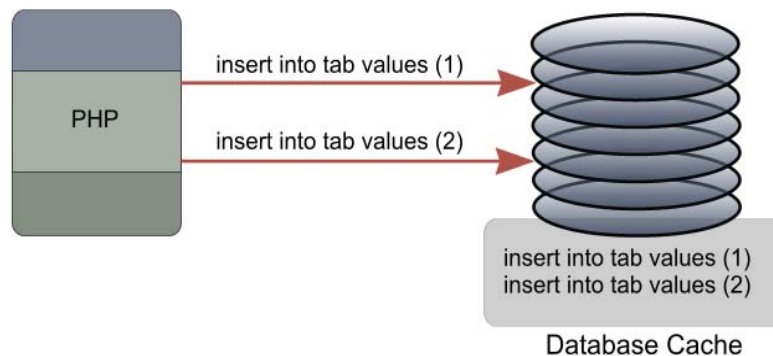


Figure 89: Not binding wastes database resources.

Binding is *highly* recommended. It can improve overall database throughput. Oracle is more likely to find the matching statement in its cache and be able to reuse the execution plan and context for that statement, even if someone else originally executed it.

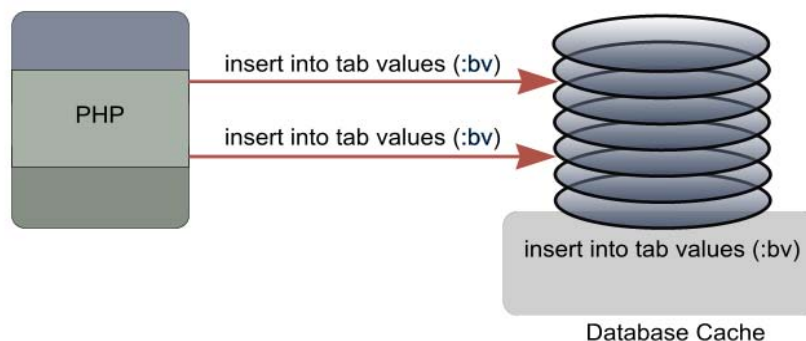


Figure 90: Binding improves performance and security.

Bind variables are also an important way to prevent SQL injection security attacks. SQL injection may occur when SQL statements are hard coded text concatenated with user input:

```
$w = "userid = 1";  
$s = oci_parse($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. A `oci_bind_by_name()` call tells Oracle which PHP variable to actually use when executing the statement.

Script 16: bindvar.php

```
<?php
$c = oci_connect("hr", "hrpwd", "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'] . "<br>\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'] . "<br>\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.

The syntax of `oci_bind_by_name()` is:

```
$rc = oci_bind_by_name($statement, $bindvarname, $phpvariable, $length, $type)
```

The length and type are 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.

It is recommended to pass the length if a query is being re-executed in a script with different bind values. For example when binding, pass the length of the largest potential string. Passing the length also avoids potential edge-case behavior differences if a script runs with multiple different Oracle character sets.

Some older PHP examples use `&` with `oci_bind_by_name()` parameters. Do not do this. Since a call-by-reference clean up in the implementation of PHP, this syntax has been deprecated and may cause problems.

Executing SQL Statements With OCI8

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.

It is a common problem with binding in a `foreach` loop:

```
$ba = array(':dname' => 'IT Support', ':loc' => 1700);
foreach ($ba as $key => $val) {
    oci_bind_by_name($s, $key, $val);
}
```

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 17: bindloop.php

```
<?php

$c = oci_connect("hr", "hrpwd", "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);

while (($row = oci_fetch_array($s, OCI_ASSOC))) {
    foreach ($row as $item) {
        echo $item." ";
    }
    echo "<br>\n";
}

?>
```

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.

Finally, Oracle does not use question mark '?' for bind variable placeholders at all and 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 with *LIKE* and *REGEXP_LIKE* Clauses

Similar to the simple example above, you can bind the value used in a pattern-matching SQL [LIKE](#) or [REGEXP_LIKE](#) clause:

Script 18: *bindlike.php*

```
<?php

$c = oci_connect("hr", "hrpwd", "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"
  }
}
```

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:

Executing SQL Statements With OCI8

Script 19: bindregexp.php

```
<?php
$c = oci_connect("hr", "hrpwd", "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 works:

```
$s = oci_parse($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";
}
```

This displays both surnames but it leads to the scaling and security issues that bind variables overcome.

Trying to emulate the query with a bind variable does not work:

```
$s = oci_parse($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);
```

All this gives is 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 the `IN` clause is to use individual bind variables. `NULLs` can be bound for any unknown values:

Script 20: bindinlist.php

```
<?php
$c = oci_connect("hr", "hrpwd", "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
```

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);
```

Executing SQL Statements With OCI8

There is also an optional fifth parameter, which is the datatype. This is mostly used for binding LOBS and result sets as shown in a later chapter. One micro-optimization when numbers are known to be integral, is to specify the datatype 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.

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, try using a SQL `CASE` statement:

```
$s = oci_parse($c, "select first_name, last_name
                    from employees
                    order by
                        case :ob
                            when 'FIRST_NAME' then first_name
                            else last_name
                        end");
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 21: rowid.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

// Fetch a record
$s = oci_parse($c,
```

```

        'select rowid, street_address
        from locations where location_id = :l_bv for update');
$locid = 3000; // location to fetch
oci_bind_by_name($s, ':l_bv', $locid);
oci_execute($s, OCI_DEFAULT);
$row = oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_NULLS);

$rid = $row['ROWID'];
$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

```
Murtenstrasse 921
```

to

```
MURTENSTRASSE 921
```

Tuning the Prefetch Size

You can tune PHP's overall query performance with the configuration parameter:

```
oci8.default_prefetch = 100
```

This parameter sets the number of rows returned in a batch when each fetch call to the database occurs. The default 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.

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 a request to the database is made. Testing will show the optimal 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 cache is managed by Oracle. The database returns the specified number of rows to a cache in its client buffers. 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 in the cache until eventually another batch of records is fetched from the database.

Executing SQL Statements With OCI8

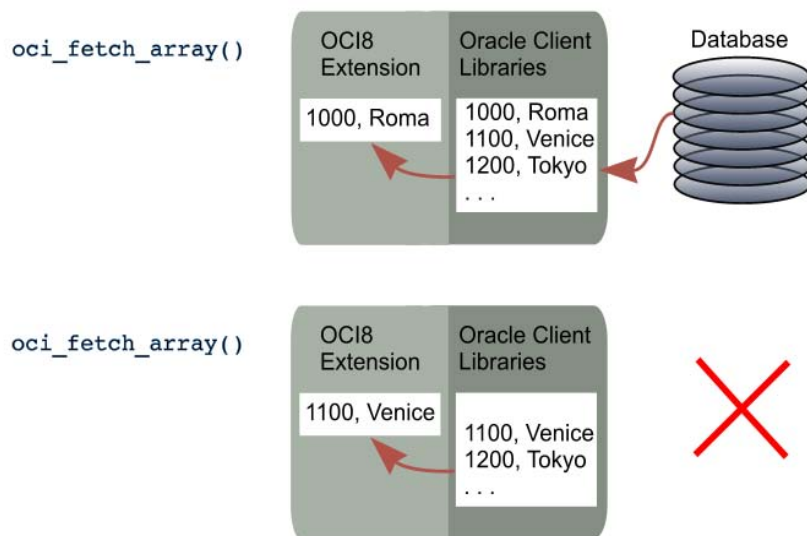


Figure 91: The first request to the database fetches multiple rows to the cache. Subsequent fetches read from the cache.

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 user, PHP OCI8 internally fetches all the records in batches of the specified prefetch size.

Prefetching is not used when queries contain LONG or LOB columns, or when fetching from a REF CURSOR.

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 = 20
```

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 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-information about the statement back to PHP.

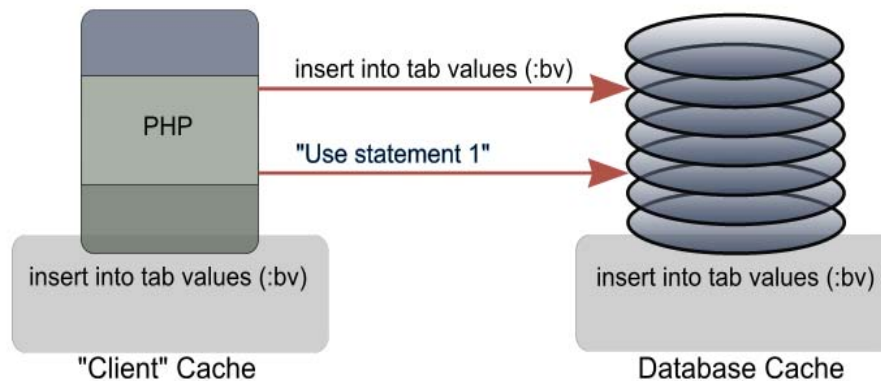


Figure 92: The second time a statement is issued, the statement text is not sent to the database.

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 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.

OCI8 clears 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.

The database cache is enabled with the `RESULT_CACHE_MODE` database parameter, which has several modes. With the default mode, queries for which you want results to be cached need a hint added:

```
$s = oci_parse($c, "select /*+ result_cache */ * from employee");
```

No PHP changes are required – applications will immediately benefit from the server cache.

The client cache is ideal for small queries from infrequently modified tables, such as lookup 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.

A key feature of the caches is that Oracle automatically handles cache entry invalidation when a database change invalidates the stored results. Oracle will check the client cache entries each time any *round trip* to the database occurs. If no round trip has happened with a configurable "lag" time, the client cache is assumed stale.

The *Oracle® Call Interface Programmer's Guide, 11g Release 1 (11.1)* contains the best description of the feature and has more about when to use it and how to manage it.

Executing SQL Statements With OCI8

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
```

In PHP, the key to using the client-cache is to pass `OCI_DEFAULT` to `oci_execute()`:

Script 22: crc.php

```
<?php
$c = oci_pconnect('hr', 'hrpwd', 'localhost/orcl');
for ($i = 0; $i < 1000; ++$i) {
    $s = oci_parse($c,
        "select /*+ result_cache */ * from employees where rownum < 2");
    oci_execute($s, OCI_DEFAULT);
    oci_fetch_all($s, $res);
}
?>
```

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
                        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 /*+ result_cache */ * 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 remove `OCI_DEFAULT` from the execute call:

```
oci_execute($s);
```

Re-run the script:

```
$ php crc.php
```

The monitoring query now shows the modified query was executed another 1000 times, or once per loop iteration. This means the client query result cache was not used and each iteration had to be processed in the database:

PARSE_CALLS	EXECUTIONS	SQL_TEXT
4	1002	select /*+ result_cache */ * from employees where rownum < 2
3	3	select parse_calls, executions, sql_text from v\$sql where sql_text like '%employees%'

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.

Limiting Rows and Creating Paged Datasets

Oracle's SQL does not have a `LIMIT` keyword. It is not in the SQL standard and the vendors that use it have different implementations. There are several alternative ways to limit the number of rows returned in OCI8.

The `oci_fetch_all()` function 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.

```
$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. 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_ROWS )
where rnum >= MIN_ROWS
```

Here, `MIN_ROWS` is the row number of first row and `MAX_ROWS` is the row number of the last row to return. In PHP you might do this:

Executing SQL Statements With OCI8

Script 23: *limit.php*

```
<?php

$c = oci_connect("hr", "hrpwd", "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` 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]=>
    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
Chung
Colmenares
Davies

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

Executing SQL Statements With OCI8

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 24: 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
    select myseq.nextval into :new.myid from dual;
end;
/
```

In Oracle Database11g you can replace the `SELECT` with:

```
:new.myid := myseq.nextval;
```

In PHP insert two rows:

Script 25: autoinc.php

```
<?php

$c = oci_connect("hr", "hrpwd", "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 26: *insertid.php*

```
<?php
$c = oci_connect("hr", "hrpwd", "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");
oci_bind_by_name($s, ":rid", $rid, -1, OCI_B_ROWID);
oci_execute($s);
```

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 couple are mentioned below.

Case Insensitive 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;
```

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. You must be connected as the *system* user in this particular example, since it depends on table data not available to *hr*:

```
select max_extents, corr(max_trans, initial_extent)
from all_tables
group by max_extents;
```

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.

USING PL/SQL WITH OCI8

PL/SQL is Oracle's procedural language extension to SQL. It is a server-side, stored 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 (in Oracle Database 11g), optimizing and debugging features, and a 'wrap' code obfuscation facility to protect the intellectual property of your applications.

PL/SQL Overview

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 client 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. Some functionality should only ever be in the database, such as data change triggers. In Oracle, you can create these to be fired when an event such as an insert or a user logon occurs.

A PL/SQL block has three basic parts:

- A declarative part ([DECLARE](#))
- An executable part ([BEGIN ... END](#))
- An exception-handling ([EXCEPTION](#)) part that handles error conditions

For example:

```
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 `/` 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');
11 end;
12 /
Salary is 2500
```

Other tools have different ways of indicating the end of the statements and how to switch server output on.

Only the executable part of a PL/SQL block is required. The declarative and exception handler parts are optional.

If an application performs several SQL statements at one time, it can be efficient to bundle them 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.

Blocks, Procedures, Packages and Triggers

PL/SQL code can be categorized as one of the following:

- Anonymous block
- Stored procedure or function
- Package
- Trigger

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 or Standalone Procedures and Functions

A stored procedure or function is a PL/SQL block that Oracle stores in the database and can be called by name from an application. Functions 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.

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 a SQL script file in SQL*Plus, it is helpful to turn [SET ECHO ON](#) to see the line numbers.

See below for handling PL/SQL errors in PHP.

Packages

Typically, stored procedures and functions are encapsulated into packages. This helps minimize 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 in the package body changes.

```
SQL> create or replace package mypack as
  2   function myfunc(i_p in number) return varchar2;
  3   end mypack;
  4   /

SQL> create or replace package body mypack as
  2   function myfunc(i_p in number) return varchar2 as
  3     d_l varchar2(20);
  4   begin
  5     select mydata into d_l from mytab where myid = i_p;
  6     return d_l;
  7   end;
```

```
8 end mypack;
9 /
```

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 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 the procedure *myproc* the code is:

Script 27: createmyproc.php

```
<?php
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$sqlsql = "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;";
$s = oci_parse($c, $sqlsql);
$r = oci_execute($s);
if ($r) {
    echo 'Procedure created';
}

?>
```

Note the last character of the PL/SQL statement is a semi-colon (after PL/SQL keyword [end](#)), which is different to the way SQL statements are terminated.

PHP string concatenation was used to build the statement. If you don't do this and are on Windows, see the end-of-line terminator issue mentioned below.

Similar to the 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

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:

```
$pysql = "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;";
```

If `showcompilationerrors()` function, shown later, is used, its additional messages will show:

```
PLS-00103: Encountered the symbol "" when expecting one of the following:
```

This error, with its 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:

```
$pysql = "create or replace procedure myproc . . .";
```

- Use PHP string concatenation with appropriate white space padding between tokens:

```
$pysql = "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

To invoke the previously created PL/SQL procedure, use `BEGIN` and `END` to create an anonymous block:

Script 28: anonpysql.php

```
<?php
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');
$s = oci_parse($c, "begin myproc('Alison', 456); 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 29: callpysql.php

```
<?php
```

Using PL/SQL With OCI8

```
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');  
  
$s = oci_parse($c, "call myproc('Chris', 123)");  
oci_execute($s);  
  
?>
```

The `call` command is actually a SQL command and does not have a trailing semi-colon.

Calling PL/SQL Functions

Calling a PL/SQL function needs a bind variable to hold the return value. Using the function `myfunc()` created previously:

Script 30: plsqlfunc.php

```
<?php  
  
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');  
  
$s = oci_parse($c, "begin :ret := mypack.myfunc(123); end;");  
oci_bind_by_name($s, ':ret', $r, 20);  
oci_execute($s);  
echo "Name is: ".$r;  
  
?>
```

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 20 bytes should be allocated to hold the result. The script output is:

```
Name is: Chris
```

This example also shows how to call a function or procedure inside a package using the syntax `packagename.functionname()`.

Binding Parameters to Procedures and Functions

PL/SQL procedure and functions 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 `myproc()` example the parameters were `IN` parameters so the code could be:

```
$s = oci_parse($c, "call myproc(:data, :id)");  
$data = "Chris";  
$id = 123  
oci_bind_by_name($s, ":data", $data);  
oci_bind_by_name($s, ":id", $id);  
oci_execute($s);
```

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.

Array Binding and PL/SQL Bulk Processing

OCI8 1.2 (PHP 5.1.2) introduced a new 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 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, `myselfproc()`, 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 31: arrayinsert.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 myselfproc(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 myselfproc(p_arr out arrtype, p_count in number) is
    begin
        select name bulk collect into p_arr from mytab where rownum <= p_count;
        end myselfproc;
end mypkg;
/
show errors
```

To insert a PHP array `$a` into *mytab*, use:

Script 32: arrayinsert.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');
```

Using PL/SQL With OCI8

```
$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 `myselfproc()` procedure can be called. The number of elements `$numelems` to be fetched is passed into `myselfproc()` by being bound to `:n`. This limits the query to return four rows. The value is also used in the `oci_bind_array_by_name()` call so the output array `$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 33: arrayfetch.php

```
<?php

$c = oci_connect("hr", "hrpwd", "localhost/XE");

$numelems = 4;
$s = oci_parse($c, "begin mypkg.myselfproc(: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]=>
  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 `SQLT_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 error 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 *plsqlerr.php*. It creates a procedure referencing a non-existent table and then queries the *USER_ERRORS* table when the ORA-24344 error occurs:

Script 34: *plsqlerr.php*

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

ini_set('display_errors', false); // do not automatically show PHP errors

// PL/SQL statement with deliberate error: not_mytab does not exist
$sqlsql = "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, $sqlsql);
$r = @oci_execute($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);
```

Using PL/SQL With OCI8

```
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 35: refcur1.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 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 like a normal query and used in a fetch loop.

Script 36: refcur1.php

```

<?php

$c = oci_connect('hr', 'hrpwd', '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 as a statement resource

echo "<table border='1'>\n";
while($row = oci_fetch_array($refcur, OCI_ASSOC)) {
    echo "<tr>";
    foreach ($row as $c) {
        echo "<td>$c</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 37: 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(
        p_emp_id      in number,
        p_contact_info out contact_info_cur_type);

```

Using PL/SQL With OCI8

```
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 *refcur1.php*, except in the call to the procedure. The procedure name has changed and, for this example, an employee identifier of **188** is used.

```
$s = oci_parse($c, "call emp_pack.get_contact_info(188, :rc)");
$rrefcur = oci_new_cursor($c);
oci_bind_by_name($s, ':rc', $rrefcur, -1, OCI_B_CURSOR);
oci_execute($s);

// Execute and fetch from the cursor
oci_execute($rrefcur); // treat as a statement resource
. . .
```

The output is the record for employee 188. The four values match the `contact_info_type`:

```
Kelly    Chung    650.505.1876    KCHUNG
```

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 38: 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', 'hrpwd', '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 query is contrived. 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`.

In Oracle Database XE you can monitor the number of cursors that each PHP connection has open by logging into Oracle Application Express as the *system* user and navigating to **Administration > Monitor > Sessions > Open Cursors**.

Converting from REF CURSOR to PIPELINED Results

Records returned by REF CURSORS cannot currently take advantage of row pre-fetching. For performance critical sections of code, 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 *refcur1.sql* to return pipelined data, create a package:

Script 39: 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 40: rc2pipeline.php

```
<?php
$c = oci_connect('hr', 'hrpwd', '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 be have been declared 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 datatypes TABLE and VARRAY, which allow you to declare index-by tables, nested tables, and variable-size arrays.

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. 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 are manipulated in PHP by methods on a collection resource, which is allocated with `oci_new_collection()`.

In a simple email address book demonstration, 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 41: 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)

```

Using PL/SQL With OCI8

```
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));
    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 42: updateaddresses.php

```
<?php

$c = oci_connect('hr', 'hrpwd', '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 manual for more information.

Using PL/SQL and SQL 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. 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, 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 parameter:

Script 43: 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/hrpwd@localhost/XE @ctx.sql
```

Using PL/SQL With OCI8

This is the “fixed” part of the problem, representing the pre-supplied functionality you need to work with.

Using OCI8 Collection Functions

To call `SuppliedProc()` using collection functions, create a wrapper function in PL/SQL to convert the PL/SQL type `SuppliedTabType` to a pair of SQL types. Use SQL*Plus to run *myproc.sql*:

Script 44: myproc.sql

```
-- Create a wrapper procedure that calls the pre-supplied
-- SuppliedProc() and converts its output to SQL types.

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 procedure MyProc
    (p_id in out MyIdRec, p_data in out MyDataRec)
as
    l_results SuppliedPkg.SuppliedTabType;
begin
    -- get results from existing procedure
    SuppliedPkg.SuppliedProc(l_results);

    -- copy to a type we can pass back to PHP
    p_id.delete;
    p_data.delete;
    for i in 1..l_results.count loop
        p_id.extend;
        p_id(i) := l_results(i).id;
        p_data.extend;
        p_data(i) := l_results(i).data;
    end loop;
end MyProc;
/
show errors
```

Now you can call `MyProc()`:

Script 45: ctx.php

```
<?php

$c = oci_connect("hr", "hrpwd", "localhost/XE");

$s = oci_parse($c, 'begin MyProc(: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<br>\n";
}

?>

```

This allocates two collections and binds them as the parameters to `MyProc()`. After `MyProc()` has been called, the 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

```

Using a REF CURSOR

If the data can be converted to a REF CURSOR, a straightforward query can be used to fetch results. This example shows an object type. The procedure `mycreatedata()` creates the original data. When it is called, it returns an object table containing five rows.

Script 46: object.sql

```

drop type mytabletype;
drop type mytype;

create or replace type mytype as object (myid number, mydata varchar2(20));
/
show errors

create or replace type mytabletype as table of mytype;
/
show errors

create or replace procedure mycreatedata(outdata out mytabletype) as
begin
    outdata := mytabletype();
    for i in 1..5 loop
        outdata.extend;
        outdata(i) := mytype(i * 2, 'some name ' || i);
    end loop;
end;
/
show errors

```

Using PL/SQL With OCI8

Data of type `mytabletype` can be converted to a REF CURSOR using a wrapper function:

Script 47: objectrc.sql

```
create or replace procedure mywrapper1(rc out sys_refcursor) as
  origdata mytabletype;
begin
  mycreatedata(origdata); -- create some values
  open rc for select * from table(cast(origdata as mytabletype));
end mywrapper1;
/
show errors
```

This can be called in PHP like:

Script 48: objectrc.php

```
<?php

$c = oci_connect("hr", "hrpwd", "localhost/XE");

$s = oci_parse($c, "begin mywrapper1(: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, $res);
var_dump($res);

?>
```

The output is:

```
array(2) {
  ["MYID"]=>
  array(5) {
    [0]=>
    string(1) "2"
    [1]=>
    string(1) "4"
    [2]=>
    string(1) "6"
    [3]=>
    string(1) "8"
    [4]=>
    string(2) "10"
  }
  ["MYDATA"]=>
  array(5) {
    [0]=>
    string(11) "some name 1"
    [1]=>
    string(11) "some name 2"
    [2]=>
```

```

    string(11) "some name 3"
    [3]=>
    string(11) "some name 4"
    [4]=>
    string(11) "some name 5"
  }
}

```

Oracle's lack of pre-fetching for REF CURSORS needs to be considered if performance is important. Using a PIPELINED function or array bind can be much faster.

Binding an Array

An alternative solution lets you use the fast `oci_bind_array_by_name()` function. The wrapper procedure looks like:

Script 49: objectba.sql

```

create or replace procedure mywrapper2 (
    pempno out dbms_sql.number_table,
    pename out dbms_sql.varchar2_table) as origdata mytabletype;
begin
    mycreatedata(origdata); -- create some values
    select myid, mydata
    bulk collect into pempno, pename
    from table(cast(origdata as mytabletype));
end mywrapper2;
/
show errors

```

This can be called from PHP with:

Script 50: obejctba.php

```

<?php
$c = oci_connect("hr", "hrpwd", "localhost/XE");

$s = oci_parse($c, "begin mywrapper2(: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:

```

array(5) {
    [0]=>
    int(2)
    [1]=>

```

Using PL/SQL With OCI8

```
int(4)
[2]=>
int(6)
[3]=>
int(8)
[4]=>
int(10)
}
array(5) {
  [0]=>
  string(11) "some name 1"
  [1]=>
  string(11) "some name 2"
  [2]=>
  string(11) "some name 3"
  [3]=>
  string(11) "some name 4"
  [4]=>
  string(11) "some name 5"
}
```

Using a PIPELINED Function

The wrapper function to convert to a PIPELINED function is:

Script 51: objectpl.sql

```
create or replace package myplpkg as
  type plrow is table of mytype;
  function mywrapper3 return plrow pipelined;
end;
/
show errors

create or replace package body myplpkg as
  function mywrapper3 return plrow pipelined is
    origdata mytabletype;
  begin
    mycreatedata(origdata); -- create some values
    for i in 1 .. origdata.count() loop
      pipe row (origdata(i));
    end loop;
  end mywrapper3;
end myplpkg;
/
show errors
```

This can be called from PHP with:

Script 52: objectpl.php

```
<?php
```

```
$c = oci_connect("hr", "hrpwd", "localhost/XE");

$s = oci_parse($c, "select * from table(myplpkg.mywrapper3())");
oci_execute($s);
oci_fetch_all($s, $res);
var_dump($res);

?>
```

The output is the same as from *objectrc.php*.

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 code turns on buffering, calls something 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;");
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 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 53: 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
```

Using PL/SQL With OCI8

```
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* along with `getdbmsoutput()` which returns an array of the output lines:

Script 54: dbmsoutput.inc

```
<?php

// Turn DBMS_OUTPUT on
function enabledbmsoutput($c)
{
    $s = oci_parse($c, "begin dbms_output.enable(null); end;");
    $r = oci_execute($s);
    return $r;
}

// 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'];
}

?>
```

The next script uses these functions to show “printing” output from PL/SQL blocks:

Script 55: dbmsoutput.php

```
<?php

include("dbmsoutput.inc");

$c = oci_connect("hr", "hrpwd", "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 Function Result Cache

Oracle Database 11g introduced a cache for PL/SQL function results, ideal for repeated lookup operations. The cache contains the generated result of a previous function call. 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 56: 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, then the cache is automatically invalidated by Oracle. The next time the function is called, it will execute completely and update the cache appropriately.

See the *Oracle Database PL/SQL Language Reference 11g Release 1 (11.1)* 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*.

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 use the tables shown in the Oracle Database Express Edition 2 *Day Plus Locator Developer Guide* sample scenario. Create these tables in SQL*Plus before continuing:

http://download.oracle.com/docs/cd/B25329_01/doc/appdev.102/b28004/xe_locator.htm#CIHEAIGJ

Inserting Locator Data

Inserting Locator data is simply a matter of executing the appropriate `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 57: loc1.php

```
<?php  
  
$c = oci_connect('hr', 'hrpwd', '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"]=>
  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"
  }
}

```

Using PL/SQL With OCI8

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 58: loc2.php

```
<?php
$c = oci_connect('hr', 'hrpwd', '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 example 1-8 of the Oracle Database Express 2 Day Plus Locator Developer Guide:

http://download.oracle.com/docs/cd/B25329_01/doc/appdev.102/b28004/xe_locator.htm#CIHEBIDA

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. The row describes a polygon of (x,y) ordinates which are given as pairs in the `SDO_ORDINATE_ARRAY` array:

Script 59: 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_ordinate_array(5,1, 8,1, 8,6, 5,7, 5,1) -- list of X,Y coordinates
  )
);

commit;
```

A decomposition function 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:

Script 60: cm2.sql

```
create or replace procedure myproc(p_id in number, p_o out sdo_ordinate_array) as
begin
  select cm.shape.sdo_ordinates
  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 61: cm.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$s = oci_parse($c, "begin myproc(:id, :ords); end;");
$id = 301;
oci_bind_by_name($s, ":id", $id);
$sords = oci_new_collection($c, "SDO_ORDINATE_ARRAY");
oci_bind_by_name($s, ":ords", $sords, -1, OCI_B_NTY);
oci_execute($s);

for ($i = 0; $i < $sords->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 the earlier section *Using PL/SQL and SQL 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.

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 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 incorrect tag *weeding*:

Script 62: dschedinit.sql

```
connect system/systempwd  
  
grant create job to hr;  
  
connect hr/hrpwd  
  
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 63: dbsched.sql

```
create or replace procedure changetagname(old in varchar2, new in varchar2) as
  b number;
begin
  for i in 1..1000000000 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 64: dsched.php

```
<?php

$c = oci_connect("hr", "hrpwd", "localhost/XE");

function doquery($c)
{
  $s = oci_parse($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);
```

Using PL/SQL With OCI8

```
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"
  }
}
```


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 65: 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>
```

To reuse the procedure directly in PHP, use SQL*Plus to create a mapping function that pipes the output from the [HTP](#) calls:

Script 66: 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
```

Using PL/SQL With OCI8

```
    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 67: mymodplsqli.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$stmt = oci_parse($c, 'select * from table(mymodplsqli(: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"]);
    }
}

?>
```

When called in a browser, the output is the expected rendition of the HTML fragment shown earlier.

USING LARGE OBJECTS IN OCI8

Oracle Character Large Object (CLOB) and Binary Large Object (BLOB) types can contain very large amounts of data. They can be used for table columns and for PL/SQL variables. There are various creation options to specify optimal Oracle table storage. A pre-supplied `DBMS_LOB` package makes manipulation in PL/SQL easy.

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 is `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 mybtab (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.

Inserting and Updating LOBs

In PHP, LOBs are generally manipulated using a descriptor. PHP code to insert into `mybtab` is:

Script 68: blobinsert.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$myblobid = 123;
$myv = 'a very large amount of binary data';

$s = oci_parse($c, 'insert into mybtab (blobid, blobdata)
                  values (:myblobid, EMPTY_BLOB())
                  returning blobdata into :blobdata');
```

Using Large Objects in OCI8

```
$lob = oci_new_descriptor($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_DEFAULT); // use OCI_DEFAULT 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_DEFAULT` 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 mybtabs set
                    blobdata = empty_blob()
                    returning blobdata into :blobdata');
```

Fetching LOBs

When fetching a LOB, OCI8 returns the LOB descriptor and the data can be retrieved by using a `load()` or `read()` method:

Script 69: blobfetch.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$myblobid = 123;

$query = 'select blobdata from mybtabs 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 (($sarr = oci_fetch_array($s, OCI_ASSOC))) {
    echo $sarr['BLOBDATA']->load();    // do something with the BLOB
    $sarr['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:

```
$sarr = oci_fetch_array($s, OCI_ASSOC+OCI_RETURN_LOBS);
echo $sarr['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 size to be limited.

Temporary LOBs

Temporary LOBs make some operations easier. Inserting data with a Temporary LOB does not use a [RETURNING INTO](#) clause:

Script 70: tempblobinsert.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$myblobid = 124;
$myv = 'a very large amount of binary data';

$s = oci_parse($c, 'insert into mybtab (blobid, blobdata)
                    values (:myblobid, :blobdata)');
$llob = oci_new_descriptor($c, OCI_D_LOB);
oci_bind_by_name($s, ':myblobid', $myblobid);
oci_bind_by_name($s, ':blobdata', $llob, -1, OCI_B_BLOB);
$llob->writeTemporary($myv, OCI_TEMP_BLOB);
oci_execute($s, OCI_DEFAULT);
oci_commit($c);
$llob->close();    // close lob descriptor to free resources

?>
```

Temporary LOBs also simplify updating values:

```
$s = oci_parse($c, 'update mybtab 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 mybtab (blobdata, blobid) values(:blobdata, :myblobid);'
    . ' exception'
    . ' when dup_val_on_index then'
```

Using Large Objects in OCI8

```
. '      update mybtab set blobdata = :blobdata where blobid = :myblobid;'
. 'end;');
```

LOBs and PL/SQL procedures

Temporary LOBs can also be used to pass data to PL/SQL [IN](#), and returned from [OUT](#) parameters. Given a PL/SQL procedure that accepts a BLOB and inserts it into [mybtab](#):

Script 71: inproc.sql

```
create or replace procedure inproc(pid in number, pdata in blob) as
begin
    insert into mybtab (blobid, blobdata) values (pid, pdata);
end;
/
show errors
```

PHP code to pass a BLOB to [INPROC](#) would look like:

Script 72: inproc.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$myblobid = 125;
$myv = 'a very large amount of binary data';

$s = oci_parse($c, 'begin inproc(:myblobid, :myblobdata); end;');
$llob = oci_new_descriptor($c, OCI_D_LOB);
oci_bind_by_name($s, ':MYBLOBID', $myblobid);
oci_bind_by_name($s, ':MYBLOBDATA', $llob, -1, OCI_B_BLOB);
$llob->writeTemporary($myv, OCI_TEMP_BLOB);
oci_execute($s);
$llob->close();

?>
```

If the PL/SQL procedure returns a BLOB as an [OUT](#) parameter:

Script 73: outproc.sql

```
create or replace procedure outproc(pid in number, pdata out blob) as
begin
    select blobdata into pdata from mybtab where blobid = pid;
end;
/
show errors
```

PHP code to fetch and display the BLOB would look like:

Script 74: outproc.php

```

<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$myblobid = 125;

$s = oci_parse($c, "begin outproc(:myblobid, :myblobdata); end;");
$llob = oci_new_descriptor($c, OCI_D_LOB);
oci_bind_by_name($s, ':MYBLOBID', $myblobid);
oci_bind_by_name($s, ':MYBLOBDATA', $llob, -1, OCI_B_BLOB);
oci_execute($s, OCI_DEFAULT);
if (is_object($llob)) { // protect against a NULL LOB
    $data = $llob->load();
    $llob->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 full list of LOB methods and functions is shown in Table 10. Check the PHP manual for usage.

Table 10: 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

PHP Function or Method	Action
<code>OCI-Lob->getBuffering</code>	Returns current state of buffering for the LOB
<code>OCI-Lob->import</code> <code>OCI-Lob->saveFile</code>	Loads data from a file to a LOB
<code>OCI-Lob->load</code>	Returns LOB contents
<code>OCI-Lob->read</code>	Returns part of the LOB
<code>OCI-Lob->rewind</code>	Moves the LOB's internal pointer back to the beginning
<code>OCI-Lob->save</code>	Saves data to the LOB
<code>OCI-Lob->seek</code>	Sets the LOB's internal position pointer
<code>OCI-Lob->setBuffering</code>	Changes LOB's current state of buffering
<code>OCI-Lob->size</code>	Returns size of LOB
<code>OCI-Lob->tell</code>	Returns current pointer position
<code>OCI-Lob->truncate</code>	Truncates a LOB
<code>OCI-Lob->write</code>	Writes data to the LOB
<code>OCI-Lob->writeTemporary</code>	Writes a temporary LOB
<code>oci_lob_copy</code>	Copies a LOB
<code>oci_lob_is_equal</code>	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.

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. Start SQL*Plus as:

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

Then run *bfile.sql*:

Script 75: bfile.sql

```
create directory TestDir AS '/tmp';
grant read on directory TestDir to hr;
connect hr/hrpwd@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 76: bfileinsert.php

```
<?php

$c = oci_connect("hr", "hrpwd", "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))");
```

Using Large Objects in OCI8

```
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_DEFAULT);
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 77: bfilequery1.php

```
<?php

$c = oci_connect("hr", "hrpwd", "localhost/XE");

$fnum = 1;

$s = oci_parse($c, "select Image from FileTest where FileNum = :fnum");
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` field like:

```
select Image from FileTest where regexp_like(FileDesc, 'somepattern')
```

Now what? 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 78: showpic.php

```
<?php

$c = oci_connect("hr", "hrpwd", "localhost/XE");
```

```

$s = oci_parse($c,
    'declare '
    . 'b_l bfile;'
    . 'da_l varchar2(255);'
    . 'begin '
    . 'select image into b_l from filetest where filenum = :fnum;'
    . 'dbms_lob.filegetname(b_l, da_l, :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 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 work with 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 XML WITH ORACLE AND PHP

Both Oracle and PHP 5 have excellent XML capabilities, allowing lots of choice for processing information. 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 returning XML data from Oracle to 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. The returned values are XML fragments, and not fully formed XML documents.

Script 79: xmlfrag.php

```
<?php
$c = oci_connect('hr', 'hrpwd', '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 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>
```

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.

There are a number of other XML functions that can be similarly used. See the *Oracle Database SQL 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 initial result needs to be treated in PHP 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 80: getxml.php

```
<?php

$c = oci_connect('hr', 'hrpwd', '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);
$mylob = $row[0]->load(); // treat result as a LOB descriptor
$row[0]->free();

echo "<pre>\n";
echo htmlentities($mylob);
echo "</pre>\n";

?>
```

The value of `$mylob` 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 5's *SimpleXML* extension to convert XML to a PHP object. Following on from the previous example the query results can be converted with:

```

$xml = simplexml_load_string((binary)$mylob);
var_dump($xml);

```

Note the cast to `binary` which ensures consistent encoding. The output is:

```

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 or 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.

As an alternative to *SimpleXML* you could use PHP's older *DOM* extension. There is an article *Using PHP5 with Oracle XML DB* by Yuli Vasiliev in the July/August 2005 Oracle Magazine that discusses this.

Fetching XMLType Columns

Data in XMLType columns could be longer than Oracle's 4000 byte string length limit. When data is fetched as a string, queries may fail in some cases depending on the data length. For example, here is a basic query on the [RESOURCE_VIEW](#) (which is a way access the Oracle XML DB repository from SQL). The *RES* column is an XMLType:

```
$s = oci_parse($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. The failure happens because the database does a conversion from XMLType to a string before returning results to PHP. When the string is too long, an error is thrown:

```
PHP Warning: oci_fetch_array(): ORA-19011: Character string buffer too small
```

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 81: *xmltype.php*

```
<?php
$c = oci_connect('hr', 'hrpwd', '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();
}

?>
```

Inserting into XMLType Columns

You can insert or update XMLType columns by binding as a CLOB.

This example updates a table without an XMLSchema, and which stores the XMLType column as a CLOB.

Script 82: *xmlinsert.sql*

```
create table xwarehouses (warehouse_id number, warehouse_spec xmltype)
                        xmltype warehouse_spec store as clob;

insert into xwarehouses (warehouse_id, warehouse_spec)
values (1,
        xmltype('<?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>')));

commit;
```

PHP code to update the number of available warehouse docks is:

Script 83: *xmlinsert.php*

```
<?php
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');
```

Using XML with Oracle and PHP

```
$s = oci_parse($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);
$llob = oci_new_descriptor($c, OCI_D_LOB);
oci_bind_by_name($s, ':clob', $llob, -1, OCI_B_CLOB);
$llob->writeTemporary($sx->asXml());
oci_execute($s);
$llob->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 shows the number of docks has been decremented from 2 to 1:

```
SQL> set long 1000 pagesize 100

SQL> select warehouse_spec from xwarehouses;

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>
```

See *Using XML in SQL Statements* in *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 function. File `xmlfunc.sql` creates a simple function returning query data for a given id value.

Script 84: 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, first create a wrapper function that maps the XML data to a CLOB:

Script 85: xmlfuncwrapper.sql

```
create or replace function myf_wrapper(p_id number) return clob as
begin
    return myf(p_id).getclobval();
end;
/
```

This can be called in PHP by binding a LOB descriptor to the return value of the function. OCI8 LOB methods like `load()` can be used on the descriptor:

Script 86: xmlfunc.php

```
<?php

$c = oci_connect('hr', 'hrpwd', 'localhost/XE');

$bd = oci_new_descriptor($c, OCI_D_LOB);
$s = oci_parse($c, "begin :bv := myf_wrapper(1); 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. Unfortunately, to keep the footprint of Oracle Database XE small, XQuery is not available in the Oracle Database XE release, but it is in the other editions of Oracle.

A basic XQuery to return the records in the *employees* table is:

```
for $i in ora:view("employees") return $i
```

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 87: xquery.php

```
<?php
$c = oci_connect('hr', 'hrpwd', 'localhost/orcl');

$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 [HEREDOC](#) with XQuery variables escaped. Note there cannot be whitespace on the last line before the token [END](#):

```
$query = <<<END
select column_value from xmltable('for \$i in ora:view("employees") return \$i')
END;
```

In PHP 5.3, use the [NOWDOC](#) syntax so the XQuery variable `$i` does not need escaping. Note the single quotes around the word [END](#) to differentiate it from a [HEREDOC](#).

```
$query = <<<'END'
select column_value from xmltable('for $i in ora:view("employees") return $i')
END;
```

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>WGIEZ</EMAIL>
    <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\(\)](#) you might want to use the [XMLTYPE.GETCLOBVAL\(\)](#) function to avoid string size limit issues:

```

$query = <<<END
select xmltype.getclobval(column_value)
from xmltable('for \${i} in ora:view("employees") return \${i}')
END;

```

and

```

$query = <<<END
select xmltype.getclobval(xmlquery('for \${i} in ora:view("hr",
                                "locations")/ROW return \${i}/CITY'
                                returning content)) from dual
END;

```

The returned column type is a LOB locator and so the fetch uses 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:

```
SQL> declare
  2   res boolean;
  3   begin
  4   begin
  5   -- delete if it already exists
  6   dbms_xdb.deleteResource('/public/test1.txt');
  7   exception
  8   when others then
  9   null;
 10   end;
 11   -- create the file
 12   res := dbms_xdb.createResource('/public/test1.txt',
 13   'the text to store');
 14   commit; -- don't forget to commit
 15   end;
 16   /
```

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`

If you are accessing Oracle Database XE from a remote browser, you may need to enable remote client connection as described in the chapter *Installing Oracle Database 10g Express Edition*.

There is extensive Oracle documentation on XML DB on the Oracle Technology network at <http://otn.oracle.com/tech/xml/xmlldb>.

PHP Scalability and High Availability

This chapter discusses two features supported by PHP OCI8 1.3 that improve scalability and high availability:

- Oracle Database 11g Database Resident Connection Pooling (DRCP)
- Oracle Database 10g Release 2 or 11g Fast Application Notification (FAN)

The Oracle features are usable separately or together.

Database Resident Connection Pooling (DRCP) is a new feature of Oracle Database 11g that addresses scalability requirements in environments requiring large numbers of connections with minimal database resource usage.

Clients that run in high availability configurations such as with Oracle Real Application Clusters (RAC) or Data Guard Physical Stand-By can take advantage of Fast Application Notification (FAN) events in PHP to allow applications to respond quickly to database node failures.

Database Resident Connection Pooling

Oracle Database 11g DRCP addresses scalability requirements in environments requiring large numbers of connections with minimal database resource usage. DRCP pools 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.

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, each process keeps connections open (“persistent connections”) 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 93.

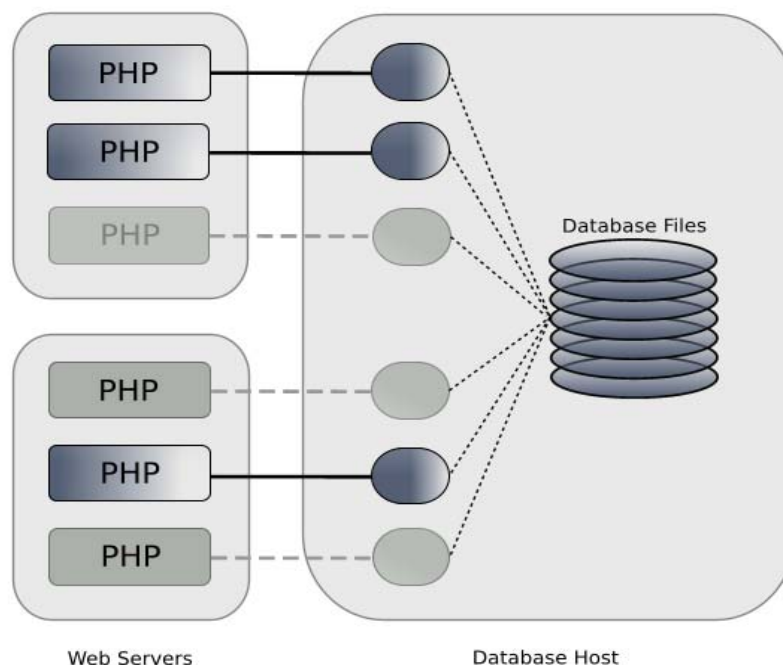


Figure 93: Without DRCP, idle persistent connections from PHP still consume database resources.

How DRCP Works

The architecture of DRCP is shown in Figure 94. A connection broker accepts incoming connection requests from PHP processes and assigns each a free server in the 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 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 take over one of the idle servers.

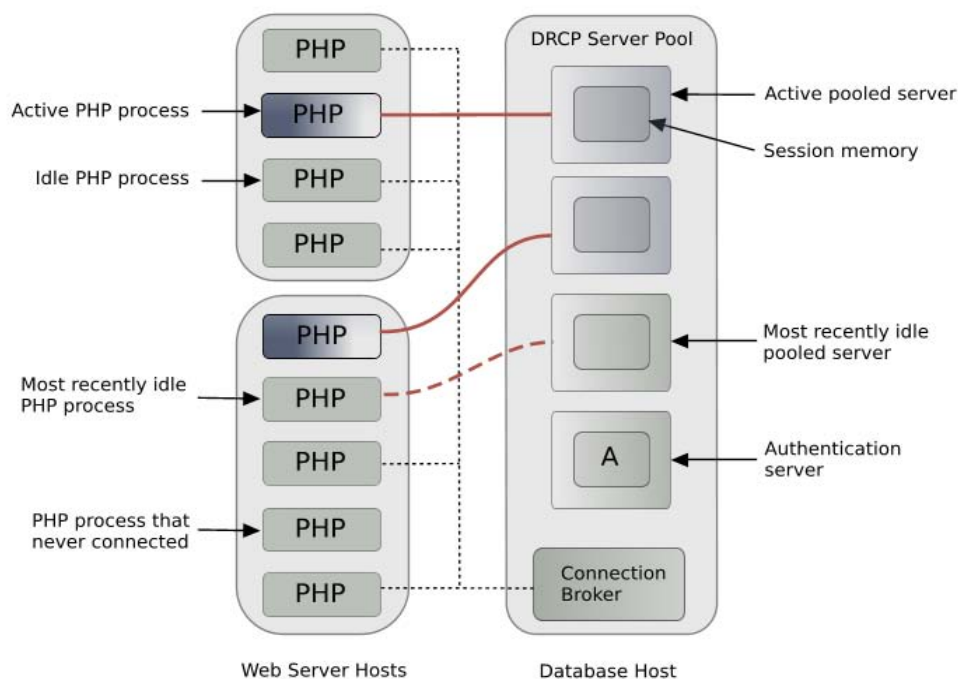


Figure 94: DRCP Architecture.

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 11: 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.

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.
Idle PHP connections hold a server process and session resources.	Idle PHP connections hold session resources but not a server process.	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 causes the session to be destroyed and the pooled server to be 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 in use 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 PHP OCI8 extension has three functions for connecting to a database: `oci_connect()`, `oci_new_connect()`, and `oci_pconnect()`. The implementation of these functions was reworked in OCI8 1.3 and all benefit from using DRCP. Table 12 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.

Table 12: Behavior of OCI8 connection functions for Dedicated and Pooled Servers.

OCI8 Function	With Dedicated Servers	With Pooled Servers
<code>oci_connect()</code>	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.
<code>oci_new_connect()</code>	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.
<code>oci_pconnect()</code>	Creates a persistent PHP connection which is cached in the PHP process. The connection is not closed at the end of the script, and the server and session are available for reuse by any subsequent <code>oci_pconnect()</code> call passing the same credentials.	Creates a persistent PHP connection. Calling <code>oci_close()</code> releases the connection and returns the server with its session intact to 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. Subsequent calls to <code>oci_pconnect()</code> reuse the existing network connection to quickly get a server and session from the pool.

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.

PHP Scalability and High Availability

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 applications with a large number of connections. Shared servers are useful for a medium number of connections and 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 can be useful when any of the following apply:

- A large number of connections need to be supported with minimum memory usage on the database host.
- 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.
- There are multiple web servers and web server hosts.
- Connections look identical in terms of session settings, for example date format settings and PL/SQL package state.

DRCP provides the following advantages:

- It enables resource sharing among multiple middle-tier client applications.
- It improves scalability of databases and applications by reducing resource usage.

For persistent PHP connections, normal dedicated servers can be fastest. There is no broker or dispatcher overhead. The server 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 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 13.

Table 13: 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

	Dedicated Servers	Shared Servers	Pooled Servers
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 bequeath connections.

With Oracle 11.1, the client result cache feature does not work with DRCP.

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 userid. DRCP also further partitions the pool into logical groups or “connection classes”. A connection class is a user chosen name set 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.

For example, applications in a suite called RPT may be willing to share pooled servers between themselves but not with an application suite called HR. The different connection classes and resulting logical partitioning of the DRCP server pool is shown in Figure 95. Connections with the same username and connection class from any host will share the same sub-pool of servers.

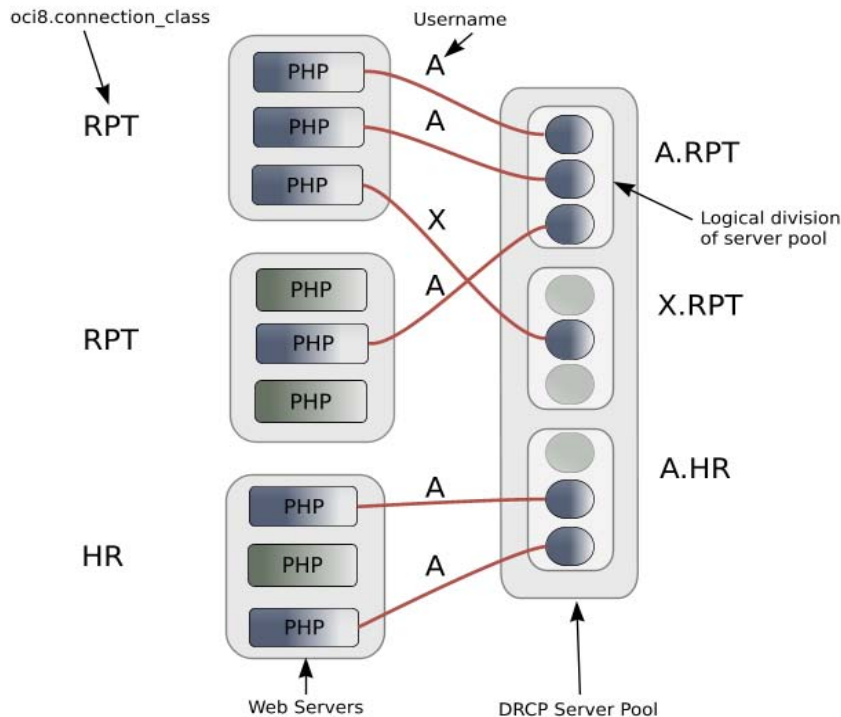


Figure 95: The DRCP pool is logically partitioned by username and connection class.

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 server will migrate to the new class. 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.

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 they remain connected to the connection broker over multiple script executions and can 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,
        incrsz            => 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 will have `maxsize` 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 DRCP configuration options are described in Table 11.

Table 14: DRCP Configuration Options.

DRCP Option	Description
<code>pool_name</code>	The pool to be configured. Currently the only supported name is the default value <code>SYS_DEFAULT_CONNECTION_POOL</code> .
<code>minsize</code>	Minimum number of pooled servers in the pool. The default is 4.
<code>maxsize</code>	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 default is 40.
<code>incrsize</code>	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.
<code>session_cached_cursors</code>	Indicates to turn on <code>SESSION_CACHED_CURSORS</code> 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.
<code>inactivity_timeout</code>	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.
<code>max_think_time</code>	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.
<code>max_use_session</code>	Maximum number of times a server can be taken and released to the pool before it is flagged for restarting. The default is 500000.
<code>max_lifetime_session</code>	Time to live for a pooled server before it is restarted. The default is 86400 seconds.
<code>num_cbrok</code>	The number of connection brokers that are created to handle connection requests. Note: this can only be set with <code>alter_param()</code> . The default is 1.
<code>maxconn_cbrok</code>	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. Note: this can only be set with <code>alter_param()</code> . The default is 40000.

Note: 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.

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.

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. This is necessary with OCI8 1.3 to avoid query errors. It also enables `LOGON` trigger support. This patch is not needed with 11.1.0.7 onwards.

Configuring PHP for DRCP

PHP must be built with the DRCP-enabled OCI8 extension. Download OCI8 1.3 from PECL, extract it and use it to replace the existing `ext/oci8` directory in PHP. Configure, build and install PHP as normal. Alternatively use the `pecl install` command to build and install the OCI8 as a shared module. PHP 5.3 and PHP 6, neither of which has been released, contain the new extension by default.

The OCI8 1.3 extension can be used with Oracle client libraries version 9.2 and higher, however DRCP functionality is only available when PHP is linked with Oracle Database 11g client libraries and connected to Oracle Database 11g.

Once installed, use PHP's `phpinfo()` function to verify that OCI8 1.3 has been loaded.

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.

Table 15: Existing `php.ini` parameters for persistent connections.

Php.ini Parameter	Behavior with DRCP
<code>oci8.persistent_timeout</code>	At the timeout of an idle PHP connection, PHP will close the Oracle connection to the broker. The default is no timeout.

Php.ini Parameter	Behavior with DRCP
<code>oci8.max_persistent</code>	The maximum number of unique persistent connections that PHP 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> .
<code>oci8.ping_interval</code>	The existing recommendation to set it to <code>-1</code> 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. The default is 60 seconds.

With OCI8 1.3, `oci8.ping_interval` is also used for non-persistent connections when pooled servers are used. This helps protect against the possibility of unusable connections being returned to PHP when there has been a database instance or node failure.

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 shipping 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, connections will not succeed and an error will be returned to PHP.

Closing Connections

With the new version of the OCI8 extension, closing a persistent connection now releases the server to the pool allowing optimal pool usage.

PHP scripts that do not currently use `oci_close()` should be examined to see if they can use it to return connections to the pool, allowing maximum use of pooled servers:

```
<?php
// 1. Do some database operations
$conn = oci_pconnect('myuser', 'mypassword', '//myhost/sales:POOLED');
. . .

oci_commit($conn);
oci_close($conn); // Release the connection to the DRCP pool

// 2. Do some non-database work
. . .

// 3. Do some more database operations
$conn = oci_pconnect('myuser', 'mypassword', '//myhost/sales:POOLED');
. . .

oci_commit($conn);
oci_close($conn);

?>
```

Remember to free statement and other resources that increase the reference count on the PHP connection and will stop a DB connection 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. With OCI8 1.3, `oci_close()` on a persistent connection rolls back any uncommitted transaction. Also the extension will do a roll back 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 existing `php.ini` parameter `oci8.old_oci_close_semantics` to `On`.

LOGON and LOGOFF Triggers with DRCP

LOGON triggers are efficient for setting common session attributes such as date formats. **LOGON** triggers are also 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()` creates the session, and the session will be reused by subsequent persistent connections.

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. This recommendation is also true for persistent connections with dedicated or shared servers, and is the existing recommendation for earlier releases of the OCI8 extension.

Database actions that must be performed exactly once per OCI8 connection call should be explicitly executed in the PHP script.

It is not possible to depend on **LOGON** triggers for tracking PHP OCI8 connect calls. The caching, pooling, timing out and recreation of sessions and connections with or without DRCP or the new extension 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.

LOGOFF triggers do not fire for pooled servers. For non-pooled servers there is no change in OCI 1.3: **LOGOFF** triggers fire when connections are closed. 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 is when the web server process terminates or restarts.

Changing Passwords with DRCP Connections

In general, PHP applications that change passwords should avoid using persistent connections because connections stored persistently in PHP may allow application connections to succeed using the old password. 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:

- `V$PROCESS`
- `V$SESSION`
- `DBA_CPOOL_INFO`
- `V$CPOOL_STATS`
- `V$CPOOL_CC_STATS`

The DRCP statistics are reset each time the pool is started.

For RAC, there are `GV$CPOOL_STATS` and `GV$CPOOL_CC_STATS` 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.

V\$PROCESS and V\$SESSION Views

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)
```

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.

DBA_CPOOL_INFO View

[DBA_CPOOL_INFO](#) displays configuration information about all DRCP pools in the database. The columns are equivalent to the `dbms_connection_pool.configure_pool()` settings described in Table 11, 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](#). For example, to check whether the pool has been started and what the maximum number of pooled servers is set to:

```
SQL> select connection_pool, status, maxsize
      from dba_cpool_info;

CONNECTION_POOL          STATUS          MAXSIZE
-----
SYS_DEFAULT_CONNECTION_POOL  ACTIVE          40
```

V\$CPOOL_STATS View

[V\\$CPOOL_STATS](#) displays information about the DRCP statistics for an instance.

Table 16: [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.

PHP Scalability and High Availability

Column	Description
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.

The `V$CPPOOL_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. Increasing the number of pooled servers would reduce the number of waits:

```
SQL> select num_requests,num_hits,num_misses,num_waits
       from v$cpool_stats
       where pool_name = 'SYS_DEFAULT_CONNECTION_POOL';
```

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

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

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 6, 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
HR.OCI:SP:SdjxIx1Ufz	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. This view shows an application known as *MYPHPAPP* using the pool effectively.

The last line of the example output shows a system generated class name for an application that did not explicitly set [oci8.connection_class](#). Pooling is not effectively used in this case. Such an entry could be an indication that a *php.ini* file is mis-configured.

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.

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 which time the application may have exceeded PHP's maximum allowed execution time.

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:

```
SQL> execute dbms_service.modify_service(  
        service_name => 'SALES',  
        aq_ha_notifications => TRUE);
```

Configuring PHP for FAN

With the OCI8 1.3 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 is linked with Oracle Database 10g Release 2 or 11g libraries and connecting to Oracle Database 10g Release 2 or 11g.

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 1.3 extension will detect the connection failure and be able to 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 connection errors so it can identify if 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`. Other connection related errors are shown in the example, but errors including standard database errors may be returned, depending on timing.

```
function isConnectionError($err)  
{  
    switch($err) {  
        case 378: /* buffer pool param incorrect */  
        case 602: /* core dump */  
        case 603: /* fatal error */  
        case 609: /* attach failed */  
        case 1012: /* not logged in */  
        case 1033: /* init or shutdown in progress */  
        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;
}

$conn = doConnect();
$error = doSomeWork($conn);
if (isConnectionError($error)) {
    // reconnect, find what was committed, and retry
    $conn = doConnect();
    $error = checkApplicationStateAndContinueWork($conn);
}
if ($error) {
    // end the application
    handleError($error);
}

```

RAC Connection Load Balancing with PHP

PHP OCI8 1.3 will automatically balance new connections across RAC instances with Oracle's Connection Load Balancing (CLB) to use resources efficiently.

It is recommended to use FAN and CLB together.

To enable CLB, the database service must be modified using `dbms_service` to send load events. Set the `clb_goal` parameter to `CLB_GOAL_SHORT` or `CLB_GOAL_LONG`. For example:

```

SQL> execute dbms_service.modify_service(
           service_name => 'SALES',
           clb_goal => dbms_service.clb_goal_long);

```

Table 17: CLB `clb_goal` parameter values.

Parameter Value	Parameter Description
CLB_GOAL_SHORT	Use for connection load balancing method for applications that have short-lived connections. This uses load balancing advisories if the <code>goal</code> parameter is not <code>GOAL_NONE</code> , otherwise a CPU utilization is used.
CLB_GOAL_LONG	Use for applications that have long-lived connections. This uses a simple metric to distribute load. This is the default.

No PHP script changes are needed. The connection balancing is handled transparently by the Oracle Net listener.

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.

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.

The character set can also be passed as a parameter to the OCI8 connection functions. Doing this is recommended for performance reasons, even if `NLS_LANG` is also set.

If the character set used by PHP does not match the character set used by the database, Oracle will try to convert when data is inserted and queried. This may reduce 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 *Configuring Apache HTTP Server on Linux* in the Apache chapter discusses how environment variables can be set for Apache.

When Zend Core for Oracle is installed on Apache, you can set `NLS_LANG` in `/etc/profile`:

Globalization

```
export NLS_LANG GERMAN_GERMANY.AL32UTF8
```

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 the default date format will have changed.

Caution: When changing the globalization settings for a persistent connection, the next time the connection is used, the altered 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
```

Manipulating Strings

PHP was designed to work with the ISO-8859-1 character set. To handle other character sets, there are several extensions that can be used. The common extensions are `mbstring` and `iconv`. Recently a new intl package has been added to PECL. It implements some International Components for Unicode (ICU) functionality and is likely to become popular.

To enable the `mbstring` extension in PHP using the Zend Core for Oracle Administration Console:

1. Open the Zend Core for Oracle Administration Console by entering the URL in a web browser:
`http://machine_name/ZendCore`
2. Login and navigate to **Configuration > Extensions > Zend Core Extensions**.
3. Enable the **mbstring** extension.
4. Save the configuration.
5. Restart Apache.

When you have enabled the `mbstring` extension and restarted the web server, several configuration options become available in the Zend Core for Oracle Administration Console. Refresh the browser on the Extension Configuration page to see these `mbstring` configuration options.

You can change the behavior of the standard PHP string functions by setting `mbstring.func_overload` to one of the *Overload* settings. For more information, see the PHP `mbstring` reference manual at <http://www.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.

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 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"]
```

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.

Globalization

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	PORTUGAL
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.

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*:

```
default_charset = UTF-8
```

To enable character set encoding of HTML pages using the Zend Core for Oracle Administration Console.

1. Open the Zend Core for Oracle Administration Console by entering the URL in a web browser:
`http://<machine_name>/ZendCore`
2. Login and navigate to **Configuration > PHP > Data Handling**.
3. Enter [UTF-8](#) for the **default_charset** parameter.
4. Save the configuration.
5. Restart Apache.

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:

Globalization

- 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 '12/11/05' implies '11th December 2005' in the United States, whereas in the United Kingdom it means '12th November 2005'. 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. This can be a particular problem in PHP when database numbers are fetched as PHP strings.

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. The following sections provide examples of locale-sensitive operations in SQL.

Oracle Number Formats

OCI8 fetches numbers as PHP strings. The conversion is done by Oracle and can be customized. It is possible to lose decimal places or get errors in PHP when it internally converts strings not using the US formatting conventions.

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
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 88: numformat.php

```
<?php
```

```
$c = oci_connect('hr', 'hrpwd', 'localhost/XE');
```

Globalization

```
$s = oci_parse($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:

```
# export NLS_NUMERIC_CHARACTERS="., "
# apachectl start
```

or 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 it in PHP is likely to be the slowest of the methods.

The tip *Do Not Set the Date Format Unnecessarily* in the chapter *Connecting to Oracle Using OCI8* shows how an `ALTER SESSION` command can be used in a database logon trigger.

Oracle Date Formats

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 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 the shell that starts your web server or PHP executable:

```

# export NLS_DATE_FORMAT='YYYY-MM-DD HH24:MI:SS'
# apachectl start

```

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:

Globalization

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 89: dateformat.php

```
<?php

$c = oci_connect('hr', 'hrpwd', '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"
    }
}
```

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

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 the language-specific database error messages will be returned to 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 XE, see *Working in a Global Environment* in the *Oracle Database Express Edition 2 Day Developer Guide*.

TESTING PHP AND THE OCI8 EXTENSION

This Chapter discusses installing and running the PHP source code tests for OCI8 on Linux. The PHP source code includes tests for all the core functionality and extensions. You should run the tests after building PHP on Linux.

You should also 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. 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. It is also 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 before each final PHP release. This ensures PHP continues doing what you need.

Running OCI8 Tests

The tests in *php-5.2.7/ext/oci8/tests* verify the behavior of the OCI8 extension. For the tests to run successfully some configuration is needed.

To run the OCI8 tests:

1. Set the Oracle connection details, either by editing the *details.inc* test configuration file, or by setting environment variables. To change the configuration file:

Edit *php-5.2.7/ext/oci8/tests/details.inc* and set the Oracle *system* user password for your database:

```
$user = "system";  
$password = "systempwd";
```

In older versions of PHP these variables are located in *connect.inc*.

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 extra privileges.

At the end of *details.inc*, set the connection string for the database:

```
$dbase = "localhost/XE";
```

If PHP is running on the same machine as the database, then also set:

```
$oracle_on_localhost = TRUE;
```

This specifies to test functionality where the Oracle database directly accesses files created by PHP. If the database and PHP are not using the same file system, this is not possible and the variable should be left `FALSE`.

Testing PHP and the OCI8 Extension

Finally, with OCI8 1.3, and you are using Oracle Database 11g Connection Pooling, set:

```
$test_drpc = TRUE
```

To use DRCP, the pool must be enabled and the connection string must specify that a pooled database server should be used.

Alternatively, with PHP 5.2.4 onwards, environment variables can be set instead of editing *details.inc*. At the shell, set these environment variables.

```
$ export PHP_OCI8_TEST_USER=system
$ export PHP_OCI8_TEST_PASS=systempwd
$ export PHP_OCI8_TEST_DB=localhost/XE
$ export PHP_OCI8_TEST_DB_ON_LOCALHOST=TRUE
$ export PHP_OCI8_TEST_DRCP=FALSE
```

The variables correspond to the settings described above in section 1.

2. Check that `variables_order` has `E` in your *php.ini*, for example:

```
variables_order = "EGPCS"
```

Without this flag, the Oracle environment variables are not propagated through the test system and tests fail to connect.

3. Set any necessary Oracle environment variables in your shell. For example, for PHP linked with Oracle Database XE enter:

```
$ export OH=/usr/lib/oracle/xep/app/oracle/product/10.2.0/server
$ . $OH/bin/oracle_env.sh
```

Note the space after the full stop. For other database versions run `oraenv` and enter the identifier of your database:

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

If Oracle is on a different machine, you may manually need to set the environment variables set by these scripts.

4. Run PHP's test suite with:

```
$ cd php-5.2.7
$ 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-5.2.7/sapi/cli/php
PHP_SAPI : cli
PHP_VERSION : 5.2.7
ZEND_VERSION: 2.2.0
PHP_OS    : Linux - Linux def 2.6.24-19-generic
INI actual : /usr/local/apache/conf/php.ini
More .INIs :
CWD       : /home/myhome/php-5.2.7
Extra dirs :
=====
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 failing tests is given at the completion of the tests.

Running a Single Test

To run only one or two tests, call the *run-tests.php* script directly and pass the test names as parameters. For example, to run the *demotest.phpt* script, do the following:

```

$ export TEST_PHP_EXECUTABLE=/home/myhome/php-5.2.7/sapi/cli/php
$ /home/myhome/php-5.2.7/sapi/cli/php run-tests.php \
>   ext/oci8/tests/demotest.phpt

```

The `TEST_PHP_EXECUTABLE` variable contains the PHP binary with which to test *demotest.phpt*. In the example, the same PHP binary is used to run the controlling *run-tests.php* script, 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 *php-5.2.7/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

Testing PHP and the OCI8 Extension

Occasionally a few tests are known to fail. These might be for unfixed bugs, or where the PHP test infrastructure doesn't practically allow tests to accept differences in versions of Oracle. If you use the latest OCI8 extension with an older version of PHP, differences in PHP's `var_dump()` output will make tests appear to fail.

Creating OCI8 Tests

To add a new OCI8 test, create a *phpt* file in *php-5.2.7/ext/oci8/tests* using the test file format. When you run `make test` the new file is automatically run. For example, create *demotest.phpt*:

Script 90: demotest.phpt

```
--TEST--
Demo to test the Test system
--SKIPIF--
<?php
if (!extension_loaded('oci8')) die("skip no oci8 extension");
?>
--FILE--
<?php
require dirname(__FILE__).'/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 *php-5.2.7/ext/oci8/tests/connect.inc*. It includes *details.inc*, connects to Oracle, and returns the connection resource in `$c`.

In PHP 5.3, you can use PHP's inbuilt constant `__DIR__` instead of `dirname(__FILE__)` to locate the correct included files, but this will limit test compatibility.

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 running the test directly in PHP a little cleaner. Some of the PHPT content is shown, but only the actual, and not the expected output is displayed. This can make it easier to quickly validate tests:

```
$ php demotest.phpt
--TEST--
Demo to test the Test system
--SKIPIF--
--FILE--
array(1) {
  ["USER"]=>
  array(1) {
    [0]=>
    string(6) "SYSTEM"
  }
}
Done
===DONE===
```

The page *Writing Tests* at <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.

Writing good units tests is an art. Making a test portable, accurate, simple and self-diagnosing requires fine judgment and tidy programming. Application level testing brings even more challenges. There are several PHP test frameworks with sophisticated features that might be more suited to your application test suites. They include *PHPUnit* and *SimpleTest*.

OCI8 Test Helper Scripts

Along with *connect.inc* and *details.inc*, there are several useful scripts in *php-5.2.7/ext/oci8/tests* for creating and dropping basic tables and types:

- *create_table.inc*
- *create_type.inc*
- *drop_table.inc*
- *drop_type.inc*

Each test should create any objects it needs and drop them at the end of the test.

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
ORA-12520: TNS:listener could not find available handler for requested type of
server
```

The solution is to configure the database to suit the load. For the errors above, the number of “processes” that Oracle can handle needs to be increased.

Testing PHP and the OCI8 Extension

To increase the number of processes in Oracle:

1. You may need to `su` as the *oracle* user so you have operating system privileges to start SQL*Plus:

```
$ su - oracle
Password:
```

2. Set the Oracle environment variables needed by SQL*Plus, for example:

```
$ export ORACLE_HOME=/usr/lib/oracle/xe/app/oracle/product/10.2.0/server
$ . $ORACLE_HOME/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 Buffers 2932736 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
```


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 91: trace.php

```
<?php
oci_internal_debug(1);          // turn on tracing

$conn = oci_connect("hr", "hrpwd", "localhost/XE");
$s = oci_parse($conn, "insert into testtable values ('my data')");
oci_execute($s, OCI_DEFAULT);  // do not auto-commit

?>
```

You get output like:

```
OCI8 DEBUG: OCINlsEnvironmentVariableGet at (/php/ext/oci8/oci8.c:1822)
OCI8 DEBUG L1: Got NO cached connection at (/php/ext/oci8/oci8.c:1867)
OCI8 DEBUG: OCIEnvNlsCreate at (/php/ext/oci8/oci8.c:2772)
OCI8 DEBUG: OCIHandleAlloc at (/php/ext/oci8/oci8.c:2632)
OCI8 DEBUG: OCIHandleAlloc at (/php/ext/oci8/oci8.c:2644)
OCI8 DEBUG: OCISessionPoolCreate at (/php/ext/oci8/oci8.c:2662)
OCI8 DEBUG: OCIAttrSet at (/php/ext/oci8/oci8.c:2674)
OCI8 DEBUG L1: create_spool: (0x8959b70) at (/php/ext/oci8/oci8.c:2690)
OCI8 DEBUG L1: using shared pool: (0x8959b70) at (/php/ext/oci8/oci8.c:2972)
OCI8 DEBUG: OCIHandleAlloc at (/php/ext/oci8/oci8.c:2983)
OCI8 DEBUG: OCIHandleAlloc at (/php/ext/oci8/oci8.c:2993)
OCI8 DEBUG: OCIAttrSet at (/php/ext/oci8/oci8.c:3002)
OCI8 DEBUG: OCIAttrSet at (/php/ext/oci8/oci8.c:3014)
OCI8 DEBUG: OCIAttrGet at (/php/ext/oci8/oci8.c:3026)
OCI8 DEBUG: OCIAttrGet at (/php/ext/oci8/oci8.c:3027)
OCI8 DEBUG L1: (numopen=0) (numbusy=0) at (/php/ext/oci8/oci8.c:3029)
OCI8 DEBUG: OCISessionGet at (/php/ext/oci8/oci8.c:3040)
OCI8 DEBUG: OCIAttrGet at (/php/ext/oci8/oci8.c:3055)
OCI8 DEBUG: OCIAttrGet at (/php/ext/oci8/oci8.c:3057)
OCI8 DEBUG: OCIContextGetValue at (/php/ext/oci8/oci8.c:3059)
OCI8 DEBUG: OCIContextGetValue at (/php/ext/oci8/oci8.c:3154)
OCI8 DEBUG: OCIMemoryAlloc at (/php/ext/oci8/oci8.c:3161)
OCI8 DEBUG: OCIContextSetValue at (/php/ext/oci8/oci8.c:3175)
```

Tracing OCI8 Internals

```
OCI8 DEBUG: OCIAttrSet at (/php/ext/oci8/oci8.c:3085)
OCI8 DEBUG L1: New Non-Persistent Connection address: (0x88e0adc) at
(/php/ext/oci8/oci8.c:2093)
OCI8 DEBUG L1: num_persistent=(0), num_links=(1) at (/php/ext/oci8/oci8.c:2095)
OCI8 DEBUG: OCIHandleAlloc at (/php/ext/oci8/oci8_statement.c:57)
OCI8 DEBUG: OCIStmtPrepare2 at (/php/ext/oci8/oci8_statement.c:72)
OCI8 DEBUG: OCIAttrSet at (/php/ext/oci8/oci8_statement.c:122)
OCI8 DEBUG: OCIAttrGet at (/php/ext/oci8/oci8_statement.c:396)
OCI8 DEBUG: OCIStmtExecute at (/php/ext/oci8/oci8_statement.c:420)
OCI8 DEBUG: OCIStmtRelease at (/php/ext/oci8/oci8_statement.c:723)
OCI8 DEBUG: OCIHandleFree at (/php/ext/oci8/oci8_statement.c:731)
OCI8 DEBUG: OCITransRollback at (/php/ext/oci8/oci8.c:2167)
OCI8 DEBUG: OCISessionRelease at (/php/ext/oci8/oci8.c:2330)
OCI8 DEBUG: OCIHandleFree at (/php/ext/oci8/oci8.c:2214)
OCI8 DEBUG: OCIHandleFree at (/php/ext/oci8/oci8.c:2217)
OCI8 DEBUG: OCISessionPoolDestroy at (/php/ext/oci8/oci8.c:3122)
OCI8 DEBUG: OCIHandleFree at (/php/ext/oci8/oci8.c:3126)
OCI8 DEBUG: OCIHandleFree at (/php/ext/oci8/oci8.c:3130)
OCI8 DEBUG: OCIHandleFree at (/php/ext/oci8/oci8.c:3134)
OCI8 DEBUG: OCIHandleFree at (/php/ext/oci8/oci8.c:1114)
OCI8 DEBUG: OCIHandleFree at (/php/ext/oci8/oci8.c:1119)
```

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_DEFAULT` 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.

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 lists the *php.ini* parameters for the OCI8 extension. Discussion of their use is covered in previous chapters.

The parameter values can be changed in the PHP configuration file *php.ini*, for example:

```
oci8.default_prefetch = 75
```

Variables can also be set in *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* and the current values of the OCI8 parameters can be found by running the command line PHP executable with the `-i` option, or by loading this script in a browser:

Script 92: *phpinfo.php*

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

If you are using Windows Vista, remember to edit *php.ini* using administrative privileges.

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 <code>ini_set()</code> . Introduced in OCI8 1.3.

OCI8 php.ini Parameters

Name	Default	Valid Range	Description
<code>oci8.events</code>	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 be configured to post events. Introduced in OCI8 1.3.
<code>oci8.max_persistent</code>	-1	≥ -1 -1 means no limit	Maximum number of persistent connections each PHP process caches.
<code>oci8.old_oci_close_semantics</code>	Off	Off or On	Toggles whether <code>oci_close()</code> uses the old behavior, which was a “no-op”.
<code>oci8.persistent_timeout</code>	-1	> -1 -1 means no timeout	How many seconds a persistent connection is allowed to remain idle before being terminated.
<code>oci8.ping_interval</code>	60	≥ 0 -1 means no extra checking	How many seconds a persistent connection can be unused before an extra check at connection verifies the database connection is still valid.
<code>oci8.privileged_connect</code>	Off	Off or On	Toggles whether SYSDBA and SYSOPER connections are permitted.
<code>oci8.statement_cache_size</code>	20	> 0	Improves database performance by caching the given number of SQL statements in PHP.

OCI8 FUNCTION NAMES IN PHP 4 AND PHP 5

In PHP 5 several extensions including OCI8 underwent name standardization. PHP 4 functions like `OCILogin()` became `oci_connect()`, `OCIParse()` became `oci_parse()`, and so on. The old OCI8 names still exist as aliases, so PHP 4 scripts do not necessarily need to be changed. PECL OCI8 releases from 1.1 onwards have the name standardization change.

Note: The OCI8 1.3 extension builds and runs with PHP 4 too. If you are 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 do this are given in this book.

One side effect of renaming the function names is that user contributed comments in the PHP manual (<http://www.php.net/oci8>) are found in two different places. The comments are either on the page for the old syntax, or on the page for the new syntax. Where a function has an alias, check both manual pages for user tips.

Function names in PHP are case insensitive and it is common to see the PHP 4 names written with a capitalized prefix.

Table 21 shows the OCI8 function names in PHP 4 and PHP 5 where names or functionality differ.

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>ocilogon()</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>

OCI8 Function Names in PHP 4 and PHP 5

Operation	Action	PHP 4 Name	PHP 5 Name
Parsing	Parse statement	<code>ociparse()</code>	<code>oci_parse()</code>
Binding	Bind variable	<code>ocibindbyname()</code>	<code>oci_bind_by_name()</code>
	Bind array	Not available	<code>oci_bind_array_by_name()</code>
Defining	Define output variables	<code>ocidefinebyname()</code>	<code>oci_define_by_name()</code>
Execution	Execute statement	<code>ociexecute()</code>	<code>oci_execute()</code>
Fetching	Fetch row	<code>ocifetch()</code>	<code>oci_fetch()</code>
	Fetch row	<code>ocifetchinto()</code>	<code>oci_fetch_array()</code> <code>oci_fetch_row()</code> <code>oci_fetch_assoc()</code> <code>oci_fetch_object()</code>
	Fetch all rows	<code>ocifetchstatement()</code>	<code>oci_fetch_all()</code>
	Fetch column	<code>ociresult()</code>	<code>oci_result()</code>
	Is the column NULL?	<code>ocicolumnisnull()</code>	<code>oci_field_is_null()</code>
	Cancel Fetch	<code>ocicancel()</code>	<code>oci_cancel()</code>
Transaction Management	Commit	<code>ocicommit()</code>	<code>oci_commit()</code>
	Rollback	<code>ocirollback()</code>	<code>oci_rollback()</code>
Descriptors		<code>ocinewdescriptor()</code>	<code>oci_new_descriptor()</code>
		<code>ocifreedesc()</code>	<code>oci_free_descriptor()</code>
Error Handling		<code>ocierror()</code>	<code>oci_error()</code>

Operation	Action	PHP 4 Name	PHP 5 Name
Long Objects (LOBs) Note: Methods on a collection object can be used in addition to these functions.		<code>ocisavelob()</code>	<code>oci_lob_save()</code>
		<code>ocisavelobfile()</code>	<code>oci_lob_import()</code>
		<code>ociwritelobtofile()</code>	<code>oci_lob_export()</code>
		<code>ociwritetemporarylob()</code>	<code>OCI-Lob->writeTemporary()</code>
		<code>ociloadlob()</code>	<code>oci_lob_load()</code>
		<code>ocicloselob()</code>	<code>OCI-Lob->close()</code>
Collections Note: Methods on a collection object can be used in addition to these functions.		<code>ocinewcollection()</code>	<code>oci_new_collection()</code>
		<code>ocifreecollection()</code>	<code>oci_free_collection()</code>
		<code>ocicollappend()</code>	<code>oci_collection_append()</code>
		<code>ocicollgetelem()</code>	<code>oci_collection_element_get()</code>
		<code>ocicollassign()</code>	<code>OCI-Collection->assign()</code>
		<code>ocicollassignelem()</code>	<code>oci_collection_element_assign()</code>
		<code>ocicollsize()</code>	<code>oci_collection_size()</code>
		<code>ocicollmax()</code>	<code>oci_collection_max()</code>
		<code>ocicolltrim()</code>	<code>oci_collection_trim()</code>
Metadata	Statement type	<code>ocistatementtype()</code>	<code>oci_statement_type()</code>

OCI8 Function Names in PHP 4 and PHP 5

Operation	Action	PHP 4 Name	PHP 5 Name
	Name of result column	<code>ocicolumnname()</code>	<code>oci_field_name()</code>
	Size of result column	<code>ocicolumnsize()</code>	<code>oci_field_size()</code>
	Datatype of result column	<code>ocicolumntype()</code>	<code>oci_field_type()</code>
	Datatype of result column	<code>ocicolumntyperaw()</code>	<code>oci_field_type_raw()</code>
	Precision of result column	<code>ocicolumnprecision()</code>	<code>oci_field_precision()</code>
	Scale of result column	<code>ocicolumnscale()</code>	<code>oci_field_scale()</code>
	Number of rows affected	<code>ocirowcount()</code>	<code>oci_num_rows()</code>
	Number of columns returned	<code>ocinumcols()</code>	<code>oci_num_fields()</code>
Changing Password		<code>ocipasswordchange()</code>	<code>oci_password_change()</code>
Tracing		<code>ociinternaldebug()</code>	<code>oci_internal_debug()</code>
Server Version		<code>ociserverversion()</code>	<code>oci_server_version()</code>
Tuning		<code>ocisetprefetch()</code>	<code>oci_set_prefetch()</code> (and new <i>php.ini</i> parameter)

THE OBSOLETE ORACLE EXTENSION

This Appendix discusses the obsolete Oracle PHP extension. Very rarely you might come across PHP scripts that use the early Oracle extension. This extension is obsolete and is no longer included with PHP. The functionality it offered was limited, and upgrading to the new OCI8 extension might be as simple as enabling the newer OCI8 extension in the PHP binary, and changing the `ora_` function calls in your scripts. Paying some attention to transaction management and connection handling is wise to make sure all your data is committed when you expect it to be.

This Chapter gives a comparison of the obsoleted Oracle PHP extension and the current OCI8 extension.

Oracle and OCI8 Comparison

Table 22 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	<code>ora_logon()</code>	<code>oci_connect()</code>
	Open new connection	Not available	<code>oci_new_connect()</code>
	Persistent connection	<code>ora_plogon()</code>	<code>oci_pconnect()</code> (and new <i>php.ini</i> parameters)
	Close connection	<code>ora_logoff()</code>	<code>oci_close()</code>
Cursor	Open cursor	<code>ora_open()</code>	<code>oci_new_cursor()</code>
	Close cursor	<code>ora_close()</code>	<code>oci_free_statement()</code>
Parsing	Parse statement	<code>ora_parse()</code>	<code>oci_parse()</code>
Binding	Bind variable	<code>ora_bind()</code>	<code>oci_bind_by_name()</code>
	Bind array	Not applicable	<code>oci_bind_array_by_name()</code>
Execution	Execute statement	<code>ora_exec()</code>	<code>oci_execute()</code>

The Obsolete Oracle Extension

Operation	Action	ORA function (obsolete)	OCI8 function
	Prepare, execute and fetch	<code>ora_do()</code>	<code>oci_parse()</code> <code>oci_execute()</code> Followed by one of: <code>oci_fetch_all()</code> <code>oci_fetch_array()</code> <code>oci_fetch_assoc()</code> <code>oci_fetch_object()</code> <code>oci_fetch_row()</code> <code>oci_fetch()</code>
Fetching	Fetch row	<code>ora_fetch()</code>	<code>oci_fetch()</code>
	Fetch row	<code>ora_fetch_into</code>	<code>oci_fetch_array()</code> <code>oci_fetch_row()</code> <code>oci_fetch_assoc()</code> <code>oci_fetch_object()</code>
	Fetch all rows	Not applicable	<code>oci_fetch_all()</code>
	Fetch column	<code>ora_getcolumn()</code>	<code>oci_result()</code>
	Is the column NULL?	Not applicable	<code>oci_field_is_null()</code>
	Cancel Fetch	Not applicable	<code>oci_cancel()</code>
Transaction Management	Commit	<code>ora_commit()</code>	<code>oci_commit()</code>
	Commit mode	<code>ora_commiton()</code> <code>ora_commitoff()</code>	Pass OCI_DEFAULT flag to <code>oci_execute()</code>
	Rollback	<code>ora_rollback()</code>	<code>oci_rollback()</code>
Error Handling		<code>ora_error()</code> <code>ora_errorcode()</code>	<code>oci_error()</code>

Operation	Action	ORA function (obsolete)	OCI8 function
Long Objects (LOBs)		Not applicable	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 applicable	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 applicable	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	<code>ora_columnsize()</code>	<code>oci_field_size()</code>
	Datatype of result column	<code>ora_columntype()</code>	<code>oci_field_type()</code> <code>oci_field_type_raw()</code>
	Precision of result column	Not applicable	<code>oci_field_precision()</code>
	Scale of result column	Not applicable	<code>oci_field_scale()</code>
	Number of rows effected	<code>ora_numrows()</code>	<code>oci_num_rows()</code>
	Number of columns returned	<code>ora_numcols()</code>	<code>oci_num_fields()</code>
Changing Password		Not applicable	<code>oci_password_change()</code>
Tracing		Not applicable	<code>oci_internal_debug()</code>
Server Version		Not applicable	<code>oci_server_version()</code>
Tuning		Not applicable	<code>oci_set_prefetch()</code> (and new <i>php.ini</i> parameter)

RESOURCES

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 online at:

<http://www.oracle.com/technology/tech/php/pdf/underground-php-oracle-manual.pdf>

General Information and Forums

PHP Developer Center on Oracle Technology Network (OTN)

<http://www.oracle.com/technology/tech/php/index.html>

OTN PHP Discussion Forum

<http://www.oracle.com/technology/forums/php.html>

Blog: Christopher Jones on OPAL

<http://blogs.oracle.com/opal/>

Blog: Alison Holloway on PHP

<http://blogs.oracle.com/alison/>

AskTom

General Oracle language and application design help

<http://asktom.oracle.com/>

Oracle Metalink

Oracle Support web site

<http://metalink.oracle.com/>

Oracle's Free and Open Source Software

<http://oss.oracle.com/>

Oracle Documentation

Oracle Documentation Portal

<http://tahiti.oracle.com/>

Resources

Oracle Call Interface Programmer's Guide

Oracle Database 11g Release 1 (11.1)

http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/toc.htm

Oracle Database Express Edition 2 Day Plus PHP Developer Guide

http://download.oracle.com/docs/cd/B25329_01/doc/appdev.102/b25317/toc.htm

Oracle Database 10g Express Edition Documentation

Oracle Database 10g Express Edition (XE) Release 2 (10.2)

<http://www.oracle.com/pls/xe102/homepage>

Oracle Database Express Edition, 2 Day Plus Locator Developer Guide

Oracle Database 10g Release 2 (10.2)

http://download.oracle.com/docs/cd/B25329_01/doc/appdev.102/b28004/toc.htm

Oracle Database Net Services Administrator's Guide

Oracle Database 11g Release 1 (11.1)

http://download.oracle.com/docs/cd/B28359_01/network.111/b28316/toc.htm

Oracle Database PL/SQL Language Reference

Oracle Database 11g Release 1 (11.1)

http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28370/toc.htm

Oracle Database SQL Language Reference

Oracle Database 11g Release 1 (11.1)

http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/toc.htm

Selected PHP and Oracle Books

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

Articles and Other References

http://www.oracle.com/technology/tech/php/htdocs/php_troubleshooting_faq.html

PHP Scalability and High Availability

Oracle Whitepaper, April 2008

<http://www.oracle.com/technology/tech/php/pdf/php-scalability-ha-twp.pdf>

Oracle Database 11g PL/SQL Programming

Michael McLaughlin, Oracle Press, 2008. Contains a PHP primer.

Improving Performance Through Persistent Connections

John Coggeshall

http://www.oracle.com/technology/pub/articles/oracle_php_cookbook/coggeshall_persist.html

Using PHP 5 with Oracle XML DB

Yuli Vasiliev

<http://www.oracle.com/technology/oramag/oracle/05-jul/o45php.html>

An Overview on Globalizing Oracle PHP Applications

http://www.oracle.com/technology/tech/php/pdf/globalizing_oracle_php_applications.pdf

The PHP 5 Data Object (PDO) Abstraction Layer and Oracle

Wez Furlong

http://www.oracle.com/technology/pub/articles/php_experts/otn_pdo_oracle5.html

Software and Source Code

PHP Distribution Releases

Source and Windows binaries

<http://www.php.net/downloads.php>

Resources

PHP Snapshots

Snapshots of PHP's source code and Windows binaries

<http://snaps.php.net/>

PECL OCI8

Source package

<http://pecl.php.net/package/oci8>

PECL PDO_OCI

Source snapshot

http://pecl.php.net/package/PDO_OCI

Zend Core for Oracle

<http://www.oracle.com/technology/tech/php/zendcore/>

Oracle Instant Client

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

OCI8 Source Code in CVS

<http://cvs.php.net/viewcvs.cgi/php-src/ext/oci8/>

Oracle SQL Developer

Downloads and documentation

http://www.oracle.com/technology/products/database/sql_developer/index.html

ADODB Database Abstraction Library for PHP (and Python)

<http://adodb.sourceforge.net/>

PHP Extension for JDeveloper

<http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/php/index.html>

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://www.php.net/>

General PHP Documentation

<http://www.php.net/docs.php>

PHP Oracle OCI8 Documentation

<http://www.php.net/oci8>

PHP PDO Documentation

<http://www.php.net/pdo>

PHP Quality Assurance Site

<http://qa.php.net/>

PHP Bug System

<http://bugs.php.net/>

PHP Wiki

<http://wiki.php.net/>

Resources

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 datatype 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 datatype stores unstructured binary data in the database.

CHAR

The CHAR datatype stores fixed-length character strings in the database.

CLOB and NCLOB

The CLOB and NCLOB datatypes store up to 8 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 datatypes 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`.

Glossary

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/hrpwd@localhost/XE`.

CVS

Concurrent Versions System, an open source version control system used for development of PHP.

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*.

Datatype

Each column value and constant in a SQL statement has a datatype, which is associated with a specific storage format, constraints, and a valid range of values. When you create a table, you must specify a datatype for each of its columns. For example, NUMBER, or DATE.

DATE

The DATE datatype stores point-in-time values (dates and times) in a database table.

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](#).

Easy Connect

A simple hostname and database name string that is used to identify which database to connect to.

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 allow you 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.

Glossary

NCHAR and NVARCHAR2

NCHAR and NVARCHAR2 are Unicode datatypes that store Unicode character data in the database.

NUMBER

The NUMBER datatype stores fixed and floating-point numbers in the database.

Packages

A package is a group of related PL/SQL procedures and functions, along with the cursors and variables they use, stored together in the database for continued use as a unit.

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.

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.

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.

PEAR

The PHP Extension and Application Repository (PEAR) is a repository for reusable packages written in PHP.

PECL

The PHP Extension Community Library (PECL) 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()`.

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.

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.

Glossary

SDK

Software Development Kit. Oracle Instant Client has an SDK for building programs that use the Instant Client libraries.

Sequence

A sequence (a sequential series of numbers) 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 they connect 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 LOB

See *LOB*.

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.

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.

Glossary

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

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 datatypes 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.

XMLType

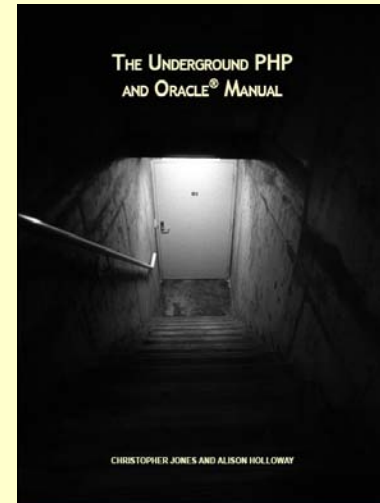
XMLType is a database datatype that can be used to store XML data in table columns.

The Underground PHP and Oracle® Manual

About this Book

This book is for PHP programmers developing applications for an Oracle database. It bridges the gap between the many PHP and the many Oracle books available. It shows how to use the PHP scripting language with the Oracle database, from installation to using them together efficiently.

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 liaises closely with the PHP community. He also helps make future versions of the Oracle database better for PHP. He is the author of various technical articles on PHP and Oracle, 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 Senior Product Manager 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.