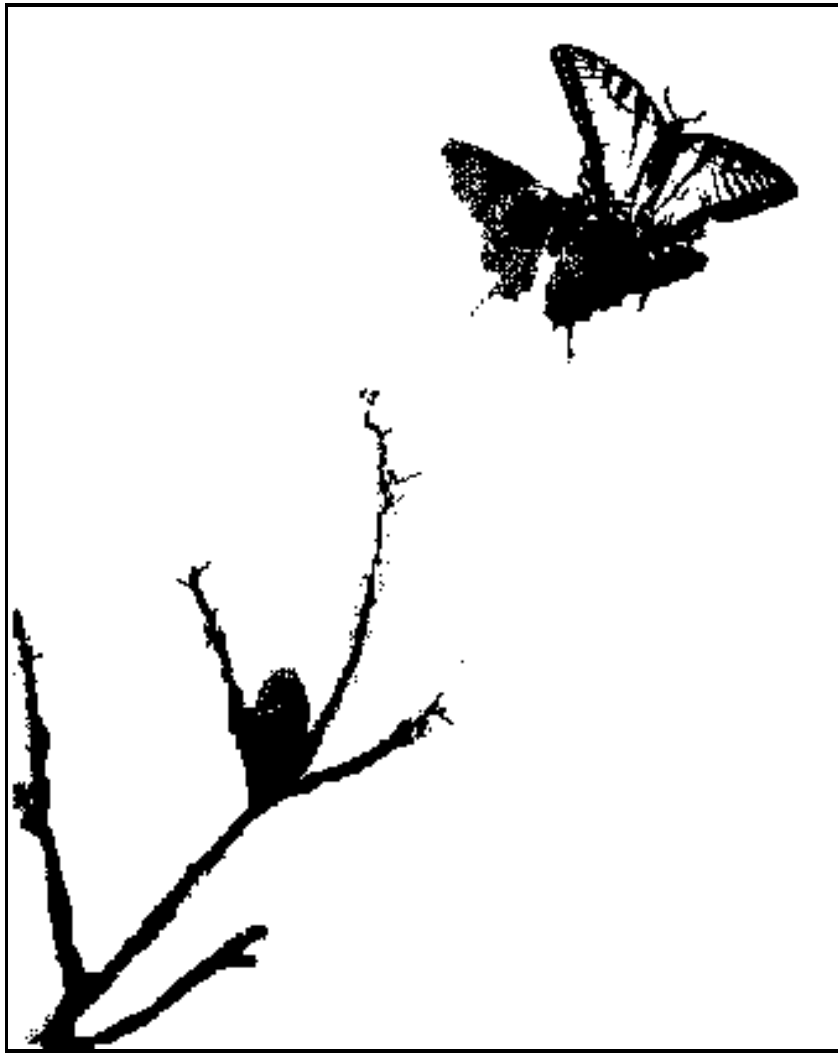




Business  
Alliance  
Programme™

Oracle Worldwide Alliances  
Design and Migration Services



ORACLE®

Optimizing OCI Applications for Client-Server Environments  
December 1994  
Part Number A25657

Author: Carolyn Wei  
Contributor: Subhash Jawaharani  
Editors: Ellen Lockett, Dan Mohler

Copyright © Oracle Corporation 1994, 1995  
**All rights reserved. Printed in the U.S.A.**

This document is provided for informational purposes only and the information herein is subject to change without notice. Please report any errors herein to Oracle Corporation. Oracle Corporation does not provide any warranties covering this information and specifically disclaims any liability in connection with this document.

Oracle, SQL\*Loader, SQL\*Module, SQL\*Net, and SQL\*Plus are registered trademarks, and Oracle7, Pro\*C, and PL/SQL are trademarks of Oracle Corporation.

All other mentioned products or company names are used for identification purposes only, and may be trademarks of their respective owners.

# Optimizing OCI Applications for Client-Server Environments

---

## Introduction

The purpose of this paper is to discuss ways to optimize OCI applications for client-server environments.

Oracle VARs commonly choose the Oracle Call Interface (OCI) to implement their Oracle7 application interfaces. As these applications are increasingly being deployed in client-server environments, the efficiency of the application interface can become a factor in the overall performance of the application. This is especially true in an environment where an application is communicating with Oracle7 over a network with limited bandwidth and high latency (i.e., a slow dial-up line). In this type of environment, excessive messaging and data exchange between the application and the Oracle server can severely impact application performance.

This paper indicates how you can measure the efficiency of your OCI application in terms of the number of OCI calls made and the SQL\*Net round-trips generated. It also addresses ways to improve application performance by reducing the number of calls from the application to Oracle7.

---

## Optimizing OCI Applications

Information about optimizing OCI applications is covered in three sections:

- OCI Calls
- Tuning the Application
- OCI Calls Revisited

### OCI Calls

The number of OCI calls that your application makes is a function of:

- how it manages sessions, cursors, and transactions
- the type of SQL statements it executes to perform application tasks that manipulate data and schema in the Oracle database.

Table 1 below lists a majority of the OCI calls with their basic parameters and a description of purpose. The calls appear in the order you are likely to see them used in an OCI application.

**Table 1: OCI Calls**

OCI call	Purpose
orlon(&lدا,&hدا,...)	establish a connection to an Oracle database
oopen(&cدا,&lدا,...)	open the specified cursor
ocon(&lدا)	enable autocommit
oparse(&cدا,sql_stmt,...)	parse a SQL statement or PL/SQL block and associate with the cursor
obndra (&cدا,&var_name,...)	bind an input variable or array to a placeholder in the SQL statement or PL/SQL block
obndrv/n(&cدا,&var_name,...)	bind an input variable to a placeholder in the SQL statement or PL/SQL block
odescr(&cدا,position,...)	describe select-list items for SQL queries
odefin(&cدا,position,...)	define an output variable for a specified select list item of a SQL query
oexec/oexn(&cدا)	execute the SQL statement or PL/SQL block associated with the cursor
oexfet(&cدا,nrows,...)	execute the SQL statement associated with the cursor and fetch nrows
ofetch/ofen(&cدا)	return row(s) of a query to the program
ocan(&cدا)	cancel a query after the desired number of rows have been fetched
ocom(&lدا)	commit the current transaction
orol(&lدا)	rollback the current transaction
oclose(&cدا)	close the specified cursor
ologof(&lدا)	disconnect from Oracle database

Table 2 lists the kinds of SQL statements OCI supports. The list is in the sequence OCI calls are typically used to process the SQL statements.

**Table 2: SQL Statements**

Type of SQL statement	OCI calls used to process SQL statement
Data Definition Language (CREATE,DROP,GRANT, REVOKE,...)	oparse, oexec
Transaction, Session and System Control (COMMIT, ROLLBACK, ALTER,...)	oparse, obndrv/n,oexec
Query (SELECT)	oparse, obndra/v/n, odescr,odefin,oexfet,ofetch/ofen
Data Manipulation Language (INSERT, UPDATE, DELETE, LOCK, EXPLAIN)	oparse, obndra/v/n, oexec/oexn
PL/SQL block	oparse, obndra/v, oexec

DML statements and PL/SQL blocks to Oracle7 in one SQL\*Net round-trip.

The ODESCR call used to describe select-list items cannot be deferred. The ODESCR call causes any outstanding PARSE, BIND, and DEFINE information to get sent to the Oracle server immediately. If your application does not need to use the ODESCR call for queries, then you can defer the EXECUTE step until the first FETCH (OEXFET) call and submit your query to Oracle7 in one SQL\*Net round-trip.

Table 3 below shows the SQL\*Net round-trips associated with each OCI call when using deferred mode linking and parsing and indicates whether or not the call can be deferred.

**Table 3: OCI calls and SQL\*Net round-trips**

OCI call	SQL*Net round-trips	Deferrable	Comment
orlon	3 for SQL*Net V1 4 for SQL*Net V2	no	
oopen	2	no	
ocon	1	no	
oparse	0	yes	Set DEFFLG flag to 1 in the oparse call. causes 1 SQL*Net round-trips if DEFLG is 0
obndra/v/n	0	yes	Causes 1 SQL*Net round-trip if application is linked in non-deferred mode
odescr	1*(number of select-list (n)/16)	no	Causes 1 SQL*Net round-trip per 16 select-list items
odefin	0	yes	Causes 1 SQL*Net round-trip if the application is linked in non-deferred mode
oexec/oexn	1	no	
oexfet	1	no	
ofetch/ofen	1	no	
ocan	1	no	
ocom	1	no	
orol	1	no	
oclose	1	no	
ologof	1 for SQL*Net V1 2 for SQL*Net V2	no	In V2, one packet is the implicit commit.

Deferred mode linking require more memory on the client system to buffer call data. If memory is scarce on your client system you may have to link in non-deferred mode. Whenever possible, link your application in deferred mode and use deferred parsing to minimize the number of SQL\*Net round-trips generated by the OCI calls in your application.

## Monitoring Application Performance

---

Determining whether or not your OCI application runs efficiently in a client-server environment often starts with some measure of the time it takes to perform application tasks in a stand-alone versus a network environment. Note that if the performance of your application in a stand-alone environment is not satisfactory it will not improve in a client-server environment. For this reason, you may benefit more by beginning with SQL statement and index tuning, subjects outside the scope of this paper.

If you determine that excessive network traffic is your application bottleneck, an investigation into the OCI calls made by your application is in order. You can start by using the SQL TRACE facility to capture the SQL statements involved in your slowest tasks.

To enable SQL TRACE from your application, add a function to your database connect routine that executes the SQL statement: "ALTER SESSION SET SQL\_TRACE = :flag" where :flag is set to TRUE to turn on SQL TRACE for the session.

The formatted trace file shows you:

1. All the SQL statements or PL/SQL blocks your application executed
2. The parse, execute, and fetch counts for each statement, and
3. The CPU time and row count associated with each step in 2).

The PARSE, EXECUTE, and FETCH counts in the trace file correspond to the equivalent OCI calls made by your application.

Look at in the trace file is the FETCH count and rows returned for each query. A high FETCH count indicates that using the array interface or increasing the array size can improve performance. The tkprof output for the statement below shows that the array interface is not being used. The fetch count is 15 and the rows returned is 14 indicating that each fetch call is returning only one row.

```
select * from emp
```

call	count	cpu	elapsed	disk	query current	rows	
Parse	1	0.14	0.26	8	32	1	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	15	0.01	0.06	1	15	2	14

Using the array interface, multiple rows can be fetched at one time. The tkprof output for the same statement using the array interface indicates that only one FETCH call was needed to fetch the rows for the query.

```
select * from ord
```

call	count	cpu	elapsed	disk	query current	rows
------	-------	-----	---------	------	---------------	------

Parse	1	0.00	0.00	0	0	0	0
-------	---	------	------	---	---	---	---

placeholders to arrays and make one OEXN call to insert the 100 accounts. This method causes only one SQL\*Net round-trip.

The tradeoffs of using the array interface in this example are:

- more memory is required in the client system to allocate space for the arrays
- the INSERT statement might not get executed until there are 100 accounts to add
- it is assumed that no intermediate processing of the data is required.

Below is an example of using the array interface in OCI:

```
/* Insert array of 10 employees */
Lda_def lda;
ubl hda[256];
Cda_def cda; /* cursor data area */
char names[10][NLEN];
char emp_nos[10];
text *sql_stmt = "INSERT INTO \
emp(ename, empno) VALUES (:N, :E)";
...
orlon(&lda, hda, "scott/tiger",...);
oopen(&cda, &lda,...);
oparse(&cda, sql_stmt,...);
obndra(&cda, ":N",-1,names,...);
obndra(&cda, ":E",-1,emp_nos,...);
/* fill arrays, then execute */
oexn(&cda, ARRSIZE, 0);
oclose(&cda);
ologof(&lda);
```

Check your application for tasks that involve processing multiple rows. If your application does batch INSERTs, UPDATEs or DELETEs you should be binding arrays instead of scalar variables and using OEXN instead of the OEXEC call. If your application executes queries that return multiple rows then you should always allocate arrays to store the return data and use the OEXFET and OFEN calls to retrieve batches of rows at a time. Use the array interface as much as you can to reduce the calls from your application to the database.

OCI also fully supports PL/SQL so you can group multiple SQL statements into one block and execute it as a unit. You can also call stored procedures and stored functions. PL/SQL blocks are processed in OCI just like a single SQL statement. The only difference is that you bind both input and output variables in PL/SQL using OBNDRA/V; you never use ODEFIN to bind output variables.

Check your application for related SQL statements. For instance, if you always do an UPDATE of an account table followed by an INSERT into a transaction table, consider either bundling those statements together in an anonymous PL/SQL block or creating a stored procedure you can call that executes both of those statements. Modifying your logical application

interface to be more function-oriented rather than statement-oriented helps reduce calls from your application to the database.

Below is an example of calling a stored procedure from OCI:

```
...
strcpy(plsblk, "
BEGIN
raise_salary(:enum, :nsal);
END;");
...
oopen(&c1, &lda,...);
oparse(&c1, plsblk, -1, 1, 2);
obndrv(&c1,":enum",-1,&empnum,...);
obndrv(&c1,":nsal",-1,&newsal,...);
...
empnum=9808; newsal=1000;
...
oexec(&c1);
ocom(&lda);
...
```

Caching data in client memory is another method you can use to improve performance. If your application has a lot of static lookup or policy information stored in the database, consider caching this information on the client system the first time it is read. Cutting down on repeated access to the data can help prevent network IO. The tradeoff of caching data in the client is the requirement of more memory in the client system to store the data.

## OCI Calls Revisited

The final section of this paper presents suggestions on the usage of OCI calls in your application. With OCI you explicitly control every part of your application's interaction with Oracle7 and every stage of SQL statement processing. With a good understanding of how your application accesses and manipulates data in the database, you can avoid repeating calls and phases of SQL statement processing and eliminate unnecessary calls from your application. Some of the suggestions made below do not reduce SQL\*Net round-trips but instead reduce processing overhead in the application and Oracle server.

### Establishing Sessions

Avoid short-lived sessions and a pattern of repeatedly connecting and disconnecting to the Oracle database from your application. Connection and session setup and break down are expensive in terms of network traffic and processing overhead in Oracle7 and the server system itself. The underlying network protocol can also affect how long these steps take. In general, once you establish a connection to Oracle7, you can keep the session open until you are finished with the database.

### Creating Cursors

In OCI, all cursors are explicitly declared and a cursor is required to process a SQL statement. An OCI application defines a cursor by declaring a cursor data area (CDA) structure and calling the OOPEN routine to associate it with a private SQL area in the Oracle server. Cursors are created independent of any SQL statements. Once a statement has been parsed into the cursor it remains there until the next statement is parsed into the cursor or the cursor is closed. Make sure that you create enough cursors in your application. It is recommended that you create enough cursors to support all the SQL statements that will be simultaneously used in your application. Consider allocating separate cursors to each frequently-executed SQL statement so that you can parse the statement just once and execute the statement repeatedly. The tradeoff in keeping multiple cursors open is that more memory is required in the application and the server. Each CDA structure currently requires 64 bytes in the client process and also memory in the server process to maintain the private SQL area.

An example of the use of multiple cursors in OCI:

```
cda_def cda1, cda2, cda3;
text *ins = "INSERT INTO emp ...";
text *upd="UPDATE emp ...";
text *edl="DELETE FROM emp ...";
...
oopen(&cda1, &lدا,...);
oopen(&cda2, &lدا,...);
oopen(&cda3, &lدا,...);
...
```

```

oparse(&cda1, ins,...);
oparse(&cda2, upd,...);
oparse(&cda3, del,...);
...
for(;;) {
if (inserting) oexn(&cda1,...);
if (updating) oexn(&cda2,...);
if (deleting) oexn(&cda3,...);
...}

```

Avoid repeatedly opening and closing cursors. Cursors can be re-used without being closed first. Re-using cursors improves performance by eliminating the SQL\*Net round-trips associated with opening and closing cursors.

### **Parsing Statements**

The OPARSE call is used to parse a SQL statement and associate it with a cursor. Every statement must be parsed before it can be executed. The PARSE stage can be deferred until execution by setting the appropriate flag in the oparse call. Otherwise the PARSE call will cause one SQL\*Net round-trip to occur. Once the parse is performed, the parsed representation is stored in the shared SQL area in Oracle server, so that subsequent OPARSE calls for the same SQL statement may not cause the statement to be re-parsed in the server. If you find that you are having to re-PARSE the same statement repeatedly then consider opening more cursors.

You can reduce the number of PARSE calls in your application by using bind variables instead of hard-coding expression or literal values in SQL statements.

### **Binding Input Variables**

The OCI bind routines are used to bind input variables and arrays to placeholders in your SQL statements. By binding variables you are telling Oracle where the values for the placeholders in the SQL statements are located in memory. For statements that are executed repeatedly, you can avoid re-binding by using static instead of automatic variables. Since BIND calls are deferred they do not affect the number of SQL\*Net round trips in your application.

### **Describing Select List Items**

The ODESCR call is used to get column datatype information for dynamic queries. You have to call ODESCR once for each column in the SELECT list. Be aware that DESCRIBE calls are not deferred. If you use the DESCRIBE call, any outstanding PARSE, BIND or DEFINE calls for that cursor are sent to the database. Fortunately, the ODESCR call does not cause one round trip per column, but returns data for up to 16 columns. A lot of SELECT list items will generate  $\frac{\text{\#select list items}}{16}$  SQL\*Net round trips. Executing a lot of dynamic queries causes a lot of overhead if you have to describe the

SELECT list for each query. You can minimize or avoid using the ODESCR call by caching table and column information in your client application.

### **Executing Statements**

Always use the array interface when possible. For queries, you should always use the OEXFET call to combine the EXECUTE and FETCH stages. Avoid using OCAN to cancel queries. In Oracle7, you do not need to cancel a query before executing the next query. You may set the cancel flag in the OEXFET call to cancel a query after you have fetched the desired number of rows. Re-executing SQL statements consists of re-executing the cursor into which the SQL statement is parsed.

### **Transaction Management**

The OCOM call commits changes for a transaction. If your transactions consists of more than one SQL statement, do not use the OCON call to turn on the auto-commit feature because it forces a COMMIT after every DML statement. Since a COMMIT is an expensive operation in Oracle7; place it carefully in your application so that it does not get executed unnecessarily. If the SQL statements inside an anonymous PL/SQL block make up a transaction, you can include the COMMIT statement along with other SQL statements. That way you do not need to call OCOM and cause another SQL\*Net round-trip to occur. When you disconnect from the database with OLOGOF, an implicit COMMIT is performed automatically.

---

## Conclusion

The challenge in designing an OCI application for a client-server environment lies in developing an application interface that takes into account and makes efficient use of the network. The efficiency of the application interface ultimately determines the effectiveness of the application in a network environment. Understanding how OCI calls and SQL\*Net round-trips are related can help VARs measure how efficiently their applications will run. Using Oracle7 features such as the array interface and stored procedures, VARs can design an application interface that requires a minimum amount of OCI calls and SQL\*Net round-trips, allowing applications to perform optimally in a client-server environment.





Oracle Worldwide Alliances  
Design and Migration Services

**ORACLE®**

Oracle Corporation  
World Headquarters  
500 Oracle Parkway  
Redwood Shores, CA 94065  
U.S.A.

Worldwide Inquiries:  
415.506.7000  
Fax: 415.506.7200  
<http://www.oracle.com/>

Copyright © Oracle Corporation 1995  
All Rights Reserved  
Printed in the U.S.A.

Part #: A25657