Building High Performance Drivers For Oracle Database 11*g*: OCI Tips and Techniques

An Oracle White Paper
[05] [2008]

Building High Performance Drivers for Oracle Database 11*g*: OCI Tips and Techniques

Building High Performance Drivers for Oracle Database 11*g*: OCI Tips and Techniques

The diverse environments that need to access the Oracle database is growing and range from Web applications to enterprise components, from Java to .Net, from PHP to Ruby, and from PERL to Python. Amidst all of this diversity, they all eventually converge to standard drivers or custom wrappers built on top of Oracle Call Interface (OCI). This document aims at being a guide for building efficient drivers/adapters/wrappers for accessing the Oracle database.

INTRODUCTION

The diverse environments that need to access the Oracle database is growing and range from Web applications to enterprise components, from Java to .Net, from PHP to Ruby, and from PERL to Python. Amidst all of this diversity, they all eventually converge to standard drivers or custom wrappers built on top of Oracle Call Interface (OCI). This document aims at being a guide for building efficient drivers/adapters/wrappers for accessing the Oracle database and offers a set of best practices, tips and techniques that lead to better performance, functionality, and ease of use.

The scope of this paper is limited to general guidelines and pointers to key technologies to use for building drivers for the Oracle database. It is not intended to be a substitute for the Oracle documentation, which is the definitive reference.

Some of the functions described here are based on features available in Oracle Database 11*g* OCI. However, most interfaces are available in older releases as well. Please consult the OCI documentation for the release that supports the specific API or call.

INITIALIZING OCI

The very first thing to do in an OCI program is to create the OCI Environment handle. Over the years, several flavors of OCI initialization functions have been provided. We recommend the flavors that are available in the most recent OCI versions.

Inheriting from `NLS_LANG`

The `NLS_LANG` environment variable specifies the client side language and character set settings. If you just want to inherit the `NLS_LANG` setup in your program, you can simply call `OCIEnvCreate()` as follows.

```
OCIEnv *envhp = (OCIEnv *)NULL;

  checkenv(envhp, OCIEnvCreate(&envhp,                 /* returned env handle */
                               OCI_DEFAULT,           /* initialization modes */
```

```
                              NULL, NULL, NULL, NULL,   /* callbacks, context */
                              (size_t) 0,      /* extra memory size: optional */
                              (void **) NULL));      /* returned extra memory */

  /* do work…. */

  OCIHandleFree(envhp, OCI_HTYPE_ENV);
```

You will notice the invocation to `checkenv(),` which is called for checking errors and will be described momentarily.

Specifying the client character set during initialization

If you want to remain independent of the `NLS_LANG` setting, OCI allows you to specify your character set programmatically. You have a choice to either fix the character set in your driver code or you can dynamically determine the character set from your own configuration file. The `OCIEnvNlsCreate()` interface allows you to specify the default client side character set programmatically.

```
OCIEnv *envhp = (OCIEnv *)NULL;
ub2     charset = getCharSetIdFromName(charSetName);

  checkenv(envhp, OCIEnvNlsCreate(&envhp,               /* returned env handle */
                                 OCI_DEFAULT,         /* initialization modes */
                                 NULL, NULL, NULL, NULL,/* callbacks, context */
                                 (size_t) 0,   /* extra memory size: optional */
                                 (void **) NULL,     /* returned extra memory */
                                 charset,             /* client side char set */
                                 charset));     /* client side NCHAR char set */

  ..do work…

  OCIHandleFree(envhp, OCI_HTYPE_ENV);
```

If you need a Unicode environment handle, you can directly use the predefined constant `OCI_UTF16ID` for the character set ids to be passed into `OCIEnvNlsCreate()`.

OCI does not publish ids for other character sets. The Oracle Globalization Support Guide publishes a comprehensive list of supported character set names.[1] OCI also provides a function `OCINlsCharSetNameToId(envhp, charSetName)` that can be used to convert a supported character set name to a character set id. This function requires an environment handle. If you have need to convert a character set name to a character set id, you will first need to create an `NLS_LANG` dependent environment handle using `OCIEnvCreate()` (as described earlier) and then call `OCINlsCharSetNameToId()` to convert the character set name to a character set id. This character set id can then be used to create specialized `charset` based environment handles using `OCIEnvNlsCreate()`.

Thread Safety

If your driver is likely to get used in a multithreaded environment, you should specify **`OCI_THREADED`** instead of `OCI_DEFAULT` in the initialization mode parameter. This will ensure that OCI internally serializes concurrent calls to connections and other OCI resources that can be shared between threads.

ERROR HANDLING

As a good programming practice, we recommend checking returned error codes from all OCI calls. OCI requires you to pass an error handle into most calls. The error handle returns diagnostic information regarding the call, if the call fails. In the following examples, you will see the usage of a `checkerr()` call that checks the returned status from every OCI call and if needed, gets more information from the error handle. You can customize this call to your error handling requirements. In this example below, we choose to stop executing the program on the first OCI_ERROR, as subsequent calls may not be meaningful if an earlier operation was unsuccessful.

For OCI calls that occur prior to allocating the error handle, it is not possible to call `checkerr()`. In such cases, a macro called `checkenv()` is called instead that can extract diagnostic info from the OCI environment handle. This is the reason why the environment handle creation calls in the earlier section called `checkenv()`.

The following example illustrates the allocation of an error handle:

```
OCIError *errhp;

  /* allocate error handle
   * note: for OCIHandleAlloc(), we always check error on environment handle
   */
  checkenv(envhp, OCIHandleAlloc(envhp,                /* environment handle */
                                (void **) &errhp,      /* returned err handle */
                                OCI_HTYPE_ERROR, /* typ of handle to allocate */
                                (size_t) 0,     /* optional extra memory size */
                                (void **) NULL));    /* returned extra memory */
```

This segment shows the definition of the error checking convenience macros.

```
/*
 * checkerr0(): This function prints a detailed error report.
 *              Used to "wrap" invocation of OCI calls.
 * Parameters:
 *   handle (IN) - can be either an environment handle or an error handle.
 *                 for OCI calls that take in an OCIError Handle:
 *                 pass in an OCIError Handle
 *
 *                 for OCI calls that don't take an OCIError Handle,
 *                 pass in an OCIEnv Handle
 *
 *   htype   (IN)- type of handle: OCI_HTYPE_ENV or OCI_HTYPE_ERROR
 *
 *   status (IN) - the status code returned from the OCI call
 *
 * Notes:
 *                 Note that this "exits" on the first
 *                 OCI_ERROR/OCI_INVALID_HANDLE.
 *                 CUSTOMIZE ACCORDING TO YOUR ERROR HANDLING REQUIREMENTS
 */
void checkerr0(void *handle, ub4 htype, sword status)
{
  /* a buffer to hold the error message */
  text errbuf[2048];
  sb4 errcode = 0;
  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) printf("Error - OCI_ERROR\n");
    if (handle)
    {
      (void) OCIErrorGet(handle, 1, (text *) NULL, &errcode,
                         errbuf, (ub4) sizeof(errbuf), htype);
      (void) printf("Error - %.*s\n", sizeof(errbuf), errbuf);
    }
    else
    {
      (void) printf("NULL handle\n");
      (void) printf("Unable to extract detailed diagnostic information\n");
    }
    exit(1);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    exit(1);
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
  default:
    break;
  }
}

/* friendly error checking macros */
/* checkerr(): meant for OCI calls that take an input error handle  */
#define checkerr(errhp, status) checkerr0((errhp), OCI_HTYPE_ERROR, (status))

/* checkenv(): meant for OCI calls that do not take an input error handle  */
#define checkenv(envhp, status) checkerr0((envhp), OCI_HTYPE_ENV, (status))
```

MANAGING CONNECTIONS

OCI provides various flavors of connection establishment calls that allow increasing degrees of control. All flavors of the calls take in a connect descriptor that has details of the database to which the connection needs to be established. Oracle has supported TNS style connect strings from an early version. More recently, Oracle has been promoting the Easy Connect string, which has an easier syntax for very common use cases.

Prior to using the easy connect naming, make sure that `EZCONNECT` is specified by the `NAMES.DIRECTORY_PATH` parameter in the sqlnet.ora file. The location of sqlnet.ora is specified by the `TNS_ADMIN` environment variable. If `TNS_ADMIN` is not set, then `$ORACLE_HOME/network/admin` is searched next. Set the `NAMES.DIRECTORY_PATH` as follows:

```
NAMES.DIRECTORY_PATH= (TNSNAMES, EZCONNECT)
```

The connect string itself takes the form:

```
"//host:[port]/[service_name]"
```

Using the same example, some valid connection strings follow:

```
"//sales-server.example.com:1521/sales"
"//sales-server.example.com/sales" (port default = 1521)
```

The following example demonstrates getting a connection for the user `"hr"`:

```
char *connstr = (char *) "//sales-server.example.com/sales";
OCIAuthInfo *authp = NULL;
OCISvcCtx   *svchp = NULL;

  /* allocate auth handle
   * note: for OCIHandleAlloc(), we check error on environment handle
   */
  checkenv(envhp, OCIHandleAlloc(envhp,
                          (void **) &authp, OCI_HTYPE_AUTHINFO,
                          (size_t) 0, (void **) NULL));

  /* setup username and password */
  checkerr(errhp, OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
                             (void *) "hr", strlen((char *) "hr"),
                             OCI_ATTR_USERNAME, errhp));

  checkerr(errhp, OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
                            apppassword, strlen((char *) apppassword),
                            OCI_ATTR_PASSWORD, errhp));

  /* get the database connection */
  checkerr(errhp, OCISessionGet(envhp, errhp,
                                &svchp,        /* returned database connection */
                                authp,    /* initialized authentication handle */
                                                           /* connect string */
                                (OraText *) connstr, strlen(connstr),
                                       /* session tagging parameters: optional */
                                NULL, 0, NULL, NULL, NULL,
                                OCI_DEFAULT));                         /* modes */

  .... do some work on the session....
  do_work_on_session(svchp);

  /* destroy the connection */
  checkerr(errhp, OCISessionRelease(svchp, errhp, NULL, 0, OCI_DEFAULT));

  (void) OCIHandleFree(authp, OCI_HTYPE_AUTHINFO);
```

Note that you could assemble the Easy Connect string dynamically from your own config file and hence you do not need to depend on tnsnames.ora (and TNS aliases) at all with the Easy Connect syntax.

Although some of the examples described here use the `OCISessionGet()` call, which allows you to deal with a database connection via a single context for simplicity, for some of the more advanced uses, you may need to use the `OCISessionBegin()` call. Please consult the Oracle documentation in case your requirements extend beyond what `OCISessionGet()` supports.

Pooling Database Connections

Pooling objects is an optimization that is widely adopted in many scenarios. Pooling an object makes sense if the object is:

- required for a relatively short duration

- expensive to create each time it is required
- expensive to tear down each time when it is no longer required
- expensive to hold on to when not being actively used, as it consumes resources

Pooling such objects enables a component that needs the object to quickly access it from the pool and release it back when no longer needed, thereby enabling other components to reuse the pooled object. Pooling drastically improves system scalability and performance.

Database connections generally satisfy all of the above-mentioned criteria; in other words, connection establishment involves: (i) creation of a network channel; (ii) spawning an associated operating system process/thread; (iii) performing the requisite authentication; and (iv) session creation with a session private context for SQL statements and other metadata. Tearing down a database connection is also expensive. And needless to say that unnecessarily maintaining them when not required eats up precious resources on the database server. That explains why database connection pooling greatly helps high traffic web applications where connections are needed for short durations. Most multithreaded systems that talk to a database offer some form of connection pooling.

OCI allows for connection pooling. The recommended connection pool for stateless web applications is called the OCI Session Pool. OCI also has a feature known as the OCI Connection Pool but Oracle recommends sticking to using the OCI Session Pool as that is designed to meet the requirements of most web applications for stateless database sessions.

There are many advantages of using OCI Session Pool:

- *No reinventing the wheel*: You get all the pool management within OCI. This involves (i) dynamically growing and shrinking the pool based on load; (ii) enforcing pool size limits; and (iii) enforcing appropriate concurrency control between threads which try to access the pool.
- *Better performance*: Seamless integration with Runtime Load Balancing (RLB) Events. This is particularly useful when you are using a RAC Database. In this scenario, any thread requesting a connection is given a connection from the pool to the best performing instance based on the most recent load balancing advisories that are published by the database.
- *Better availability*: Seamless integration with Fast Application Notification (FAN) Events. When FAN Events are enabled, OCI Session Pool terminates any idle connections in the pool upon database node/instance failure so that any new connection requests are directed to surviving nodes, thereby preventing the application from faulting due to a dead/orphan connection. It also provides immunity from waits in the TCP stack typically caused during node failures
- *Better scalability*: Transparent integration with the Database Resident Connection Pool (DRCP) functionality in Oracle Database 11*g*. DRCP allows

sharing of database connections across middle tier hosts, yielding extreme scalability and optimal memory consumption.

Using OCI Session Pool involves the creation of the pool by providing a connect string containing the target database and service identification e.g. "//sales-server.example.com/sales"

This step returns a pool-name, which uniquely identifies the session pool. After that, checking connections out of the pool and checking connections back into the pool look quite similar to the earlier example except that the `OCISessionGet()` call takes in the pool-name instead of the database connect string.

```
char *connstr = (char *) "//sales-server.example.com/sales";
OCIAuthInfo *authp = NULL;
OCISvcCtx   *svchp = NULL;
char *poolName = NULL;
ub4   poolNameLen;
OCISPool *spoolhp;
ub4  min;                                 /* minimum number of sessions in pool */
ub4  max;                                 /* maximum number of sessions in pool */
ub4  increment;                          /* the number to increment the pool by */

  printf ("demonstrating session pool creation\n");

  /* allocate session pool handle
   * note: for OCIHandleAlloc(), we check error on environment handle
   */
  checkenv(envhp, OCIHandleAlloc(envhp, (void **) &spoolhp,
           OCI_HTYPE_SPOOL, (size_t) 0, (void **) NULL));

  min = 0;                            /* not applicable for heterogeneous pools */
  max = 10;                               /* maximum number of sessions in pool */
  increment = 0;                      /* not applicable for heterogeneous pools */

  checkerr(errhp,
           OCISessionPoolCreate(envhp, errhp,
                                spoolhp,               /* session pool handle */
                                                 /* returned poolname, length */
                                (OraText **) poolName, &poolNameLen,
                                                            /* connect string */
                                (const OraText *) connstr, strlen(connstr),
                                min, max, increment,  /* pool size constraints */
                                (OraText *) "", 0,    /* hetero pool - no user */
                                (OraText *) "", 0,  /* hetero pool - no passwd */
                                OCI_DEFAULT));                       /* modes */
  /* allocate auth handle
   * note: for OCIHandleAlloc(), we check error on environment handle
   */
  checkenv(envhp, OCIHandleAlloc(envhp,
                        (void **) &authp, OCI_HTYPE_AUTHINFO,
                        (size_t) 0, (void **) NULL));

  /* setup username and password */
  checkerr(errhp, OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
                           (void *) "hr", strlen((char *) "hr"),
                           OCI_ATTR_USERNAME, errhp));

  checkerr(errhp, OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
                          apppassword, strlen((char *) apppassword),
                          OCI_ATTR_PASSWORD, errhp));

  /* get the database connection */
  checkerr(errhp, OCISessionGet(envhp, errhp,
                                &svchp,        /* returned database connection */
                                authp,    /* initialized authentication handle */
                                                         /* session pool name */
                                (OraText *) poolName, poolNameLen,
                                       /* session tagging parameters: optional */
                                NULL, 0, NULL, NULL, NULL,
                                OCI_SESSGET_SPOOL));                 /* modes */

  ....do some work on the session....
```

```
do_work_on_session(svchp);

/* release session back to pool */
checkerr(errhp, OCISessionRelease(svchp, errhp, NULL, 0, OCI_DEFAULT));

(void) OCIHandleFree( authp, OCI_HTYPE_AUTHINFO);
```

The example above illustrates only one thread. A multithreaded system, on the other hand, should typically do the `OCISessionPoolCreate()` once (per unique connect string) for a given process and every thread operating on the database should do the `OCISessionGet()` and `OCISessionRelease()` calls every time it needs a database connection from the pool or wants to give back a database connection.

The above example also illustrates the case wherein connections for multiple users are allowed to exist in the session pool. If it is so desired, one could make the session pool `OCI_SPC_HOMOGENEOUS`. In this case, the session pool only creates connections for the user given in the `OCISessionPoolCreate()` call.

```
min = 2;                                /* minimum number of sessions in pool */
max = 10;                               /* maximum number of sessions in pool */
increment = 2;                         /* the number to increment the pool by */

/* create a homogeneous session pool */
checkerr(errhp,
         OCISessionPoolCreate(envhp, errhp,
                              spoolhp,               /* session pool handle */
                                             /* returned poolname, length */
                              (OraText **) poolName, &poolNameLen,
                                                        /* connect string */
                              (const OraText *) connstr, strlen(connstr),
                              min, max, increment,  /* pool size constraints */
                                                              /* username */
                              (OraText *) "hr", strlen((char *) "hr"),
                                                              /* password */
                              (OraText *) apppassword,
                              strlen((char *) apppassword),
                                                                 /* modes */
                              OCI_SPC_HOMOGENEOUS));
```

Integrating with Oracle 11*g* Database Resident Connection Pool

Database Resident Connection Pooling (DRCP) provides a connection pool in the database server for typical Web application usage scenarios where the application acquires a database connection, works on it for a relatively short duration, and then releases it. DRCP pools a set of "dedicated" servers, which are referred to as "pooled" servers.

DRCP complements middle-tier connection pools, which share connections between threads in a middle-tier process. Unlike middle-tier connection pools, DRCP enables sharing of database connections across middle-tier processes on the same middle-tier host and across middle-tier hosts. This results in significant reduction in key database resources needed to support a large number of client connections, thereby reducing the database tier memory footprint and boosting the scalability of both middle-tier and database tiers. Having a pre-spawned pool of readily available servers also has the additional benefit of reducing the cost of creating and tearing down client connections.

DRCP is especially needed for architectures with multi-process single threaded application servers (such as PHP/Apache) that cannot perform middle-tier connection pooling. Applications can scale as high as tens of thousands of simultaneous connections with DRCP on a single commodity database server.

OCI Session Pool provides support for integrating with DRCP. With minimal changes, you can instruct an existing application using OCI Session Pool to also use the server side pool. This can be done by setting the `(SERVER=POOLED)` option in the connect string or the "`:POOLED`" option in the Easy Connect syntax. For more details on making the best use of Oracle DRCP, refer to the DRCP white paper.[2]

PREPARING STATEMENTS

Once you have the connection pool going, you need to get ready to prepare statements. Applications typically tend to issue the same statements repeatedly. Creating a statement from scratch is an expensive operation. Hence, it makes a lot of sense to keep statements around after preparing them, effectively pooling prepared statements.

Although OCI allows users to explicitly cache statements on their own, OCI furnishes built in statement caching functionality through the `OCIStmtPrepare2()` call. The statement cache size can be specified after obtaining a database connection:

```
ub4       stmt_cachesize = 20;

/* get the database connection */
checkerr(errhp, OCISessionGet(envhp, errhp,
                              &svchp,        /* returned database connection */
                              authp,    /* initialized authentication handle */
                                                           /* connect string */
                              (OraText *) connstr, strlen(connstr),
                                     /* session tagging parameters: optional */
                              NULL, 0, NULL, NULL, NULL,
                              OCI_DEFAULT));                        /* modes */
/* set the statement cache size */
checkerr(errhp, OCIAttrSet(svchp, OCI_HTYPE_SVCCTX,
                           &stmt_cachesize, 0, OCI_ATTR_STMTCACHESIZE, errhp));
```

or better, it can simply be set when the OCI Session Pool is created, which means that every session in the pool gets to cache that many statements:

```
OCISPool *spoolhp;
ub4       stmt_cachesize = 20;

/* allocate session pool handle
 * note: for OCIHandleAlloc(), we check error on environment handle
 */
checkenv(envhp, OCIHandleAlloc(envhp, (void **) &spoolhp,
         OCI_HTYPE_SPOOL, (size_t) 0, (void **) NULL));


/* set the statement cache size for all sessions in the pool */
checkerr(errhp, OCIAttrSet(spoolhp, OCI_HTYPE_SPOOL,
                &stmt_cachesize, 0, OCI_ATTR_SPOOL_STMTCACHESIZE, errhp));

min = 0;                         /* not applicable for heterogeneous pools */
max = 10;                            /* maximum number of sessions in pool */
increment = 0;                   /* not applicable for heterogeneous pools */

checkerr(errhp,
         OCISessionPoolCreate(envhp, errhp,
                              spoolhp,                /* session pool handle */
                                            /* returned poolname, length */
                              (OraText **) poolName, &poolNameLen,
                                                           /* connect string */
```

```
                                (const OraText *) connstr, strlen(connstr),
                                min, max, increment,  /* pool size constraints */
                                (OraText *) "", 0,    /* hetero pool - no user */
                                (OraText *) "", 0,  /* hetero pool - no passwd */
                                OCI_SPC_STMTCACHE));                /* modes */
```

Once you have done the appropriate statement cache setup, `OCIStmtPrepare2()` first looks for a prepared statement handle in the cache. If it finds a matching handle in the cache, it just returns the cached handle. Otherwise, it prepares a new statement handle. Once you are done with the statement handle, you can call `OCIStmtRelease()` that returns that statement handle back into the cache. Releasing the statement handle means that you are no longer actively using the statement handle and the OCI Statement Cache can age it out, if required, to make room for new statement handles in the cache.

```
char *MY_SELECT = (char *) "select employee_id, last_name, salary from \
                           employees where employee_id = :EMPNO";

void select_emp_details(OCISvcCtx *svchp, ub4 empno)
{
  OCIStmt *stmthp;
  ub4 num_rows = 1;


  printf ("demonstrating single row select\n");

  /* get a prepared statement handle */
  checkerr(errhp,
           OCIStmtPrepare2 (svchp,
                            &stmthp,                  /* returned stmt handle */
                            errhp,                            /* error handle */
                            (const OraText *) MY_SELECT,     /* statement text */
                            strlen((char *) MY_SELECT),      /* length of text */
                            NULL, 0,          /* tagging parameters: optional */
                            OCI_NTV_SYNTAX, OCI_DEFAULT));

  /* do work on the statement */
  execute_statement(svchp, stmthp, errhp);

  /* release the statement handle */
  checkerr(errhp, OCIStmtRelease(stmthp, errhp,
                                 (OraText *) NULL, 0,         /* tag: optional */
                                 OCI_DEFAULT));

}
```

If the statement text is known at compile time, as in this example, you could have pre-computed the length at compile-time instead of calling `strlen()` at run-time. However, for general-purpose drivers where strings are dynamic, you either need to compute the length at run-time or alternatively, you could expose an interface in your driver that gets the statement length from the caller.

BINDING VARIABLES

Oracle recommends using bind variables in SQL statements instead of literal values. The are several advantages with using Bind variables:

- It is a mandatory step that you should take to secure the code from SQL injection attacks
- The statement can be prepared once and executed several times with different bind values

- Both the Oracle backend and the client perform and scale better because there are fewer parse calls

The process of "binding" involves associating a variable in your program to a bind variable in the statement text. OCI provides the `OCIBindByName()` call that allows you to identify the bind-variable in the statement text by name. You need to specify the address of your program variable and its size and its datatype code. A detailed treatment of Datatypes is beyond the scope of this whitepaper. For more details on Oracle Datatypes, please see the section on Introduction to Oracle Datatypes[3] in the Concepts Guide. Also, see the Datatypes[4] section in the OCI documentation.

You can optionally associate an indicator variable (that indicates whether the Bind value is `NULL`). The indicator variable being set to `-1` means that the value is `NULL`. For bind values whose size can vary, such as character strings, you can also pass in a pointer to an actual length value. This value can be reset on every execution to be the actual length. For more details, look-up the OCI documentation.

```
char *MY_SELECT = (char *) "select employee_id, last_name, salary from \
                           employees where employee_id = :EMPNO";
OCIBind *bndp1;

ub4 empno = 1;                  /* variable in program that needs to be bound */
sb2 empno_ind = 0;                         /* corresponding indicator variable */

/* bind :EMPNO */
checkerr(errhp,
         OCIBindByName(stmthp, &bndp1, errhp,
                         (text *) ":EMPNO",           /* name of bind variable */
                         -1,  /* length of bind name: -1 means NULL terminated */
                         (void *) &empno,       /* address of program variable */
                         sizeof(empno),             /* size of program variable */
                         SQLT_INT,        /* type-code of the program variable */
                         (void *) &empno_ind,          /* address of indicator */
                         (ub2 *) NULL,             /* address of actual length */
                         (ub2 *) NULL,               /* address of return code */
                         0,       /* max array length for PLSQL indexed tables */
                         (ub4 *) NULL,   /* current array len for PLSQL arrays */
                         OCI_DEFAULT));
```

With `OCIBindByName()`, you only need to specify the name of your bind variable such as `":X"` and all occurrences of `":X"` in the statement will get bound to the same value.

OCI also provides a position based `OCIBindByPos()` call, which identifies the bind variable in the statement text by its position as opposed to its name. `OCIBindByPos()` allows you increased flexibility in terms of binding duplicate bind-parameters separately, if you need it. You have the option of binding any of the duplicate occurrences of a bind parameter separately. Any unbound duplicate occurrences of a parameter inherit the value from the first occurrence of the bind parameter with the same name. The first occurrence must be explicitly bound.

In the context of SQL statements, the position "n" indicates the bind parameter at the nth position. However, in the context of PL/SQL statements, `OCIBindByPos()` has a different interpretation for the "position" parameter: the position "n" in the bind call indicates a binding for the nth unique parameter name in the statement when scanned left to right.

`OCIBindByName()` has the advantage that if your program evolves to add more bind variables in your statement text, you do not have to touch your existing bind calls in order to update bind positions.

EXECUTING DMLS

Once you have a prepared and statement with all variables bound appropriately, the next step would be to execute it. Let us first take the example of a simple DML.

```
char *MY_DML = (char *) "update employees set salary = :sal \
                        where employee_id = :id";
void do_dml_statement(OCISvcCtx *svchp, ub4 sal, ub4 id)
{
  OCIBind *bndp1, *bndp2;
  OCIStmt *stmthp;
  ub4 num_iterations = 1;


  printf ("demonstrating simple dml\n");
  /* get a prepared statement handle */
  checkerr(errhp,
           OCIStmtPrepare2 (svchp,
                            &stmthp,                    /* returned stmt handle */
                            errhp,                              /* error handle */
                            (const OraText *) MY_DML,  /* input statement text */
                            strlen((char *) MY_DML),         /* length of text */
                            NULL, 0,           /* tagging parameters: optional */
                            OCI_NTV_SYNTAX, OCI_DEFAULT));

  /* do parameter bindings */
  checkerr(errhp, OCIBindByPos(stmthp, &bndp1, errhp, 1,
                               (void *) &sal, (sword) sizeof(sal), SQLT_INT,
                               (void *) NULL, (ub2 *) NULL, (ub2 *) NULL, 0,
                               (ub4 *) NULL, OCI_DEFAULT));

  checkerr(errhp, OCIBindByPos(stmthp, &bndp2, errhp, 2,
                               (void *) &id, (sword) sizeof(id), SQLT_INT,
                               (void *) NULL, (ub2 *) NULL, (ub2 *) NULL, 0,
                               (ub4 *) NULL, OCI_DEFAULT));

  /* execute the statement and commit */
  checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, num_iterations, 0,
                                 (OCISnapshot *) NULL, (OCISnapshot *) NULL,
                                 OCI_COMMIT_ON_SUCCESS));

  /* release the statement handle */
  checkerr(errhp, OCIStmtRelease(stmthp, errhp,
                                 (OraText *) NULL, 0, OCI_DEFAULT));
  printf("updated salary successfully\n");
}
```

Note: You must ensure that all program variables, indicators and lengths whose addresses you pass into OCI bind calls remain in scope at least until the end of the `OCIStmtExecute()` call for DML statements. Oracle recommends keeping the bind metadata information, such as the maximum length of the bind and its datatype, stable across repeat executions for best performance in order to minimize repeat bind processing overheads.

As an example, when statement caching is enabled, repeatedly executing `do_dml_statement()` procedure above will repeatedly find the statement handle in the OCI Statement Cache. The bound program variables are passed on the stack and hence remain in scope until the `OCIStmtExecute()` call is done. The bind metadata, such as the maximum size of the bind buffer (`sizeof(sal)`) and its datatype (`SQLT_INT`) do not change across executions but the actual bind addresses/values may change. When a prepared statement is found in the statement cache and the bind metadata remains stable for repeat executions, OCI provides best performance.

Array Executes of DML statements

In many cases, you need to bulk insert/update or delete several rows at once. For such cases, we strongly recommend that you use the OCI Array DML capabilities as it reduces the roundtrips to do the operation. Note that the bind variables are arrays of values and the `"iters"` parameter in the `OCIStmtExecute()` is set appropriately to indicate the size of the array. In this example, it is set to `NUM_ITERATIONS`.

```
char *MY_DML = (char *) "update employees set salary = :sal \
                        where employee_id = :id";

void do_array_dml_statement(OCISvcCtx *svchp)
{
 #define NUM_ITERATIONS 2
  ub4 sal[NUM_ITERATIONS] = {31000, 35000};
  ub4  id[NUM_ITERATIONS] = {100, 101};
  OCIBind *bndp1, *bndp2;
  OCIStmt *stmthp;

  printf ("demonstrating array dml\n");

  /* get a prepared statement handle */
  checkerr(errhp,
           OCIStmtPrepare2 (svchp,
                            &stmthp,                   /* returned stmt handle */
                            errhp,                             /* error handle */
                            (const OraText *) MY_DML,  /* input statement text */
                            strlen((char *) MY_DML),          /* length of text */
                            NULL, 0,           /* tagging parameters: optional */
                            OCI_NTV_SYNTAX, OCI_DEFAULT));

  /* do parameter bindings */
  checkerr(errhp, OCIBindByPos(stmthp, &bndp1, errhp, 1,
                               (void *) sal, (sword) sizeof(sal[0]), SQLT_INT,
                               (void *) NULL, (ub2 *) NULL, (ub2 *) NULL, 0,
                               (ub4 *) NULL, OCI_DEFAULT));

  checkerr(errhp, OCIBindByPos(stmthp, &bndp2, errhp, 2,
                               (void *) id, (sword) sizeof(id[0]), SQLT_INT,
                               (void *) NULL, (ub2 *) NULL, (ub2 *) NULL, 0,
                               (ub4 *) NULL, OCI_DEFAULT));

  /* execute the statement and commit */
  checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, NUM_ITERATIONS, 0,
                                 (OCISnapshot *) NULL, (OCISnapshot *) NULL,
                                 OCI_COMMIT_ON_SUCCESS));

  /* release the statement handle */
  checkerr(errhp, OCIStmtRelease(stmthp, errhp,
                                 (OraText *) NULL, 0, OCI_DEFAULT));

  printf("updated salaries successfully\n");
}
```

EXECUTING QUERIES AND FETCHING RESULTS

Binding variables into queries is similar to DMLs. However, you have the additional steps of defining output buffers and fetching from the result set.

```
char *MY_SELECT = (char *) "select employee_id, last_name, salary from \
                           employees where employee_id = :EMPNO";

void select_emp_details(OCISvcCtx *svchp, ub4 empno)
{
  OCIBind *bndp1;
  OCIDefine *defhp1, *defhp2, *defhp3;
  OCIStmt *stmthp;
  ub4 emp_id;
  char last_name[100];
  ub4  salary;
  ub4 num_rows = 1;
  sword status;


  printf ("demonstrating single row select\n");
  /* get a prepared statement handle */
  checkerr(errhp,
           OCIStmtPrepare2 (svchp,
                            &stmthp,                   /* returned stmt handle */
                            errhp,                             /* error handle */
                            (const OraText *) MY_SELECT,     /* statement text */
                            strlen((char *) MY_SELECT),      /* length of text */
                            NULL, 0,           /* tagging parameters: optional */
                            OCI_NTV_SYNTAX, OCI_DEFAULT));

  /* bind input parameters */
  checkerr(errhp, OCIBindByName(stmthp, &bndp1, errhp, (text *) ":EMPNO",
                                -1, (void *) &empno,
                                sizeof(empno), SQLT_INT, (void *) NULL,
                                (ub2 *) NULL, (ub2 *) NULL, 0, (ub4 *) NULL,
                                OCI_DEFAULT));

  /* Execute the query */
  checkerr(errhp, OCIStmtExecute (svchp, stmthp, errhp, 0, 0,
                                  (OCISnapshot *) NULL, (OCISnapshot *) NULL,
                                  OCI_DEFAULT));

  /* Define output buffers */
  checkerr(errhp, OCIDefineByPos (stmthp, &defhp1, errhp, 1,
                                  (void *) &emp_id, (sb4) sizeof(emp_id),
                                  SQLT_INT, (void *) NULL, (ub2 *) NULL,
                                  (ub2 *) NULL, OCI_DEFAULT));

  checkerr(errhp, OCIDefineByPos (stmthp, &defhp2, errhp, 2,
                                  (void *) last_name, (sb4) sizeof(last_name),
                                  SQLT_STR, (void *) NULL, (ub2 *) NULL,
                                  (ub2 *) NULL, OCI_DEFAULT));


  checkerr(errhp, OCIDefineByPos (stmthp, &defhp3, errhp, 3,
                                  (void *) &salary, (sb4) sizeof(salary),
                                  SQLT_INT, (void *) NULL, (ub2 *) NULL,
                                  (ub2 *) NULL, OCI_DEFAULT));

  /* Fetch one row */
  status = OCIStmtFetch(stmthp, errhp, num_rows, OCI_FETCH_NEXT,
                        OCI_DEFAULT);
  if (status == OCI_SUCCESS)
    printf("fetched results: employee_id=%d, last_name=%s, salary=%d\n",
           emp_id, last_name, salary);
  else
    checkerr(errhp, status);

  /* release the statement handle */
  checkerr(errhp, OCIStmtRelease(stmthp, errhp,
                                 (OraText *) NULL, 0,         /* tag: optional */
                                 OCI_DEFAULT));

}
```

However, in many cases, such as the example above, you know the shape of the SQL text and the types of the define buffers. In such cases, you can save one call by bundling the execute and fetch calls together. In this case, the output buffers need to be provided prior to execution as follows:

```
char *MY_SELECT = (char *)"select employee_id, last_name,salary from \
                           employees where employee_id = :EMPNO";

void select_emp_details_bundled(OCISvcCtx *svchp, ub4 empno)
{
  OCIBind *bndp1;
  OCIDefine *defhp1, *defhp2, *defhp3;
  OCIStmt *stmthp;
  ub4 emp_id;
  char last_name[100];
  ub4  salary;
  ub4 num_rows = 1;

  printf ("demonstrating single row select with bundled exec/fetch\n");

  /* get a prepared statement handle */
  checkerr(errhp, OCIStmtPrepare2 (svchp, &stmthp, errhp,
                                   (const OraText *) MY_SELECT,
                                   strlen((char *) MY_SELECT),
                                   (OraText*) NULL, 0,
                                   OCI_NTV_SYNTAX, OCI_DEFAULT));

  /* bind input parameters */
  checkerr(errhp, OCIBindByName(stmthp, &bndp1, errhp, (text *) ":EMPNO",
                                -1, (void *) &empno,
                                sizeof(empno), SQLT_INT, (void *) NULL,
                                (ub2 *) NULL, (ub2 *) NULL, 0, (ub4 *) NULL,
                                OCI_DEFAULT));

  /* Define output buffers */
  checkerr(errhp, OCIDefineByPos (stmthp, &defhp1, errhp, 1,
                                  (void *) &emp_id, (sb4) sizeof(emp_id),
                                  SQLT_INT, (void *) NULL, (ub2 *) NULL,
                                  (ub2 *) NULL, OCI_DEFAULT));

  checkerr(errhp, OCIDefineByPos (stmthp, &defhp2, errhp, 2,
                                  (void *) last_name, (sb4) sizeof(last_name),
                                  SQLT_STR, (void *) NULL, (ub2 *) NULL,
                                  (ub2 *) NULL, OCI_DEFAULT));

  checkerr(errhp, OCIDefineByPos (stmthp, &defhp3, errhp, 3,
                                  (void *) &salary, (sb4) sizeof(salary),
                                  SQLT_INT, (void *) NULL, (ub2 *) NULL,
                                  (ub2 *) NULL, OCI_DEFAULT));


  /* Execute the query */
  checkerr(errhp, OCIStmtExecute (svchp, stmthp, errhp, num_rows, 0,
                                  (OCISnapshot *) NULL, (OCISnapshot *) NULL,
                                  OCI_DEFAULT));


  printf("fetched results: employee_id=%d, last_name=%s, salary=%d\n",
         emp_id, last_name, salary);

  /* release the statement handle */
  checkerr(errhp, OCIStmtRelease(stmthp, errhp,
                                 (OraText *) NULL, 0, OCI_DEFAULT));

}
```

In both of the above examples, we omitted the indicator variables in the OCI define calls for simplicity. If you know that your column value cannot be `NULL`, you too can omit the corresponding indicator. However, if there is a possibility that a fetched column can be `NULL`, you should pass an indicator variable in the `OCIDefineByPos()` call. The next example will illustrate this usage.

Note: The same scoping rules described earlier for bind addresses apply to define addresses as well. You need to ensure that all program variables, indicators and lengths whose addresses you pass into OCI bind and define calls remain in scope all the way until the end of the last `OCIStmtFetch()` call that depends on those bind/define buffers.

Array Fetching

Oracle recommends using array fetching when the result set contains several rows. Instead of fetching one row at a time, which results in many roundtrips to the database, you can fetch the rows in batches using array fetches.

```
char *MY_SELECT2 = (char *) "select employee_id, last_name from employees\
                     where employee_id > :EMPNO order by employee_id";

void array_fetch(OCISvcCtx *svchp, ub4 empno)
{
#define ARRAY_SIZE 10
  OCIStmt *stmthp;
  OCIBind *bndp;
  OCIDefine *defhp1, *defhp2;
  ub4 empid[ARRAY_SIZE];
  sb2 empid_ind[ARRAY_SIZE];
  char lname[ARRAY_SIZE][30];
  sb2  lname_ind[ARRAY_SIZE];
  ub2  lname_len[ARRAY_SIZE];
  boolean done=FALSE;
  ub4 rows = 0;
  ub4 i =0;
  sb4 status;

  printf ("demonstrating array fetching\n");

  /* get a prepared statement handle */
  checkerr(errhp, OCIStmtPrepare2 (svchp, &stmthp, errhp,
                                  (const OraText *) MY_SELECT2,
                                  strlen((char *) MY_SELECT2),
                                  (OraText*) NULL, 0,
                                  OCI_NTV_SYNTAX, OCI_DEFAULT));

  /* bind input parameters */
  checkerr(errhp, OCIBindByName(stmthp, &bndp, errhp, (text *) ":EMPNO",
                               -1, (void *) &empno,
                               sizeof(int), SQLT_INT, (void *) NULL,
                               (ub2 *) NULL, (ub2 *) NULL, 0, (ub4 *) NULL,
                               OCI_DEFAULT));

  /* Execute the query */
  checkerr(errhp, OCIStmtExecute (svchp, stmthp, errhp, 0, 0,
                                 (OCISnapshot *) NULL, (OCISnapshot *) NULL,
                                 OCI_DEFAULT));

  /* Define output buffers */
  checkerr(errhp, OCIDefineByPos (stmthp, &defhp1, errhp, 1,
                                 (void *) empid, (sb4) sizeof(empid[0]),
                                 SQLT_INT, (void *) empid_ind,
                                 (ub2 *) NULL, (ub2 *) NULL, OCI_DEFAULT));

  checkerr(errhp, OCIDefineByPos (stmthp, &defhp2, errhp, 2,
                                 (void *) lname[0], (sb4) sizeof(lname[0]),
                                 SQLT_STR, (void *) lname_ind,
                                 (ub2 *) lname_len, (ub2 *) NULL,
                                 OCI_DEFAULT));

  /* Fetch ten rows at a time */
  while (!done)
  {
    status = OCIStmtFetch(stmthp, errhp, ARRAY_SIZE,
                         OCI_FETCH_NEXT, OCI_DEFAULT);

    if ((status == OCI_SUCCESS) || (status == OCI_NO_DATA))
    {
      if (status == OCI_SUCCESS)
        rows = ARRAY_SIZE;                 /* all rows asked for were obtained */
      else if (status == OCI_NO_DATA)
      {
        /* might have gotten fewer rows */
        checkerr(errhp, OCIAttrGet(stmthp, OCI_HTYPE_STMT,
                           &rows, (ub4 *) NULL,
                           OCI_ATTR_ROWS_FETCHED, errhp));
        done = TRUE;
      }
      for (i=0; i < rows; i++)
        printf ("employee_id=%d, last_name=%s\n", empid[i], lname[i]);

    }
    else
    {
      checkerr (errhp, status);
      done =TRUE;
    }
  }

  /* release the statement handle */
  checkerr(errhp, OCIStmtRelease(stmthp, errhp,
                                 (OraText *) NULL, 0, OCI_DEFAULT));
  printf ("array fetch successful\n");
}
```

CHARACTER SET CONVERSIONS

When doing OCI bind and define calls, you need to be aware that the data can expand/contact due to character set conversions taking place as the data is transferred between the client and server. Please see the section on Character Conversion in OCI Binding and Defining[5] for more details.

OCI normally does all character set conversions for data transparently as data goes back and forth between the client and server from the server character set to client character set and vice versa. You can override the initialization time character set (specified during `OCIEnvNlsCreate()` or picked up by `OCIEnvCreate()`) for bind/define buffers programmatically. This example shows a client binding/defining Unicode data.

```
  ub2 csid = OCI_UTF16ID;

  /* inserting unicode data */
  /* bind input parameters */
  checkerr(errhp, OCIBindByName(stmthp, &bndp1, errhp, (text *) ":ENAME",
                                -1, (void *) ename,
                                sizeof(ename), SQLT_STR, (void *) &ename_ind,
                                (ub2 *) NULL, (ub2 *) NULL, 0, (ub4 *) NULL,
                                OCI_DEFAULT));

  /* specify char set id for the bind */
  checkerr(errhp, OCIAttrSet(bndp1, OCI_HTYPE_BIND, &csid,  0,
                             OCI_ATTR_CHARSET_ID, errhp));

  /* retrieving unicode data */
  /* define output parameters */
  checkerr(errhp, OCIDefineByPos (stmthp, &defhp1, errhp, 2,
                                  (void *)lname[0], (sb4)sizeof(lname[0]),
                                  SQLT_STR,(void *)lname_ind,
                                  (ub2 *)lname_len, (ub2 *)0,
                                  OCI_DEFAULT));

  /* specify the character set id for the define */
  checkerr(errhp, OCIAttrSet(defhp1, OCI_HTYPE_DEFINE, &csid, 0,
                             OCI_ATTR_CHARSET_ID, errhp));
```

TRANSACTION CONTROL

OCI provides interfaces for transaction commit and rollback. These operations, `OCITransCommit()` and `OCITransRollback()` cause a roundtrip to the database. Oracle recommends that you use `OCI_COMMIT_ON_SUCCESS`, where possible, on the last `OCIStmtExecute()` in a transaction so that the commit gets piggybacked with the same call to the Database.

ORACLE SECUREFILES

Oracle Database 11*g* SecureFiles bring a significant leap in large objects performance. With SecureFiles, Oracle has also improved the performance of transporting large objects across the network. Some of these optimizations are transparent and will benefit existing applications. Some others are not transparent and require using new APIs.

OCI allows fetching LOB locators followed by explicit OCI lob read calls in order to lookup the contents of the locator. This however requires you to perform additional round-trips during the OCI LOB read calls in order to get to the contents of the LOB. OCI provides two alternatives to deal with this:

- You can choose to implicitly convert the LOB to character/raw data (for CLOBS/BLOBS respectively) right when fetching the data from the server. You can do this by setting the appropriate data type in the OCI define call: instead of requesting the locator, request the datatype of the underlying contents. You also need to provide an appropriately sized buffer. In such case, OCI call to fetch the data from the server returns the data in the desired format and you do not need additional round-trips. For more information, please see the section on Data Interface for Persistent LOBs.[6]

- On the other hand, it is possible that your application requires you to fetch LOB locators and you need to issue additional LOB read calls. In such cases, if you set a prefetch size for LOBs, OCI can prefetch LOB data along with the locator to improve performance of smaller LOBs. Setting a LOB prefetch size can help eliminate network roundtrips, which can be a large performance win. The `OCILobGetLength2(), OCILobRead2()`, `OCILobArrayRead()` and `OCILobGetChunkSize()` calls can be processed locally with LOB prefetching.[7] This can be done at a session level, the moment a session is checked out of a session pool or a new connection is established. It can also be done at a statement column level, if you desire to override the session level setting:

```
ub4 default_lobprefetch_size = 2000;              /* set default size to 2K */

/* set lob prefetch attribute to session */
checkerr(errhp, OCIAttrSet (sesshp, OCI_HTYPE_SESSION,
                            &default_lobprefetch_size,    /* attribute value */
                            0,
                            OCI_ATTR_DEFAULT_LOBPREFETCH_SIZE,  /* attr type */
                            errhp));

/* later … */
/* override the default prefetch size to 4K on define handle */
ub4 prefetch_size = 4000;
checkerr(errhp, OCIAttrSet (defhp,  OCI_HTYPE_DEFINE,
                            &prefetch_size,             /* attribute value */
                            0,
                            OCI_ATTR_LOBPREFETCH_SIZE,          /* attr type */
                            errhp));
```

RECEIVING DATABASE EVENTS

The Oracle Database publishes various events and sends them to clients. These events are related to high availability, namely Fast Application Notification (FAN), and scalability, namely Runtime Load Balancing (RLB) in RAC environments. OCI clients can also receive Advanced Queue Notifications and Database Change Notifications.

In order to receive and process these database events, you must initialize your Environment handle in `OCI_EVENTS` mode. Oracle suggests making the `OCI_EVENTS` initialization an option that is configurable when your driver is deployed rather than having `OCI_EVENTS` always on by default.

If `OCI_EVENTS` is set, FAN and RLB notifications are received and processed transparently if you use OCI Session Pool APIs (please refer to earlier discussion on "Pooling Database Connections"). On the other hand, if you build your own pool, you can plug into OCI to receive FAN events (but not the RLB events). In case you have your own custom pool, we strongly suggest that you consider using the OCI Session Pool APIs as an alternative as it is pre-integrated with these events as already described. However, if you must have your own custom pool, the next section describes what you need to do to process FAN events.

Consuming FAN Events

You need to explicitly consume FAN Events only if you have a custom pool. You can supply a callback that will enable you to clean up dead connections from your pool data structure:

```
void evtcallback_fn(void *evtctx, OCIEvent *eventhp);
```

where `evtctx` is the client context, which is opaque to the OCI library. The other input argument is `eventhp`, the event handle: that captures the attributes associated with an event.

If registered, this function will be called once for each event. In the case of RAC FAN events, this callback will be invoked after the affected connections have been disconnected. The following environment handle attributes are used to register an event callback and context, respectively:

`OCI_ATTR_EVTCBK` is of datatype `OCIEventCallback *`.
`OCI_ATTR_EVTCTX` is of datatype `void *`.

```
static char *myctx = (char *) "dummy context";   /* context passed to callback */
```

```
  /* set HA Event callback and context */
  checkerr(errhp, OCIAttrSet(envhp, OCI_HTYPE_ENV, (void *) evtcallback_fn,
                             0, OCI_ATTR_EVTCBK, errhp));

  checkerr(errhp, OCIAttrSet(envhp, OCI_HTYPE_ENV, myctx,
                             0, OCI_ATTR_EVTCTX, errhp));
```

Within the OCI Event callback, the list of affected server handles is encapsulated in the `OCIEvent` handle. For RAC High Availability (HA) DOWN events, client applications can iterate over a list of server handles that are affected by the event by using `OCIAttrGet()` with attribute types `OCI_ATTR_HA_SVRFIRST` and `OCI_ATTR_HA_SVRNEXT`:

```
void evtcallback_fn (void *evtctx, OCIEvent *eventhp)
{
  OCIServer *srvhp;
  sword status;

  printf ("received HA event: %s", (char *) evtctx);

  /* get the first dead connection from the event handle */
  status = OCIAttrGet(eventhp, OCI_HTYPE_EVENT, &srvhp, (ub4 *) NULL,
                      OCI_ATTR_HA_SRVFIRST, errhp);

  /* status will be OCI_NO_DATA when no more dead connections left */
  while (status == OCI_SUCCESS)
  {
    /* got a dead connection */

    /* cleanup srvhp, which is a dead connection */

    /* get next dead connection from the event handle */
    status = OCIAttrGet(eventhp, OCI_HTYPE_EVENT, &srvhp, (ub4 *) NULL,
                        OCI_ATTR_HA_SRVNEXT, errhp);
  }
}
```

The work done in the callback should be kept to a minimum in order to process the event very quickly. The application should avoid doing extensive processing in the callback. For more details on the callback approach, please see the High Availability Event Notification[8] section in the OCI docs.

An alternative approach for custom connection pool implementations, starting with OCI 11*g*, is to not register the callback at all. Instead, just before giving out a connection, the pool can check `OCI_ATTR_SERVER_STATUS`. This attribute is sensitive to FAN events and tells you if the connection is alive. `OCI_ATTR_SERVER_STATUS` is an attribute on the OCI Server Handle (which can be obtained from the OCI Service Context). If the attribute does indicate that the connection is dead, the custom connection pool can remove the dead connection from the pool and continue to look for a different connection, until it finds one that is alive.

```
/* checks if a given connection is valid */
boolean is_connection_valid(OCISvcCtx *svchp)
{
  ub4 serverStatus = 0;
  OCIServer *srvhp = (OCIServer *) NULL;

  /* get the server handle */
  checkerr(errhp, OCIAttrGet(svchp, OCI_HTYPE_SVCCTX,
                      &srvhp, 0, OCI_ATTR_SERVER, errhp));

  /* get the connection status */
  checkerr(errhp, OCIAttrGet(srvhp, OCI_HTYPE_SERVER,
        &serverStatus, (ub4 *) NULL, OCI_ATTR_SERVER_STATUS, errhp));

  /* check if the connection is gone */
  if (serverStatus == OCI_SERVER_NOT_CONNECTED)
  {
     printf("Connection is down.\n");
     return FALSE;
  }
  printf("Connection is up.\n");
  return TRUE;
}
```

Note that this attribute isn't an absolute guarantee of connection validity. There are various reasons why a connection may become invalid, e.g. a DBA could kill the server process, or the network may go down, thus making the corresponding connection invalid. In such cases, this attribute will not detect that the connection is invalid; you will know that the connection is dead only when you attempt the next OCI call to the server. However, this attribute is sensitive to high availability events on the database that result in dead connections from the client (e.g. a database instance going down) and to some of the specific errors (such as ORA-3113) that are known to drop connections.

END-TO-END DIAGNOSTIC ATTRIBUTES

In many circumstances, system administrators need to retrace incidents on the database (such as a trace file or a runaway query) back to some application, module, driver and end-user in the middle-tier. This can help narrow the cause of the problem, such as potentially unsupported/uncertified versions of drivers or missing patches on clients. Any identification of end user information also can help in establishing contact with the end user responsible for the sequence of application steps leading to the problem.

Starting with Oracle Database 11*g*, OCI exposes an attribute that allows the driver to specify its name using the `OCI_ATTR_DRIVER_NAME` attribute that may be set before creating a session:

```
OCIAuthInfo *authp;
OCIError *errhp;
…
#define CLIENT_DRIVER  “ruby-x.y.z”
...
  /* setup name of client side driver */
  checkerr(errhp, OCIAttrSet(authp, OCI_HTYPE_AUTHINFO,
                            (void *) CLIENT_DRIVER, (sizeof(CLIENT_DRIVER) -1),
                            OCI_ATTR_DRIVER_NAME, errhp));
```

This driver name attribute is passed to the server and the server tracks this for diagnostic purposes and also displays it in the `CLIENT_DRIVER` column in `V$SESSION_CONNECT_INFO` and `GV$SESSION_CONNECT_INFO`. Oracle encourages OCI-based drivers to pass this attribute during connection establishment. There are other columns of interest in the same view that are populated by default. They are:

`CLIENT_CHARSET` – Shows the client side character set

`CLIENT_OCI_LIBRARY` – Whether home-based or instant client

`CLIENT_VERSION` – Version of the client library, for 11*g* Release 1, it is 11.1.0.6.0.

OCI also exposes attributes called `OCI_ATTR_CLIENT_IDENTIFIER`, `OCI_ATTR_MODULE` and `OCI_ATTR_ACTION` that can be set on the session handle (which is obtained from the service context, as shown in the example below) before any call is made to the database. This is also useful for diagnostic purposes in correlating SQL statements issued in the database back to the end-user, module-names in the middle tier code and also to specific actions within a given module.

```
/* sets a string attribute on the session handle */
void set_session_attribute(OCISvcCtx *svchp,
                           const char *attribute_value, ub4 attribute_code)
{
  OCISession *userhp = (OCISession *) NULL;
  checkerr(errhp,
           OCIAttrGet(svchp,
                      OCI_HTYPE_SVCCTX,
                      &userhp,
                      (ub4 *) NULL,
                      OCI_ATTR_SESSION,
                      errhp));

  checkerr(errhp,
           OCIAttrSet(userhp,
                      OCI_HTYPE_SESSION,
                      (void *) attribute_value,
                      strlen((char *) attribute_value),
                      attribute_code,
                      errhp));
}

{
  /* ………get the database connection */
  checkerr(errhp, OCISessionGet(envhp, errhp,
                                &svchp,        /* returned database connection */
                                authp,    /* initialized authentication handle */
                                                          /* session pool name */
                                (OraText *) poolName, poolNameLen,
                                       /* session tagging parameters: optional */
                                NULL, 0, NULL, NULL, NULL,
                                OCI_SESSGET_SPOOL));                  /* modes */


    /* set the client id */
    set_session_attribute(svchp, "mary.smith", OCI_ATTR_CLIENT_IDENTIFIER);

      …

    /* set the module attribute on entering procedure array_fetch */
    set_session_attribute(svchp, "array_fetch", OCI_ATTR_MODULE);

    /* set appropriate action attribute prior to database calls */
    set_session_attribute(svchp, "execute_query", OCI_ATTR_ACTION);

    do_work_on_session(svchp);
}
```

This information appears in `V$SESSION`:

```
SQL> select client_identifier, module, action from v$session  where
client_identifier like 'mary%';

CLIENT_IDENTIFIER MODULE          ACTION
----------------- --------------- ---------------
mary.smith        array_fetch     execute_query
```

This information can also appear in oracle traces files and is very useful for diagnosing and correlating database side issues back to a particular end user and also to a certain module/action issued by the middle tier:

```
*** SESSION ID:(86.1903) 2008-04-15 14:01:19.382
*** CLIENT ID:(mary.smith) 2008-04-15 14:01:19.382
*** SERVICE NAME:(sales.us.acme.com) 2008-04-15 14:01:19.382
*** MODULE NAME:(array_fetch) 2008-04-15 14:01:19.382
*** ACTION NAME:(execute_query) 2008-04-15 14:01:19.382
```

Starting with Oracle Database 11*g*, the OCI client is also integrated into the Automatic Diagnostic Repository (ADR)[9] framework. An occurrence of a problem (such as a crash) on the OCI client is captured without user intervention in the form of diagnostic data in trace files. The trace files go into the `ADR_BASE` location specified in sqlnet.ora. Lookup Fault Diagnosability [10] in the OCI documentation for details on the location of the trace files.

CACHING QUERY RESULTS ON THE CLIENT

If your driver has a requirement to cache query results for frequently issued queries on read-only or read-mostly tables, you should consider using the 11*g* OCI Consistent Client Cache.

OCI creates and manages the cache in process memory and the cache is shared by all the sessions/threads in a process. OCI also does all of the cache memory management and concurrency control. Using a combination of Oracle's unique Snapshot based Read consistency and Database Change Notification technologies, OCI maintains the cache consistency transparently by detecting all updates to cached objects and invalidating them as necessary.

When the cache is enabled, all queries with the hint `/*+ result_cache */` begin to get cached on the client side. If the query result is found on the client, the `OCIStmtExecute()` and subsequent `OCIStmtFetch()` calls on the client are purely local calls and eliminate the round-trip and any related computation on the database tier.

The OCI client cache can be enabled at a particular deployment by setting the `CLIENT_RESULT_CACHE_SIZE` (default 0, cache disabled) in your init.ora on the database. You can also specify `CLIENT_RESULT_CACHE_LAG` (optional, 3000ms default), which sets an upper bound on how far back the client cache can lag behind the database in terms of time. There are also additional client cache parameters you can set in sqlnet.ora on the client tier to override the database setting: `OCI_RESULT_CACHE_MAX_SIZE, OCI_RESULT_CACHE_MAX_RSET_SIZE and OCI_RESULT_CACHE_MAX_RSET_ROWS.`

Using the OCI Consistent Client Cache allows you and your users to focus on your business problems instead of spending development effort on building caching and invalidation infrastructure. The OCI cache is certified with Oracle supplied drivers such as JDBC-OCI, ODBC, ODP.Net and OCCI.

PREFERED PROGRAMMING MODELS

How many Environment Handles should I have?

In general, if you are using OCI Session Pool you should have one environment handle being used by all threads that need to share the same OCI Session Pool.

If your driver or application needs to support different client character sets simultaneously, you may need to go with a separate `OCIEnv` handle per character set.

How many Error Handles should I have?

The main consideration is that if a thread does an `OCIXYZ()` call with an error handle and if the `OCIXYZ()` call returns a certain error, the thread should have sole access to the error handle in order to do a following `OCIErrorGet()` call to retrieve the error code and error message. If some other thread gets to use the error handle for a different concurrent call in the meantime, before the first thread can do the `OCIErrorGet()` call, the second thread could end up overwriting the error code intended for the first thread.

In order to avoid such race conditions, Oracle recommends that you have one error handle per thread. Alternatively, you could also keep error handles that could be shared between threads (e.g. one per connection) but you need to serialize access to these handles appropriately to avoid the race condition described above.

Should I perform Defines before or after Execute?

We recommend that if the shape of the SQL statement is known in advance, it is more efficient to perform the `OCIDefine` calls before doing the `OCIStmtExecute()`. Also, the `OCIStmtExecute()` call could specify the number of rows to be fetched and that avoids calling `OCIStmtFetch()` altogether.

On the other hand, if the driver deals with arbitrary SQL statements whose shape is not known in advance and if it needs to describe the statement before allocating and assigning `OCIDefine` buffers, then there is no choice but to do the `OCIDefine` calls after `OCIStmtExecute()`.

Should I pre-fetch or should I array-fetch?

If the application cannot bundle an execute and fetch together as described in the earlier section, then it makes sense to set `OCI_ATTR_PREFETCH_SIZE` and `OCI_ATTR_PREFETCH_ROWCNT` when executing the statement for the very first time. OCI will prefetch the rows during execution and some or all of the following fetch calls can be dealt with locally. This reduces roundtrips to the database.

However, once you have described the SELECT list and performed the `OCIDefine` calls, we recommend that you rely on Array Fetching instead of prefetching. This is because Array Fetching also gives you the benefits of reduced roundtrips and doesn't require an additional data copy from OCI prefetch buffers into the `OCIDefine` buffers.

Should I use non-blocking OCI?

Oracle discourages using non-blocking OCI in favor of using blocking connections in conjunction with multi-threading. In a multi-threaded application or in a driver capable of multithreading, Oracle recommends sticking to the default (blocking) mode for connections. In such cases, any task that needs to be done when you are blocked on a database connection for network i/o can be designated to a different thread.

Usage of non-blocking OCI should only be considered in single threaded programs if the sole thread of execution absolutely has to do the multitasking all by itself and hence cannot afford to block on a connection.

How do I interrupt an OCI call?

Many drivers have the need to interrupt calls to the Database if a timeout expires. Our recommendation is to create a monitor thread that is responsible for keeping track of active connections on which a request has been initiated and if the timeout expires, it should initiate an `OCIBreak()` on that connection. Once the `OCIBreak()` is done, the blocked thread is woken up and the OCI call that was in progress at that time gets an ORA-1013.

How do I handle connection failures?

FAN helps minimize application exposure to dead connections. However, even with FAN enabled, you could still get a connection failure right when your database call is in flight. In that case, you need to return the dead connection back to the pool so that the connection pool can reclaim any resources associated with that connection.

Whether the application can recover from the error depends on the semantics of the operation in progress. If the operation being performed on the database is idempotent (e.g. a set of queries on the database), then the operation might be safe to retry. On the other hand, if the operation being done is not idempotent (e.g. if it involves transactions), then the application may need to do further checks before determining that the operation is safe to retry.

BUILDING YOUR OCI PROGRAM

For compiling the above examples, you will need to include the following header files:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>
```

Add to this list based on your requirements. You can use the demo_rdbms.mk makefile to build your program. If you are on Windows, you can follow the build process in the OCI Windows Documentation[11].

Many drivers have a requirement to build for different Oracle client releases but at the same time need to leverage newer calls and functionality specific to later releases. Instead of resorting to maintaining different code bases for each release of the client, we recommend that you consider using conditionally compiled code based on `OCI_MAJOR_VERSION` and `OCI_MINOR_VERSION`. As an example:

```
#if ( (OCI_MAJOR_VERSION > 10) || ((OCI_MAJOR_VERSION == 10) &&
                             (OCI_MINOR_VERSION >= 2)) )    /* OCIXYZ available */
   Do OCIXYZ call
#else
   Do older call
#endif
```

INSTANT CLIENT

The Instant Client feature makes it extremely easy to deploy OCI based applications by eliminating the need for an `ORACLE_HOME`. The installation involves copying just three libraries. You can download the OCI Instant Client for free from OTN.[12] For more details, please look up the Instant Client Documentation.[13]

UPGRADING THE CLIENT

OCI applications that are dynamically linked with the client shared library from 10*g* and later releases are drop-in compatible with the current release. That is, the current Oracle client-side shared library is compatible with the previous version of the library in terms of preserving the documented behavior of all the OCI calls. Starting with Oracle Database 11*g*, the Oracle Installer creates a symbolic link for the previous version of the library that resolves to the current version. Therefore, an application that is dynamically linked with the previous version of the Oracle client-side dynamic library (from 10*g* and later) does not need to be re-linked to operate with the current version of the Oracle client-side library.

With each release of the Database, OCI introduces newer interfaces that provide access to the latest Oracle Database functionality. You are encouraged to leverage these enhancements in the latest release of your application/driver. For more details, please see Upgrading Your Applications.[14]

UPGRADING THE SERVER

Oracle recommends that you upgrade your client software to match the current server software. For example, if you upgrade your server to Oracle Database 11*g* Release 1 (11.1), then Oracle recommends upgrading the client software to Oracle Database 11*g* Release 1 (11.1) as well. Keeping the server and client software at the same release number ensures the maximum stability for your applications. In addition, the latest Oracle Database client software might provide added features, performance enhancements and fixes that were not available with previous releases.

However, Oracle also supports a mixed client/server environment as well. If your client and server do not share the `ORACLE_HOME`, and you upgrade the database, you are not required to re-code, re-compile or re-link your OCI applications. Look up note 207303.1 for the currently supported client/server interoperability matrix at MetaLink.[15]

CONCLUSION

This paper has provided a wide range of pointers on making the best use of the Oracle Call Interface for building the best possible drivers for emerging dynamic languages and environments for the Oracle Database. Our hope is that the reader would have benefited from the tips and techniques presented in the paper. These touch upon a wide range of topics and questions that we at Oracle have seen various driver developers repeatedly encounter and for which we have, over time, established clear guidelines and best practices. It is our hope that the pointers can be made use of by driver maintainers and designers as they plan and implement the future versions of such drivers.

---

[1] Oracle Database 11*g* Globalization Support Guide:
http://download.oracle.com/docs/cd/B28359_01/server.111/b28298/applocaledata.htm#i635016
[2] DRCP white paper:
http://www.oracle.com/technology/tech/oci/pdf/oracledrcp11g.pdf
[3] Introduction to Oracle Datatypes
http://download.oracle.com/docs/cd/B28359_01/server.111/b28318/datatype.htm#CNCPT113
[4] Datatypes in OCI
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/oci03typ.htm#LNOCI030
[5] Character Conversion in OCI Binding and Defining
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/oci05bnd.htm#sthref657

[6] Data Interface for Persistent LOBs
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28393/adlob_data_interface.htm#g1029381
[7] Prefetching of LOB Data, Length and Chunk Size
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/oci07lob.htm#LNOCI07100
[8] HA Event Notification
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/oci09adv.htm#CJGDCEFG
[9] Automatic Diagnostic Repository
http://download.oracle.com/docs/cd/B28359_01/server.111/b28310/diag001.htm#ADMIN11261
[10] Fault Diagnosability in OCI:
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/oci10new.htm#LNOCI1020
[11] Compiling OCI Applications for Windows:
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/ociadwin.htm#sthref5771
[12] Instant Client on Oracle Technology Network:
http://www.oracle.com/technology/tech/oci/instantclient/index.html
[13] Instant Client Documentation:
http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/oci01int.htm#LNOCI019
[14] Upgrading Your Applications:
http://download.oracle.com/docs/cd/B28359_01/server.111/b28300/app.htm#UPGRD006
[15] MetaLink:
http://metalink.oracle.com/

USEFUL POINTERS

OCI Documentation:

http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28395/toc.htm

OCI OTN Page:

http://www.oracle.com/technology/tech/oci/index.html

OCI Forum:

http://forums.oracle.com/forums/forum.jspa?forumID=67

Instant Client OTN Page:

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

Instant Client Forum:

http://forums.oracle.com/forums/forum.jspa?forumID=190

Building High Performance Drivers for Oracle Database 11*g*: OCI Tips and Techniques
[April] 2008
Author: Luxi Chidambaran
Contributors: Santanu Datta, Christopher Jones, Srinath Krishnaswamy, Kuassi Mensah, Krishna Mohan, Kevin Neel

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

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com