Using OCCI: Best Practices

An Oracle White Paper
May 2003

ORACLE

OCCI Best Practices

INTRODUCTION

OCCI offers the convenience of having direct access to the Oracle Database using C++. While developers can use OCI, OCCI offers seamless integration with the C++ language.

Some of the benefits of using OCCI over OCI include a cleaner object oriented interface for managing the environment as well as a more intuitive interface to Oracle Objects. It is possible to use OCCI for the more mundane setup tasks (environment setup, connection handling, etc.) and still use OCI for the more granular operations. An introduction to using Oracle Objects with OCCI is the topic of another paper: *Using OCCI: An Introduction to Objects*.

This paper describes some of the "best practices" programmers may consider when designing their applications and is targeted toward a more advanced development audience. Most of the tips involve caching or amortizing an expensive operation over many executions. For example, connection pooling amortizes the cost of all the network initiation by keeping a set of physical connections open for multiple logical connections. Caching is used when setting the prefetching parameters of a ResultSet. Some tips concern efficient memory management, including using Ref pointers rather than standard C++ pointers for Oracle Objects.

STATEMENT OBJECT OPTIMIZATIONS

Statement Object Reuse

Each time a Statement object is created, resources such as memory and cursors must be allocated on both the client and server sides to store the object along with its data. In order to save reallocations of memory, try to reuse statement objects. After creating statements objects, they can be reused simply by using the setSQL method as in the following example:

```
Connection* conn = env->createConnection();
Statement* stmt = conn->createStatement();
stmt->setSQL("INSERT INTO fruit_basket_tab VALUES('Apples', 3)");
stmt->executeUpdate();
stmt->setSQL("INSERT INTO fruit_basket_tab VALUES('Oranges', 4)");
stmt->executeUpdate();
stmt->setSQL("INSERT INTO fruit_basket_tab VALUES('Bananas', 1)");
stmt->executeUpdate();'
stmt->setSQL("SELECT * FROM fruit_basket_tab WHERE quantity > 2");
ResultSet* rs = stmt->executeQuery();
```

Statement Parameterization

To save even more memory reallocations, it is possible to parameterize the first three SQL statements into one, set the parameters, and then execute. Be wary of changing input parameter types, however, since a rebind must be performed each time the type changes. The following example demonstrates parameterization:

```
stmt->setSQL("INSERT INTO fruit_basket_tab VALUES(:1, :2)");
stmt->setString( 1, "Apples" );
stmt->setInt( 2, 3 );
stmt->executeUpdate();
stmt->setString( 1, "Oranges" );
stmt->setInt( 2, 4 );
stmt->executeUpdate();
stmt->setString( 1, "Bananas" );
stmt->setInt( 2, 1 );
stmt->executeUpdate();
```

Batch Update

For some operations that often occur in batches, much time is wasted in the network roundtrip communication time to the server. OCCI provides an efficient mechanism for sending multiple rows of information in a single network roundtrip. This optimization applies to INSERTs, UPDATEs, and DELETEs. First, you must set the maximum number of iterations, and then set the maximum parameter size for variable length parameters. Parameters cannot change types in iterative operations. See *OCCI Programmers* Guide, Chapter 2 for more details. Following is the optimized form of the above INSERTs:

```
//prepare the batching process
stmt->setMaxIterations( 3 );
stmt->setMaxParamSize( 1, 8 ); //"Bananas" is longest param

//batch the statements
stmt->setSQL("INSERT INTO fruit_basket_tab VALUES(:1, :2)");
```

```
stmt->setString( 1, "Apples" );
stmt->setInt( 2, 3 );
stmt->addIteration();
stmt->setString( 1, "Oranges" );
stmt->setInt( 2, 4 );
stmt->addIteration();
stmt->setString( 1, "Bananas" );
stmt->setInt( 2, 1 );

//execute the statements
stmt->executeUpdate();
```

Use Appropriate Accessors and Character Set Settings

Rather than retrieving everything as a string, you should use the appropriate setXXX and getXXX methods for the column data that you are operating upon, saving on unnecessary conversions.

Make sure to use the appropriate character set settings are in the NLS_LANG environment setting to avoid unnecessary character set conversions when retrieving strings.

AutoCommit Mode

Since all SQL DML is executed in the context of a transaction, all DML must be committed. You can use "Connection::commit" and "Connection::rollback" methods accordingly. The "Statement::setAutoCommit" method can be used to commit every statement that follows. It is possible to save a network roundtrip time by using the "Statement::setAutoCommit" method:

```
//code with AutoCommit
//transaction 1
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Apples",3));
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Oranges",4));
stmt->setAutoCommit( TRUE );
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Bananas",1));
stmt->setAutoCommit( FALSE );

//transaction 2
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Apples",5));
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Oranges",6));
stmt->setAutoCommit( TRUE );
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Bananas",2));
stmt->setAutoCommit( FALSE );
```

which is equivalent to this, but saves 2 network roundtrips, 1 per transaction:

```
//code without AutoCommit
//transaction 1
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Apples",3));
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Oranges",4));
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Bananas",1));
conn->commit();

//transaction 2
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Apples",5));
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Oranges",6));
stmt->executeUpdate("INSERT INTO fruit_basket_tab VALUES("Bananas",2));
conn->commit();
```

We recommend that AutoCommit only be used right before the last statement in a transaction.

Result Set Object Optimizations

The result set is returned in response to a query. We can manipulate the results with the "next" and "status" methods:

```
ResultSet* rs = stmt->executeQuery( "SELECT * FROM fruit_basket_tab" );
ResultSet::Status stat = rs->status(); //status is DATA_AVAILABLE
while( rs->next() ) { //process data }
```

setPrefetchRowCount and setPrefetchMemorySize

Although you can only get a single row to process at a time using rs->next(), there is no reason that you cannot prefetch many rows for the client side cache in one roundtrip to the server. Using either "setPrefetchRowCount" or "setPrefetchMemorySize" from the "Statement" class, you can fetch more than 1 row at a time. An optimized form of the above example is:

```
stmt->setPrefetchRowCount( 3 );
ResultSet* rs = stmt->executeQuery( "SELECT * FROM fruit_basket_tab" );
while ( rs->next() ) { //process data }
```

Using the preceding code, we can now fetch 3 rows in one network roundtrip. By default, prefetching is turned on, with 1 extra row prefetched. To turn off prefetching completely, you MUST use both "setPrefetchRowCount" and "setPrefetchMemorySize" with the setting of 0. If both setPrefetchXXX parameters are set, then the actual amount prefetched is the minimum of the size of the setPrefetchRowCount and the setPrefetchMemorySize parameters. See the *OCCI Programmer's Guide* for further details.

setMaxColumnSize

When retrieving results with large columns, it is possible to use ResultSet::setMaxColumnSize to limit how much data is retrieved from a particular column. This can be useful if you are only interested in a small amount of the actual data in the column or have limited buffer space.

```
ResultSet *rs = stmt->executeQuery( "SELECT description FROM
       fruit_basket_tab" );

//want only first 80 characters from the description column
rs->setMaxColumnSize( 1, 80 );
```

Closing the ResultSet

Use the Statement::closeResultSet method to manually force a close on a ResultSet as soon as you are done with it, so that the database and OCCI resources are freed instead of waiting for an automatic release.

Statement Termination

Finally, to make sure there is no memory leakage, and to close the associated server-side cursor, remember to free all statement objects as soon as you are finished using them:

```
conn->terminateStatement( stmt );
```

REF OBJECT OPTIMIZATIONS

The REF object is a very powerful mechanism for manipulating Oracle Objects. It is effectively a smart pointer, and can only be used when instantiating or retrieving persistent objects when employing the navigational access pattern. The following general tips apply to both the Ref and RefAny classes, except where noted.

Use REFs when possible

Refs are designed to be smart pointers. For persistent objects, it should be the only way to access them. Thus use:

```
Ref<Person> p = new (conn, table) Person();
```

rather than

```
Person* p = new (conn, table) Person();
```

The type specific methods can still be accessed using the "->" dereferencing operator:

```
Ref<Person> p = new (conn, table) Person();
//access the Person::setName method
p->setName("John");
```

Refs keep track of how many references there are to a specific object, unpinning them at the appropriate time. In addition, there are specific methods you can operate on Refs to make object access faster.

markDelete

When deleting persistent objects from the database, use the Ref or RefAny markDelete method. While all Oracle persistent objects have the markDelete method inherited from PObject, the Ref::markDelete method can be invoked before an Object is pinned, saving network roundtrips and copying time. For example:

```
//ok:  PObject's markDelete
Ref<Person> p = rs->getRef(1);
p->markDelete();
conn->commit();

//BETTER:  Ref/RefAny markDelete
Ref<Person> p = rs->getRef(1);
p.markDelete();
conn->commit();
```

setPrefetch

The setPrefetch method allows customizing the prefetching of Complex Object Retrieval (COR). Complex Objects are Objects that contain other Objects. As an example, suppose we have a type OrderList that contains a REF to a type Customer and another REF to a type Address. It is possible to setPrefetch on just the Customer if that information is always used when retrieving the OrderList:

```
Ref<OrderList> r = rs->getRef(1);
```

```
//retrieve all Customer Objects with a depth of 1
r.setPrefetch( "Customer", 1 );
```

or retrieve all objects to a certain depth:

```
Ref<OrderList> r = rs->getRef(1);
//retrieve all Objects within a depth of 2
r.setPrefetch( 2 );
```

Limit the Scope of a Ref

Try to keep the scope of Refs as local as possible when declaring them. That way, when the scope is done, the Ref can be deallocated and the associated object unpinned as soon as possible. This helps manage the object cache optimally.

Declare and Initialize Refs Simultaneously

If possible, declare a Ref and initialize it immediately. This practice saves creating a copy for construction and another for assignment, replacing it simply with a copy constructor:

```
//good
Ref<Person> p = rs->getRef(1);

//bad
Ref<Person> p;
p = rs->getRef(1);
```

Unpin Objects that have been explicitly Pinned

Since the Ref mechanism automatically pins and unpins objects, it should be unnecessary to ever explicitly pin an object with the PObject::pin method. However, if an application has explicitly pinned an object, then the application must unpin the object with the PObject::unpin method. **We strongly discourage ever explicitly pinning a persistent object in the first place, since Refs should always be used and will automatically pin as necessary.**

Using Ref versus RefAny

There are two types of REFs: Ref and RefAny. RefAny should be used when the logic is independent of the specific type. In such instances where the logic passing the Objects around, use the RefAny, since it has less instantiation overhead than a Ref<T> instantiation. For example:

```
/* processes an object through a series of steps, but does not access
        Type specific functionality */
void dispatch( RefAny object )
{
        //perform different actions
        step1( object );
        step2( object );
        step3( object );

        return;
}
```

However, when fetching REFs from a ResultSet, make sure to use Ref as follows to avoid unnecessary Ref instantiations when you are using Type specific functionality:

```
//good
Ref<Person> r = rs->getRef(1);
r->setName("John");

//bad
RefAny r = rs->getRef(1);
Ref<Person> rp = r;
r->setName("John");
```

MISCELLANEOUS AND ADVANCED TECHNIQUES

Use isNull on OCCI Objects

A generally good practice for programmers is to check whether an OCCI Object is null before accessing it. This applies to all OCCI objects, including the Statement, Environment, Connection, and PObject classes.

Deleting Objects

Persistent Objects: Do NOT delete objects that are obtained as a result of calling the Ref::ptr method. In general, do not get Ref pointers. Use only Refs. However, if a rollback occurs, you must delete the persistent objects.

Transient Objects: Always delete transient objects manually. They cannot and will not be managed by Refs.

Statement::setDataBuffer Method

Binding values to parameters of parameterized statements carries with it a memory copying cost, since the values must be copied to internal buffers in order to protect the information from overwrites during mid-execution. Especially for large strings, the overhead of copying can be expensive, both in terms of memory consumed and time spent. If the application manages its own memory, it can use some OCCI provided methods to minimize some of this overhead.

Many OCI developers use OCCI for its ability to cleanly create environments and statement objects, but still use many OCI types. The setDataBuffer method allows these OCI developers to use their OCI types to execute array updates, minimizing the number of network roundtrips required. The setDataBuffer works very differently from the setXXX methods. Normally, the setXXX methods copy the data passed in to internal buffers, and so the arguments can be altered as soon as the setXXX methods return. However, by using the setDataBuffer method, the application can completely avoid the cost of copying the data to internal buffers. In exchange for this savings, the application cannot modify the buffer until the after the execution of the statement. For example:

```
// insert Bananas
char buf[BUF_SIZE] = "Bananas";
int quantity = 1;
ub2 buflen = strlen( buf ) + 1;
ub2 quantlen = sizeof(int);

stmt->setDataBuffer(1, (dvoid*)buf, OCCI_SQLT_STR, buflen, &buflen);
stmt->setDataBuffer(2, (dvoid*)&quantity, OCCIINT, quantlen,
       &quantlen);

stmt->executeUpdate(); // executeArrayUpdate(1) also would work.

// insert Apples
strcpy( buf, "Apples" );
quantity = 3;
buflen = strlen( buf ) + 1;
quantlen = sizeof( int );
```

```
stmt->setDataBuffer(1, (dvoid*)buf, OCCI_SQLT_STR, buflen, &buflen);
stmt->setDataBuffer(2, (dvoid*)&quantity, OCCIINT, quantlen,
        &quantlen);

stmt->executeUpdate(); // executeArrayUpdate(1) also would work.

//commit the transaction
conn->commit();
```

The setDataBuffer method can be used in conjunction with iterative executes and the executeArrayUpdate method detailed below. See below for more details and an example.

executeArrayUpdate Method

When doing many INSERTs or UPDATEs, it is possible to batch the operation using the executeArrayUpdate in conjunction with the setDataBuffer method. This saves on many network roundtrips to the server, leading to higher throughput. Following is an example:

```
char fruit[][BUF_SIZE] = { "Apples","Oranges","Bananas","Grapes" };
int int_arr[]={ 3,4,1,5 };
ub2 fruitlen[4]; // array of size of individual elements
ub2 intsize[4];

for(int i=0 ; i<4 ; i++)
{
        intsize[i] = sizeof(int);
        fruitlen[i] = strlen( fruit[i] ) + 1 ; // include the null
}

stmt->setDataBuffer(1, (dvoid*)fruit, OCCI_SQLT_STR, BUF_SIZE,
        fruitlen);
stmt->setDataBuffer(2, (dvoid*)int_arr, OCCIINT, sizeof(int), intsize);
stmt->executeArrayUpdate(4);
conn->commit();
```

executeArrayUpdate() does not work unless **ALL** buffers are set using the setDataBuffer method. If any parameters need to use the setXXX methods, instead, use the setMaxIterations and setMaxParamSize methods, along with the addIteration method as detailed here:

```
char fruits[][BUF_SIZE] = {"Apples", "Oranges", "Bananas"};
ub2 fruitLen[3];
for( int j=0; j<3; j++ )
{
        fruitLen[j] = strlen( fruits[j] ) + 1; //include the null
}
stmt->setMaxIterations(3);

//setDataBuffer only needs to be executed once
//while all the other variables need to be set for each iteration

stmt->setDataBuffer( 1, fruits, OCCI_SQLT_STR, sizeof(fruits[0]),
        fruitLen );
stmt->setInt(2, 3); //Apple's quantity
stmt->addIteration();
stmt->setInt(2, 4); //Orange's quantity
stmt->addIteration();
stmt->setInt(2, 1); //Banana's quantity

//execute the iterative update
stmt->executeUpdate(3);
```

Object Cache Management

Use the Environment::setCacheOptSize and Environment::setCacheMaxSize methods to set the optimal and maximum size of the client-side Object Cache. The setCacheMaxSize takes a percentage over the optimal size to allow the Object Cache to grow (i.e. 10% = 10). Use as:

```
//set the optimal cache size to 8MB
env->setCacheOptSize( 8388608 );
//set the max cache size to 10% over the optimal size
env->setCacheMaxSize( 10 );
```

If you have many objects with quite a few string attributes or collection attributes, most of the actual memory allocated for these objects comes from the C++ heap due to OCCI using the STLs. Thus, the cache size should be set to low values to avoid large memory usage before the garbage collector activates.

When using only associative access, take care to set the cache size limits as accurately as possible. Even though OCCI is not allowed to manage the memory of the transient objects used in associative access, OCCI will nonetheless run the garbage collector each time more memory needs to be allocated and the current memory footprint exceeds the limits, finding that it cannot free any of the memory currently in use. Thus, it is better in this instance to err on the side of a slightly larger cache than needed.

Connection Management

Flushing the Cache

Flush the cache on a connection using the Connection::flushCache method. This call flushes all new and modified objects to the database server. Calling this method reduces the time for commits later. We recommend that you periodically flush the cache on the connection.

Connection Pooling

When managing multiple connections, use the ConnectionPool API. This API allows the multiplexing of logical connections over a pool of physical connections. OCCI manages the actual number of physical connections under the covers, within the ranges set by the application. The benefits include amortizing the setup of connections, and keeping the number of connections in some optimal range. Here is how to create and use it:

```
//create a ConnectionPool
//0 connections is the min
//10 connections is the max
//1 connection is the incremental number to increase when more needed
ConnectionPool* connpool = env->createConnectionPool( "scott", "tiger",
       "orcl", 0, 10, 1 );

//get a Connection from the ConnectionPool
Connection* conn = connpool->createConnection( "scott", "tiger" );
```

```
//process data

//return Connection to ConnectionPool
connpool->terminateConnection( conn );
```

The creation of Connections from a Connection Pool is very lightweight, and often involves only assigning a physical connection to a logical Connection pointer.

Exception Handling

Use separate try-catch blocks for different parts of the application functionality, handling exceptions appropriately. Also, catch the OCCI exceptions, SQLException objects, by reference, saving time and memory allocations for copying the exceptions:

```
try{    //process data  }
catch( SQLException &e )
{
        //exception handling
        cout << "Error: " << e.getMessage() << endl;
}
```

Using OCCI: Best Practices
May 2003
Author: Toliver Jue
Contributors: Shankar Iyer, Krishna Mohan, Chetan Parampalli, Rekha Vallam

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
www.oracle.com