Making efficient use of Oracle thru Apache and PHP 4

(c) Thies C. Arntzen <thies@thieso.net> 2001-2002

introduction

this paper will give you an overview of the points covered in my "Making efficient use of Oracle" session. it will cover:

- connecting
- transactions
- error-handling
- binding & defining
- autoincrement
- efficient "select for update" using ROWID
- large objects
- nested tables
- stored procedures
- refcursors
- collections

connecting

when you connect to an oracle-instance thru PHP you get back a db-handle:

```
$db = OCILogon("scott","tiger");
```

this db-handle consists of a few individual oracle structures. the important ones are the server-handle which holds the connection to the oracle-server second we need the session-handle which carries the authenticated user/password and the third one is our so called service-context.

during the login the most expensive operation is setting up the server-handle as this actually creates the oracle shadow process on the server side. starting a session over this already established link is still somehow expensive but relatively cheap compared to the former. last but not least creating the service-context is a no-brainer as the creation of the service-context does not even need a sever-round-trip (= the client lib needs to talk to the server)

therefore PHP tries it's best to avoid the creation of unneeded server-handles and session-handles. all of this complexity is hidden inside the three available log-on functions within PHP:

OCILogon($username,$password[, $tnsname]);

    gets you a standard, non-persistent connection that will be closed
    automatically by PHP on script-end. PHP will reuse existing
    server-handles and session-handles.

OCIPLogon($username,$password[, $tnsname]);

    will do the same as OCILogon() but mark the sever and session handle
    as persistent, which means that PHP won't close them on script-end. if
    your script does a OCILogon("s","t") and later a OCIPLogon("s","t")
    _no_ new server or session handle will be created but instead the
    existing ones will be marked persistent.

OCINLogon($username,$password[, $tnsname]);

    will reuse server-handles but always create a new session handle. this
    is because oracle has tied the transaction context to the
    session-handle. so if you need to isolate transactions on one page you
    would use OCILogon() or OCIPLogon() for the "main" connection and
    create an additional connection using OCINLogon(). the OCINLogon()
    connection would be handled thru the same server-handle as the other
    connection but would have it's own session-handle. side-note: the
    session-handle created by OCINLogon() will always be freed at the end
    of the script.

if we have a page like this:

```
<?php

  $scottDB = OCILogon("scott","tiger","TC");

  // will create a server-handle and connect to to database "TC"

  $sysDB = OCILogon("sys","manager","TC");

  // will reuse the "TC" server-handle from our 1st OCILogon() call but
  // create a new session-handle

  $paulDB = OCIPLogon("paul","lion","TC");

  // will reuse the "TC" server-handle from our 1st OCILogon() _and_ mark
  // it persistent. as a new session-handle is created for "paul"

  $scott2DB = OCIPLogon("scott","tiger","TC");

  // will reuse "TC" (which is already marked as persistent) and reuse
  // the "scott" session-handle and mark that as persistent as well
```

```
  $scott3DB = OCINLogon("scott","tiger","TC");


  // will reuse "TC" (which is already marked as persistent) and create a
  // new session-handle as we used OCINLogon()

  ....
?>
```

is this page (which doesn't make any sense in it's current form) gets loaded the picture changes:

```
<?php

  $scottDB = OCILogon("scott","tiger","TC");

  // "TC" from cache, "scott" from cache

  $sysDB = OCILogon("sys","manager","TC");

  // "TC" from cache, create session-handle fro "sys"

  $paulDB = OCIPLogon("paul","lion","TC");

  // "TC" from cache, create session-handle fro "paul"

  $scott2DB = OCIPLogon("scott","tiger","TC");

  // "TC" from cache, "scott" from cache

  $scott3DB = OCINLogon("scott","tiger","TC");

  // "TC" from cache, new "scott" session as we want to isolate
  // transactions.

  ....
?>
```

the important thing is that we only need one server-connection per database-server that we connect and that we can run as many session thru this one channel as we like and that PHP does the thinking for us!

transactions

by default PHP works in the COMMIT_ON_SUCCESS mode (auto-commit-mode).

names.sql

```
create table names (id number, name varchar2(32));
```

```
<?php
    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"insert into names values (1,'thies')");
    OCIExecute($stmt);

    // PHP does _all_ cleanups for us!
?>
```

if we want to do the commit ourself:

```
<?php
    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"insert into names values (1,'thies')");
    OCIExecute($stmt,OCI_DEFAULT);

    OCICommit($db); // or OCIRollback($db);

    // PHP does _all_ cleanups for us!
?>
```

note: see the OCI_DEFAULT parameter in the OCIExecute() call!

as said before oracle ties the transaction-context to the session-handle which means that if we only use OCILogon() (and not OCINLogon()) we won't get any transaction-isolation.

if we need to have a second isolated transaction-context in our sample we would have to write:

```
<?php
    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"insert into names values (1,'thies')");
    OCIExecute($stmt,OCI_DEFAULT);

    $idb = OCINLogon("scott","tiger");
    $istmt = OCIParse($idb,"select name from names where id = 1");
    OCIExecute($istmt);
    while (OCIFetch($istmt))
        echo OCIResult($istmt,"NAME")."\n"; // will not show the just inserted
                                            // row as we live i a different
                                            // transaction context and the
                                            // insert is not committed yet!

    OCICommit($db);
    echo "\n";

    OCIExecute($istmt);                     // reexecute the "select" statement
    while (OCIFetch($istmt))
        echo OCIResult($istmt,"NAME")."\n"; // will now show the just inserted
                                            // row!
```

```
        // PHP does _all_ cleanups for us!
    ?>

    BTW: on script end all uncommitted transactions will get rolled-back!

error-handling
==============

    PHP stores oracle-errors in the context they happened. you have to call
    OCIError() to retrieve the error-code and error-message from oracle. the
    function OCIError() returns FALSE if no error has occurred in the selected
    context or an associative array containing the "code" and "message" of the
    error. this little example illustrates how:

    <?php

        $db = OCILogon("scott","tiger");
        if (! $db) {                    // connection failed
            // as we don't have a connection yet the error is stored in the
            // module global error-handle
            $err = OCIError();

            if ($err[ "code" ] == "12545") {
                echo "target host or object does not exist\n";
            }
            die();
        }

        $stmt = OCIParse($db,"select sysdate from dual");
        if (! $stmt) {
            // when OCIParse() reports an error (usually a parse-error) the
            // error is stored in the connection-handle that was used for
            // parsing.

            $err = OCIError($db);
            echo $err[ "code" ]." ".$err[ "message" ]."\n";
            die();
        }

        if (! OCIExecute($stmt)) {
            // when OCIExecute() fails the error is stored in the supplied
            // statement-handle

            $err = OCIError($stmt);
            echo $err[ "code" ]." ".$err[ "message" ]."\n";
            die();
        }
    ?>

    all errors that are reported by oracle are also sent to the standard PHP
    error-handler with E_WARNING priority. so if you want to handle all oracle
```

related errors yourself it's maybe a good idea to prefix all OCI*() calls with an '@'.

binding & defining

one of the more advanced features of oracle and the PHP interface is that it allows user defined bind- and define-variables. binding is the technique that allows to have placeholder in your sql-statement and oracle will take the value of the bound-variable directly from script-space. this has two major advantages: you don't have to escape quotes in you field-values as would have to do if you were in-lining the values in your sql-statement. example: you want to insert O'Hare into your oracle DB so you have to say "insert into names values ('O''Hare')". even though PHP can easily escape the apostrophe it's still an extra step.

the bigger advantage is that oracle is very smart about reusing already parsed statements. parsing statements is considered an expensive operation on oracle. user-rights, field-name etc are checked against the db during statement parsing. therefore oracle caches the parsed representation of your sql in the so called SGA (Server Global-Area). if you issue the _exact_ same sql again, oracle would find the already parsed statement and reexecuted that. so instead of saying:

```
<?php

    $data = array("2" => "larry","3" => "bill", "4" => "steve");

    $db = OCILogon("scott","tiger");
    while (list($id,$name) = each($data)) {
        $id = strtr($id,array("'","''"));
        $name = strtr($name,array("'","''"));

        $stmt = OCIParse($db,"insert into names values ('$id','$name')");
        OCIExecute($stmt);
    }
?>
```

you would say:

```
<?php

    $data = array("2" => "larry","3" => "bill", "4" => "steve");

    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"insert into names values (:id,:name)");
    OCIBindByName($stmt,":ID",$id,32);
    OCIBindByName($stmt,":NAME",$name,32);

    while (list($id,$name) = each($data))
        OCIExecute($stmt);
```

```
?>
```

voila! we'll see more about binds later!

defining on the other hand allows us to receive rows from oracle into predefined PHP-variables.

instead of saying:

```
<?php

    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"select * from names");
    OCIExecute($stmt);

    while (OCIFetch($stmt)) {
        echo "id:".OCIResult($stmt,"ID")."\n";
        echo "name:".OCIResult($stmt,"NAME")."\n";
    }
?>
```

we can say:

```
<?php

    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"select * from names");

    // note that oracle converts all colum-names to UPPERCASE
    OCIDefineByName($stmt,"ID",$id);
    OCIDefineByName($stmt,"NAME",$name);

    OCIExecute($stmt);

    while (OCIFetch($stmt)) {
        echo "id:$id\n";
        echo "name:$name\n";
    }
?>
```

this combined with OCIBindByName() would allow us to copy one table to another without any script-level assignments:

```
names2.sql
create table names2 (id number, name varchar2(32));

<?php

    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"select * from names");
```

```
        $insert = OCIParse($db,"insert into names2 values (:ID,:NAME)");

        // note that oracle converts all column-names to UPPERCASE
        OCIDefineByName($stmt,"ID",$id);
        OCIDefineByName($stmt,"NAME",$name);

        OCIBindByName($insert,"ID",$id,32);
        OCIBindByName($insert,"NAME",$name,32);

        OCIExecute($stmt);

        while (OCIFetch($stmt)) {
            OCIExecute($insert);
        }
    ?>

    note: the PHP oracle interface fully supports the SQL NULL-value when
    using bind & define!

autoincrement
=============

    so called sequencers are very useful for generating the unique-id (primary
    key) that every GoodTable(tm) should have. oracle allows you to have triggers
    auto-fill the next value of a sequencer into your primary-key field but
    when you insert a record into your table they won't tell the actual
    sequencer-value used (see mysql_insert_id()). to get the oci_insert_id()
    _without_ an extra server-round-trip you can use the oracle "RETURNING"
    clause together with a bound-variable:

    seq.sql
    create sequence myid;

    <?php

        $data = array("larry","bill","steve");

        $db = OCILogon("scott","tiger");
        $stmt = OCIParse($db,"insert into names values (myid.nextval,:name) returning
id into :id");
        OCIBindByName($stmt,":ID",$id,32);
        OCIBindByName($stmt,":NAME",$name,32);

        while (list(,$name) = each($data)) {
            OCIExecute($stmt);
            echo "$name got id:$id\n";
        }
    ?>

efficient "select for update" using ROWID
=========================================
```

if you need to update your chicken table you usually do a select
to get the old row values from your database, then you do some
calculations based on the just read values and then you update that very record
in the database with the updated values. as said before having a unique
field as the primary-key in every table is a SmartThing(tm) to do but to
do the most efficient update you don't need that when you use oracle.
in oracle every row has an "address" inside the database which is called
the ROWID. this ROWID is unique over all tables in the database and filled
in automatically by oracle. so why not use this as the primary key?
simple answer, the ROWID changes when you export and re-import the table.
but for a simple update the ROWID is still the most efficient way of
addressing the record to be updated!

```
chicken.sql
create table chicken (id number, chickens_sold number);
insert into chicken values (1,100);

<?php

    $db = OCILogon("scott","tiger");

    $stmt = OCIParse($db,"select rowid,chickens_sold from chicken for update");
    OCIDefineByName($stmt,"CHICKENS_SOLD",$chickens_sold);
    OCIDefineByName($stmt,"ROWID",$rid);
    OCIExecute($stmt,OCI_DEFAULT);
    OCIFetch($stmt);

    echo "chicken sold so far: $chickens_sold\n";

    $update = OCIParse($db,"update chicken set chickens_sold=:chickens_sold where
rowid = :rid");
    OCIBindByName($update,"CHICKENS_SOLD",$chickens_sold,32);
    OCIBindByName($update,"RID",$rid,-1,OCI_B_ROWID);

    $chickens_sold += 100;

    OCIExecute($update);
    OCICommit($db);
?>
```

note: ROWID is handled as an opaque data-type within PHP (you cannot print
it as a string). if you need the ROWID as a string do
"select ROWIDTOCHAR(rowid) from chicken"

large objects

PHP has full support for using internal and external LOBs in oracle. LOBs
are different from "normal" data-types as they require extra programming on
the script side. when you need to store large amount of data in one field

LOBs are the ideal choice. you could also store bigger fields (up to 2GB) in a "LONG" or "LONG RAW" field (which is as good supported be PHP) but oracle plans to drop support for those types in future releases. "LONG" and "LONG RAW" fields can not be replicated across servers and they will always get loaded into memory when the row containing the "LONG" or "LONG RAW" is touched. LOBs don't have this limitation but cause a tiny bit more headache when used. oracle has CLOBs (character-LOB) BLOBs (binary-LOB) and BFILEs (external files - only path to file is stored in database).

before you can use a LOB oracle needs to create it - lets illustrate:

blobdemo.sql

```
create table blobdemo (id number, lob blob);
```

```
<?php

    $data = array("/lib/libc-2.2.2.so","/lib/libncurses.so.5.2");

    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"insert into blobdemo values
                    (myid.nextval,EMPTY_BLOB()) returning id,lob into :id,:lob");
    $lob = OCINewDescriptor($db);
    OCIBindByName($stmt,":ID",$id,32);
    OCIBindByName($stmt,":LOB",$lob,-1,SQLT_BLOB);

    while (list(,$file) = each($data)) {
        OCIExecute($stmt,OCI_DEFAULT);          // we cannot use autocommitt here
        $lob->save(`cat $file`);
        echo "$file id:$id\n";
        OCICommit($db);
    }
?>
```

now we have loaded our libc and ncurses into oracle - makes sense;-)

getting them back is more trivial:

```
<?php

    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"select * from blobdemo");
    OCIExecute($stmt);

    while (OCIFetchInto($stmt,$arr,OCI_ASSOC)) {
        echo "id: ".$arr[ "ID" ]."\n";
        echo "size: ".strlen($arr[ "LOB" ]->load())."\n";
    }
?>
```

to update a lob you have to load the LOB-descriptor first:

```
    <?php

        $db = OCILogon("scott","tiger");
        $stmt = OCIParse($db,"select blob from blobdemo for update");
        OCIExecute($stmt,OCI_DEFAULT);

        while (OCIFetchInto($stmt,$arr,OCI_ASSOC)) {
            $content = $arr[ "LOB" ]->load();

            echo "id: ".$arr[ "ID" ]."\n";
            echo "size: ".strlen($content)."\n";

            $lob->save(strrev($content));
        }
        OCICommit($db);
    ?>
```

there are functions to just replace a part of a LOB, you can spool a LOB to the browser or a file _without_ buffering it a PHP-variable. the OCIFetchInto function also allows you to inline the LOB values into your result-set, this saves you the call to ->load() but LOB-data which is returned instead of the locator can not be modified like shown above.

the oracle-BFILE type can be read like a normal LOB/CLOB but can't be written to. basically you can store the path to a file on the oracle-server in a table-field and the LOB functions allow you to read this file thru the oracle server-connection. this can be extremely helpful if your web-server is in front of a firewall and you only want to allow oracle-traffic thru this firewall (no NFS-traffic). the BFILE allows you to keep your images in the file-system so they don't clutter your table-space but you can still access them as if they were a part of your database!

nested tables

lets say we have a user and a duty table:

```
create table family (id number, name varchar2(32));
create table duties (id number, user_id number, duty varchar2(32));
```

and we preload some data:

```
insert into family values (1, 'thies');
insert into duties values (1, 1, 'cook');
insert into duties values (1, 1, 'make tea');
insert into duties values (1, 1, 'fix the code');

insert into family values (2, 'antje');
insert into duties values (1, 2, 'change dipers');
```

```
    insert into duties values (1, 2, 'bring kid to school');

    we can now say (in sqlplus):

    select id,name,CURSOR(select duty from duties where duties.user_id = family.id)
as duties
    from family;

    in PHP we would have to say:

    <?
        $db = OCILogon("scott","tiger");
        $stmt = OCIParse($db,"
                select name,CURSOR(select duty from duties where duties.user_id =
family.id) as duties
                from family");

        OCIExecute($stmt);

        while (OCIFetchinto($stmt,$arr,OCI_ASSOC)) {
            echo $arr[ "NAME" ]."\n";
            echo "-----\n";

            // Now execute the sub-query
            OCIExecute($arr[ "DUTIES" ]);
            while (OCIFetchinto($arr[ "DUTIES" ],$duties,OCI_ASSOC)) {
                echo $duties[ "DUTY" ]."\n";
            }
            echo "\n";
        }
    ?>

    gets us:

    thies
    -----
    make tea
    fix the code
    cook

    antje
    -----
    change dipers
    bring kid to school
```

stored procedures
==========

calling stored-procedures from PHP is trivial once you understood the
binding of variables:

```
inoutdemo.sql
CREATE OR REPLACE PROCEDURE inoutdemo (
    par_in IN VARCHAR2,
    par_in_out IN OUT VARCHAR2,
    par_out OUT VARCHAR2)
IS
BEGIN
    par_out := par_in;
    par_in_out := par_in || ' ' || par_in_out;
END;

<?
    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"BEGIN inoutdemo(:in,:inout,:out); END;");
    OCIBindByName($stmt,":in",$in,32);
    OCIBindByName($stmt,":inout",$inout,32);
    OCIBindByName($stmt,":out",$out,32);

    $in = "Hello ";
    $inout = "World!";

    OCIExecute($stmt);

    echo $in."\n";
    echo $inout."\n";
    echo $out."\n";
?>
```

refcursors

very similar to the nested-tables are refcursors which can be passed out of
stored-procedures:

```
CREATE OR REPLACE PACKAGE info AS
    TYPE the_data IS REF CURSOR RETURN all_users%ROWTYPE;
    PROCEDURE output(return_data IN OUT the_data);
END info;
/
CREATE OR REPLACE PACKAGE BODY info AS
    PROCEDURE output(return_data IN OUT the_data) IS
      BEGIN
        OPEN return_data FOR SELECT * FROM all_users;
    END output;
END info;
/
```

to call this procedure from PHP we would have to say:

```
<?php
    $db = OCILogon("scott","tiger");

    $curs = OCINewCursor($db);

    $stmt = OCIParse($db,"begin info.output(:data); end;");
    OCIBindByName($stmt,":data",$curs,-1,OCI_B_CURSOR);
    OCIExecute($stmt);
    OCIExecute($curs);

    while (OCIFetchinto($curs,$duties,OCI_ASSOC)) {
        var_dump($duties);
    }
?>
```

collections

collections are a way to pass a variable number of items to a stored procedure. they are very helpful to reduce to number of roundtrips to the database when inserting more than one record at a time. if we have a table with user-responsibilities like this:

```
CREATE TABLE user_responsibilities(
    user_id VARCHAR2(100),
    responsibility VARCHAR2(100)
);
```

we might want to update the responsibilities of one user with a new set - in PHP wo would do:

```
<?php
    $duties = array("clean","cook","buy-food");
    $user = "andy";

    $db = OCILogon("scott","tiger");
    $stmt = OCIParse($db,"DELETE FROM user_responsibilities WHERE user_id =
'$user'");
    OCIExecute($stmt);

    while (list(,$v) = each($duties)) {
        $stmt = OCIParse($db,"INSERT INTO user_responsibilities
                              (user_id,responsibility)
                              values('$user','$v')");
        OCIExecute($stmt);
   }
```

```
    ?>


    we would end up doing a lot of database-roundtrips doing it this way! if we
    use the collection feature of oracle we could save most of the roundtrips
    and let oracle do all the work for us:


    CREATE OR REPLACE TYPE str_vec AS VARRAY(100) OF VARCHAR2(100);

    CREATE OR REPLACE PROCEDURE update_responsiblities(
            i_user_id IN VARCHAR2,
            i_responsibilities IN str_vec)
    IS
    BEGIN
       DELETE FROM user_responsibilities WHERE user_id = i_user_id;
       FOR i IN 1 .. i_responsibilities.count LOOP
          INSERT INTO user_responsibilities(user_id,responsibility) values
                (i_user_id,i_responsibilities(i));
       END LOOP;
    END;

    <?php
        $duties = array("clean","cook","buy-food");
        $user = "andy";

        $db = OCILogon("scott","tiger");

        $arr = OCINewCollection($db,'STR_VEC');
        $stmt = OCIParse($db,"begin update_responsiblities(:user,:duties); end;");

        while (list(,$v) = each($duties)) {
           $arr->append($v);
        }

        OCIBindByName($stmt,':user',$user);
        OCIBindByName($stmt,':duties',$arr,32,OCI_B_SQLT_NTY);

        OCIExecute($stmt);
    ?>


outlook into the future
=======================

    - connection pooling
    - failover
```