

Brewing Benchmarks



James Morle

Using some custom developed tools, a 10046 trace file and a little time it is possible to create fully functional, accurate simulations of *your* application for benchmarking and testing purposes.

INTRODUCTION

Benchmarking is a generic term for many activities. Included in this list of activities are:

- Performance Evaluation
- Stress Testing
- Upgrade Testing

It is unfortunate, but it is rare to find performance evaluations occurring inside corporations. Though it is a tremendously valuable exercise, it is often considered too costly and/or time consuming to consider. The latter two points are often neglected also, and their absence presents a huge risk to the enterprise. For example, prior to installing a new server, or upgrading to a new OS or RDBMS release, would it not be nice to have some comfort that **your application** will run with acceptable performance, and without continual failure?

So what actually is a benchmark? There are really two kinds - artificial benchmarks used as competitive yardsticks (such as TPC-C), and specific application benchmarks, typically used internally. For the purposes of this paper, we are focused upon the latter category. A better terminology for this type, perhaps, would be an Application Simulation, but we will stick with the benchmark terminology.

Traditional methods of producing application benchmarks typically fall into one of two categories:

- Custom written, 'synthetic' benchmarks
- Capture-based benchmarks

The second category provides the most accurate reproduction of the application by ensuring the simulation is submitting the actual load of the application to the database. However, there are a number of negative attributes to this approach:

- License Cost: The software harnesses to provide such facilities are expensive, niche products

-
- Infrastructure cost: In order to test the database server, hardware must be provided for the upper tier(s) of equal magnitude to the real production system, plus additional hardware to execute the benchmark 'driver'.
 - Flexibility: Screen-capture techniques are arguably sensitive to minor changes in the application, such as field relocation.

The approach of using a custom-written benchmark circumvents many of these issues, but has even more serious issues:

- Accuracy: Custom written benchmarks typically only bear a passing resemblance to the real application. This has historically resulted in errors approaching an order of magnitude in hardware sizing...
- Stagnation: Once written, custom benchmarks are typically never updated. Thus, any resemblance they had to the production application fades rapidly.

After many years experience using both these approaches, there just had to be a better way!

UNDERLYING, MAD, RAMBLING THOUGHTS

A better way? Surely these are the thoughts of a madman? Well, perhaps, but Oracle can be a pretty verbose beast when prompted, and it seemed that there must be a way of using one of these information sources to capture specific requests from an application.

One such journal of activity is the redo log, but this is purely a write-centric journal, storing little information as a result of SELECT requests¹. Accordingly, this is of no use whatsoever, as most applications have a significantly greater read content than they have write content.

Another form of activity journal is that of a SQL_TRACE trace file, which represents nearly all calls made to the database by a specific connection. This is nearly perfect, but is missing two attributes that would be required if we were to use this information for replaying the transaction: bind variable values and calls made using PL/SQL RPC calls from a client-side PL/SQL engine.

The solutions to both these omissions are available:

- Use event 10046 level 4 (more later)
- Apply patch for bug 2425312 (again, more later)

With both of these issues addressed, trace files form the ideal vehicle with which to build some kind of replay engine. Now all we need is some kind of recipe with which to brew our benchmark...

RECIPE

Ingredients:

10MB or so of SQL_TRACE file, Level 4

A dash of Preprocessing

A powerful script interpreter

A powerful execution harness to bind it all together

Method: Bah, it's time to stop this tenuous analogy and get on with the real nuts and bolts. Let's dive into some of the detail!

GATHERING 10046 TRACE FILES

A so-called 10046 trace file is merely a standard sql_trace file collected with a different level of detail. With each release of Oracle, there seems to also be an additional method of collecting these trace files, which is no bad thing; these files are perhaps the single most useful piece of information on the operation of Oracle.

The traditional way of getting a 10046 file is by setting an event in a session. This can either be performed upon your own session, or upon any other user's session. Though no longer necessary in recent releases of Oracle, this method will be presented first because it works in every release for the last 15 years or so!

If the session is your own, the syntax is as follows:

1. The only information stored in the redo log as a result of a SELECT is purely the side-effect of a prior write-based request.

```
alter session set events '10046 trace name context forever, level 4';
```

This will produce a trace file in your `user_dump_dest` of the form `<SID>_ora_<PID>.trc`.

The method to dump a trace file for another session is a little more complex. Note: these directions assume the use of dedicated server connections to the database.¹

First, find the OSPID of the session you wish to trace. This can be achieved by joining `v$session` to `v$process`. For this example, let's assume it is '12345'. Next, go into `SQLPLUS`:

```
SQL> oradebug setospid 12345
Statement processed.
SQL> oradebug event 10046 trace name context forever, level 4
Statement processed.
```

Note the lack of quotes in this example. To turn tracing off, in either situation, the syntax is as follows: "10046 trace name context off".

THE EASY WAY

That was the hard, backward compatible, way. The easy way is to use the `DBMS_SUPPORT` package, supplied with the database. To start tracing within your own session, the `DBMS_SUPPORT.START_TRACE(waits =>false, binds => true)` call provides the equivalent of an event 10046, level 4 trace. To start tracing in another session, the `start_trace_in_session(sid => <SID>, serial => <SERIAL>, waits =>false, binds => true)` will have the desired effect.

Starting with 10g, there is a new way to turn on tracing through the use of `DBMS_MONITOR`. This is the official home for all tracing-related procedures, and allows many approaches to enabling traces, such as starting tracing for a given `CLIENT-ID`, or service name. The one we are interested in though is `session_trace_enable`, which has the same args as the `DBMS_SUPPORT.start_trace_in_session` procedure.

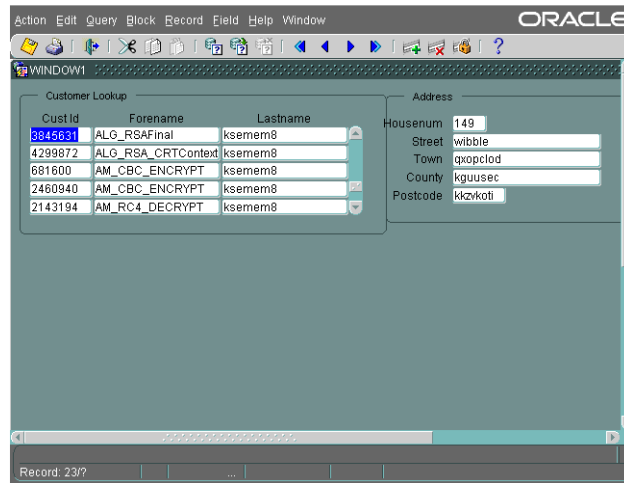
Some people are a little confused over the different 'levels' available with 10046 tracing. It is really straightforward when you know the secret, so I'll briefly explain. The number supplied for 'level' is treated by Oracle as a bitmask, where the first three bits enable different attributes of the trace functionality. Bits two and three relate to tracing bind operations and waits respectively. As you will of course know, bit two of a binary number has a value of 4, and bit three has a value of 8. In our case, we only want bind operations, so we set bit two by supplying a value of 4. If we wanted wait information, we would specify a value of 8, thus setting bit three. If we wanted both, we would set both bits by specifying a value of 12.

THE TEST APPLICATION

For the purposes of this paper, I created a very simple form-based application using `Forms 9i`. Not being a `Forms` developer, this form hardly qualifies for any design or coding awards. However, it does provide all the attributes needed for a good demon-

1. It is also possible to trace MTS connections, but this is a more complex affair that involves the combining of multiple trace files into one. I recommend that, simply for the purposes of generating a clean trace file for the simulation, that a dedicated connection is configured and used.

stration of the techniques presented here. Not only is it incredibly ‘bite-sized’, making it easier to explain, but it also uses PL/SQL RPC calls in addition to standard SQL. The Form looks like this:



The first thing you will notice is that the data is a little strange. This is because I used the symbol table of both the Linux and Oracle kernel’s in order to generate a reasonably sized list of unique text for this application’s test data. We can just ignore that, and assume they are real names and addresses. The flow of this application is as follows:

1. User logs in, and enters a query against the customer’s last name (pretend they are talking on the phone, for extra realism)
2. The form retrieves all the name records that match, and allows the user to scroll through them until the one with the correct address is found. The address is populated automatically each time the user moves to the next name record.
3. The user then hits the ‘Next Block’ key and, quite unhelpfully to the customer, the application automatically creates an order for that customer comprising of four random line items, then commits.
4. The application returns a blank form to the user, at which point the customer slams the phone down in disgust.

Steps 1 & 2 are performed with standard SQL calls from Forms. Step 3 is all performed in a Forms trigger, and comprises of standard SQL AND a PL/SQL RPC call to a server-side procedure which allocates the stock. Step 4 consists of non-database action items.

LOOKING AT A TRACE FILE

Once a trace file has been produced for an interesting piece of database workload, it can be used for many useful things, notably performance tuning of the application and even the database. However, that is not the purpose of these particular trace files. These files are to be used to reproduce the workload against the database, so let’s take a look at the actual content from this test application to determine the possibilities.

The trace file has multiple sections, so I will present those individually for reasons of clarity. First of all, the first thing the form does is to set its Application Info with the database, using a PL/SQL RPC call to the DBMS_APPLICATION_INFO package on the server. This is the ideal time to discuss the effect of bug 2425312, as mentioned earlier.

Bug 2425312 caused the RDBMS to omit trace information for PL/SQL RPC calls. When these occurred, there was no indication of them in the trace file, thus the picture of the workload was incorrect. Prior to the fix of this bug, this section of the trace file would appear as follows:

```
*** ACTION NAME:(test action) 2003-11-04 16:50:57.213
*** MODULE NAME:(test form) 2003-11-04 16:50:57.213
```

If the trace were performed at level 12, to include the wait information in addition to binds, there would also have been one wait for each of SQL*Net message from client and SQL*Net message to client, showing some kind of communication with the client prior to this magical setting of the application info. If one were to turn on event 10051 to show Oracle Programmatic Interface (OPI) calls, one would see the following:

```
OPI CALL: type=76 argc=2 cursor= 0 name=PL/SQL RPC
*** ACTION NAME:(test action) 2003-11-04 16:50:57.213
*** MODULE NAME:(test form) 2003-11-04 16:50:57.213
```

This, at least, shows the RPC call happening, but not any specifics about it. Once the patch is applied (for 8i and 9i databases - the bug is permanently fixed in 10g), the output is as follows:

```
RPC CALL:PROCEDURE SYS.DBMS_APPLICATION_INFO.SET_MODULE(MODULE_NAME IN VARCHAR2,
ACTION_NAME IN VARCHAR2);
RPC BINDS:
  bind 0: dty=1 bfp=0b8e7c54 flg=08 avl=09 mxl=09 val="test form"
  bind 1: dty=1 bfp=0b8e7c7c flg=08 avl=11 mxl=11 val="test action"
*** ACTION NAME:(test action) 2003-11-04 16:50:57.213
*** MODULE NAME:(test form) 2003-11-04 16:50:57.213
RPC EXEC:c=0,e=19681
```

Now we can see the call to the server-side package, along with its bind variable values, and some statistics about the CPU and elapsed times for its execution.

The next section of the trace file is pretty straightforward, the initial search on the customer's last name:

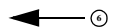
```
=====
PARSING IN CURSOR #4 len=81 dep=0 uid=59 oct=3 lid=59 tim=1042934250222095
hv=4157578275 ad='5de26dfc'
SELECT ROWID,addr_id,CUST_ID,FORENAME,LASTNAME FROM CUST WHERE (LASTNAME LIKE :1)
END OF STMT
PARSE #4:c=0,e=176,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1042934250222085
BINDS #4:
  bind 0: dty=96 mxl=32(12) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0010 size=32 offset=0
    bfp=b6ba62f8 bln=32 avl=04 flg=05
    value="ksm%"
EXEC #4:c=0,e=218,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1042934250222438
FETCH #4:c=0,e=385,p=0,cr=11,cu=0,mis=0,r=7,dep=0,og=4,tim=1042934250222998
```

The two classic characteristics of Forms are both there: always getting the ROWID in addition to the data (1), and the “:n” format for bind variables (2). The important things to note here are the inclusion of ADDR_ID in the select list (3), the search string entered by the user, used as the predicate value (4), and that seven rows were returned in a single array fetch(5).

The next section is where the user is scrolling through the list of names. For each name the user visits, the application performs a look up of the corresponding

address. I have included quite a chunk of trace file this time to demonstrate an important point:

```
=====
PARSING IN CURSOR #5 len=86 dep=0 uid=59 oct=3 lid=59 tim=1042934250227339
hv=803003199 ad='5de21630'
SELECT ROWID,ADDR_ID,HOUSENUM,STREET,TOWN,COUNTY,POSTCODE FROM ADDR WHERE
(ADDR_ID=:1)
END OF STMT
PARSE #5:c=0,e=133,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1042934250227333
BINDS #5:
  bind 0: dty=2 mxl=22(05) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset=
  ① bfp=b6ba45dc bln=22 avl=05 flg=05
    value=3709053
EXEC #5:c=0,e=243,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1042934250227690
FETCH #5:c=0,e=120,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1042934250227953
BINDS #5:
  bind 0: dty=2 mxl=22(05) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset=
  ② bfp=b6ba3514 bln=22 avl=05 flg=05
    value=4643064
EXEC #5:c=0,e=217,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1042934252270618
FETCH #5:c=0,e=152,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1042934252270816
BINDS #5:
  bind 0: dty=2 mxl=22(04) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset=
  ③ bfp=b6ba484c bln=22 avl=04 flg=05
    value=592137
EXEC #5:c=0,e=208,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1042934252443017
FETCH #5:c=0,e=142,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1042934252443204
BINDS #5:
  bind 0: dty=2 mxl=22(04) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset=
  ④ bfp=b6ba484c bln=22 avl=04 flg=05
    value=796574
EXEC #5:c=0,e=146,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1042934252595025
FETCH #5:c=0,e=100,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1042934252595160
BINDS #5:
  bind 0: dty=2 mxl=22(05) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset=
  ⑤ bfp=b6ba484c bln=22 avl=05 flg=05
    value=1341180
EXEC #5:c=0,e=134,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1042934252756299
FETCH #5:c=0,e=111,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1042934252756445
FETCH #4:c=10000,e=443,p=0,cr=8,cu=0,mis=0,r=6,dep=0,og=4,tim=1042934252919941
```



Nearly all of this trace file relates to the repeated execution of the same statement. Markers 1-5 in this segment show different bind values being assigned to cursor number 5, which is then executed and fetched from each time. This is Forms in action, populating the address details as the user scrolls through the list. Notice that it is NOT reparsing the statement each time, just rebinding and executing - perfect coding technique. The final piece of interest in this segment is (6), where cursor number 4 is fetched from once more. This is to get the next batch of customer details, since the user has nearly got to the bottom of the list. The trace file then continues to rebind and execute cursor 5, but this is not shown here for reasons of brevity.

The next (and final) section of the trace file exists entirely within the automatic order creation portion of the application, which all occurs within a Forms trigger. The trigger code looks like this:

```

declare
    i number;
    stockLevel number;
    nextItem number;
    nextRand number;
    my_item_id number;
    my_stock_id number;
    dummy number;
    nameRand varchar2(10);
begin
    select order_seq.nextval into :PARAMETER.onum from sys.dual;
    INSERT INTO orders values (:PARAMETER.onum, :cust.cust_id,
                             :cust.addr_id , sysdate, 'N', 'OPEN');
    sys.dbms_random.initialize(42);
    for i in 0..4 LOOP
        -- Get SKU
        nextRand:=floor(sys.dbms_random.value(1,60000));
        select substr(name,1,3)||'%' INTO NAMERAND from nameroots
            where name_id=nextRand;
        SELECT item_id
        into nextItem
        FROM items
        WHERE name like namerand
        AND rownum=1;

        SELECT item_id, stock_id, whs_id, stock_level
        INTO my_item_id, my_stock_id, dummy, dummy
        FROM stock
        WHERE item_id =nextItem
        FOR UPDATE OF stock_level;

        alloc_stock(:PARAMETER.onum, my_item_id, my_stock_id);
    end loop;
    commit;
    clear_form(No_Validate);
END;
```

This trigger code (a client-side PL/SQL block) creates the order, picks four random items (by leading part of its name), and allocates that stock to the order. In the trace file, things look a little different, however, because all the work done by the `alloc_stock` stored procedure (which only exists in the database) occurs at a different recursive depth to the rest of the SQL in the block. This is because the PL/SQL RPC call triggered by the call to `alloc_stock` is the only user call, and the SQL within the procedure is only implicit. Here's a small sample of that in action:

```

=====
PARSING IN CURSOR #10 len=106 dep=0 uid=59 oct=3 lid=59 tim=1042934256163144
hv=4237262396 ad='5d78923c'
SELECT ITEM_ID,STOCK_ID,WHIS_ID,STOCK_LEVEL FROM STOCK WHERE ITEM_ID = :b1
UPDATE OF STOCK_LEVEL
END OF STMT
PARSE #10:c=10000,e=10860,p=0,cr=55,cu=0,mis=1,r=0,dep=0,og=4,tim=10429342561631:
BINDS #10:
  bind 0: dty=2 mxl=22(21) mal=00 scl=00 pre=00 oacflg=03 oacfl2=0000 size=24 offe
    bfp=b6b9ffc4 bln=22 avl=03 flg=05
    value=1315
EXEC #10:c=0,e=1552,p=2,cr=4,cu=1,mis=0,r=0,dep=0,og=4,tim=1042934256165354
FETCH #10:c=0,e=103,p=0,cr=3,cu=0,mis=0,r=1,dep=0,og=4,tim=1042934256165506
RPC CALL:PROCEDURE ORDERS.ALLOC_STOCK(IN_ORDER_ID IN NUMBER, IN_ITEM_ID IN NUMBEI
IN_STOCK_ID IN NUMBER);
RPC BINDS:
  bind 0: dty=6 bfp=0b8e7c48 flg=00 avl=05 mxl=22 val=1000008
  bind 1: dty=6 bfp=0b8e7c70 flg=00 avl=03 mxl=22 val=1315
  bind 2: dty=6 bfp=0b8e7c98 flg=00 avl=03 mxl=22 val=1315
=====
PARSING IN CURSOR #11 len=61 dep=1 uid=59 oct=6 lid=59 tim=1042934256168423
hv=1134532186 ad='5d91df7c'
UPDATE STOCK SET STOCK_LEVEL=STOCK_LEVEL-1 WHERE ITEM_ID=:B1
END OF STMT
PARSE #11:c=0,e=467,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=4,tim=1042934256168413
```

As you can see, the `SELECT` from `STOCK` occurs at a recursive depth of zero (`dep=0`), marked by (1). When the RPC call is made (2), the very next statement is occurring at recursive depth of 1 (`dep=1`), as marked by (3). The RPC call is not complete in the trace file until the `RPC EXEC` tag is emitted. All SQL between `RPC CALL` and `RPC EXEC` is a result of the RPC call itself, and not directly submitted by the application itself.

Now that we have a good idea of the content of the trace files, and of the operation of the demo application, let's see what we can do with them.

PREPROCESSING

The first stage in converting a trace file into something we can replay is to preprocess the file itself, and convert it into something we can use more readily. In this case, I opted to use the text processing language `awk` to perform the preprocessing, because it is tremendously powerful at doing exactly this kind of operation. In fact, I used the GNU version of `awk`, named `gawk` of course, because it is significantly more flexible in terms of both line-length restrictions, and diagnostic messages.

The idea behind the preprocessing phase was to turn the trace file into some kind of scripting language. This script could then be edited very easily and replayed back against the database. Of course, there isn't a scripting language that can accurately replay the various phases of execution, so I had to develop one using the TCL language as a base. We'll come on to that later.

As time has gone on, the preprocessing script has grown quite large, mostly to cover the large number of corner cases that have emerged during real use of this facility. It is now around 24KB in size, which is rather large for an `awk` script. It is also quite tightly coupled with the replay language, so that it generates exactly the right syntax to handle special cases such as problematic data types (e.g. `DATE`, `BOOLEAN` and `NULL`.) It also embeds a number of useful little attributes, such as the ability to track sequence number usage, which we will see when we generate our script. Before doing that, though, let's have a look at the replay engine, so that the script is more intelligible.

THE REPLAY ENGINE

The replay engine is called 'dbaman', so called for historical reasons. It started life as a generic shell for doing DBA MANagement functions in the early days of Oracle7, and its name has stuck. It is an extension to the TCL Language (Toolset Control Language), a fast scripting language that was specifically designed for this kind of extension. In its vanilla form, TCL is a lot like a UNIX shell in its facilities, with a few notable differences (such as the ability to use network sockets!). It also includes a JIT compiler, but we typically get little benefit from it in this case, because we very seldom revisit code (such as in loops). However, the native interpreter is very fast.

Over the top of TCL, I built a number of extensions. These extensions fall into two broad areas: Oracle Integration and O/S Integration. All these extensions are built with the express purpose of replaying trace files accurately against the database.

The following table shows all the Oracle Integration extensions by name:

TABLE 1. DBAMAN Oracle Extensions

Name	Description
oci_bind_begin	Start the bind process for the specified cursor - client-side only operation
oci_bind_end	End the bind process for the specified cursor - sends bind information to server.
oci_bind_pos	Bind a value to a variable based upon its position in the SQL statement.
oci_bind_name	Bind a value to a variable based upon its name.
oci_cancel	Cancel execution of a cursor.
oci_close	Close a cursor.
oci_commit	Commit. This is handled using the OCI commit call rather than sending a SQL 'commit' statement.
oci_rollback	As commit, only for rollback!
oci_exec	Execute a cursor.
oci_fetch	Fetch rows from the database by specified array size.
oci_list	List open cursors.
oci_logon	Logon to the database via TNS.
oci_parse	Parse the specified statement and assign a cursor handle.
oci_date	Convert the current time to internal Oracle format used in trace files.
oci_rpccall	Issue PL/SQL rpc call. This is actually simulated using anonymous PL/SQL blocks.
oci_vars	List bind variables and assigned values for specified cursor.

This table shows all the O/S Integration extensions by name:

TABLE 2. DBAMAN O/S Extensions

Name	Description
os_cpu	Return CPU usage information.
os_fork	Fork a process.
os_loadavg	Return the run queue and load average information.
os_mem	Return the physical and available memory statistics.
os_procs	Return per-processor statistics (OS dependent)
os_setpgid	Make current process a process group leader. Used when creating daemon processes
os_utime	Return current time in μ s
sem_get	Create a semaphore or start using existing.
sem_delete	Destroy a semaphore.
sem_incr	Increment the specified semaphore.
sem_decr	Decrement the specified semaphore.
sem_set	Set the value of the specified semaphore to an absolute value.
sem_val	Get the current value of the semaphore.

TABLE 2. DBAMAN O/S Extensions

Name	Description
shm_get	Create or attach to existing shared memory segment.
shm_delete	Destroy specified shared memory segment.
shm_link	Link named program variable to a shared memory location.
shm_size	Returns the segment size in terms of the number of values of type <code>double</code> it can store.
shm_getDouble	Read the <code>double</code> value at specified location.
shm_setDouble	Set the <code>double</code> value at specified location.
msg_get	Create message queue or start using existing.
msg_delete	Destroy specified message queue.
msg_qsize	Returns number of messages currently on queue.
msg_rcv	Pop message of queue, or block until one is available.
msg_send	Push message onto named queue.

Phew. Now let's get back to looking at the script we produced from our processing of the demo application's trace file.

THE SCRIPT

To keep things manageable, we will view the script in pieces similar to those in which we viewed the trace file. The total script file size generated was 390 lines long, compared to 1091 for the raw trace file. The 390 will get reduced, however, as part of the editing process, as will soon become apparent.

The first section is the call to `DBMS_APPLICATION_INFO`:

```
package require dbaman 2.0
#oci_parse -assign 256 { alter session set events '10046 trace name context fore
level 4'}
#oci_exec 256
#oci_close 256
# RPC: PROCEDURE SYS.DBMS_APPLICATION_INFO.SET_MODULE(MODULE_NAME IN VARCHAR2,
ACTION_NAME IN VARCHAR2);
log_pos 1
oci_rpccall {PROCEDURE SYS.DBMS_APPLICATION_INFO.SET_MODULE( :MODULE_NAME,
:ACTION_NAME) } { { MODULE_NAME IN "test form" } { ACTION_NAME IN "test action"
oci_parse -assign 4 { SELECT 'in startup trigger' FROM SYS.DUAL }
set int4 [ time {
set int1 [ time {
```

The first thing the script does is to tell the TCL interpreter that it will be needing the dbaman extensions. Then, as an 'ease of use' facility, the preprocessor has entered a section to turn on tracing for the execution of the script! It is commented out by default, because you would only really turn it on as a diagnostic aid, rather than have everything traced.

Next up is the RPC call to set the application info, which is fairly self-explanatory. This is followed by a standard SQL call that was also in the Form startup code. Note that it is assigned the same cursor number as was used in the trace file.

The last two lines merit a little explanation. By default, the preprocessor divides the whole script into three pieces, and brackets those pieces with a timing harness. This means that we can easily determine the response times of key pieces of the application when we are executing it. In addition to the three timed segments, the whole script is also timed, as shown by the `int 4` line.

We'll skip over the next few pieces of SQL and go straight to the address look up piece:

```
oci_parse -assign 5 { SELECT ROWID,ADDR_ID,HOUSENUM,STREET,TOWN,COUNTY,POSTCODE
ADDR WHERE (ADDR_ID=:1) }
oci_bind_begin 5
oci_bind_pos 5 0 3709053
oci_bind_end 5
# SELECT ROWID,ADDR_ID,HOUSENUM,STREET,TOWN,COUNTY,POSTCODE FROM ADDR W ...
log_pos 4
oci_exec 5
oci_fetch 5 1
oci_cancel 5
oci_bind_begin 5
oci_bind_pos 5 0 4643064
semsleep 2042
oci_bind_end 5
# SELECT ROWID,ADDR_ID,HOUSENUM,STREET,TOWN,COUNTY,POSTCODE FROM ADDR W ...
log_pos 5
oci_exec 5
oci_fetch 5 1
oci_cancel 5
oci_bind_begin 5
oci_bind_pos 5 0 592137
oci_bind_end 5
# SELECT ROWID,ADDR_ID,HOUSENUM,STREET,TOWN,COUNTY,POSTCODE FROM ADDR W ...
log_pos 6
oci_exec 5
oci_fetch 5 1
oci_cancel 5
oci_bind_begin 5
oci_bind_pos 5 0 796574
oci_bind_end 5
# SELECT ROWID,ADDR_ID,HOUSENUM,STREET,TOWN,COUNTY,POSTCODE FROM ADDR W ...
log_pos 7
oci_exec 5
oci_fetch 5 1
oci_cancel 5
oci_bind_begin 5
oci_bind_pos 5 0 1341180
oci_bind_end 5
# SELECT ROWID,ADDR_ID,HOUSENUM,STREET,TOWN,COUNTY,POSTCODE FROM ADDR W ...
log_pos 8
oci_exec 5
oci_fetch 5 1
oci_fetch 4 6
oci_cancel 5
```

Bind operation

Sleep

Fetch more customer details

There are quite a few interesting observations here. First of all, you can see the bind mechanism in operation. The preprocessor always uses positional binding because that eliminates the need to parse each statement, and the positions always start at ZERO, not ONE. Whenever the preprocessor emits an `oci_exec` call, it will enter a comment line showing an extract of the statement it is executing. This is purely for the benefit of somebody editing the script, and just makes life easier.

The next thing of interest is the call to `semsleep`. This is a function that resides in the set of utility procedure scripts that support `dbaman`. It will sleep for the specified number of milliseconds after funnelling through a semaphore. This serves two purposes: The semaphore functionality is used to provide the capability to pause a benchmark with great simplicity; if the pause button is pressed, each executing script will detect it at its next sleep point. The second purpose is to play back the actual user think time. The value for the sleep function is derived from the trace file itself, and is the *actual* time the user waited before proceeding with the transaction.

The final thing of interest is that this is a segment that is just *begging* for some kind of loop. The raw trace file shows that the user pressed the Down Arrow key some 18 times before he/she found the right customer details. It would be nice if a) this was in some way randomised (it surely isn't always 18 records down before the right customer is found), and b) it was a long series of repeated lines. Here's what it looks

like after it's been edited, combined with the customer look up query we skipped over earlier:

```
# Lookup customer details. Scroll through up to 18 similar customer and
# then select one.
#set max_search 18
set max_search [expr int(rand()*18)+1]
set curind 0
oci_parse -assign 4 { SELECT ROWID,addr_id,CUST_ID,FORENAME,LASTNAME FROM CUST WH
(LASTNAME LIKE :1) }
while {1} {
    oci_bind_begin 4
    oci_bind_pos 4 0 $in_surname
    oci_bind_end 4
    # SELECT ROWID,addr_id,CUST_ID,FORENAME,LASTNAME FROM CUST WHERE (LASTN ..
    log_pos 3
    oci_exec 4
    oci_fetch 4 6
    if { $results(4,rpc)>0 } {
        break
    } else {
        puts "No customer found for $in_surname"
        set in_surname [getVal surname]%
    }
}
set cur_rpc 0
while { $curind< $max_search } {

    # Check how many rows we got back.
    set cur_rpc [ expr $results(4,rpc) - $cur_rpc ]
    # Check our max_search is <= the available rowcount
    if { $cur_rpc<7 && $results(4,rpc) > $max_search } {
        set max_search $results(4,rpc)
    }
    for {set i 0} {$curind<$max_search && $i<$cur_rpc} {incr i} {
        if {$curind==0} {
            oci_parse -assign 5 { SELECT ROWID,ADDR_ID,HOUSE-
NUM,STREET,TOWN,COUNTY,POSTCODE FROM ADDR WHERE (ADDR_ID=:1) }
        }
        semsleep [expr int(rand()*3000)+1]
        oci_bind_begin 5
        oci_bind_pos 5 0 [ lindex $results(4,$curind) 1 ]
        oci_bind_end 5
        oci_exec 5
        oci_fetch 5 1
        set cust_id [ lindex $results(4,$curind) 2 ]
        set addr_id [ lindex $results(4,$curind) 1 ]
        oci_cancel 5
        incr curind
        log_pos [ expr 3 + $curind ]
    }
    oci_fetch 4 6
}
}
```

Note the addition of a `semsleep` on each address fetch, to simulate the user looking at each address as it is displayed. This is a random sleep, somewhere between 0 and 3 seconds, to more accurately reflect reality.

It's a good thing to check that this new fragment is actually performing the same task as the original Forms application, and also to see whether the simulated version looks anything like the original, at the tracefile level.

```
PARSING IN CURSOR #2 len=89 dep=0 uid=59 oct=3 lid=59 tim=1043518597703799
hv=3969767686 ad='5e453340'
  SELECT ROWID,ADDR_ID,HOUSENUM,STREET,TOWN,COUNTY,POSTCODE FROM ADDR WHERE
(ADDR_ID=:1)
END OF STMT
PARSE #2:c=0,e=170,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043518597703785
BINDS #2:
  bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset
  bfp=b6ba9710 bln=22 avl=05 flg=05
  value=4238106
EXEC #2:c=0,e=227,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043518597704169
FETCH #2:c=0,e=141,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1043518597704967
BINDS #2:
  bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset
  bfp=b6ba99c4 bln=22 avl=05 flg=05
  value=3910340
EXEC #2:c=0,e=197,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043518597705742
FETCH #2:c=0,e=95,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1043518597706136
BINDS #2:
  bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset
  bfp=b6ba99c4 bln=22 avl=04 flg=05
  value=759761
EXEC #2:c=0,e=173,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043518597706752
FETCH #2:c=0,e=89,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1043518597707123
BINDS #2:
  bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset
  bfp=b6ba99c4 bln=22 avl=05 flg=05
  value=3688132
EXEC #2:c=0,e=167,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043518597707711
FETCH #2:c=0,e=79,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1043518597708070
BINDS #2:
  bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset
  bfp=b6ba99c4 bln=22 avl=05 flg=05
  value=4439656
EXEC #2:c=0,e=166,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043518597708653
FETCH #2:c=0,e=78,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1043518597709006
BINDS #2:
  bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset
  bfp=b6ba99c4 bln=22 avl=05 flg=05
  value=3702708
EXEC #2:c=0,e=171,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043518597709598
FETCH #2:c=0,e=74,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1043518597709951
BINDS #2:
  bind 0: dty=2 mxl=22(22) mal=00 scl=00 pre=00 oacflg=01 oacfl2=0000 size=24 offset
  bfp=b6ba99c4 bln=22 avl=05 flg=05
  value=2163776
EXEC #2:c=0,e=167,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043518597710533
FETCH #2:c=0,e=81,p=0,cr=4,cu=0,mis=0,r=1,dep=0,og=4,tim=1043518597710891
FETCH #1:c=0,e=265,p=0,cr=8,cu=0,mis=0,r=7,dep=0,og=4,tim=1043518597711565
```

It's pretty close! Actually, the only real difference is that Forms appears to fetch seven rows the first time, and after five have been consumed it fetches six more. A little logic tweaking, and the script would follow exactly this pattern.

The next entry in the trace is the creation of the actual order. This entire section is performed by the trigger code listed earlier. You may remember that the first thing this trigger does is to grab NEXTVAL from a sequence number:

```
PARSING IN CURSOR #6 len=40 dep=0 uid=59 oct=3 lid=59 tim=1043522864579902
hv=2520645797 ad='5e8d09b4'
  SELECT ORDER_SEQ.NEXTVAL FROM SYS.DUAL
END OF STMT
PARSE #6:c=0,e=270,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043522864579891
BINDS #6:
EXEC #6:c=0,e=94,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1043522864581638
```

This raises an interesting point. It is extremely common for a client-side application to request a sequence number, and then proceed to use it. Wouldn't it be nice if we could spot this trend and automate the scripting somewhat? Well, that's something that is also taken care of in the preprocessing stage, **provided** that all the sequence

As you can see, comments are inserted in the generated script to clearly denote where automatic substitutions are made. This is to cater for the case where sequence values coincide with other values and a manual repair has to be made.

The other interesting thing to note in this example is that the `cust_id` and `addr_id` binds are made with literal values: Obviously this is not acceptable for a benchmark (always using the same customer is not a good idea!), and so manual editing will need to take place. This is probably the biggest known flaw in using the 10046 output to generate benchmarks; returned data is not present in the trace file, and so it cannot be 100% automated. That being said, it is believed to be possible with a little more investigation - see the appendix for details.

The only remaining piece of the application is that of the loop to lookup and allocate five line items for the order. This is another area that is begging for a loop, and so let's cut straight to the finished version.

```
# Cursors pulled out of loop - real tracefile only parses once, so we preserve the
oci_parse -assign 8 { SELECT SUBSTR(NAME,1,3) || '%' FROM NAMEROOTS WHERE
NAME_ID = :b1 }
oci_parse -assign 9 { SELECT ITEM_ID FROM ITEMS WHERE NAME LIKE :b1 AND ROWNUM
= 1 }
oci_parse -assign 10 { SELECT ITEM_ID,STOCK_ID,WHS_ID,STOCK_LEVEL FROM STOCK
WHERE ITEM_ID = :b1 FOR UPDATE OF STOCK_LEVEL }

#Start line items loop. Use between 1 and 10 lines per order.
}}
set int3 [ time {
for {set totlines [ expr int(rand()*10)+1]} {$totlines>0} {incr totlines -1} {
  set got_item false
  while { $got_item=="false" } {
    # RPC: FUNCTION SYS.DBMS_RANDOM.VALUE(LOW IN NUMBER, HIGH IN NUMBER
RETURN NUMBER;
    log_pos 25
    oci_rpccall {FUNCTION SYS.DBMS_RANDOM.VALUE( :LOW, :HIGH) } { {
dbaman_ret OUT "" } { LOW IN 1 } { HIGH IN 60000 } }
    oci_bind_begin 8
    oci_bind_pos 8 0 [ expr int($SYS_DBMS_RANDOM_VALUE_dbaman_ret)]
    oci_bind_end 8
    # SELECT SUBSTR(NAME,1,3) || '%' FROM NAMEROOTS WHERE NAME_ID =
:b1 ...
    log_pos 26
    oci_exec 8
    oci_fetch 8 1
    oci_bind_begin 9
    oci_bind_pos -vcforce 9 0 $results(8,0)
    oci_bind_end 9
    # SELECT ITEM_ID FROM ITEMS WHERE NAME LIKE :b1 AND ROWNUM = 1
...
    log_pos 27
    oci_exec 9
    oci_fetch 9 1
semsleep 1000
    if { $results(9,rpc) != 0 } {
      oci_bind_begin 10
      oci_bind_pos 10 0 $results(9,0)
      oci_bind_end 10
      # SELECT ITEM_ID,STOCK_ID,WHS_ID,STOCK_LEVEL FROM STOCK
WHERE ITEM_I ...
      log_pos 28
      oci_exec 10
      oci_fetch 10 1
      set got_item true
    } else {
      puts "retrying for line item"
    }
  }
}
# RPC: PROCEDURE ORDERS.ALLOC_STOCK(IN_ORDER_ID IN NUMBER, IN_ITEM_ID IN
NUMBER, IN_STOCK_ID IN NUMBER);
log_pos 29
set stock_id [ lindex $results(10,0) 1 ]
oci_rpccall {PROCEDURE ALLOC_STOCK( :IN_ORDER_ID, :IN_ITEM_ID, :IN_STOCK_ID
) { { IN_ORDER_ID IN $seq(order_seq) } { IN_ITEM_ID IN $results(9,0) } {
IN_STOCK_ID IN $stock_id } }
oci_commit
}
```

One thing to observe here is that the preprocessor automatically creates a return variable for RPC called `FUNCTIONS`. It always assigns the value to a variable named `dbaman_ret`. In fact, to have some kind of hope of avoiding namespace collisions, `dbaman` prefixes the name of the variable with the name of the function, making the variable name `SYS_DBMS_RANDOM_VALUE_dbaman_ret` in this case. We use this value as a bind in the next query, which was manually edited.

It's unlikely that your real customers will always have five line items, so I've put a randomisation on the line item count (between 1 and 10) to make it a bit more realistic. Of course, for a real application, you could determine what the correct range of values should be here.

All that remains in the script is Forms desperately trying to rollback its already committed work (twice). We will keep that in, because it's all counted as calls to the database, and we want it to be as realistic as possible.

INITIAL TESTING

The first thing to do when testing an edited script is to run it through using library functions from a utility script named `dummy_library.tcl`. This script is a set of stubbed procedures that will allow the script to run as a single session. Once the script is running as a single session you can compare the trace file it produces with the original trace file, to see if you are close. I also made a change to both the original form and the simulation script to dump session statistics into a table at completion, so that we can compare those. First, though, let's look at a `tkprof` output for

both. This output is from the version of tkprof supplied with Oracle9i, and so kindly provides a summary of all the waits in addition to the call statistics:

ACTUAL:

OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call	count	cpu	elapsed	disk	query	current	rows
Parse	11	0.01	0.00	0	2	0	0
Execute	41	0.04	0.01	0	21	43	72
Fetch	40	0.02	0.04	2	191	4	61
total	92	0.07	0.06	2	214	47	133

Misses in library cache during parse: 1

Elapsed times include waiting on following events:

Event waited on	Times Waited	Max. Wait	Total Waited
SQL*Net message to client	106	0.00	0.00
SQL*Net message from client	106	10.72	24.56
db file sequential read	2	0.02	0.03
log file sync	8	0.00	0.00

SIMULATED:

OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call	count	cpu	elapsed	disk	query	current	rows
Parse	23	0.01	0.01	0	2	0	0
Execute	53	0.02	0.04	0	38	125	79
Fetch	38	0.03	0.00	0	179	3	53
total	114	0.06	0.06	0	219	128	132

Misses in library cache during parse: 1

Misses in library cache during execute: 1

Elapsed times include waiting on following events:

Event waited on	Times Waited	Max. Wait	Total Waited
SQL*Net message to client	99	0.00	0.00
SQL*Net message from client	99	0.00	0.04
log file sync	6	0.00	0.00

Starting with the Parse line, there is immediately a discrepancy, one which will also affect the other statistics. There is a difference of 12 parse calls between the actual and simulated trace files. The good news is that this can be explained relatively easily!

The tkprof utility used to produce these summaries has no idea what a PL/SQL RPC call is, and so omits these from its output. The simulated trace file is using an emulation of RPC calls by submitting an anonymous PL/SQL block making the call. This appears in the trace file as a standard SQL statement, which is counted by tkprof in its output. To be more scientific, I manually counted all parse calls for the simulated run **excluding** the emulated RPC calls, and found a total of 11 there also. Phew.

For the Execute section, I could count 40 Executes excluding the RPC calls. The error of one execute can be explained by pollution in scientific method: the Forms run where I produced this trace file had one extra cursor-down in the customer look up, thus facilitating an additional address look up. The differences evident in the 'current' and 'query' columns for Execute can both be explained by the absence of RPC calls in tkprof's counting, and seem reasonable. So far, so good.

Fetches are more straightforward, because they aren't affected by the RPC calls. The difference of two fetches is explained by a) our algorithm in the first loop we created is slightly erroneous, thus missing one of the fetches and b) the extra address look up mentioned above.

The only other thing of interest in these summaries is that the amount of time spent waiting on SQL*Net message from client is very different. This is because the `dummy_library.tcl` script used to facilitate testing ignores all sleeps and applies the script as quickly as possible!

So, through the slightly distorted eyes of `tkprof`, the simulation looks pretty accurate. Let's complicate matters by looking at a selection of session statistics for the actual and simulated sessions. I took all statistics that were non-zero, and then removed the irrelevant ones, such as `session connect time`:

TABLE 3. Statistical Differences Between Forms and Simulated Forms

Statistic	Simulator	Forms	Diff	Comments
redo synch time	1	0		Probably rounding error
cursor authentications	0	8	-100%	
physical read IO requests	0	2	-100%	Normal caching side-effect
physical reads	0	2	-100%	Ditto
physical reads cache	0	2	-100%	Ditto
redo ordering marks	0	1	-100%	
user I/O wait time	0	3	-100%	A result of zero sleep time
DB time	9	17	-47%	This is the most alarming difference of all. Unless there is something wrong with this new (in 10g) statistic, something is quite different.
free buffer requested	6	9	-33%	
workarea executions - optimal	7	9	-22%	
opened cursors current	12	15	-20%	
sorts (memory)	9	11	-18%	
SQL*Net roundtrips to/from client	99	115	-14%	This suggests that Forms is performing other untraced activity, OR that it is not bundling calls as effectively as OCI.
session pga memory max	458444	523980	-13%	I'm surprised about this one. I thought we would have a comparable pga size.
table fetch by rowid	65	73	-11%	
buffer is not pinned count	104	115	-10%	
buffer is pinned count	52	58	-10%	
no work - consistent read gets	80	88	-9%	
calls to kcmgas	16	17	-6%	

TABLE 3. Statistical Differences Between Forms and Simulated Forms

Statistic	Simulator	Forms	Diff	Comments
consistent gets	248	260	-5%	A difference of 5% seems reasonable.
consistent gets from cache	248	260	-5%	
enqueue releases	35	37	-5%	
enqueue requests	36	38	-5%	
shared hash latch upgrades - no wait	42	44	-5%	
calls to get snapshot scn: kcmgss	91	95	-4%	
consistent gets - examination	108	112	-4%	
rows fetched via callback	23	24	-4%	
session logical reads	378	393	-4%	
index fetch by key	29	30	-3%	
sorts (rows)	64	66	-3%	
db block gets	130	133	-2%	
db block gets from cache	130	133	-2%	
recursive calls	162	165	-2%	
db block changes	144	145	-1%	
IMU commits	5	5	0%	All the following statistics are identical, and need no further explanation.
IMU undo allocation size	22380	22380	0%	
commit cleanouts	53	53	0%	
commit cleanouts successfully completed	53	53	0%	
deferred (CURRENT) block cleanout applic	40	40	0%	
enqueue conversions	5	5	0%	
immediate (CURRENT) block cleanout appli	2	2	0%	
index scans kdiixs1	42	42	0%	
logons cumulative	1	1	0%	
logons current	1	1	0%	
messages sent	6	6	0%	
parse time cpu	1	1	0%	
recursive cpu usage	2	2	0%	
redo entries	19	19	0%	
redo size	19704	19764	0%	
redo synch writes	6	6	0%	
session pga memory	392908	392908	0%	
session uga memory	142728	142728	0%	
session uga memory max	142728	142728	0%	
switch current to new buffer	5	5	0%	
table scan blocks gotten	38	38	0%	
table scan rows gotten	2435	2435	0%	
table scans (long tables)	5	5	0%	
undo change vector size	6932	6932	0%	
user commits	5	5	0%	
user calls	116	105	10%	The difference of 11 makes me highly suspicious that this is another statistic which is not updated when the call is an RPC call.
execute count	78	70	11%	RPC Calls

TABLE 3. Statistical Differences Between Forms and Simulated Forms

Statistic	Simulator	Forms	Diff	Comments
CPU used by this session	9	8	13%	For once, these are actually the same. 13% is not too bad, and is probably artificially high due to rounding error.
CPU used when call started	9	8	13%	
opened cursors cumulative	37	28	32%	The dbaman harness uses a 'lazy close' algorithm, which could result in this difference.
parse count (total)	37	28	32%	RPC Calls
bytes sent via SQL*Net to client	17729	12106	46%	These two are interesting.
bytes received via SQL*Net from client	16144	10799	49%	There is clearly a significant amount of network traffic associated with Forms that is still not simulated using this technique.
parse count (hard)	2	1	100%	Caching side-effect.
parse time elapsed	2	1	100%	Increased parsing plus rounding error.
user rollbacks	4	1	300%	This is a side effect of the way the preprocessor creates the script. It looks for all SQL (including 'rollback'), but also looks for the tracefile tags that denote rollbacks through the programmatic interface (which do not show up as SQL). However, when an SQL rollback is performed, BOTH appear in the trace file! Another discrepancy is that one of the Forms rollback commands occurs after the statistics are gathered.

The only statistic of these that **really** disturbs me is that of DB Time. I did run some tests where the script actually slept at the `semsleep` points, and the DB time **did** increase. This is not supposed to be how this statistic works, and could indicate an implementation problem for this statistic.

MULTI-USER TESTING

Now that this initial testing is complete, we can start to combine this script into the full execution harness and run multiple sessions. The first thing to do is to supply some driving data that the script can use. In the case of this simple demo application, we only need the first few letters of a surname, which I generated at random from the database. In the process of doing this, I encountered a few problems related to my driving data being unsuitable. The worst of these was when surnames did not exist in my database, which I had originally not catered for in the script loops. Once these issues were addressed, I could plug the script into the multi-user framework, and execute multiple users.

Before we go into the gory details of what the multi-user framework entails, let's have a quick look at the front end in action:

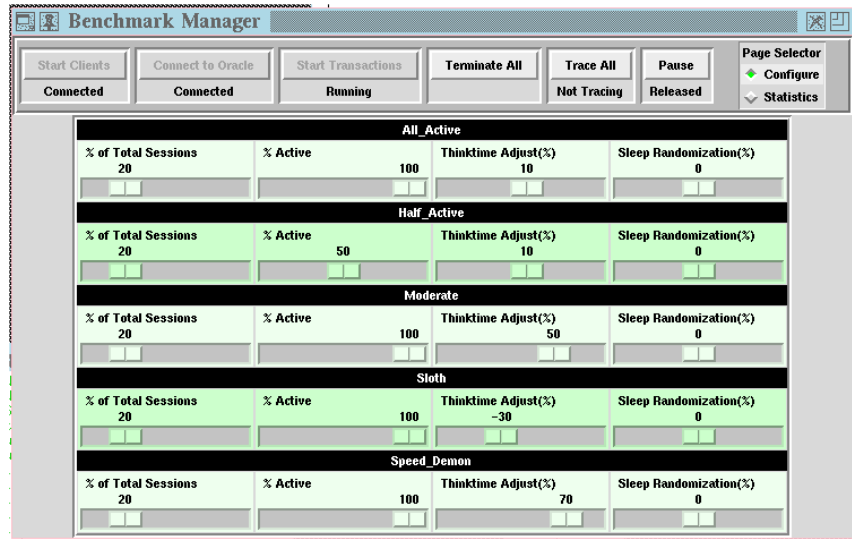
Client Processes		50			
Oracle Connections		50			
Type	Name	Count	1m	5m	15m
0	All_Active	242.0	10.0	9.7	0.0
1	Half_Active	117.0	5.0	4.9	0.0
2	Sloth	167.0	6.0	6.3	0.0
3	Moderate	436.0	19.0	18.0	0.0
4	Speed_Demon	680.0	28.9	27.9	0.0

ID	TX	Count	Pos	Inter1	Inter2	Inter3	Inter4	Last Inter
0	0	24	51	29.3	1.2	5.4	35.9	1.0
1	0	24	51	14.6	0.0	14.5	29.1	28.3
2	0	21	51	28.3	0.3	9.1	37.7	0.0
3	0	24	8	25.3	1.5	9.9	36.7	16.8
4	0	25	9	21.5	0.6	9.0	31.1	15.7
5	2	17	51	45.0	0.3	11.6	56.9	16.8
6	0	23	2	19.4	1.0	14.5	34.9	34.5
7	3	43	5	10.0	0.0	1.9	11.9	19.9
8	0	26	27	28.4	0.1	10.1	31.3	2.1
9	2	18	10	19.2	0.5	14.3	34.0	71.1
10	3	42	9	11.9	3.8	3.1	18.7	12.6
11	3	44	51	8.7	0.3	2.5	11.5	2.1
12	3	44	51	11.8	0.0	2.5	14.3	0.0
13	0	24	51	22.1	1.7	6.3	30.2	10.5
14	0	26	12	25.8	1.3	10.1	37.2	36.6

It does not have the prettiest face in the world, but it is quite nicely functional. It is also hiding the complexity of the multi user framework from the user. The controls are all placed at the top of the screen, and on another screen which is selected, surprisingly, with the 'Page Selector' switch on the right. The startup process can be as manual or automatic as desired, based upon the configuration in the single setup file. If desired, the client startup (process creation), connection to Oracle, and the workload can all be started in sequence as soon as the frontend is booted. Alternatively, they can all be started manually with the buttons, which is mostly preferable on large benchmarks with thousands of users.

The next button is the Terminate All button, for closing down the test. After that is a button to signal all the sessions to start producing 10046 trace files, for diagnostic purposes. This can be turned on or off at will. Finally, and very importantly, is the Pause button. Remember the `semsleep` function? Every time a client session goes for a sleep, it checks the status of the semaphore that is controlled by this button. Therefore, it is very easy to pause and resume the testing at any time.

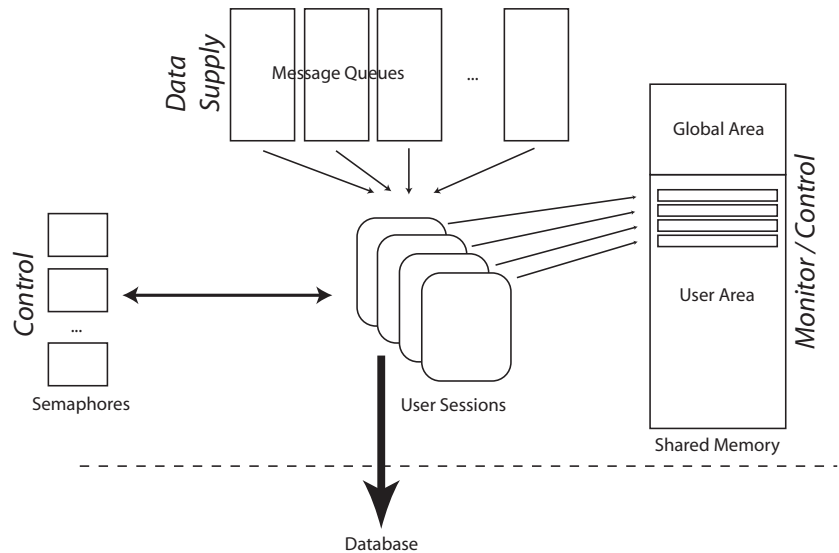
The Configure screen echoes the setup parameters that it might be nice to fiddle with while the benchmark is running:



For each of the 'types' of transaction (more later), the mix, percentage active, think times and sleep randomization can be changed. The first two can only be changed when the benchmark is in 'prestartup' mode, whereas the latter two can be changed on the fly. We will cover the function of all these parameters in the next section.

MULTI-USER FRAMEWORK

Now we've seen the face of the framework, let's have a look at what's going on behind the scenes. A high level diagram will be of no help at all, but perhaps it will make sense after all the words, so I'll include it anyway:



It's probably best to start with the directory layout for the execution harness:

TABLE 4. Directory Structure

Directory	Contents
<code>./bin</code>	All executables, both binary and non-editable scripts.
<code>./conf</code>	Configuration file
<code>./data</code>	Driving data
<code>./include</code>	Tcl include files
<code>./lib</code>	Shared libraries
<code>./log</code>	Log file infrastructure
<code>./script</code>	Application scripts, and pre- and post-processing scripts
<code>./tcllib</code>	Tcl libraries

There are very few edit touchpoints for the user of the harness. The only directories that require any attention are the 'script' and 'conf' directories - all the other directories should remain static.

To boot the harness, one simply needs to execute `frontend.tcl`, which starts the GUI. This script performs the following steps at startup time:

- Reads the `setup.tcl` file to determine required configuration.
- Creates a new directory structure for the logs from this run, and a symbolic link to them named 'lastrun' for ease of use.
- Creates all the Inter-Process Communication (IPC) facilities to support the execution.
- Starts the data pumps to feed driving data to the scripts.
- Creates the GUI interface

The IPC facilities are worthy of explanation. The harness uses all three UNIX System V IPC facilities: Shared Memory, Semaphores and Message Queues. The shared memory is used for transfer of information between the execution slaves and the front end. For example, the front end can signal the slaves to start SQL tracing by setting a flag in the segment. Conversely, the slaves report back all their timing and positional information through the segment.

The semaphores are used for the implementation of serially threaded functions such as login, the Pause functionality, and for control over the number of active slaves. In addition, it can be used in the scripts themselves to serialize on certain activity, if so desired.

The message queues are used almost exclusively for data supply from the pumps to the slaves. The pumps keep the queues full, and the slaves read data from them as required. In addition, a queue is used to send the slaves the name of the script they are required to execute.

The only files that need to be edited before execution (apart from the application script itself) are the configuration file and (optionally) the pre- and post-processing scripts. The pre/post-processing scripts are executed by the front end at the appropriate times, and can be used to do things like statistics collection, log file archiving,

and so forth. The master configuration file is a little complex to the uninitiated, so we will go through it a little.

The first section in the file is where we define the different ‘types’ of transaction. Each of these types can run a different script, have different think times, a different mix of active/inactive and so forth. This supports the creation of different logical areas of testing, such as reservations, check-in and check-out, all as independently tracked entities. This fragment shows the definition of two transaction types:

```
# Transaction definitions
set txttype_thinkadj(login) 0

set txttype_ind(All_Active) 0
set txttype_pct(All_Active) 20
set txttype_pctActive(All_Active) 100
set txttype_thinkadj(All_Active) 10
set txttype_randadj(All_Active) 0
set txttype_script(All_Active) newOrder.tcl

set txttype_ind(Half_Active) 1
set txttype_pct(Half_Active) 20
set txttype_pctActive(Half_Active) 50
set txttype_thinkadj(Half_Active) 10
set txttype_randadj(Half_Active) 0
set txttype_script(Half_Active) newOrder.tcl
```

There is a ‘special’ transaction type for the login process, which only has one parameter: thinkadj. This allows the login process to be slowed somewhat if problems are encountered with login timeouts. The following two sections define the ‘real’ transaction types named ‘All_Active’ and ‘Half_Active’. Both transaction types specify that, of all the connected sessions, they should represent 20%. All_Active requests that 100% of its allocated slaves should be active at any one time, whereas Half_Active, amazingly, only wants 50% of the slaves to be busy at any one time.

Each transaction specifies a ‘thinkadj’ of 10%. This means that any sleeps in the script will be shortened by 10%. In a similar way, the ‘randadj’ parameter varies the sleep times by +/- the percentage specified. This allows measures to be taken against simulations ‘rollercoasting’. Both of these parameters can be varied on the fly from the front end GUI.

Finally, the name of the script is given. In our case, we are using the same transaction for all our ‘types’, newOrder.tcl, but varying the execution attributes.

There are many other configurable options in the setup file, but I will just focus on the remaining ones that are critical for getting up and running:

```
# Number of slave processes to start
set procs 50

# Oracle authentication
set userprefix USR
set passprefix USR
set tnsname micro

# Message Queues
set scriptQueue 1

set feedQID(surname) 2
set feedFile(surname) data/surname.dat
```

The ‘procs’ parameter determines the total number of sessions that will be started and connected to the database. They will connect using the ‘userprefix’ and ‘passprefix’ for the username and password, respectively, each of which will have a sequential number appended to it. They connect to the specified TNS alias.

The driving data is provided by specifying the feed by name in the 'feedQID' and 'feedFile' parameters. The QID is just an incremental value, and the file is simply a file containing one value or set of values per line. These can then be retrieved in the application script using the 'getVal <name>' call.

POSTSCRIPT

From a very rough standpoint, that's all there is to it. The hardest remaining part of the whole process is really the creation and use of driving data that accurately represents your application's usage. The same can be said of any of the other methods of benchmarking, as this is really something that could ever be automated. Of course, there are numerous techniques that can be employed to make this process easier, and it is possible that these could one day form part of this toolkit.

The availability of this software is still a little in flux, as of this writing. An early version of it, with notable omissions such as PL/SQL RPC support and the multi user harness, was included in my book, *Scaling Oracle8i*. The full, current toolkit will hopefully become Open Source at some point in the near future, once some small issues have been resolved. For the time being, I would be interested in releasing the beta code to interested parties, provided that it is used to generate real benchmarks and that good feedback is returned in order to improve further versions. If you are interested in using the toolkit, please feel free to contact me by email at James@Morle.com for the latest status.

ABOUT THE AUTHOR

With 15 years in professional computing, James has been personally responsible for the architecture, implementation and tuning of some of the world's largest and most complex business systems. James is a well respected member of the Oracle community, and is the author of the critically acclaimed book, *Scaling Oracle8i*. In 2000, James founded Scale Abilities Ltd, a small consultancy specializing in high-end Oracle systems implementation and large consolidation projects. James is currently working in Oracle development, focused on Enterprise Manager. He is also a founder member of the OakTable Network (www.oaktable.net).

Appendix: Futures

Shortly before writing this paper, I was alerted by my good friend Joakim Treugut to the existence of Oracle event 10079. This event could well hold the key to closing the automation loop during the preprocessing stage, in that it dumps the SQL*Net packets into the same tracefile as the 10046 output. Included, somewhere, in this hex dump is the information that is returned from a fetch operation, and could at least in theory be used to determine most of the reuse of values throughout the application. This would be a similar process to that performed today in the preprocessing with sequence number values being automagically substituted in the correct places in the generated script.

I fully intend on looking carefully at this data to try and determine how difficult this final step would be to make, and will incorporate any findings into the toolkit.