

Oracle Instrumentation

Andy Rivenes, AppsDBA Consulting
andy@appsdba.com

Revised 2008/10/16

[Note: An edited version of this article appeared in the first quarter of 2009 issue of the IOUG SELECT Journal]

It has become popular to say that applications written to run on Oracle databases should be instrumented. But other than using the database package `DBMS_APPLICATION_INFO`, what does this really mean? This article will explore instrumenting applications to run on Oracle databases and will make a first pass at defining an instrumentation framework. It will also pass along some guidelines the author has learned while instrumenting Oracle based applications.

Introduction

What does it really mean to instrument an application? A Google search on the terms “Oracle” and “instrumentation” will result primarily in hits about performance related measurement. The intent of this paper is to show that this is really only one small part of a well instrumented application. Anyone who has written even a basic application knows that some amount of debugging must be available to troubleshoot application logic and data related bugs. The ability to log messages from debugging and possibly run time operation is usually desirable as well. Ask any database administrator and they will surely acknowledge how difficult it is to monitor an application when you have no idea what it’s doing or what code is being run, so let’s add runtime registration to the list of desirable instrumentation qualities. And as we’ve seen from our Google search, we will probably want some ability to perform metric collection about the functioning of the application from a performance perspective, and probably a workload measurement and characterization perspective as well for an application of any size and importance.

Clearly we’ve now gone way beyond simply instrumenting our application to create performance metrics. We want to know what it’s doing, how it’s doing it, how long it took and how much it costs. To gather this kind of information requires a more disciplined approach to instrumentation than just writing and deploying some application code. So to summarize, we can classify instrumentation into the following categories:

1. Debugging
2. Logging
3. Runtime registration
4. Metric collection

These categories form the basis for an instrumentation framework that will create a robust operational environment for applications that run in an Oracle database. The following will describe the beginnings of such an instrumentation framework. I’ve said beginnings because I don’t believe that this will be an all inclusive list and I do believe that this framework will evolve over time as different people add their ideas and experiences.

Debugging

Application debugging tools will vary from environment to environment, but will in general encompass some basic characteristics. But before we begin let's differentiate between debugging and logging. Many times the two are used interchangeably and many implementations combine the functionality. However, there are some underlying differences that are worth describing. Debugging really boils down to being able to quickly fix coding errors. Whether those errors are the result of errors in coding or errors encountered at runtime, debugging is all about fixing problems. The following is the start of a list of debugging characteristics:

- Should be able to enable debugging for one or more “modules” in the code.
- Should be able to activate debugging remotely for one or more “users”.
- Debugging should be written to one or more “persistent” sources and preferably to a console for initial debugging.
- There needs to be enough debugging code to be useful.

The fourth item is a little harder to quantify and really will be code dependent. However, you should write enough debugging information so that you can tell what's going on in the code. This probably means identifying inputs and outputs and what has occurred. Of course, a lot of this will probably become more obvious once you actually start debugging your code. Probably the biggest point to stress is, don't remove your debugging code once you've got it running, and while you are debugging keep in mind how you would perform that debugging once your application has been deployed.

There are examples of the first three characteristics available. All three are available publicly in Tom Kyte's “debug” package for PL/SQL. Tom's package supports setting debugging for one or more users and for one or multiple packages. It will also output debugging information to an operating system file. The following is an example using the debug package:

```
SQL> create or replace procedure debug_test
 2  as
 3  v_name  global_name.global_name%TYPE;
 4  begin
 5      debug.f('Enter debug_test');
 6      select global_name into v_name
 7      from global_name;
 8      debug.f('Global name: '||v_name);
 9      debug.f('End debug_test');
10  end;
11  /
```

Procedure created.

```
SQL> set serveroutput on;
SQL> exec debug.init(p_file=>'c:\temp\temp.txt');
```

PL/SQL procedure successfully completed.

```
SQL> exec debug.status;
```

Debug info for UTILITY

```
-----
USER:                UTILITY
MODULES:             ALL
FILENAME:            c:\temp\temp.txt
SHOW DATE:           YES
DATE FORMAT:         MMDYYYYY HH24MISS
NAME LENGTH:        30
SHOW SESSION ID:    NO
```

```
PL/SQL procedure successfully completed.
SQL> exec debug_test;
PL/SQL procedure successfully completed.
SQL>
```

The resulting output file looks like:

```
$ cat temp.txt
10022008 133501(          UTILITY.DEBUG_TEST      5) Enter debug_test
10022008 133501(          UTILITY.DEBUG_TEST      8) Global name: ORCL
10022008 133501(          UTILITY.DEBUG_TEST      9) End debug_test
$
```

We see the time stamp, the calling package and procedure, the line number and then the message.

Logging

Logging should not be confused with a logging tool or a logging framework. Logging in this context refers to application level logging or the output of status and/or informational messages as to the progress of an application task. This might be considered as a type of “auditing” or operational tracking of a task or set of tasks. This is different than debugging in both purpose and the types and detail of messages output. Debugging is all about fixing coding errors or logic. Logging is all about tracking what happened. Now, not all application code may require logging, but I would argue that all code requires debugging, unless of course you write perfect code.

So logging is really the program code’s method of recording what happened. It may be as simple as outputting a message at startup and shutdown of a daemon task or it may be more sophisticated with messages that some number of rows has been processed or some task has been completed. The other critical purpose of logging is to provide a mechanism to record errors and diagnostic information. Exceptions that are raised or runtime anomalies need to be captured in a persistent state to allow for proper handling and debugging.

Although logging is dependent on the nature of the application here are some common requirements for application logging:

- Output to a persistent source
- Provide logging levels (i.e. informational or error)
- The ability to record errors and anomalies
- The ability to record runtime information

There are several “logging” tools available for Oracle and many, many tools available industry wide. Tools like log4j and log4perl are designed for Java and Perl respectively. For Oracle PL/SQL apps there is log4plsqli, oralog and even Tom’s Debug package could be used. Of course there are many logging/debugging tools built on the fly on a per application basis. Anything as simple as a DBMS_OUTPUT.PUT_LINE command can be considered a form of logging. However, to meet the basic requirements listed above does require some amount of sophistication over a simple PUT_LINE statement. The point is that logging is really just a tool to be used to provide a persistent mechanism for a program to communicate with the outside world. As we’ve seen in the previous debugging section a

logging framework provides the ability to disseminate runtime information to a persistent place for later analysis.

The following will show a logging example using the open source log4plsqli tool.

```
SQL> exec plog.info('mess info');
```

```
PL/SQL procedure successfully completed.
```

```
SQL> select * from vlog;
```

```
LOG
```

```
-----  
[Oct 02, 14:12:30:34][INFO][LOG4PLSQL][block][mess info]
```

```
SQL> create or replace procedure TestProc is  
2     cpt number;  
3     begin  
4         plog.info('this select raise ORA-01403:No Data Found');  
5         select id into cpt from tlog where id = -1;  
6         exception  
7             when others then  
8                 plog.error; -- default message is SQLCODE SQLERRM  
9     end;  
10 /
```

```
Procedure created.
```

```
SQL> exec testproc;
```

```
PL/SQL procedure successfully completed.
```

```
SQL> select * from vlog;
```

```
LOG
```

```
-----  
[Oct 02, 14:12:30:34][INFO][LOG4PLSQL][block][mess info]  
[Oct 02, 14:13:19:75][INFO][LOG4PLSQL][block-->LOG4PLSQL.TESTPROC][this select raise ORA-01403:No Data Found]  
[Oct 02, 14:13:19:75][ERROR][LOG4PLSQL][block-->LOG4PLSQL.TESTPROC][SQLCODE:100 SQLERRM:ORA-01403: no data found]
```

```
SQL>
```

Runtime Registration

We've explored debugging and logging and now we're going to focus on the topic that is probably most associated with Oracle based instrumentation, run time registration. Run time registration is the name that I believe best describes the use of the calls to set services, client id, client info, module and action in the various Oracle V\$ views, the extended SQL trace file and the AWR and ASH interfaces. Oracle supplies various methods to set these fields, but one of the main purposes in doing so is to enable the identification, at run time, of the code or process in the database.

Services

Services provide a mechanism for grouping database connections. Usually services can be implemented without having to make changes to application code. By assigning connections to services based on application or logical functions the services can be used to classify and track database workload. A service is assigned to every database session at connect time. This occurs either implicitly, the defaults are SYS\$USERS for user sessions not associated with any application service and SYS\$BACKGROUND for background sessions, or explicitly through Oracle Networking connect strings.

Services are also assigned to DBMS_SCHEDULER jobs, through the assignment of a job class, and parallel operations by virtue of inheriting the invoking process' service, and can be associated with any distributed operations as well, again through Oracle Networking connect strings.

Oracle Networking connect strings can exist in many different forms and places, but in the simplest terms will either be associated with a "tnsnames.ora" entry or the "DISPATCHERS" init.ora parameter. Services can be created in Enterprise Manager or by setting the "service_names" init.ora parameter in single instance Oracle or using DBCA when using RAC.

One of my favorite examples of using services is to create a backup service and always run backups through that service. It can be very eye opening to see the resource usage that backups consume. The following shows a sample tnsnames.ora entry for a backup service called "backup.appsdba.com":

```
ORCL_BACKUP =
  (DESCRIPTION =
    (ADDRESS =
      (PROTOCOL = BEQ)
      (PROGRAM = oracle)
      (ARGV0 = oraclearcl)
      (ARGS = '(DESCRIPTION=(LOCAL=YES) (ADDRESS=(PROTOCOL=beq)))')
    )
    (CONNECT_DATA =
      (SERVICE_NAME = backup.appsdba.com)
    )
  )
```

Not all connections make use of a tnsnames.ora file, but the connect string syntax for most tools supports the use of services. For example, the latest JDBC connections support the following:

JDBC Thin driver:

```
String URL = "jdbc:oracle:thin:scott/tiger@//myhost:1521/my servicename";
ods.setURL(URL);
Connection conn = ods.getConnection();
```

JDBC OCI driver:

```
ods.setUrl(
"jdbc:oracle:oci:@(DESCRIPTION=
  (ADDRESS=(PROTOCOL=TCP) (HOST=cluster_alias)
  (PORT=1521))
  (CONNECT_DATA=(SERVICE_NAME=service_name)))");
```

Once assigned, a connection's service name is visible in the V\$SESSION view as well as other V\$ and DBA_ views, and various AWR and ASH views.

Client Identifier

The "client identifier" field in the V\$SESSION view can be used to provide a means to identify the "real" application user when using shared connections. In other words, if all application connections connect to the same database user, then the client identifier field can be used to identify the authentication user. It then becomes much easier to associate a database session with an actual application user. There is nothing more frustrating when trying to associate a specific user to a database connection than to see all of the connections in the V\$SESSION view connected as the same user. The client identifier field can be set using the DBMS_SESSION.set_identifier database procedure. It can also be set using the OCI attribute OCI_ATTR_CLIENT_IDENTIFIER or the Java method Oracle.jdbc.OracleConnection.setClientIdentifier.

Client_Info, Module and Action

The client_info, module and action fields can be set using the DBMS_APPLICATION_INFO database package. The client_info field is available as part of the V\$SESSION view and the module and action fields are recorded in the V\$SESSION and V\$SQLAREA views according to the documentation, but actually show up in a variety of views (i.e. ASH and AWR and derivatives of V\$SESSION and V\$SQLAREA).

The Oracle description of the client_info field is that its purpose is to supply additional information about the client application. This makes it an ideal companion to the client_identifier field described above for use in fully identifying the client session attributes. The module and action fields are described as naming the module currently running and the current action. This leaves room for quite a bit of interpretation as to how to use these fields in a running application.

Usage Guidelines

Having now created several applications dedicated to using these four fields, I offer the following usage guidelines.

- Only call DBMS_APPLICATION_INFO through an interface that supports call nesting. I can't stress this strongly enough. As good as the idea of identifying database calls with module and action is, without the ability to support a hierarchical call structure it is basically useless. The problem with DBMS_APPLICATION_INFO is that there is no inherent ability to set module and action back to what they were before a subsequent call is made. In other words, if a procedure or function sets module and action, and then calls another procedure or function that in turn also sets module and/or action, when that procedure or function returns the prior module and action will not be reset back to their values. Instead the values will remain at whatever setting they were last set to. The following illustrates the problem:

```
SQL> create or replace procedure set_modact
 2  as
 3    v_mod      v$session.module%TYPE;
 4    v_act      v$session.action%TYPE;
 5  begin
 6    dbms_application_info.set_module(
 7      module_name => 'Called module',
 8      action_name => 'Called action' );
 9    --
10    select module, action
11    into v_mod, v_act
12    from v$session
13    where sid = SYS_CONTEXT('USERENV','SID');
14    --
15    dbms_output.put_line('set_modact module: '||v_mod||', action: '||v_act);
16  end;
17  /
SQL>
SQL> set serveroutput on;
SQL> declare
 2    v_premod   v$session.module%TYPE;
 3    v_postmod  v$session.module%TYPE;
 4    v_preact   v$session.action%TYPE;
 5    v_postact  v$session.action%TYPE;
 6  begin
 7    dbms_application_info.set_module(
 8      module_name => 'Calling module',
 9      action_name => 'Calling action' );
10    --
```

```

11  select module, action
12  into v_premod, v_preact
13  from v$session
14  where sid = SYS_CONTEXT('USERENV','SID');
15  --
16  dbms_output.put_line('Calling module: '||v_premod||', action: '||v_preact);
17  --
18  set_modact;
19  --
20  select module, action
21  into v_postmod, v_postact
22  from v$session
23  where sid = SYS_CONTEXT('USERENV','SID');
24  --
25  dbms_output.put_line('Returned module: '||v_postmod||', action: '||v_postact);
26  end;
27  /
Calling module: Calling module, action: Calling action
set_modact module: Called module, action: Called action
Returned module: Called module, action: Called action
SQL>

```

- Do not call instrumentation for row by row processing. This may require a little clarification, and may or may not be a hard and fast rule, but in general you don't want to instrument calls, functions specifically, that are called as each row of a result set is returned. There's really no value in it since module and action will never be set long enough to do any good, and the practice will introduce quite a bit of overhead. Unfortunately this precludes a consistent format for all procedures and functions, but the benefit of not introducing wasted overhead far outweighs any philosophical desire to instrument all procedures and functions.
- Create a consistent naming policy for module and align it with application functionality. Creating a useful naming policy can be difficult. What this guideline is trying to state, is that you will probably not want to blindly name all of your DBMS_APPLICATION_INFO.set_module calls with the name of the package being called. I have found that it is much more useful to name module settings with the logical function being performed. In most of the applications that I've seen the package name doesn't provide a broad enough scope to easily categorize the basic functions of the application. The action field can be used to provide a finer granularity of what actual code is running or what task is being performed. Following this guideline will also make the OEM or Grid Control screens more useful.
- Conditionally compile your instrumentation calls. This can be a huge benefit for any code that you develop that you want to be portable to other databases. If the instrumentation calls that you are making are not available in the database, or to the schema, that you're working with then your code will not compile. However, if you've added conditional compilation directives to your code you can simply "turn off" your instrumentation and your code will still function. Of course you will lose the benefits of instrumentation, but at least your code will run. The following is an example of conditionally compiled code referencing the very useful Hotsos Instrumentation Library for Oracle which is an open source tool that may not be installed in the database:

```

create or replace procedure test_cond
is
begin
  $IF $$hotsos_ilo $THEN
    HOTSOS_ILO_TASK.BEGIN_TASK(
      module => 'Set Module',
      action => 'Set Action');
  $END
  --
  dbms_lock.sleep(5);

```

```
--
$IF $$hotsos_ilo $THEN
  HOTSOS_ILO_TASK.END_TASK;
$END
end;
```

Metric Collection

Metric collection can be done for an individual session, a grouping of sessions or the entire database. There are several collection methods available in the Oracle database and most are greatly enhanced using the instrumentation methods described in the preceding section. Being able to collect metrics for connections grouped around application based logical functions can be invaluable. Identifying how much CPU and I/O that the users of an application are generating, or within an application being able to categorize the resource usage of the logical tasks, can be very valuable capacity planning information.

Individual Sessions

For the individual session the most precise facility available is extended SQL trace data. When used along with DBMS_APPLICATION_INFO calls, very precise metrics can be collected. The following is a sample of the type of information available in an extended SQL trace file.

```
SQL> exec dbms_application_info.set_client_info('appsdba');

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.00
SQL> exec dbms_application_info.set_module('Trace Demo','Set trace');

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.00
SQL> exec DBMS_MONITOR.SESSION_TRACE_ENABLE(NULL,NULL,TRUE,TRUE);

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.18
SQL> select global_name from global_name;

GLOBAL_NAME
-----
DB1.APPSDBA.COM

Elapsed: 00:00:01.57
SQL> exit
```

The resulting trace file has been shortened to include only the actual SQL statements issued and to show the informational statements now included. These can be used as demarcation points within the trace file, and with a profiler can produce very detailed timing information for specific module and/or action groupings.

```
*** 2008-10-07 21:58:19.171
*** ACTION NAME:(Set trace) 2008-10-07 21:58:19.125
*** MODULE NAME:(Trace Demo) 2008-10-07 21:58:19.125
*** SERVICE NAME:(DB1.appsdba.com) 2008-10-07 21:58:19.125
*** SESSION ID:(156.12) 2008-10-07 21:58:19.125
=====
PARSING IN CURSOR #1 len=68 dep=0 uid=5 oct=47 lid=5 tim=435038286200 hv=1690018796 ad='20dbb880'
BEGIN DBMS_MONITOR.SESSION_TRACE_ENABLE(NULL,NULL,TRUE,TRUE); END;
END OF STMT
EXEC #1:c=0,e=2199,p=0,cr=50,cu=0,mis=1,r=1,dep=0,og=1,tim=435038286195
WAIT #1: nam='SQL*Net message to client' ela= 5 driver id=1413697536 #bytes=1 p3=0 obj#=-1
tim=435038381272
```

```

*** 2008-10-07 21:58:30.531
WAIT #1: nam='SQL*Net message from client' ela= 11313245 driver id=1413697536 #bytes=1 p3=0 obj#=-1
tim=435049699866
=====

```

< Lots of recursive SQL tracing deleted >

```

=====
PARSING IN CURSOR #2 len=35 dep=0 uid=5 oct=3 lid=5 tim=435051264781 hv=1415804991 ad='20da6610'
select global_name from global_name
END OF STMT
PARSE #2:c=187500,e=1552169,p=21,cr=402,cu=0,mis=1,r=0,dep=0,og=1,tim=435051264776
BINDS #2:
EXEC #2:c=0,e=72,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,tim=435051265014
WAIT #2: nam='SQL*Net message to client' ela= 4 driver id=1413697536 #bytes=1 p3=0 obj#=-1
tim=435051265072
FETCH #2:c=0,e=74,p=0,cr=3,cu=0,mis=0,r=1,dep=0,og=1,tim=435051265198
WAIT #2: nam='SQL*Net message from client' ela= 3795 driver id=1413697536 #bytes=1 p3=0 obj#=-1
tim=435051269066
FETCH #2:c=0,e=25,p=0,cr=1,cu=0,mis=0,r=0,dep=0,og=1,tim=435051269171
WAIT #2: nam='SQL*Net message to client' ela= 2 driver id=1413697536 #bytes=1 p3=0 obj#=-1
tim=435051269254

```

The V\$SESSION view shows summarized session information. The following examples include rates of work and wait event summarization. The following is an example using the V\$SESSION view:

User Name	SID	Service Name	Module	Status	Total CPU Time (Sec)	Logical Reads	Physical Reads	Redo Size (KBytes)	Sorts (Disk)
SYSTEM	156	SYSS\$USERS	sqlplus	ACTIVE	0.17	607	9	0	0

And the V\$SESSION_WAIT view:

Time Waited

SID	Wait Event	Total Waits	Time Waited	Avg Wait	Max Wait	Micro	Event ID
156	SQL*Net message from client	24	12,195	508	11,166	121,953,226	1421975091
156	db file sequential read	14	29	2	5	288,212	2652584166
156	SQL*Net more data from client	2	0	0	0	19	3530226808
156	SQL*Net message to client	25	0	0	0	87	2067390145

Session Aggregates

For groups of sessions, either based on service, module/action, or classified by some other attribute there are several options available. Oracle introduced Active Session History (ASH) in Release 10g along with the Automatic Workload Repository (AWR). ASH was a big improvement because it became possible to catch sessions that might have otherwise been missed using the V\$SESSION view. Once a session disconnects it is no longer available through the V\$SESSION based views, but with ASH the session data remains available for some time after the session terminates. The data is sampled at 1 second intervals as well, so the data is more granular than most attempts at sampling the V\$SESSION based views. The following example was taken from an Oracle 10g database for a session that was running an RMAN archivelog file backup.

```

select
  ash.sample_id,
  ash.sample_time,
  ash.session_id,
  ash.user_id,
  (select name from dba_services
   where name_hash = ash.service_hash) sname,
  ash.module,
  ash.program,
  ash.session_state state
from
  v$active_session_history ash
where
  session_id = 181
  and user_id = 0
order by
  ash.sample_id,
  session_id,
  user_id
/

```

SAMPLE_ID	SAMPLE_TIME	SID	UID	Name	MODULE	PROGRAM	STATE
64223806	16-OCT-08 04.01.29.256 AM	181	0	BACKUP	rman	rman	WAITING
64223809	16-OCT-08 04.01.32.256 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223811	16-OCT-08 04.01.34.266 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223812	16-OCT-08 04.01.35.266 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223813	16-OCT-08 04.01.36.266 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223814	16-OCT-08 04.01.37.266 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223815	16-OCT-08 04.01.38.266 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223816	16-OCT-08 04.01.39.266 AM	181	0	BACKUP	backup archivelog	rman	ON CPU
64223817	16-OCT-08 04.01.40.266 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223818	16-OCT-08 04.01.41.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223819	16-OCT-08 04.01.42.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223820	16-OCT-08 04.01.43.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223821	16-OCT-08 04.01.44.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223822	16-OCT-08 04.01.45.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223823	16-OCT-08 04.01.46.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223824	16-OCT-08 04.01.47.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223825	16-OCT-08 04.01.48.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223826	16-OCT-08 04.01.49.276 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223827	16-OCT-08 04.01.50.286 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223828	16-OCT-08 04.01.51.286 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223829	16-OCT-08 04.01.52.286 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223830	16-OCT-08 04.01.53.286 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223894	16-OCT-08 04.02.57.376 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223895	16-OCT-08 04.02.58.376 AM	181	0	BACKUP	backup archivelog	rman	WAITING
64223896	16-OCT-08 04.02.59.376 AM	181	0	BACKUP	backup archivelog	rman	WAITING

Note that each sample has been taken at approximately a one second interval. It's also interesting to note that a sample isn't necessarily taken at every sample id. Notice that there are gaps in time. This occurs because Oracle only samples "active" sessions, those that are either using CPU or are actively waiting for a resource.

System Level

For the entire database the AWR facility can be used along with the V\$ "system" views like V\$SYSSTAT and V\$SYS_TIME_MODEL. This can provide overall workload and capacity planning information based on resource usage information, and can also be classified by service, module/action or some other attribute just like session data. And, as with the ASH views, one of the advantages of the AWR views over the V\$ "system" views are that the snapshots have already been performed. Of course the disadvantage of using both the ASH and AWR data is that they are separately licensed by Oracle.

Taken from session_time.sql (available on appsdba.com):

User	Service Name	Module	Calculated		DB Wait Time (sec)	DB Elapsed Time (sec)	DB CPU Time (sec)	SQL Time (sec)	Tim
			Elapsed Time (sec)	Total Wait Time (sec)					
DBSNMP	SYSS\$USERS	emagent	98,553.45	98,215.77	9,012.97	9,238.23	337.68	9,235.84	
SYS	SYSS\$USERS	racgimon	837,214.66	837,121.86	686.99	697.91	92.80	696.66	
SYS	SYSS\$USERS	racgimon	841,967.74	841,882.82	15.24	89.06	84.92	4.13	
SYS	SYSS\$USERS	racgimon	842,758.86	842,714.32	2.90	45.41	44.54	1.09	
APP_DATA	LOAD1	preLoader	163,212.50	163,201.21	2.42	13.24	11.29	11.47	
APP_DATA	LOAD1	preLoader	163,210.88	163,199.85	2.07	12.47	11.03	10.86	
APP_DATA	LOAD1	preLoader	163,224.80	163,213.60	2.01	13.00	11.20	11.18	
APP_DATA	LOAD1	preLoader	162,584.85	162,567.21	1.89	18.53	17.64	15.90	
APP_DATA	LOAD1	preLoader	162,959.02	162,947.27	1.57	13.17	11.75	11.26	
APP_DATA	LOAD1	preLoader	163,180.41	163,169.89	1.01	11.40	10.52	9.88	
APP_DATA	LOAD1	preLoader	162,801.48	162,784.62	0.88	17.53	16.86	14.88	
APP_DATA	LOAD1	preLoader	162,657.78	162,642.20	0.87	16.26	15.58	13.87	
APP_DATA	LOAD1	preLoader	163,195.65	163,181.12	0.85	15.18	14.53	12.90	
APP_DATA	LOAD1	preLoader	163,235.99	163,224.57	0.85	12.15	11.42	10.26	
APP_DATA	LOAD1	preLoader	162,997.16	162,981.68	0.63	15.93	15.48	13.56	
SYS	SYSS\$USERS	sqlplus	1,151.81	1,150.12	0.03	1.73	1.69	1.68	
SYS	SYSS\$USERS	racgimon	610,970.99	610,970.09	0.02	0.96	0.90	0.84	
SYS	SYSS\$USERS	racgimon	615,061.94	615,061.09	0.01	0.89	0.85	0.83	
SYS	SYSS\$USERS	sqlplus	7.40	7.39	0.00	0.01	0.01	0.00	
SYS	SYSS\$USERS	racgimon	609,223.03	609,222.24	0.00	0.80	0.79	0.74	
SYS	SYSS\$USERS	racgimon	609,773.05	609,772.27	0.00	1.16	0.78	1.10	

There are other facilities plus what has been integrated into the Oracle Enterprise Manager products (e.g. dbconsole and Grid Control). For instance, service/module/action based workload classification can be accomplished using the DBMS_MONITOR package. The following will show an example of using a service and module statistic collection using the following command:

```
EXECUTE DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE('LOAD', 'Loader');
```

The results are available in the V\$SERV_MOD_ACT_STATS view with a query like the following:

```
select
  aggregation_type,
  service_name,
  module,
  action,
  stat_name,
  value
from
  v$serv_mod_act_stats
where
  value > 0
order by
  aggregation_type,
  service_name,
  module,
  action,
  stat_name
```

```

/

```

AGGREGATION_TYPE	SERVICE_NAME	MODULE	ACTION	STAT_NAME	VALUE
SERVICE_MODULE	LOAD	Loader		DB CPU	491207
SERVICE_MODULE	LOAD	Loader		DB time	502795
SERVICE_MODULE	LOAD	Loader		cluster wait time	6973
SERVICE_MODULE	LOAD	Loader		concurrency wait time	11607
SERVICE_MODULE	LOAD	Loader		execute count	14320
SERVICE_MODULE	LOAD	Loader		gc cr blocks received	8
SERVICE_MODULE	LOAD	Loader		gc current blocks received	2
SERVICE_MODULE	LOAD	Loader		opened cursors cumulative	6
SERVICE_MODULE	LOAD	Loader		parse count (total)	6
SERVICE_MODULE	LOAD	Loader		parse time elapsed	6216
SERVICE_MODULE	LOAD	Loader		session logical reads	126
SERVICE_MODULE	LOAD	Loader		sql execute elapsed time	379150
SERVICE_MODULE	LOAD	Loader		user calls	9709

V\$SESSION_LONGOPS

The V\$SESSION_LONGOPS view provides a method of monitoring long running database processes. Many Oracle tasks are written to output status to the V\$SESSION_LONGOPS view. Recovery Manager backup and restore jobs, Data Pump jobs, materialized view refreshes to name a few. The DBMS_APPLICATION_INFO.set_session_longops procedure is provided by Oracle to allow any application program to use the V\$SESSION_LONGOPS view to output the progress of a task. The following example shows a simple loop that outputs it's progress to the V\$SESSION_LONGOPS view:

```

SQL> DECLARE
  2  --
  3  -- Session Longops variables
  4  --
  5  rindex      BINARY_INTEGER;
  6  slno       BINARY_INTEGER;
  7  --
  8  var_proc_cnt NUMBER := 0; -- Records processed
  9  BEGIN
 10  rindex := DBMS_APPLICATION_INFO.SET_SESSION_LONGOPS_NOHINT;
 11  --
 12  FOR j IN 1 .. 10 LOOP
 13    FOR i IN 1 .. 100 LOOP
 14      var_proc_cnt := var_proc_cnt + 1;
 15    END LOOP;
 16    --
 17    DBMS_APPLICATION_INFO.SET_SESSION_LONGOPS(
 18      rindex => rindex,
 19      slno => slno,
 20      op_name => 'Bulk insert',
 21      sofar => var_proc_cnt,
 22      target_desc => 'TABLE_NAME',
 23      units => 'ROWS' );
 24    --
 25    DBMS_LOCK.SLEEP(2); -- Sleep for 2 seconds just to slow down
 26  END LOOP;
 27 END;
 28 /

```

PL/SQL procedure successfully completed.

SQL>

The following shows a partial listing of the output taken from querying the V\$SESSION_LONGOPS view while running the sample code above:

SQL> /

USERNAME	SID	SERIAL#	INST_ID	OPNAME	TARGET_DESC	SOFAR	Total Work	UNITS	Elapsed Seconds
UTILITY	170	12899	1	Bulk insert	TABLE_NAME	200	0	ROWS	2

SQL> /

USERNAME	SID	SERIAL#	INST_ID	OPNAME	TARGET_DESC	SOFAR	Total Work	UNITS	Elapsed Seconds
UTILITY	170	12899	1	Bulk insert	TABLE_NAME	300	0	ROWS	4

SQL> /

USERNAME	SID	SERIAL#	INST_ID	OPNAME	TARGET_DESC	SOFAR	Total Work	UNITS	Elapsed Seconds
UTILITY	170	12899	1	Bulk insert	TABLE_NAME	400	0	ROWS	6

Wrap Up

To wrap up we've covered four categories of instrumenting Oracle applications: debugging, logging, runtime registration and metric collection. Each of the categories provides a key part of an overall instrumentation framework. I've tried to show some examples for each category and point out that in many cases there are tools, both open source and commercial, that can be used to facilitate the use of the techniques shown. I've also tried to provide some guidelines based on personal experience. Hopefully this information will keep evolving and that this information will be helpful to others trying to get a handle on how to instrument their applications.

About The Author

Andy Rivenes is an Oracle DBA living in the San Francisco Bay Area. He has been working with Oracle technology since 1992, and is active in both the OAUG and the IOUG and maintains the AppsDBA.com web site. He has authored many articles, utilities and open source software for Oracle databases and Oracle Applications systems. He can be reached at andy@appsdba.com.

References and Resources

Rivenes, A., 2006, "Oracle Workload Characterization", <http://www.appsdba.com>

Millsap, C., 2005, "How to Make an Application Easy to Diagnose", <http://www.hotsos.com>

Schneider, J., 2007, "Unleashing Oracle Services: A Comprehensive Review of "Services" in Oracle Databases", <http://www.ardentperf.com>

session_time.sql – Can be found at www.appsdba.com

debug.f – Can be found at asktom.oracle.com

log4pql – Can be found on sourceforge.net