Secret ORACLE

Unleashing the Full Potential of the ORACLE DBMS by Leveraging Undocumented Features

Norbert Debes

ORADBPRO Publishing

Chapter 11

X$KSLED and Enhanced Session Wait Data

Status: X$KSLED is an undocumented X$ fixed table. V$SESSION_WAIT is based on X$KSLED and X$KSUSECST. X$KSLED contributes the wait event name to V$SESSION_WAIT whereas X$KSUSECST holds timing information. Neither Oracle9*i* nor Oracle10*g* have a V$ view that provides more than centisecond resolution for a single wait event at session level[1]. There is also no view which integrates information on operating system processes found in V$PROCESS with wait information. It's a bit cumbersome to correctly interpret the columns WAIT_TIME and SECONDS_IN_WAIT of the view V$SESSION_WAIT depending on the value of the column STATE.

Benefit: Direct access to X$ fixed tables makes it possible to get microsecond resolution for wait events without the need to enable SQL trace. Furthermore, an enhanced version of V$SESSION_WAIT, which combines information from V$SESSION, V$SESSION_WAIT and V$PROCESS and is easier to interpret, may be built. Note that V$SESSION in Oracle9*i* does not include information on wait events, whereas wait event information has been incorporated into V$SESSION in Oracle10*g*. The enhanced session wait view presented in this chapter is compatible with both Oracle9*i* and Oracle10*g*.

Drilling Down From V$SESSION_WAIT

By drilling down from V$SESSION_WAIT, as presented in Chapter 9, it becomes apparent that this view is based on X$KSLED and X$KSUSECST. Adding column alias names, which correspond to the column names of GV$SESSION, to the view definition retrieved from V$FIXED_VIEW_DEFINITION, yields the following SELECT statement:

```
SELECT s.inst_id AS inst_id,
s.indx AS sid,
```

1. In Oracle11*g*, the dynamic performance view V$SESSION_WAIT has the following new columns: WAIT_TIME_MICRO, TIME_REMAINING_MICRO, and TIME_SINCE_LAST_WAIT_MICRO.

```
s.ksussseq AS seq#,
e.ksllednam AS event,
e.ksledp1 AS p1text,
s.kssusp1 AS p1,
s.kssusp1r AS p1raw,
e.ksledp2 AS p2text,
s.kssusp2 AS p2,
s.kssusp2r AS p2raw,
e.ksledp3 AS p3text,
s.kssusp3 AS p3,
s.kssusp3r AS p3raw,
decode(s.ksusstim,0,0,-1,-1,-2,-2,   decode(round(s.ksusstim/10000),0,-
1,round(s.ksusstim/10000))) AS wait_time,
s.ksusewtm AS seconds_in_wait,
decode(s.ksusstim, 0, 'WAITING', -2, 'WAITED UNKNOWN TIME',  -1, 'WAITED SHORT TIME',
'WAITED KNOWN TIME') AS state
FROM x$ksusecst s, x$ksled e
WHERE bitand(s.ksspaflg,1)!=0
and bitand(s.ksuseflg,1)!=0
and s.ksussseq!=0
and s.kssussopc=e.indx
```

Note how the microsecond resolution in `X$KSUSECST` is artificially reduced to centisecond resolution through the division by 10000. At reduced resolution, it is impossible to learn how long short wait events such as *db file sequential read*, *db file scattered read*, or global cache related wait events in Real Application Clusters (RAC) were. Wait times shorter than 1 centisecond are displayed as -1 by `V$SESSION_WAIT`. At this resolution, it is impossible to see disk access times at session level. Peaks in I/O service time also remain unnoticed, as long as the duration of the wait events stays below 1 centisecond, which it normally will. Below is an example:

```
SQL> SELECT event, wait_time, seconds_in_wait, state
FROM v$session_wait
WHERE (state='WAITED KNOWN TIME' or state='WAITED SHORT TIME')
AND event !='null event';
EVENT                          WAIT_TIME SECONDS_IN_WAIT STATE
------------------------------ --------- -------------- ----------------
db file sequential read             -1               0 WAITED KNOWN TIME
SQL*Net message from client         -1               0 WAITED KNOWN TIME
SQL*Net message to client           -1               0 WAITED KNOWN TIME
```

An Improved View

Now that the restrictions of `V$SESSION_WAIT` have become apparent, we may set goals for an improved view. The goals are to:

- provide wait event duration at microsecond resolution
- integrate process, session and session wait information
- present the wait status and wait time in a readily accessible format without requiring further decoding by users of the view

Information on processes and sessions is available from the X$ tables underlying `V$PROCESS` and `V$SESSION`. These are `X$KSUSE` and `X$KSUPR` respectively. It requires some perseverance to construct the largish view, which meets the goals set above. Below is the DDL to create the view, which I have called `X_$SESSION_WAIT` (script file name `x_session_wait.sql`):

```
CREATE OR REPLACE view x_$session_wait AS
SELECT s.inst_id AS inst_id,
s.indx AS sid,
se.ksuseser AS serial#,
-- spid from v$process
p.ksuprpid AS spid,
-- columns from v$session
```

Chapter 19

DBMS_SCHEDULER

Status: The database scheduler is an advanced job scheduling capability built into Oracle10*g* and subsequent releases. The package `DBMS_SCHEDULER` is the interface to the job scheduler. It is extensively documented in the *Oracle Database Administrator's Guide* and the *PL/SQL Packages and Types Reference*. Important aspects concerning the execution of external jobs, such as exit code handling, removal of environment variables, details of program argument passing, requirements to run external programs owned by users other than SYS, and default privileges of external jobs defined by the configuration file `externaljob.ora` are undocumented.

Benefit: This chapter provides all the details on the subject of external jobs: how they are run, which environment variables and privileges are available to them, how to signal success and failure, and how to integrate custom logging with the scheduler's own logging.

Running External Jobs with the Database Scheduler

The database scheduler is an advanced job scheduling capability, which ships with Oracle10*g* and subsequent releases. The PL/SQL package `DBMS_SCHEDULER` is the interface to a rich set of job scheduling features. Oracle10*g* Release 1 was the first ORACLE DBMS release, which had the capability to run jobs outside of the database. The scheduler supports three types of jobs:

- stored procedures
- PL/SQL blocks
- executables, i.e. external programs, which run outside of the database engine

Job chains are another new feature introduced with Oracle10*g*. Chains consist of several jobs. Rules are used to decide which job within a chain to execute next. Since the scheduler supports jobs that run within as well as outside of the database engine, it makes sense to use it for controlling complex processing that involves job steps at operating sys-

Chapter 24

Extended SQL Trace File Format Reference

Status: The SQL trace file format is undocumented, even though the utilities TKPROF and TRCSESS are based on the analysis of SQL trace files.

Benefit: Understanding the format of extended SQL trace files is an essential skill for any DBA who is confronted with performance problems or troubleshooting tasks. Since formatting trace files with TKPROF obscures important information, such as statement hash values, timestamps, dependency levels, and SQL identifiers (emitted by Oracle11*g*), it is often mandatory to read and understand the trace files themselves.

Introduction to Extended SQL Trace Files

Extended SQL trace files are by and large a statement by statement account of SQL and PL/SQL executed by a database client[1]. Entries found in such files fall into four major categories:

- database calls (parse, execute, and fetch)
- wait events
- bind variable values
- miscellaneous (timestamps, session, module, action, and client identification)

Database calls, session identification, and other details from category miscellaneous are logged when tracing is enabled at the lowest level 1, e.g. with `ALTER SESSION SET SQL_TRACE=TRUE`, whereas recording of wait events and

1. Background processes may be traced too, but they are normally not responsible for performance problems.

bind variable values may be enabled independently. How to obtain trace files at various levels of detail is the topic of Chapter 28. The trace levels and the type of trace file entries they enable are summarized in Table 42.

Table 42: SQL Trace Levels

SQL Trace Level	*Database Calls*	*Bind Variable Values*	*Wait Events*
1	yes	no	no
4	yes	yes	no
8	yes	no	yes
12	yes	yes	yes

Sometimes extended SQL trace files are referred to as raw SQL trace files. Both terms are indeed synonymous. Since there is nothing particularly raw about the files—they are perfectly human readable—I have decided not to use the adjective raw and will stick with the term extended SQL trace file or simply trace file for the sake of conciseness.

SQL and PL/SQL Statements

The term cursor is often used in conjunction with `SELECT` statements and the iterative fetching of rows returned by queries. However, the ORACLE DBMS uses cursors to execute any SQL or PL/SQL statement, not just `SELECT` statements. SQL and PL/SQL statements in a trace file are identified by their cursor number. Cursor numbers for SQL statements sent by clients start at 1. The cursor number is the figure behind the pound sign (`#`) in entries such as `PARSING IN CURSOR #1`, `PARSE #1`, `EXEC #1`, `FETCH #1`, `WAIT #1`, and `STAT #1`. These examples all refer to the same cursor number 1. Each additional SQL statement run by the client receives another cursor number, unless reuse of a cursor number is taking place after the cursor has been closed. Entries relating to the same statement are interrelated through the cursor number.

Not all operations executed are assigned a proper cursor number. One notable exception is the use of large objects (LOBs) through Oracle Call Interface (OCI). When working with LOBs, you may see cursor number 0 or cursor numbers for which a parse call is missing, although tracing was switched on right after connecting. This does not apply to the PL/SQL LOB interface `DBMS_LOB`.

Cursor numbers may be reused within a single database session. When the client closes a cursor, the DBMS writes `STAT` entries, which represent the execution plan, into the trace file. At this stage, the cursor number can be reused for a different SQL statement. The SQL statement text for a certain cursor is printed after the first `PARSING IN CURSOR #`*n* entry above any `EXEC #`*n*, `FETCH #`*n*, `WAIT #`*n*, or `STAT #`*n* entries with the same cursor number *n*.

Dependency Level

Anyone who has worked with the TKPROF utility is presumably familiar with the concept of recursive and internal SQL statements. SQL Statements sent by a database client are executed at dependency level 0. Should a SQL statement fire other statements, such as an `INSERT` statement, which fires the execution of an insert trigger, then these other statements would be executed at dependency level 1. A trigger body may then execute additional statements, which may cause recursive SQL at the next higher dependency level. Below is an example of an `INSERT` statement executed at dependency level 0. The `INSERT` statement fires a trigger. Access to a sequence in the trigger body is at dependency level 1. Note how the execution of the top level `INSERT` statement (`EXEC #3`) is written to the trace file after the execution of the dependent `SELECT` from the sequence has completed (`EXEC #2`):

```
PARSING IN CURSOR #3 len=78 dep=0 uid=61 oct=2 lid=61 tim=771237502562 hv=3259110965
ad='6c5f86dc'
INSERT INTO customer(name, phone) VALUES (:name, :phone) RETURNING id INTO :id
END OF STMT
PARSE #3:c=0,e=1314,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=771237502553
=====================
PARSING IN CURSOR #2 len=40 dep=1 uid=61 oct=3 lid=61 tim=771237506650 hv=1168215557
ad='6c686178'
SELECT CUSTOMER_ID_SEQ.NEXTVAL FROM DUAL
```

```
END OF STMT
PARSE #2:c=0,e=1610,p=0,cr=0,cu=0,mis=1,r=0,dep=1,og=1,tim=771237506643
EXEC #2:c=0,e=57,p=0,cr=0,cu=0,mis=0,r=0,dep=1,og=1,tim=771237507496
FETCH #2:c=0,e=54,p=0,cr=0,cu=0,mis=0,r=1,dep=1,og=1,tim=771237507740
EXEC #3:c=0,e=4584,p=0,cr=1,cu=3,mis=1,r=1,dep=0,og=1,tim=771237508046
```

When a client runs an anonymous PL/SQL block, the block itself is executed at dependency level 0, but the statements inside the block will have dependency level 1. Another example is auditing entries inserted into table SYS.AUD$. These are executed at one dependency level higher than the statement that triggered the auditing.

Recursive parse, execute, and fetch operations are listed before the execution of the statement that triggered the recursive operations. Statistics for SQL statements executed at dependency level 0 contain the cost of dependent statements in terms of CPU time, elapsed time, consistent reads, etc. This must be taken into consideration to avoid double counting when evaluating SQL trace files.

Database Calls

The database call category consists of the three subcategories parsing, execution, and fetching. Note that these entries correspond with the three stages of running dynamic SQL with the package DBMS_SQL by calling the package subroutines DBMS_SQL.PARSE, DBMS_SQL.EXECUTE, and DBMS_SQL.FETCH_ROWS.

Among other metrics, database call entries represent the CPU and wall clock time (elapsed time) a server process spends inside the ORACLE kernel on behalf of a database client. The aggregated CPU and wall clock times from database calls in a trace file are closely related to the session level statistics DB CPU and DB time in the dynamic performance view V$SESS_TIME_MODEL, which is available in Oracle10*g* and subsequent releases.

Parsing

Parsing involves syntactic and semantic analysis of SQL statements as well as determining a well suited execution plan. Since this may be a costly operation, the ORACLE DBMS has the capacity to cache the results of parse calls in the so called library cache within the System Global Area (SGA) for reuse by other statements that use the same SQL statement text.

The use of bind variables in SQL statements is crucial for the reuse of cached statements. Failure to use bind variables causes increased parse CPU consumption, contention for the library cache, excessive communication round–trips between client and server due to repeated parse calls of non–reusable statements with literals, and difficulties in diagnosing performance problems due to the inability of the TKPROF utility to aggregate statements, which are identical apart from literals. I recommend reading the section *Top Ten Mistakes Found in Oracle Systems* on page 3–4 of *Oracle Database Performance Tuning Guide 10g Release 2* before beginning design and coding of an application. While bind variables are mandatory to achieve scalability in high volume transaction processing (OLTP), literals are usually preferred in data warehousing applications to provide the CBO with as much information as possible and to avoid the unpredictability inherent in bind variable peeking. The CBO looks at bind variable values when it first encounters a statement, but not on subsequent executions of the same statement, such that the plan chosen may be optimal for the initial execution, but inappropriate for subsequent executions. This functionality is called bind variable peeking. It is enabled by default with the hidden parameter setting _OPTIM_PEEK_USER_BINDS=TRUE.

Parsing is usually represented by two adjacent entries in the trace file. The first is PARSING IN CURSOR, and the second is PARSE. The minimum SQL trace level for enabling parse related entries is 1. Here's an example of a PARSING IN CURSOR followed by a PARSE from an Oracle10*g* trace file:

```
PARSING IN CURSOR #3 len=92 dep=0 uid=30 oct=2 lid=30 tim=81592095533 hv=1369934057
ad='66efcb10'
INSERT INTO poem (author, text) VALUES(:author, empty_clob())
   RETURNING text INTO :lob_loc
END OF STMT
PARSE #3:c=0,e=412,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,tim=81592095522
```

Parameters associated with the PARSING IN CURSOR entry are explained in Table 43.

PARSING IN CURSOR Entry Format

Caching of SQL statements in the shared pool is based on a hash value that is derived from the SQL or PL/SQL statement text. Changes in optimizer settings have no effect on the hash value, whereas slight changes to the statement text

such as insertion of a blank or tab character do. Rarely, two statements with different statements texts may have the same hash value.

The hash value may be retrieved from many V$ views such as VSQL, VSQLTEXT, V$SQLAREA, V$OPEN_CURSOR, and V$SESSION. It remains constant across instance startups, but might change after an upgrade to a new release. In fact, the algorithm for computing hash values has changed in Oracle10*g*. The hash value compatible with previous releases is available in the column OLD_HASH_VALUE of the views V$SQL and V$SQLAREA. Merely the hash value is emitted to trace files. Since Statspack stuck with the "old school" hash value but merely the new hash value is emitted to trace files, this adds the complexity of translating from the new hash value to the old hash value when searching a Statspack repository for information pertinent to statements in a trace file (more on this in Chapter 25).

Oracle10*g* introduced the new column SQL_ID to some of the aforementioned V$ views. The value of this new column is not written to SQL trace files in releases prior to Oracle11*g*, but is used in Active Workload Repository reports, such that translation from the new hash value (column HASH_VALUE) to the SQL_ID may be required when looking up information on a statement in AWR. For cached SQL statements, translation among SQL_ID, HASH_VALUE, and OLD_HASH_VALUE may be performed using V$SQL. For statements that are no longer cached, but were captured by a Statspack snapshot, STATS$SQL_SUMMARY serves as a translation table (the Rosetta stone of SQL statement identifiers). AWR has no facility for translating the hash value found in trace files to the corresponding SQL_ID. In releases prior to Oracle11*g*, matching the statement text between both types of capture is the only time consuming approach for extracting historical information on a statement, such as past execution time and plan, from AWR (see Chapter 26). Considering that Statspack requires no extra license and includes session level capture and reporting (watch out for bug 5145816; see Table 60 on page 273), this shortcoming of AWR might be another reason for favoring Statspack.

Oracle11*g* is the first DBMS release that emits the SQL identifier (V$SQL.SQL_ID) in addition to the hash value to trace files. Hence the statement matching issue between extended SQL trace and AWR is a thing of the past for users of Oracle11*g*. Below is an example of an Oracle11*g* PARSING IN CURSOR entry:

```
PARSING IN CURSOR #3 len=116 dep=0 uid=32 oct=2 lid=32 tim=15545747608 hv=1256130531
ad='6ab5ff8c' sqlid='b85s0yd5dy1z3'
INSERT INTO customer(id, name, phone) VALUES (customer_id_seq.nextval, :name, :phone)
RETURNING id INTO :id
END OF STMT
```

Table 43: PARSING IN CURSOR Parameters

Parameter	*Meaning*
len	Length of the SQL statement text in bytes
dep	Dependency level
uid	Parsing user identity; corresponds to ALL_USERS.USER_ID and V$SQL.PARSING_USER_ID
oct	ORACLE command type; corresponds to V$SQL.COMMAND_TYPE and V$SESSION.COMMAND
lid	Parsing schema identity; corresponds to ALL_USERS.USER_ID and V$SQL.PARSING_SCHEMA_ID; may differ from uid (see Chapter 14 on ALTER SESSION SET CURRENT_SCHEMA)
tim	Timestamp in microseconds; often slightly earlier than the value of tim in the associated PARSE entry
hv	Hash value; corresponds to V$SQL.HASH_VALUE
ad	Address; corresponds to V$SQL.ADDRESS
sqlid	SQL identifier; corresponds to V$SQL.SQL_ID (emitted by Oracle11*g*)

The SQL statement parsed is printed on a new line after the line starting with the string PARSING IN CURSOR. A line starting with END OF STMT marks the end of the SQL statement. The mapping from the numeric command type (parameter oct) to the command name is available by running SELECT action, name FROM audit_actions. Table

44 contains the most common command types plus some additional command types that may be used by applications. Please note that these numeric command types do not correspond with Oracle Call Interface SQL command codes (*Oracle Call Interface Programmer's Guide 10g Release 2*, *Appendix A*).

Table 44: SQL and PL/SQL Command Types

Numeric Command Type	*SQL or PL/SQL Command*
2	`INSERT`
3	`SELECT`
6	`UPDATE`
7	`DELETE`
26	`LOCK TABLE`
44	`COMMIT`
45	`ROLLBACK`
46	`SAVEPOINT`
47	PL/SQL block
48	`SET TRANSACTION`
55	`SET ROLE`
90	`SET CONSTRAINTS`
170	`CALL`
189	`MERGE`

PARSE Entry Format

Among other metrics, `PARSE` entries represent CPU and wall clock time consumed by parse operations. By looking at the parameter `mis` (library cache miss) it is possible to derive the library cache hit ratio from trace files. A low hit ratio usually indicates that the application does not use bind variables. Details on the parameters of the `PARSE` entry are in Table 45.

Table 45: PARSE Parameters

Parameter	*Meaning*
`c`	CPU consumption
`e`	Elapsed time
`p`	Physical reads
`cr`	Consistent reads
`cu`	Current blocks processed
`mis`	Cursor misses, 0=soft parse, i.e. statement found in library cache, 1=hard parse, i.e. statement not found
`r`	Rows processed
`dep`	Dependency level
`og`	Optimizer goal, `1=ALL_ROWS`, `2=FIRST_ROWS`, `3=RULE`, `4=CHOOSE`; Oracle9*i* default is `CHOOSE`; Oracle10*g* and Oracle11*g* default is `ALL_ROWS`
`tim`	Timestamp in microseconds

Chapter 27

ESQLTRCPROF Extended SQL Trace Profiler

Status: To the best of my knowledge ESQLTRCPROF is currently the only free profiler for Oracle9*i*, Oracle10*g*, and Oracle11*g* extended SQL trace files.

Benefit: ESQLTRCPROF is capable of parsing the undocumented extended SQL trace file format. It calculates a resource profile for an entire SQL trace file and for each cursor in the trace file. Essentially, it is a replacement for TKPROF, since it addresses several shortcomings of Oracle Corporation's own SQL trace analysis tool.

Categorizing Wait Events

The ultimate goal is to automatically create a resource profile from an extended SQL trace file. Even though the file format has been discussed in Chapter 24, some more preparations are necessary to achieve this goal. An issue that has not yet been discussed is the categorization of wait events into intra database call wait events and inter database call wait events. This is necessary for correct response time accounting. Intra database call wait events occur within the context of a database call. The code path executed to complete a database call consists not only of CPU consumption, but may also engender waiting for resources such as disks, latches, or enqueues. Time spent waiting within a database call is accounted for by intra database call wait events. Examples of such wait events are *latch free, enqueue*, *db file sequential read*, *db file scattered read*, and *buffer busy waits*. In fact, most wait events are intra database call wait events. Inter database call wait events occur when the DBMS server is waiting to receive the next database call. In other words, the DBMS server is idle, since the client does not send a request.

According to Millsap and Holt ([MiHo 2003], page 88), the wait events below are inter (or between) database call wait events:

- *SQL*Net message from client*
- *SQL*Net message to client*

Chapter 28

The MERITS Performance Optimization Method

Status: The MERITS performance optimization method is built around a sophisticated assessment of extended SQL trace files. The extended SQL trace profiler ESQLTRCPROF, which is capable of parsing the undocumented trace file format, is used in the assessment phase of the method. The MERITS method uses undocumented features predominantly in the assessment, reproduction, and extrapolation phases.

Benefit: The MERITS method is a framework for solving performance problems. The goal of the method is to identify the root cause of slow response time and to subsequently modify parameters, database objects, or application code until the performance goal is met.

Introduction to the MERITS Method

The MERITS performance optimization method is an approach to performance optimization that consists of six phases and relies on undocumented features in several phases. MERITS is a designed acronym derived from the six phases of the method, which are:

1. **M**easurement
2. Ass**e**ssment
3. **R**eproduction
4. **I**mprovement
5. Ex**t**rapolation
6. In**s**tallation

The first step in any performance optimization project should consist of measuring the application or code path that is too slow (phase 1). Measurement data are assessed in the second phase. In some cases this assessment may already reveal the cause of the performance problem. More intricate cases need to be reproduced by a test case, potentially on a test system (phase 3). If a SQL statement takes excessive time to execute, then the test case consists of reproducing

the response time of the SQL statement. The fourth phase is concerned with improving the response time of the application, code path, or test case. This may involve creating a new index, changing optimizer parameters, changing the SQL statement, changing database objects with DDL, introducing previously unused features (e.g. Partitioning option, stored outlines, SQL profiles), etc. This is phase 4. Effects of the improvement are measured in the same way as the original code path. Comparing the measurement data of the original code path with the measurement data of the improvements achieved in phase 4 may be used to extrapolate the magnitude of the performance improvement (phase 5). In other words, it is possible to forecast the effect of an improvement in a test case on the code path that was measured in phase 1. If the improvement is deemed sufficient, the necessary changes need to be approved and installed on the target (production) system at some point. Discussing each phase of the MERISTS method in full detail is a subject for a separate book. However, I provide enough information on each phase to allow the reader to use the method as a framework for performance optimization tasks.

Measurement

Since extended SQL trace is the most complete account of where a database session spent its time and a resource profile may be compiled from extended SQL trace data, this data source is at the core of the measurements taken. However, an extended SQL trace file does not provide a complete picture of an application, system, or DBMS instance. Some aspects which are not covered by an extended SQL trace file are:

- load at the operating system level (I/O bottlenecks, paging, network congestion, waiting for CPU)
- ORACLE DBMS parameters
- session statistics (`V$SESSSTAT`)
- contending database sessions

To capture a complete picture of the system, I recommend using tools such as `sar`, `iostat`, `vmstat`, and `top` to record activity at the operating system level. Concerning the DBMS, I advocate taking a Statspack or AWR snapshot, which spans the same interval as the extended SQL trace file. The Statspack snapshot should include the traced session (`STATSPACK.SNAP` parameter `i_session_id`). If AWR is preferred, an active session history report may be used to get additional information on the session. It may be necessary to take several measurements and to compute an average to compensate for fluctuations in response time. Both AWR and Statspack reports contain a list of all initialization parameters with non–default values. An ASH report contains a section on contending sessions entitled “Top Blocking Sessions”.

Measurement Tools

This section presents two SQL scripts, which may serve as measurement tools at session and instance level. The script `awr_capture.sql` is based on AWR and ASH, while `sp_capture.sql` is based on Statspack. Both scripts require `SYSDBA` privileges. The scripts do not invoke any operating system tools to collect operating system statistics. Yet, an Oracle10*g* Statspack report includes CPU and memory statistics at operating system level in the “Host CPU” and “Memory Statistics” sections and an AWR report includes a section titled “Operating System Statistics”.

Both AWR and ASH are included in the extra–cost Diagnostics Pack. The downside of ASH is that it does not provide a resource profile, since it is built with sampling. A session–level Statspack report does include a rudimentary resource profile, although the report does not use the term resource profile. Session level data are based on `V$SESSION_EVENT` and `V$SESSTAT`, which afford the calculation of a resource profile. Interesting sections from an ASH report, which a session–level Statspack as well as a TKPROF report lack, are “Top Service/Module”, “Top SQL using literals”, “Top Blocking Sessions”, and “Top Objects/Files/Latches”.

Extended SQL Trace, AWR, and ASH

The script `awr_capture.sql` temporarily sets the hidden parameter `_ASH_SAMPLE_ALL=TRUE` to cause ASH to sample idle wait events for improved diagnostic expressiveness. Then the script takes an AWR snapshot and enables level 12 SQL trace for the session which exhibits a performance problem. Next, the script asks the user for how long it should trace the session. There are two ways of using the script:

- Tracing the session for a predetermined interval, such as 300 or 600 seconds. This is achieved by entering the desired length of the interval. The script calls `DBMS_LOCK.SLEEP` to pause for the specified number of seconds, takes another AWR snapshot, and disables SQL trace.