23

Performing Application Tracing

This chapter explains what end-to-end application tracing is, and how to generate and read trace files.

See Also:

*SQL*Plus User's Guide and Reference* to learn about the use of Autotrace to trace and tune SQL*Plus statements

Overview of End-to-End Application Tracing

End-to-end application tracing can identify the source of an excessive database workload, such as a high load SQL statement, by client identifier, service, module, action, session, instance, or an entire database.

In multitier environments, the middle tier routes a request from an end client to different database sessions, making it difficult to track a client across database sessions. End-to-end application tracing is an infrastructure that uses a client ID to uniquely trace a specific end-client through all tiers to the database and provides information about the operation that an end client is performing in the database.

Purpose of End-to-End Application Tracing

End-to-end application tracing simplifies diagnosing performance problems in multitier environments.

For example, you can identify the source of an excessive database workload, such as a high-load SQL statement, and contact the user responsible. Also, a user having problems can contact you. You can then identify what this user session is doing at the PDB level

End-to-end application tracing also simplifies management of application workloads by tracking specific modules and actions in a service. The module and action names are set by the application developer. For example, you would use the `SET_MODULE` and `SET_ACTION` procedures in the `DBMS_APPLICATION_INFO` package to set these values in a PL/SQL program.

End-to-end application tracing can identify workload problems in a database for:

- Client identifier

  Specifies an end user based on the logon ID, such as `HR.HR`
- Service

  Specifies a group of applications with common attributes, service level thresholds, and priorities; or a single application, such as `ACCTG` for an accounting application
- Module

  Specifies a functional block, such as Accounts Receivable or General Ledger, of an application

- Action

  Specifies an action, such as an `INSERT` or `UPDATE` operation, in a module
- Session

  Specifies a session based on a given database session identifier (SID), on the local instance
- Instance

  Specifies a given database instance based on the instance name
- Container

  Specifies the container

End-to-End Application Tracing for PDBs

`V$` views enable read access to trace files that are specific to a container.

The primary use cases are as follows:

- CDB administrators must view trace files from a specific PDB.

  The `V$DIAG_TRACE_FILE` view lists all trace files in the ADR trace directory that contain trace data from a specific PDB. `V$DIAG_TRACE_FILE_CONTENTS` displays the contents of the trace files.
- PDB administrators must view trace files from a specific PDB.

  You can use SQL Trace to collect diagnostic data for the SQL statements executing in a PDB application. The trace data includes SQL tracing (event 10046) and optimizer tracing (event 10053). Using `V$` views, developers can access only SQL or optimizer trace records without accessing the entire trace file.

To enable users and tools to determine which PDB is associated with a file or a part of a file, PDB annotations exist in trace files, incident dumps, and log files. The PDB information is part of the structured metadata that is stored in the `.trm` file for each trace file. Each record captures the following attributes:

- `CON_ID`, which is the ID of the container associated with the data
- `CON_UID`, which is the unique ID of the container
- `NAME`, which is the name of the container

See Also:

"Views for Application Tracing"

Tools for End-to-End Application Tracing

The SQL Trace facility and TKPROF are two basic performance diagnostic tools that can help you accurately assess the efficiency of the SQL statements an application runs.

For best results, use these tools with `EXPLAIN PLAN` rather than using `EXPLAIN PLAN` alone. After tracing information is written to files, you can consolidate this data with the TRCSESS utility, and then diagnose it with TKPROF or SQL Trace.

The recommended interface for end-to-end application tracing is Oracle Enterprise Manager Cloud Control (Cloud Control). Using Cloud Control, you can view the top consumers for each consumer type, and enable or disable statistics gathering and SQL tracing for specific consumers. If Cloud Control is unavailable, then you can manage this feature using the `DBMS_MONITOR` APIs.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about the `DBMS_MONITOR`, `DBMS_SESSION`, `DBMS_SERVICE`, and `DBMS_APPLICATION_INFO` packages

Overview of the SQL Trace Facility

The SQL Trace facility provides performance information on individual SQL statements.

SQL Trace generates the following statistics for each statement:

- Parse, execute, and fetch counts
- CPU and elapsed times
- Physical reads and logical reads
- Number of rows processed
- Misses on the library cache
- User name under which each parse occurred
- Each commit and rollback
- Wait event data for each SQL statement, and a summary for each trace file

If the cursor for the SQL statement is closed, then SQL Trace also provides row source information that includes:

- Row operations showing the actual execution plan of each SQL statement
- Number of rows, number of consistent reads, number of physical reads, number of physical writes, and time elapsed for each operation on a row

Although you can enable the SQL Trace facility for a session or an instance, Oracle recommends that you use the `DBMS_SESSION` or `DBMS_MONITOR` packages instead. When the SQL Trace facility is enabled for a session or for an instance, performance statistics for all SQL statements executed in a user session or in the instance are placed into trace files. Using the SQL Trace facility can affect performance and may result in increased system overhead, excessive CPU usage, and inadequate disk space.

The TRCSESS command-line utility consolidates tracing information from several trace files based on specific criteria, such as session or client ID.

See Also:

- "TRCSESS"
- "Enabling End-to-End Application Tracing" to learn how to use the `DBMS_SESSION` or `DBMS_MONITOR` packages to enable SQL tracing for a session or an instance

Overview of TKPROF

To format the contents of the trace file and place the output into a readable output file, run the TKPROF program.

TKPROF can also do the following:

- Create a SQL script that stores the statistics in the database
- Determine the execution plans of SQL statements

TKPROF reports each statement executed with the resources it has consumed, the number of times it was called, and the number of rows which it processed. This information enables you to locate those statements that are using the greatest resource. With baselines available, you can assess whether the resources used are reasonable given the work done.

Enabling Statistics Gathering for End-to-End Tracing

To gather the appropriate statistics using PL/SQL, you must enable statistics gathering for client identifier, service, module, or action using procedures in `DBMS_MONITOR`.

The default level is the session-level statistics gathering. Statistics gathering is global for the database and continues after a database instance is restarted.

Enabling Statistics Gathering for a Client ID

The procedure `CLIENT_ID_STAT_ENABLE` enables statistics gathering for a given client ID, whereas the procedure `CLIENT_ID_STAT_DISABLE` disables it.

You can view client identifiers in the `CLIENT_IDENTIFIER` column in `V$SESSION`.

Assumptions

This tutorial assumes that you want to enable and then disable statistics gathering for the client with the ID `oe.oe`.

To enable and disable statistics gathering for a client identifier:

1. Start SQL*Plus, and then connect to the database with the appropriate privileges.
2. Enable statistics gathering for `oe.oe`.

   For example, run the following command:

   ```
   EXECUTE DBMS_MONITOR.CLIENT_ID_STAT_ENABLE(client_id => 'OE.OE');
   ```

3. Disable statistics gathering for `oe.oe`.

   For example, run the following command:

   ```
   EXECUTE DBMS_MONITOR.CLIENT_ID_STAT_DISABLE(client_id => 'OE.OE');
   ```

Enabling Statistics Gathering for Services, Modules, and Actions

The procedure `SERV_MOD_ACT_STAT_ENABLE` enables statistic gathering for a combination of service, module, and action, whereas the procedure `SERV_MOD_ACT_STAT_DISABLE` disables statistic gathering for a combination of service, module, and action.

When you change the module or action using the preceding `DBMS_MONITOR` procedures, the change takes effect when the next user call is executed in the session. For example, if a module is set to `module1` in a session, and if the module is reset to `module2` in a user call in the session, then the module remains `module1` during this user call. The module is changed to `module2` in the next user call in the session.

Assumptions

This tutorial assumes that you want to gather statistics as follows:

- For the `ACCTG` service
- For all actions in the `PAYROLL` module
- For the `INSERT ITEM` action within the `GLEDGER` module

To enable and disable statistics gathering for a service, module, and action:

1. Start SQL*Plus, and then connect to the database with the appropriate privileges.
2. Enable statistics gathering for the desired service, module, and action.

   For example, run the following commands:

```
BEGIN
  DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE(
    service_name => 'ACCTG'
,   module_name  => 'PAYROLL' );
END;

BEGIN
  DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE(
    service_name => 'ACCTG'
,   module_name  => 'GLEDGER'
,   action_name  => 'INSERT ITEM' );
END;
```

3. Disable statistic gathering for the previously specified combination of service, module, and action.

   For example, run the following command:

```
BEGIN
  DBMS_MONITOR.SERV_MOD_ACT_STAT_DISABLE(
    service_name => 'ACCTG'
,   module_name  => 'GLEDGER'
,   action_name  => 'INSERT ITEM' );
END;
```

Enabling End-to-End Application Tracing

To enable tracing for client identifier, service, module, action, session, instance or database, execute the appropriate procedures in the `DBMS_MONITOR` package.

With the criteria that you provide, specific trace information is captured in a set of trace files and combined into a single output trace file. You can enable tracing for specific diagnosis and workload management by the following criteria:

See Also:

Oracle Database Administrator's Guide for information about how to locate trace files

Enabling Tracing for a Client Identifier

To enable tracing globally for the database for a specified client identifier, use the `DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE` procedure.

The `CLIENT_ID_TRACE_DISABLE` procedure disables tracing globally for the database for a given client identifier.

Assumptions

This tutorial assumes the following:

- `OE.OE` is the client identifier for which SQL tracing is to be enabled.
- You want to include wait information in the trace.
- You want to exclude bind information from the trace.

To enable and disable tracing for a client identifier:

1. Start SQL*Plus, and then connect to the database with the appropriate privileges.
2. Enable tracing for the client.

   For example, execute the following program:

   ```
   BEGIN
     DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE(
       client_id => 'OE.OE' ,
       waits     => true    ,
       binds     => false   );
   END;
   ```

3. Disable tracing for the client.

   For example, execute the following command:

   ```
   EXECUTE DBMS_MONITOR.CLIENT_ID_TRACE_DISABLE(client_id => 'OE.OE');
   ```

Enabling Tracing for a Service, Module, and Action

The DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE procedure enables SQL tracing for a specified combination of service name, module, and action globally for a database, unless the procedure specifies a database instance name.

The SERV_MOD_ACT_TRACE_DISABLE procedure disables the trace at all enabled instances for a given combination of service name, module, and action name globally.

Assumptions

This tutorial assumes the following:

- You want to enable tracing for the service ACCTG.
- You want to enable tracing for all actions for the combination of the ACCTG service and the PAYROLL module.
- You want to include wait information in the trace.
- You want to exclude bind information from the trace.
- You want to enable tracing only for the inst1 instance.

To enable and disable tracing for a service, module, and action:

1. Start SQL*Plus, and then connect to the database with the appropriate privileges.
2. Enable tracing for the service, module, and actions.

   For example, execute the following command:

   ```
   BEGIN
     DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE(
       service_name  => 'ACCTG'   ,
       module_name   => 'PAYROLL' ,
       waits         =>  true     ,
       binds         =>  false    ,
       instance_name => 'inst1'   );
   END;
   ```

3. Disable tracing for the service, module, and actions.

   For example, execute the following command:

   ```
   BEGIN
     DBMS_MONITOR.SERV_MOD_ACT_TRACE_DISABLE(
       service_name  => 'ACCTG'   ,
       module_name   => 'PAYROLL' ,
       instance_name => 'inst1'   );
   END;
   ```

Enabling Tracing for a Session

The SESSION_TRACE_ENABLE procedure enables the trace for a given database session identifier (SID) on the local instance.

Whereas the DBMS_MONITOR package can only be invoked by a user with the DBA role, users can also enable SQL tracing for their own session by invoking the DBMS_SESSION.SESSION_TRACE_ENABLE procedure, as in the following example:

```
BEGIN
  DBMS_SESSION.SESSION_TRACE_ENABLE(
    waits => true
  , binds => false);
END;
```

Assumptions

This tutorial assumes the following:

- You want to log in to the database with administrator privileges.
- User OE has one active session.
- You want to temporarily enable tracing for the OE session.
- You want to include wait information in the trace.
- You want to exclude bind information from the trace.

To enable and disable tracing for a session:

1. Start SQL*Plus, and then log in to the database with the administrator privileges.
2. Determine the session ID and serial number values for the session to trace.

   For example, query V$SESSION as follows:

   ```
   SELECT SID, SERIAL#, USERNAME
   FROM   V$SESSION
   WHERE  USERNAME = 'OE';

          SID    SERIAL# USERNAME
   ---------- ---------- ------------------------------
           27         60 OE
   ```

3. Use the values from the preceding step to enable tracing for a specific session.

   For example, execute the following program to enable tracing for the OE session, where the true argument includes wait information in the trace and the false argument excludes bind information from the trace:

   ```
   BEGIN
     DBMS_MONITOR.SESSION_TRACE_ENABLE(
       session_id => 27
     , serial_num => 60
     , waits      => true
   ```

```
  , binds      => false);
END;
```

4. Disable tracing for the session.

   The `SESSION_TRACE_DISABLE` procedure disables the trace for a given database session identifier (SID) and serial number. For example:

```
BEGIN
  DBMS_MONITOR.SESSION_TRACE_DISABLE(
    session_id => 27
  , serial_num => 60);
END;
```

Enabling Tracing for an Instance or Database

The `DBMS_MONITOR.DATABASE_TRACE_ENABLE` procedure overrides all other session-level traces, but is complementary to the client identifier, service, module, and action traces. Tracing is enabled for all current and future sessions.

All new sessions inherit the wait and bind information specified by this procedure until you call the `DATABASE_TRACE_DISABLE` procedure. When you invoke this procedure with the `instance_name` parameter, the procedure resets the session-level SQL trace for the named instance. If you invoke this procedure without the `instance_name` parameter, then the procedure resets the session-level SQL trace for the entire database.

Prerequisites

You must have administrative privileges to execute the `DATABASE_TRACE_ENABLE` procedure.

Assumptions

This tutorial assumes the following:

- You want to enable tracing for all SQL the `inst1` instance.
- You want wait information to be in the trace.
- You do not want bind information in the trace.

To enable and disable tracing for an instance or database:

1. Start SQL*Plus, and then log in to the database with the necessary privileges.
2. Call the `DATABASE_TRACE_ENABLE` procedure to enable SQL tracing for a given instance or an entire database.

   For example, execute the following program, where the `true` argument specifies that wait information is in the trace, and the `false` argument specifies that no bind information is in the trace:

```
BEGIN
  DBMS_MONITOR.DATABASE_TRACE_ENABLE(
    waits         => true
  , binds         => false
  , instance_name => 'inst1' );
END;
```

3. Disable tracing.

   The `DATABASE_TRACE_DISABLE` procedure disables the trace. For example, the following program disables tracing for `inst1`:

   ```
   EXECUTE DBMS_MONITOR.DATABASE_TRACE_DISABLE(instance_name => 'inst1');
   ```

   To disable SQL tracing for an entire database, invoke the `DATABASE_TRACE_DISABLE` procedure without specifying the `instance_name` parameter:

   ```
   EXECUTE DBMS_MONITOR.DATABASE_TRACE_DISABLE();
   ```

Generating Output Files Using SQL Trace and TKPROF

This section explains the basic procedure for using SQL Trace and TKPROF.

The procedure for generating output files is as follows:

1. Set initialization parameters for trace file management.

   See "Step 1: Setting Initialization Parameters for Trace File Management".
2. Enable the SQL Trace facility for the desired session, and run the application. This step produces a trace file containing statistics for the SQL statements issued by the application.

   See "Step 2: Enabling the SQL Trace Facility".
3. Run TKPROF to translate the trace file created in Step 2 into a readable output file. This step can optionally create a SQL script that you can use to store the statistics in a database.

   See "Step 3: Generating Output Files with TKPROF".
4. Optionally, run the SQL script produced in Step 3 to store the statistics in the database.

   See "Step 4: Storing SQL Trace Facility Statistics".

Step 1: Setting Initialization Parameters for Trace File Management

To enable trace files, you must ensure that specific initialization parameters are set.

When the SQL Trace facility is enabled for a session, Oracle Database generates a trace file containing statistics for traced SQL statements for that session. When the SQL Trace facility is enabled for an instance, Oracle Database creates a separate trace file for each process.

To set initialization parameters for trace file management:

1. Check the settings of the `TIMED_STATISTICS`, `MAX_DUMP_FILE_SIZE`, and `DIAGNOSTIC_DEST` initialization parameters, as indicated in "Table 23-1".

   Table 23-1 Initialization Parameters to Check Before Enabling SQL Trace

Parameter	Description
`DIAGNOSTIC_DEST`	Specifies the location of the Automatic Diagnostic Repository (ADR) Home. The diagnostic files for each database instance are located in this dedicated directory.

Table 23-1 (Cont.) Initialization Parameters to Check Before Enabling SQL Trace

Parameter	Description
`MAX_DUMP_FILE_SIZE`	When the SQL Trace facility is enabled at the database instance level, every call to the database writes a text line in a file in the operating system's file format. The maximum size of these files in operating system blocks is limited by this initialization parameter. The default is 32M on Oracle Database Free, and 1G on all other Oracle Database offerings.
`TIMED_STATISTICS`	Enables and disables the collection of timed statistics, such as CPU and elapsed times, by the SQL Trace facility, and the collection of various statistics in the `V$` views. If `STATISTICS_LEVEL` is set to `TYPICAL` or `ALL`, then the default value of `TIMED_STATISTICS` is `true`. If `STATISTICS_LEVEL` is set to `BASIC`, then the default value of `TIMED_STATISTICS` is `false`. Enabling timing causes extra timing calls for low-level operations. This is a dynamic parameter. It is also a session parameter.

2. Devise a way of recognizing the resulting trace file.

   Be sure you know how to distinguish the trace files by name. You can tag trace files by including in your programs a statement such as `SELECT` *`program_name`*`' FROM DUAL`. You can then trace each file back to the process that created it.

   You can also set the `TRACEFILE_IDENTIFIER` initialization parameter to specify a custom identifier that becomes part of the trace file name. For example, you can add *`my_trace_id`* to subsequent trace file names for easy identification with the following:

   ```
   ALTER SESSION SET TRACEFILE_IDENTIFIER = 'my_trace_id';
   ```

3. If the operating system retains multiple versions of files, then ensure that the version limit is high enough to accommodate the number of trace files you expect the SQL Trace facility to generate.
4. If the generated trace files can be owned by an operating system user other than yourself, then ensure that you have the necessary permissions to use TKPROF to format them.

See Also:

- *Oracle Database Reference* to learn about the `DIAGNOSTIC_DEST`, `STATISTICS_LEVEL`, `TIMED_STATISTICS`, and `TRACEFILE_IDENTIFIER` initialization parameters
- *Oracle Database Administrator's Guide* to learn how to control the trace file size

Step 2: Enabling the SQL Trace Facility

You can enable the SQL Trace facility at the instance or session level.

The package to use depends on the level:

- Database instance

Use `DBMS_MONITOR.DATABASE_TRACE_ENABLE` procedure to enable tracing, and `DBMS_MONITOR.DATABASE_TRACE_DISABLE` procedure to disable tracing.

- Database session

  Use `DBMS_SESSION.SET_SQL_TRACE` procedure to enable tracing (`true`) or disable tracing (`false`).

> **Note:**
>
> Because running the SQL Trace facility increases system overhead, enable it only when tuning SQL statements, and disable it when you are finished.

To enable and disable tracing at the database instance level:

1. Start SQL*Plus, and connect to the database with administrator privileges.
2. Enable tracing at the database instance level.

   The following example enables tracing for the `orcl` instance:

   ```
   EXEC DBMS_MONITOR.DATABASE_TRACE_ENABLE(INSTANCE_NAME => 'orcl');
   ```
3. Execute the statements to be traced.
4. Disable tracing for the database instance.

   The following example disables tracing for the `orcl` instance:

   ```
   EXEC DBMS_MONITOR.DATABASE_TRACE_DISABLE(INSTANCE_NAME => 'orcl');
   ```

To enable and disable tracing at the session level:

1. Start SQL*Plus, and connect to the database with the desired credentials.
2. Enable tracing for the current session.

   The following example enables tracing for the current session:

   ```
   EXEC DBMS_SESSION.SET_SQL_TRACE(sql_trace => true);
   ```
3. Execute the statements to be traced.
4. Disable tracing for the current session.

   The following example disables tracing for the current session:

   ```
   EXEC DBMS_SESSION.SET_SQL_TRACE(sql_trace => false);
   ```

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_MONITOR.DATABASE_TRACE_ENABLE`

Step 3: Generating Output Files with TKPROF

TKPROF accepts as input a trace file produced by the SQL Trace facility, and it produces a formatted output file. TKPROF can also generate execution plans.

After the SQL Trace facility has generated trace files, you can:

- Run TKPROF on each individual trace file, producing several formatted output files, one for each session.
- Concatenate the trace files, and then run TKPROF on the result to produce a formatted output file for the entire instance.
- Run the TRCSESS command-line utility to consolidate tracing information from several trace files, then run TKPROF on the result.

TKPROF does not report `COMMIT` and `ROLLBACK` statements recorded in the trace file.

Note:

The following SQL statements are truncated to 25 characters in the SQL Trace file:

```
SET ROLE
GRANT
ALTER USER
ALTER ROLE
CREATE USER
CREATE ROLE
```

Example 23-1 TKPROF Output

```
SELECT * FROM emp, dept
WHERE emp.deptno = dept.deptno;

call    count       cpu     elapsed      disk     query current     rows
---- -------  -------  ---------- --------- --------- -------  ------
Parse       1      0.16       0.29         3        13       0        0
Execute     1      0.00       0.00         0         0       0        0
Fetch       1      0.03       0.26         2         2       4       14

Misses in library cache during parse: 1
Parsing user id: (8) SCOTT

Rows     Execution Plan
-------  --------------------------------------------------- 14  MERGE JOIN
 4   SORT JOIN
 4     TABLE ACCESS (FULL) OF 'DEPT'
14    SORT JOIN
14      TABLE ACCESS (FULL) OF 'EMP'
```

For this statement, TKPROF output includes the following information:

- The text of the SQL statement

- The SQL Trace statistics in tabular form
- The number of library cache misses for the parsing and execution of the statement.
- The user initially parsing the statement.
- The execution plan generated by `EXPLAIN PLAN`.

TKPROF also provides a summary of user level statements and recursive SQL calls for the trace file.

Step 4: Storing SQL Trace Facility Statistics

You might want to keep a history of the statistics generated by the SQL Trace facility for an application, and compare them over time.

`TKPROF` can generate a SQL script that creates a table and inserts rows of statistics into it. This script contains the following:

- A `CREATE TABLE` statement that creates an output table named `TKPROF_TABLE`.
- `INSERT` statements that add rows of statistics, one for each traced SQL statement, to `TKPROF_TABLE`.

After running `TKPROF`, run this script to store the statistics in the database.

Generating the TKPROF Output SQL Script

When you run `TKPROF`, use the `INSERT` parameter to specify the name of the generated SQL script.

If you omit the `INSERT` parameter, then `TKPROF` does not generate a script.

Editing the TKPROF Output SQL Script

After `TKPROF` has created the SQL script, you might want to edit the script before running it.

If you have created an output table for previously collected statistics, and if you want to add new statistics to this table, then remove the `CREATE TABLE` statement from the script. The script then inserts the new rows into the existing table. If you have created multiple output tables, perhaps to store statistics from different databases in different tables, then edit the `CREATE TABLE` and `INSERT` statements to change the name of the output table.

Querying the Output Table

After you have created the output table, query using a `SELECT` statement.

The following `CREATE TABLE` statement creates the `TKPROF_TABLE`:

```
CREATE TABLE TKPROF_TABLE (

DATE_OF_INSERT    DATE,
CURSOR_NUM        NUMBER,
DEPTH             NUMBER,
USER_ID           NUMBER,
PARSE_CNT         NUMBER,
PARSE_CPU         NUMBER,
PARSE_ELAP        NUMBER,
PARSE_DISK        NUMBER,
PARSE_QUERY       NUMBER,
PARSE_CURRENT     NUMBER,
PARSE_MISS        NUMBER,
EXE_COUNT         NUMBER,
EXE_CPU           NUMBER,
EXE_ELAP          NUMBER,
EXE_DISK          NUMBER,
EXE_QUERY         NUMBER,
EXE_CURRENT       NUMBER,
EXE_MISS          NUMBER,
EXE_ROWS          NUMBER,
FETCH_COUNT       NUMBER,
FETCH_CPU         NUMBER,
FETCH_ELAP        NUMBER,
FETCH_DISK        NUMBER,
FETCH_QUERY       NUMBER,
FETCH_CURRENT     NUMBER,
FETCH_ROWS        NUMBER,
CLOCK_TICKS       NUMBER,
SQL_STATEMENT     LONG);
```

Most output table columns correspond directly to the statistics that appear in the formatted output file. For example, the `PARSE_CNT` column value corresponds to the count statistic for the parse step in the output file.

The columns in the following table help you identify a row of statistics.

Table 23-2   TKPROF_TABLE Columns for Identifying a Row of Statistics

Column	Description
`SQL_STATEMENT`	This is the SQL statement for which the SQL Trace facility collected the row of statistics. Because this column has data type `LONG`, you cannot use it in expressions or `WHERE` clause conditions.
`DATE_OF_INSERT`	This is the date and time when the row was inserted into the table. This value is different from the time when the SQL Trace facility collected the statistics.
`DEPTH`	This indicates the level of recursion at which the SQL statement was issued. For example, a value of 0 indicates that a user issued the statement. A value of 1 indicates that Oracle Database generated the statement as a recursive call to process a statement with a value of 0 (a statement issued by a user). A value of *n* indicates that Oracle Database generated the statement as a recursive call to process a statement with a value of *n*-1.
`USER_ID`	This identifies the user issuing the statement. This value also appears in the formatted output file.
`CURSOR_NUM`	Oracle Database uses this column value to keep track of the cursor to which each SQL statement was assigned.

The output table does not store the statement's execution plan. The following query returns the statistics from the output table. These statistics correspond to the formatted output shown in "Example 23-7".

```
SELECT * FROM TKPROF_TABLE;
```

Sample output appears as follows:

```
DATE_OF_INSERT CURSOR_NUM DEPTH USER_ID PARSE_CNT PARSE_CPU PARSE_ELAP
-------------- ---------- ----- ------- --------- --------- ----------
21-DEC-2017             1     0      8          1        16          22

PARSE_DISK PARSE_QUERY PARSE_CURRENT PARSE_MISS EXE_COUNT EXE_CPU
---------- ----------- ------------- ---------- --------- -------
    3          11            0             1           1         0

EXE_ELAP EXE_DISK EXE_QUERY EXE_CURRENT EXE_MISS EXE_ROWS FETCH_COUNT
-------- -------- --------- ----------- -------- -------- -----------
    0        0         0          0          0        0          1

FETCH_CPU FETCH_ELAP FETCH_DISK FETCH_QUERY FETCH_CURRENT FETCH_ROWS
--------- ---------- ---------- ----------- ------------- ----------
     2        20          2           2             4            10

SQL_STATEMENT
---------------------------------------------------------------------
SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO
```

Guidelines for Interpreting TKPROF Output

While `TKPROF` provides a useful analysis, the most accurate measure of efficiency is the performance of the application. At the end of the `TKPROF` output is a summary of the work that the process performed during the period that the trace was running.

Guideline for Interpreting the Resolution of Statistics

Timing statistics have a resolution of one hundredth of a second. Therefore, any operation on a cursor that takes a hundredth of a second or less might not be timed accurately.

Keep the time limitation in mind when interpreting statistics. In particular, be careful when interpreting the results from simple queries that execute very quickly.

Guideline for Recursive SQL Statements

Recursive SQL is additional SQL that Oracle Database must issue to execute a SQL statement issued by a user.

Conceptually, recursive SQL is “side-effect SQL.” For example, if a session inserts a row into a table that has insufficient space to hold that row, then the database makes recursive SQL calls to allocate the space dynamically. The database also generates recursive calls when data dictionary information is not available in memory and so must be retrieved from disk.

If recursive calls occur while the SQL Trace facility is enabled, then `TKPROF` produces statistics for the recursive SQL statements and marks them clearly as recursive SQL statements in the output file. You can suppress the listing of Oracle Database internal recursive calls (for example, space management) in the output file by setting the `SYS` command-line parameter to `NO`. The statistics for a recursive SQL statement are included in the listing for that statement, not in the listing for the SQL statement that caused the recursive call. So, when you are calculating the total resources required to process a SQL statement, consider the statistics for that statement and those for recursive calls caused by that statement.

Note:

Recursive SQL statistics are not included for SQL-level operations.

Guideline for Deciding Which Statements to Tune

You must determine which SQL statements use the most CPU or disk resource.

If the `TIMED_STATISTICS` parameter is enabled, then you can find high CPU activity in the `CPU` column. If `TIMED_STATISTICS` is not enabled, then check the `QUERY` and `CURRENT` columns.

With the exception of locking problems and inefficient PL/SQL loops, neither the CPU time nor the elapsed time is necessary to find problem statements. The key is the number of block visits, both query (that is, subject to read consistency) and current (that is, not subject to read consistency). Segment headers and blocks that are going to be updated are acquired in current mode, but all query and subquery processing requests the data in query mode. These are precisely the same measures as the instance statistics `CONSISTENT GETS` and `DB BLOCK GETS`. You can find high disk activity in the `disk` column.

The following listing shows `TKPROF` output for one SQL statement as it appears in the output file:

```
SELECT *
FROM emp, dept
WHERE emp.deptno = dept.deptno;

call    count      cpu    elapsed     disk     query current    rows
---- --------  -------  --------- --------- --------- -------  ------
Parse      11     0.08       0.18        0         0       0        0
Execute    11     0.23       0.66        0         3       6        0
Fetch      35     6.70       6.83      100     12326       2      824
------------------------------------------------------------------
total      57     7.01       7.67      100     12329       8      826

Misses in library cache during parse: 0
```

If it is acceptable to have 7.01 CPU seconds and to retrieve 824 rows, then you need not look any further at this trace output. In fact, a major use of `TKPROF` reports in a tuning exercise is to eliminate processes from the detailed tuning phase.

The output indicates that 10 unnecessary parse call were made (because 11 parse calls exist for this single statement) and that array fetch operations were performed. More rows were fetched than there were fetches performed. A large gap between `CPU` and `elapsed` timings indicates Physical I/Os.

See Also:

"Example 23-4"

Guidelines for Avoiding Traps in TKPROF Interpretation

When interpreting `TKPROF` output, it helps to be aware of common traps.

Guideline for Avoiding the Argument Trap

If you are not aware of the values being bound at run time, then it is possible to fall into the argument trap.

`EXPLAIN PLAN` cannot determine the type of a bind variable from the text of SQL statements, and it always assumes that the type is `VARCHAR`. If the bind variable is actually a number or a date, then `TKPROF` can cause implicit data conversions, which can cause inefficient plans to be executed. To avoid this situation, experiment with different data types in the query, and perform the conversion yourself.

Guideline for Avoiding the Read Consistency Trap

Without knowing that an uncommitted transaction had made a series of updates to a column, it is difficult to see why so many block visits would be incurred.

Such cases are not normally repeatable. If the process were run again, it is unlikely that another transaction would interact with it in the same way.

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';

call     count      cpu     elapsed     disk     query current     rows
----     -----      ---     -------     ----     ----- -------     ----
Parse        1     0.10        0.18        0         0       0        0
Execute      1     0.00        0.00        0         0       0        0
Fetch        1     0.11        0.21        2       101       0        1

Misses in library cache during parse: 1
Parsing user id: 01 (USER1)

Rows     Execution Plan
----     --------- ----
   0     SELECT STATEMENT
   1       TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
   2         INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON_UNIQUE)
```

Guideline for Avoiding the Schema Trap

In some cases, an apparently straightforward indexed query looks at many database blocks and accesses them in current mode.

The following example shows an extreme (and thus easily detected) example of the schema trap:

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';
```

```
call        count       cpu      elapsed     disk  query current rows
--------  -------    --------    ---------  ------- ------ ------- ----
Parse           1       0.06         0.10        0      0       0    0
Execute         1       0.02         0.02        0      0       0    0
Fetch           1       0.23         0.30       31     31       3    1

Misses in library cache during parse: 0
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT
   2340    TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
      0      INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON-UNIQUE)
```

Two statistics suggest that the query might have been executed with a full table scan: the current mode block visits, and the number of rows originating from the Table Access row source in the plan. The explanation is that the required index was built after the trace file had been produced, but before `TKPROF` had been run. Generating a new trace file gives the following data:

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';

call     count    cpu   elapsed  disk  query current     rows
-----   ------ ------  -------- ----- ------ -------    -----
Parse        1   0.01      0.02     0      0       0        0
Execute      1   0.00      0.00     0      0       0        0
Fetch        1   0.00      0.00     0      2       0        1

Misses in library cache during parse: 0
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT
      1    TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
      2      INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON-UNIQUE)
```

In the correct version, the parse call took 10 milliseconds of CPU time and 20 milliseconds of elapsed time, but the query apparently took no time to execute and perform the fetch. These anomalies arise because the clock tick of 10 milliseconds is too long relative to the time taken to execute and fetch the data. In such cases, it is important to get many executions of the statements, so that you have statistically valid numbers.

Guideline for Avoiding the Time Trap

In some cases, a query takes an inexplicably long time.

For example, the following update of 7 rows executes in 19 seconds:

```
UPDATE cq_names
  SET ATTRIBUTES = lower(ATTRIBUTES)
  WHERE ATTRIBUTES = :att
```

```
call       count       cpu    elapsed      disk     query current        rows
-------- -------  --------  --------- --------- --------- -------  ----------
Parse          1      0.06       0.24         0         0       0           0
Execute        1      0.62      19.62        22       526      12           7
Fetch          0      0.00       0.00         0         0       0           0

Misses in library cache during parse: 1
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  UPDATE STATEMENT
   2519  TABLE ACCESS (FULL) OF 'CQ_NAMES'
```

The explanation is interference from another transaction. In this case, another transaction held a shared lock on the table `cq_names` for several seconds before and after the update was issued. It takes experience to diagnose that interference effects are occurring. On the one hand, comparative data is essential when the interference is contributing only a short delay (or a small increase in block visits in the previous example). However, if the interference contributes only modest overhead, and if the statement is essentially efficient, then its statistics may not require analysis.

Application Tracing Utilities

The Oracle tracing utilities are TKPROF and TRCSESS.

TRCSESS

The TRCSESS utility consolidates trace output from selected trace files based on user-specified criteria.

After TRCSESS merges the trace information into a single output file, TKPROF can process the output file.

Purpose

TRCSESS is useful for consolidating the tracing of a particular session for performance or debugging purposes.

Tracing a specific session is usually not a problem in the dedicated server model because one process serves a session during its lifetime. You can see the trace information for the session from the trace file belonging to the server process. However, in a shared server configuration, a user session is serviced by different processes over time. The trace for the user session is scattered across different trace files belonging to different processes, which makes it difficult to get a complete picture of the life cycle of a session.

Guidelines

You must specify one of the `session`, `clientid`, `service`, `action`, or `module` options.

If you specify multiple options, then TRCSESS consolidates all trace files that satisfy the specified criteria into the output file.

Syntax

```
trcsess  [output=output_file_name]
         [session=session_id]
         [clientid=client_id]
         [service=service_name]
         [action=action_name]
         [module=module_name]
         [trace_files]
```

Options

TRCSESS supports a number of command-line options.

Argument	Description
`output`	Specifies the file where the output is generated. If this option is not specified, then the utility writes to standard output.
`session`	Consolidates the trace information for the session specified. The session identifier is a combination of session index and session serial number, such as `21.2371`. You can locate these values in the `V$SESSION` view.
`clientid`	Consolidates the trace information for the specified client ID.
`service`	Consolidates the trace information for the specified service name.
`action`	Consolidates the trace information for the specified action name.
`module`	Consolidates the trace information for the specified module name.
`trace_files`	Lists the trace file names, separated by spaces, in which TRCSESS should look for trace information. You can use the wildcard character (`*`) to specify the trace file names. If you do not specify trace files, then TRCSESS uses all files in the current directory as input.

Examples

This section demonstrates common TRCSESS use cases.

Example 23-2 Tracing a Single Session

This sample output of TRCSESS shows the container of traces for a particular session. In this example, the session index and serial number equals `21.2371`. All files in current directory are taken as input.

```
trcsess session=21.2371
```

Example 23-3 Specifying Multiple Trace Files

The following example specifies two trace files:

```
trcsess session=21.2371 main_12359.trc main_12995.trc
```

The sample output is similar to the following:

```
[PROCESS ID = 12359]
*** 2014-04-02 09:48:28.376
PARSING IN CURSOR #1 len=17 dep=0 uid=27 oct=3 lid=27 tim=868373970961
hv=887450622 ad='22683fb4'
select * from cat
END OF STMT
PARSE #1:c=0,e=339,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=868373970944
EXEC #1:c=0,e=221,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=868373971411
FETCH #1:c=0,e=791,p=0,cr=7,cu=0,mis=0,r=1,dep=0,og=4,tim=868373972435
FETCH #1:c=0,e=1486,p=0,cr=20,cu=0,mis=0,r=6,dep=0,og=4,tim=868373986238
*** 2014-04-02 10:03:58.058
XCTEND rlbk=0, rd_only=1
STAT #1 id=1 cnt=7 pid=0 pos=1 obj=0 op='FILTER  '
STAT #1 id=2 cnt=7 pid=1 pos=1 obj=18 op='TABLE ACCESS BY INDEX ROWID OBJ$ '
STAT #1 id=3 cnt=7 pid=2 pos=1 obj=37 op='INDEX RANGE SCAN I_OBJ2 '
STAT #1 id=4 cnt=0 pid=1 pos=2 obj=4 op='TABLE ACCESS CLUSTER TAB$J2 '
STAT #1 id=5 cnt=6 pid=4 pos=1 obj=3 op='INDEX UNIQUE SCAN I_OBJ# '
[PROCESS ID=12995]
*** 2014-04-02 10:04:32.738
Archiving is disabled
```

TKPROF

The TKPROF program formats the contents of the trace file and places the output into a readable output file.

TKPROF can also do the following:

- Create a SQL script that stores the statistics in the database
- Determine the execution plans of SQL statements

> **Note:**
>
> If the cursor for a SQL statement is not closed, then TKPROF output does not automatically include the actual execution plan of the SQL statement. In this situation, use the `EXPLAIN` option with TKPROF to generate an execution plan.

TKPROF reports each statement executed with the resources it has consumed, the number of times it was called, and the number of rows which it processed.

Purpose

TKPROF can locate statements that are consuming the greatest resources.

With baselines available, you can assess whether the resources used are reasonable given the work performed.

Guidelines

The input and output files are the only required arguments.

If you invoke TKPROF without arguments, then the tool displays online help.

Syntax

```
tkprof input_file output_file
  [ waits=yes|no ]
  [ sort=option ]
  [ print=n ]
  [ aggregate=yes|no ]
  [ insert=filename3 ]
  [ sys=yes|no ]
  [ table=schema.table ]
  [ explain=user/password ]
  [ record=filename4 ]
  [ width=n ]
```

Options

TKPROF supports a number of command-line options.

Table 23-3 TKPROF Arguments

Argument	Description
input_file	Specifies the input file, a trace file containing statistics produced by the SQL Trace facility. This file can be either a trace file produced for a single session, or a file produced by concatenating individual trace files from multiple sessions.
output_file	Specifies the file to which `TKPROF` writes its formatted output.
`WAITS`	Specifies whether to record summary for any wait events found in the trace file. Valid values are `YES` (default) and `NO`.

Table 23-3 (Cont.) TKPROF Arguments

Argument	Description
`SORT`	Sorts traced SQL statements in descending order of specified sort option before listing them in the output file. If multiple options are specified, then the output is sorted in descending order by the sum of the values specified in the sort options. If you omit this parameter, then `TKPROF` lists statements into the output file in order of first use. Sort options are listed as follows: • `PRSCNT` - Number of times parsed • `PRSCPU` - CPU time spent parsing • `PRSELA` - Elapsed time spent parsing • `PRSDSK` - Number of physical reads from disk during parse • `PRSQRY` - Number of consistent mode block reads during parse • `PRSCU` - Number of current mode block reads during parse • `PRSMIS` - Number of library cache misses during parse • `EXECNT` - Number of executions • `EXECPU` - CPU time spent executing • `EXEELA` - Elapsed time spent executing • `EXEDSK` - Number of physical reads from disk during execute • `EXEQRY` - Number of consistent mode block reads during execute • `EXECU` - Number of current mode block reads during execute • `EXEROW` - Number of rows processed during execute • `EXEMIS` - Number of library cache misses during execute • `FCHCNT` - Number of fetches • `FCHCPU` - CPU time spent fetching • `FCHELA` - Elapsed time spent fetching • `FCHDSK` - Number of physical reads from disk during fetch • `FCHQRY` - Number of consistent mode block reads during fetch • `FCHCU` - Number of current mode block reads during fetch • `FCHROW` - Number of rows fetched • `USERID` - ID of user that parsed the cursor
`PRINT`	Lists only the first integer sorted SQL statements from the output file. If you omit this parameter, then `TKPROF` lists all traced SQL statements. This parameter does not affect the optional SQL script. The SQL script always generates insert data for all traced SQL statements.
`AGGREGATE`	If you specify `AGGREGATE` = `NO`, then `TKPROF` does not aggregate multiple users of the same SQL text.
`INSERT`	Creates a SQL script that stores the trace file statistics in the database. `TKPROF` creates this script with the name *`filename3`*. This script creates a table and inserts a row of statistics for each traced SQL statement into the table.
`SYS`	Enables and disables the listing of SQL statements issued by the user `SYS`, or recursive SQL statements, into the output file. The default value of `YES` causes `TKPROF` to list these statements. The value of `NO` causes `TKPROF` to omit them. This parameter does not affect the optional SQL script. The SQL script always inserts statistics for all traced SQL statements, including recursive SQL statements.

Table 23-3 (Cont.) TKPROF Arguments

Argument	Description
`TABLE`	Specifies the schema and name of the table into which `TKPROF` temporarily places execution plans before writing them to the output file. If the specified table exists, then `TKPROF` deletes all rows in the table, uses it for the `EXPLAIN PLAN` statement (which writes more rows into the table), and then deletes those rows. If this table does not exist, then `TKPROF` creates it, uses it, and then drops it. The specified user must be able to issue `INSERT`, `SELECT`, and `DELETE` statements against the table. If the table does not exist, then the user must also be able to issue `CREATE TABLE` and `DROP TABLE` statements. This option enables multiple users to run `TKPROF` concurrently with the same database user account in the `EXPLAIN` value. These users can specify different `TABLE` values and avoid destructively interfering with each other when processing the temporary plan table. `TKPROF` supports the following combinations: • The `EXPLAIN` parameter without the `TABLE` parameter `TKPROF` uses the table `PROF$PLAN_TABLE` in the schema of the user specified by the `EXPLAIN` parameter • The `TABLE` parameter without the `EXPLAIN` parameter `TKPROF` ignores the `TABLE` parameter. If no plan table exists, then `TKPROF` creates the table `PROF$PLAN_TABLE` and then drops it at the end.
`EXPLAIN`	Determines the execution plan for each SQL statement in the trace file and writes these execution plans to the output file. `TKPROF` also displays the number of rows processed by each step of the execution plan. `TKPROF` determines execution plans by issuing the `EXPLAIN PLAN` statement after connecting to Oracle Database with the user and password specified in this parameter. The specified user must have `CREATE SESSION` system privileges. `TKPROF` takes longer to process a large trace file if the `EXPLAIN` option is used. **Note:** Trace files generated immediately after instance startup contain data that reflects the activity of the startup process. In particular, they reflect a disproportionate amount of I/O activity as caches in the system global area (SGA) are filled. For the purposes of tuning, ignore such trace files.
`RECORD`	Creates a SQL script with the specified *`filename`* with all of the nonrecursive SQL in the trace file. You can use this script to replay the user events from the trace file.
`WIDTH`	An integer that controls the output line width of some `TKPROF` output, such as the explain plan. This parameter is useful for post-processing of `TKPROF` output.

Output

This section explains the TKPROF output.

Identification of User Issuing the SQL Statement in TKPROF

`TKPROF` lists the user ID of the user issuing each SQL statement.

If the SQL Trace input file contained statistics from multiple users, and if the statement was issued by multiple users, then `TKPROF` lists the ID of the last user to parse the statement. The user ID of all database users appears in the data dictionary in the column `ALL_USERS.USER_ID`.

Tabular Statistics in TKPROF

TKPROF lists the statistics for a SQL statement returned by the SQL Trace facility in rows and columns.

Each row corresponds to one of three steps of SQL statement processing. Statistics are identified by the value of the `CALL` column. See Table 23-4.

Table 23-4 CALL Column Values

CALL Value	Meaning
`PARSE`	Translates the SQL statement into an execution plan, including checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.
`EXECUTE`	Actual execution of the statement by Oracle Database. For `INSERT`, `UPDATE`, `DELETE`, and `MERGE` statements, this modifies the data. For `SELECT` statements, this identifies the selected rows.
`FETCH`	Retrieves rows returned by a query. Fetches are only performed for `SELECT` statements.

The other columns of the SQL Trace facility output are combined statistics for all parses, executions, and fetches of a statement. The sum of `query` and `current` is the total number of buffers accessed, also called Logical I/Os (LIOs). See Table 23-5.

Table 23-5 SQL Trace Statistics for Parses, Executes, and Fetches.

SQL Trace Statistic	Meaning
`COUNT`	Number of times a statement was parsed, executed, or fetched.
`CPU`	Total CPU time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if `TIMED_STATISTICS` is not enabled.
`ELAPSED`	Total elapsed time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if `TIMED_STATISTICS` is not enabled.
`DISK`	Total number of data blocks physically read from the data files on disk for all parse, execute, or fetch calls.
`QUERY`	Total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls. Usually, buffers are retrieved in consistent mode for queries.
`CURRENT`	Total number of buffers retrieved in current mode. Buffers are retrieved in current mode for statements such as `INSERT`, `UPDATE`, and `DELETE`.

Statistics about the processed rows appear in the `ROWS` column. The column shows the number of rows processed by the SQL statement. This total does not include rows processed by subqueries of the SQL statement. For `SELECT` statements, the number of rows returned appears for the fetch step. For `UPDATE`, `DELETE`, and `INSERT` statements, the number of rows processed appears for the execute step.

Note:

The row source counts are displayed when a cursor is closed. In SQL*Plus, there is only one user cursor, so each statement executed causes the previous cursor to be closed; therefore, the row source counts are displayed. PL/SQL has its own cursor handling and does not close child cursors when the parent cursor is closed. Exiting or reconnecting causes the counts to be displayed.

Library Cache Misses in TKPROF

TKPROF also lists the number of library cache misses resulting from parse and execute steps for each SQL statement.

These statistics appear on separate lines following the tabular statistics. If the statement resulted in no library cache misses, then `TKPROF` does not list the statistic. In "Examples", the statement resulted in one library cache miss for the parse step and no misses for the execute step.

Row Source Operations in TKPROF

In the TKPROF output, row source operations show the number of rows processed for each operation executed on the rows, and additional row source information, such as physical reads and writes.

Table 23-6 Row Source Operations

Row Source Operation	Meaning
`cr`	Consistent reads performed by the row source.
`r`	Physical reads performed by the row source
`w`	Physical writes performed by the row source
`time`	Time in microseconds

In the following sample `TKPROF` output, note the `cr`, `r`, `w`, and `time` values under the Row Source Operation column:

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  DELETE  (cr=43141 r=266947 w=25854 time=60235565 us)
  28144   HASH JOIN ANTI (cr=43057 r=262332 w=25854 time=48830056 us)
  51427    TABLE ACCESS FULL STATS$SQLTEXT (cr=3465 r=3463 w=0 time=865083 us)
 647529    INDEX FAST FULL SCAN STATS$SQL_SUMMARY_PK
                       (cr=39592 r=39325 w=0 time=10522877 us) (object id 7409)
```

Wait Event Information in TKPROF

If wait event information exists, then the `TKPROF` output includes a section on wait events.

Output looks similar to the following:

```
Elapsed times include waiting on following events:
  Event waited on                  Times Waited   Max. Wait  Total Waited
  ---------------------------      ------------   ---------  ------------
  db file sequential read                  8084        0.12          5.34
  direct path write                         834        0.00          0.00
  direct path write temp                    834        0.00          0.05
  db file parallel read                       8        1.53          5.51
  db file scattered read                   4180        0.07          1.45
  direct path read                         7082        0.00          0.05
  direct path read temp                    7082        0.00          0.44
  rdbms ipc reply                            20        0.00          0.01
  SQL*Net message to client                   1        0.00          0.00
  SQL*Net message from client                 1        0.00          0.00
```

In addition, wait events are summed for the entire trace file at the end of the file.

To ensure that wait events information is written to the trace file for the session, run the following SQL statement:

```
ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
```

Examples

This section demonstrates common TKPROF use cases.

Example 23-4 Printing the Most Resource-Intensive Statements

If you are processing a large trace file using a combination of `SORT` parameters and the `PRINT` parameter, then you can produce a `TKPROF` output file containing only the highest resource-intensive statements. The following statement prints the 10 statements in the trace file that have generated the most physical I/O:

```
TKPROF ora53269.trc ora53269.prf SORT = (PRSDSK, EXEDSK, FCHDSK) PRINT = 10
```

Example 23-5 Generating a SQL Script

This example runs `TKPROF`, accepts a trace file named `examp12_jane_fg_sqlplus_007.trc`, and writes a formatted output file named `outputa.prf`:

```
TKPROF examp12_jane_fg_sqlplus_007.trc OUTPUTA.PRF EXPLAIN=hr
  TABLE=hr.temp_plan_table_a INSERT=STOREA.SQL SYS=NO SORT=(EXECPU,FCHCPU)
```

This example is likely to be longer than a single line on the screen, and you might need to use continuation characters, depending on the operating system.

Note the other parameters in this example:

- The `EXPLAIN` value causes `TKPROF` to connect as the user `hr` and use the `EXPLAIN PLAN` statement to generate the execution plan for each traced SQL statement. You can use this to get access paths and row source counts.

Note:

If the cursor for a SQL statement is not closed, then `TKPROF` output does not automatically include the actual execution plan of the SQL statement. In this situation, you can use the `EXPLAIN` option with `TKPROF` to generate an execution plan.

- The `TABLE` value causes `TKPROF` to use the table `temp_plan_table_a` in the schema `scott` as a temporary plan table.
- The `INSERT` value causes `TKPROF` to generate a SQL script named `STOREA.SQL` that stores statistics for all traced SQL statements in the database.
- The `SYS` parameter with the value of `NO` causes `TKPROF` to omit recursive SQL statements from the output file. In this way, you can ignore internal Oracle Database statements such as temporary table operations.
- The `SORT` value causes `TKPROF` to sort the SQL statements in order of the sum of the CPU time spent executing and the CPU time spent fetching rows before writing them to the output file. For greatest efficiency, always use `SORT` parameters.

Example 23-6 TKPROF Header

This example shows a sample header for the TKPROF report.

```
TKPROF: Release 12.1.0.0.2

Copyright (c) 1982, 2012, Oracle and/or its affiliates.  All rights reserved.

Trace file: /disk1/oracle/log/diag/rdbms/orcla/orcla/trace/orcla_ora_917.trc
Sort options: default

***************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
***************************************************************************
```

Example 23-7 TKPROF Body

This example shows a sample body for a TKPROF report.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  --------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.01       0.00          0          0          0           0

Misses in library cache during parse: 1
Optimizer mode: FIRST_ROWS
```

```
Parsing user id: 44

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                     1       28.59         28.59
********************************************************************************

select condition
from
 cdef$ where rowid=:1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          2          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          2          0           1

Misses in library cache during parse: 1
Optimizer mode: CHOOSE
Parsing user id: SYS   (recursive depth: 1)

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS BY USER ROWID OBJ#(31) (cr=1 r=0 w=0 time=151 us)

********************************************************************************

SELECT last_name, job_id, salary
  FROM employees
WHERE salary =
  (SELECT max(salary) FROM employees)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.02       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0         15          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.02       0.01          0         15          0           1

Misses in library cache during parse: 1
Optimizer mode: FIRST_ROWS
Parsing user id: 44

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS FULL EMPLOYEES (cr=15 r=0 w=0 time=1743 us)
      1   SORT AGGREGATE (cr=7 r=0 w=0 time=777 us)
    107    TABLE ACCESS FULL EMPLOYEES (cr=7 r=0 w=0 time=655 us)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       2        0.00          0.00
  SQL*Net message from client                     2        9.62          9.62
********************************************************************************

********************************************************************************
 delete
        from stats$sqltext st
       where (hash_value, text_subset) not in
            (select --+ hash_aj
                    hash_value, text_subset
               from stats$sql_summary ss
              where (   (   snap_id     < :lo_snap
                         or snap_id     > :hi_snap
                        )
                        and dbid            = :dbid
                        and instance_number = :inst_num
                    )
                 or (   dbid            != :dbid
                     or instance_number != :inst_num)
             )

call     count       cpu    elapsed       disk      query    current rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0          0
Execute      1     29.60      60.68     266984      43776     131172      28144
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total        2     29.60      60.68     266984      43776     131172      28144

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: CHOOSE
Parsing user id: 22

Rows     Row Source Operation
-------  ---------------------------------------------------
      0  DELETE  (cr=43141 r=266947 w=25854 time=60235565 us)
  28144   HASH JOIN ANTI (cr=43057 r=262332 w=25854 time=48830056 us)
  51427    TABLE ACCESS FULL STATS$SQLTEXT (cr=3465 r=3463 w=0 time=865083 us)
 647529    INDEX FAST FULL SCAN STATS$SQL_SUMMARY_PK
                      (cr=39592 r=39325 w=0 time=10522877 us) (object id 7409)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  db file sequential read                      8084        0.12          5.34
  direct path write                             834        0.00          0.00
  direct path write temp                        834        0.00          0.05
  db file parallel read                           8        1.53          5.51
  db file scattered read                       4180        0.07          1.45
  direct path read                             7082        0.00          0.05
  direct path read temp                        7082        0.00          0.44
  rdbms ipc reply                                20        0.00          0.01
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                    1        0.00          0.00
********************************************************************************
```

Example 23-8   TKPROF Summary

This example that shows a summary for the TKPROF report.

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        4      0.04       0.01          0          0          0           0
Execute      5      0.00       0.04          0          0          0           0
Fetch        2      0.00       0.00          0         15          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11      0.04       0.06          0         15          0           1

Misses in library cache during parse: 4
Misses in library cache during execute: 1
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       6        0.00          0.00
  SQL*Net message from client                     5       77.77        128.88

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          2          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          2          0           1

Misses in library cache during parse: 1
    5  user  SQL statements in session.
    1  internal SQL statements in session.
    6  SQL statements in session.
********************************************************************************
Trace file: main_ora_27621.trc
Trace file compatibility: 9.00.01
Sort options: default
       1  session in tracefile.
       5  user  SQL statements in trace file.
       1  internal SQL statements in trace file.
       6  SQL statements in trace file.
       6  unique SQL statements in trace file.
      76  lines in trace file.
     128  elapsed seconds in trace file.
```

Views for Application Tracing

You can use data dictionary and `V$` views to monitor tracing.

This section includes the following topics:

Views Relevant for Trace Statistics

You can display the statistics that have been gathered with the following `V$` and `DBA` views.

Table 23-7 Diagnostic Views

View	Description
`DBA_ENABLED_AGGREGATIONS`	Accumulated global statistics for the currently enabled statistics
`V$CLIENT_STATS`	Accumulated statistics for a specified client identifier
`V$SERVICE_STATS`	Accumulated statistics for a specified service
`V$SERV_MOD_ACT_STATS`	Accumulated statistics for a combination of specified service, module, and action
`V$SERVICEMETRIC`	Accumulated statistics for elapsed time of database calls and for CPU use
`V$DIAG_TRACE_FILE`	Information about all trace files in ADR for the current container
`V$DIAG_APP_TRACE_FILE`	Information about all trace files that contain application trace data (`SQL_TRACE` or `OPTIMIZER_TRACE` event data) in ADR for the current container
`V$DIAG_TRACE_FILE_CONTENTS`	Trace data in the trace files in ADR
`V$DIAG_SQL_TRACE_RECORDS`	`SQL_TRACE` data in the trace files in ADR
`V$DIAG_OPT_TRACE_RECORDS`	Optimizer trace event data in the trace files in ADR
`V$DIAG_SESS_SQL_TRACE_RECORDS`	`SQL_TRACE` data in the trace files in ADR for the current user session
`V$DIAG_SESS_OPT_TRACE_RECORDS`	Optimizer trace event data in the trace files in ADR for the current user session
`V$DIAG_ALERT_EXT`	Contents of the XML-based alert log in ADR for the current container

See Also:

Oracle Database Reference for information about `V$` and data dictionary views

Views Related to Enabling Tracing

A Cloud Control report or the `DBA_ENABLED_TRACES` view can display outstanding traces.

In the `DBA_ENABLED_TRACES` view, you can determine detailed information about how a trace was enabled, including the trace type. The trace type specifies whether the trace is enabled for client identifier, session, service, database, or a combination of service, module, and action.

Controls Over Application Tracing and Access to Trace Events

The DBMS_USERDIAG package lets you enable traces in specific sessions based on database session identifier (SID) and serial number (SER) application tracing.

Some diagnostic mechanisms in Oracle Database are restricted outside of a given PDB. This is done using lockdown profiles, which prevent misuse in which arbitrary events can be enabled from user sessions in a shared tenancy in CDB deployments, particularly in cloud instances. You can use DBMS_USERDIAG to selectively enable trace events for specified levels within a user session. DBMS_USERDIAG is available as a common interface on all types of Oracle Database deployments including cloud instances. The APIs to enable tracing and to use additional functionality the package are briefly described here.

Table 23-8 DBMS_USERDIAG Package Subprograms

Subprogram	Description
`ENABLE_SQL_TRACE_EVENT` procedure	Enables `sql_trace` event at a given `level` in user session, and generates SQL traces into respective process trace files.
`CHECK_SQL_TRACE_EVENT` procedure	Checks the current `sql_trace` event and retrieves the level.
`SET_TRACEFILE_IDENTIFIER` procedure	Sets a custom trace file identifier for the active trace file in the current ADR home.
`SET_EXCEPTION_MODE` procedure	Raises an exception on any error or silently ignores the same (default).
`TRACE` procedure	Traces message to the trace file or alert log.
`GET_CALL_STATUS` function	Obtains the status of the last call to the `DBMS_USERDIAG` API.
`GET_CALL_ERROR_MSG` function	Obtains the error message if the last call to `DBMS_USERDIAG` API returned an error.

See Also:

- DBMS_USERDIAG in the *Database PL/SQL Packages and Types Reference* provides full details on the APIs in this package.