Application Tracing

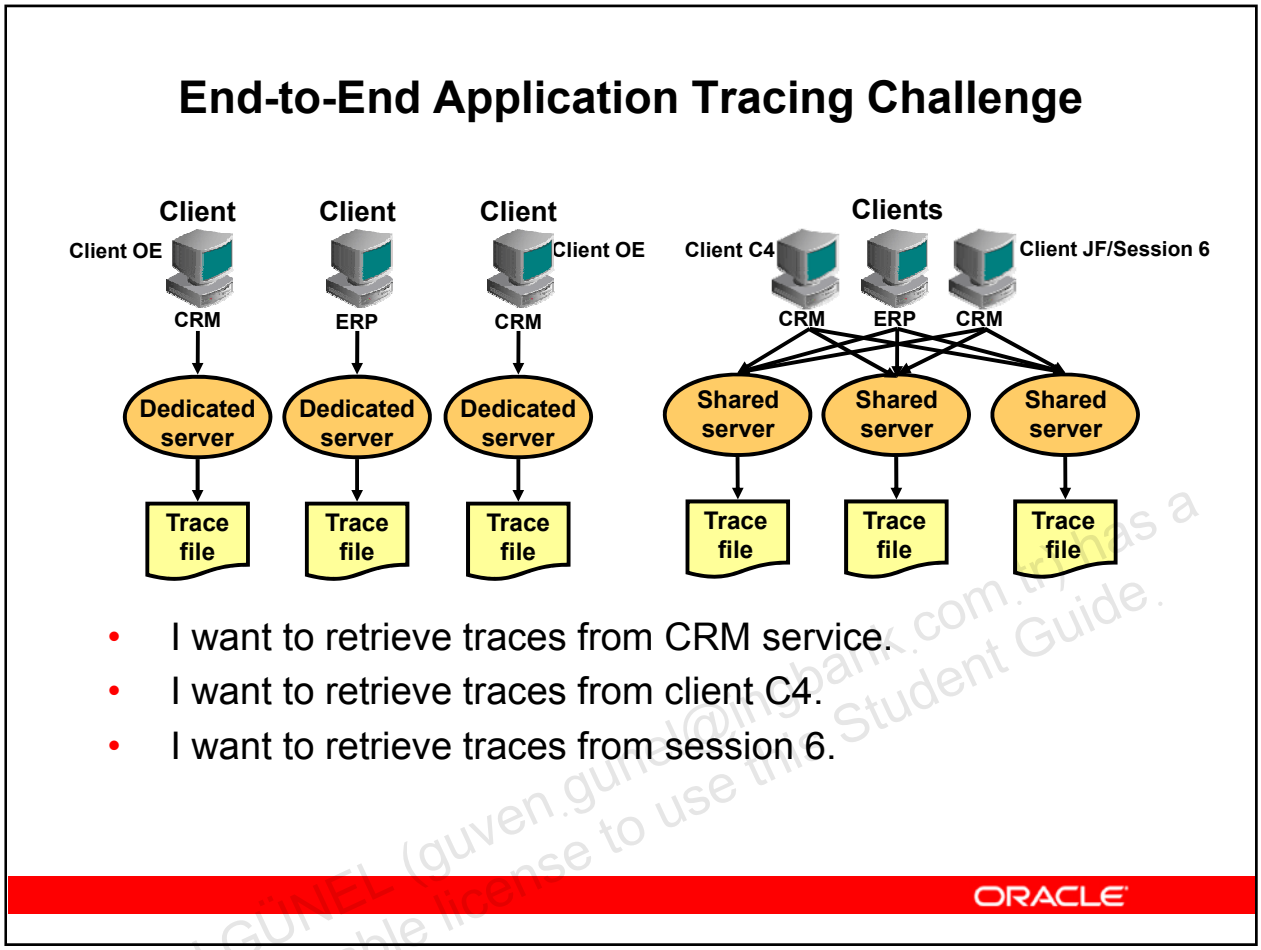
Chapter 5

Objectives

After completing this lesson, you should be able to do the following:

- Configure the SQL Trace facility to collect session statistics
- Use the `trcsess` utility to consolidate SQL trace files
- Format trace files using the `tkprof` utility
- Interpret the output of the `tkprof` command

ORACLE

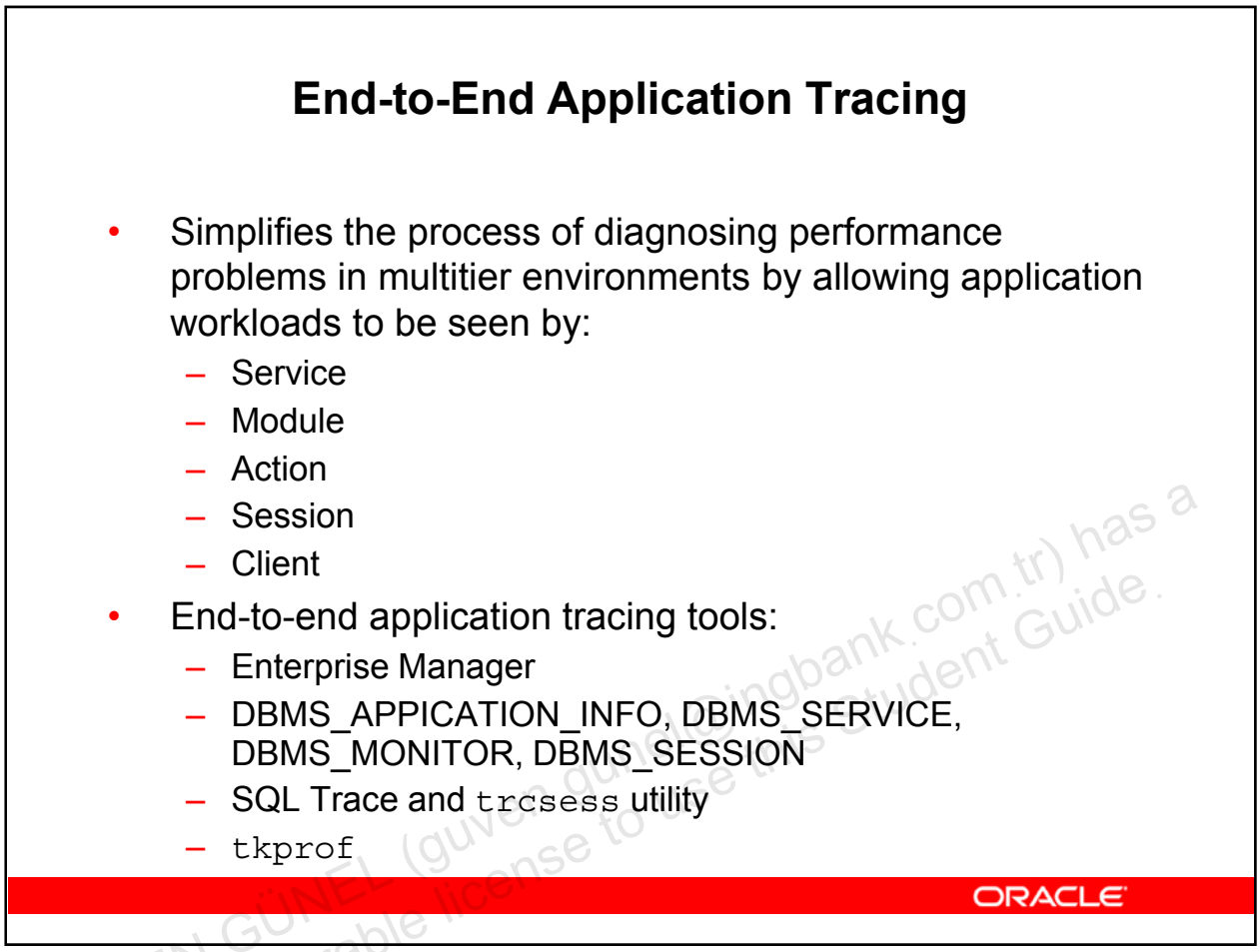


End-to-End Application Tracing Challenge

Oracle Database implements tracing by generating a trace file for each server process when you enable the tracing mechanism.

Tracing a specific client is usually not a problem in the dedicated server model as a single dedicated process serves a session during its lifetime. All the trace information for the session can be seen from the trace file belonging to the dedicated server serving it. However, in a shared server configuration, a client is serviced by different processes from time-to-time. The trace pertaining to the user session is scattered across different trace files belonging to different processes. This makes it difficult for you to get a complete picture of the life cycle of a session.

Moreover, what if you want to consolidate trace information for a particular service for performance or debugging purposes? This is also difficult because you have multiple clients using the same service and each generating trace files belonging to the server process serving it.



End-to-End Application Tracing

End-to-end application tracing simplifies the diagnosis of performance problems in multitier environments. In multitier environments, a request from an end client is routed to different database sessions by the middle tier, making it difficult to track a specific client. A client identifier is used to uniquely trace a specific end client through all tiers to the database server.

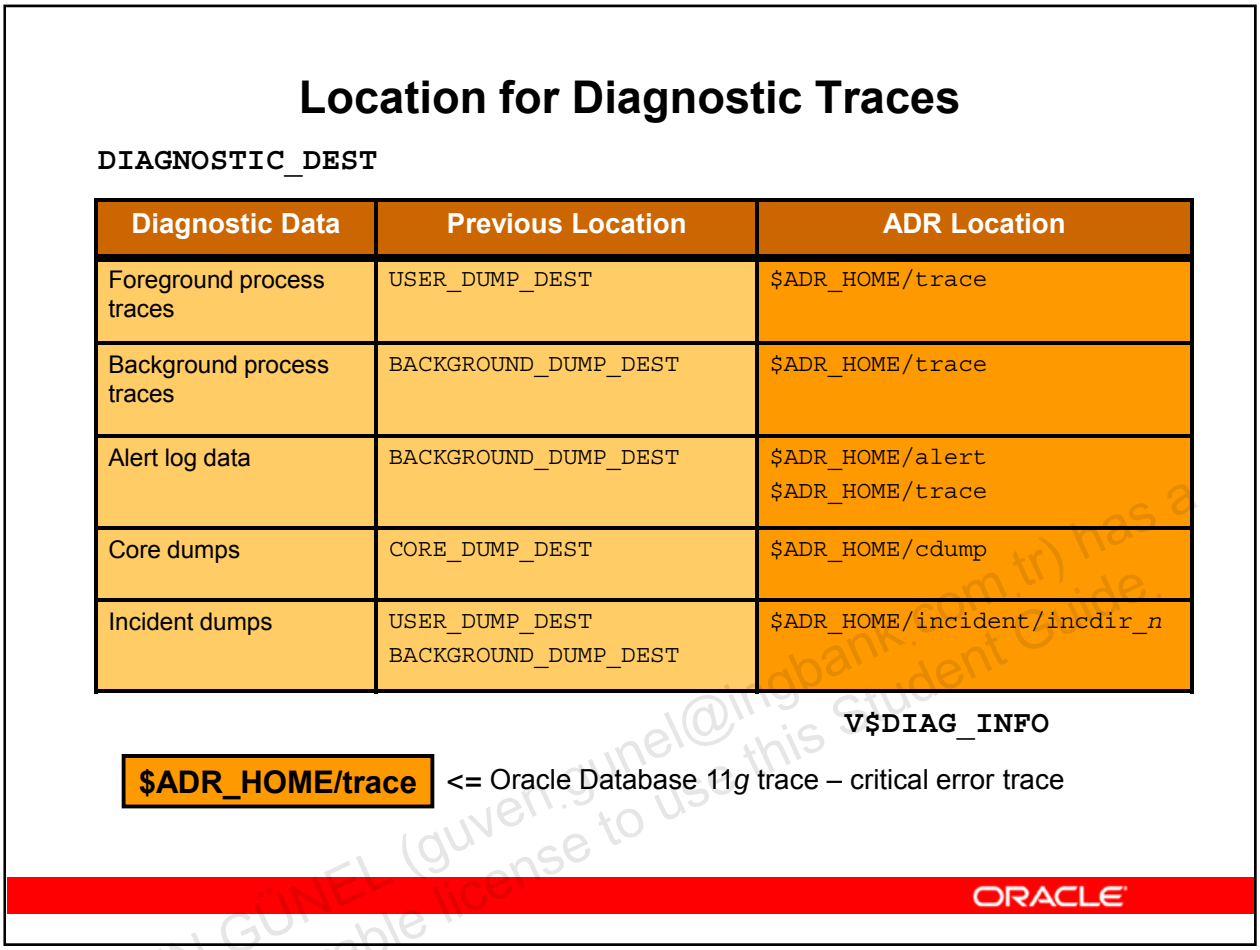
You can used end-to-end application tracing to identify the source of an excessive workload, such as a high-load SQL statement. Also, you can identify what a user's session does at the database level to resolve that user's performance problems.

End-to-end application tracing also simplifies management of application workloads by tracking specific modules and actions in a service. Workload problems can be identified by:

- **Client identifier:** Specifies an end user based on the logon ID, such as `HR`
- **Service:** Specifies a group of applications with common attributes, service-level thresholds, and priorities; or a single application
- **Module:** Specifies a functional block within an application
- **Action:** Specifies an action, such as an `INSERT` or an `UPDATE` operation, in a module
- **Session:** Specifies a session based on a given database session identifier (SID)

The primary interface for end-to-end application tracing is Enterprise Manager. Other tools listed in the slide are discussed later in this lesson.

Location for Diagnostic Traces

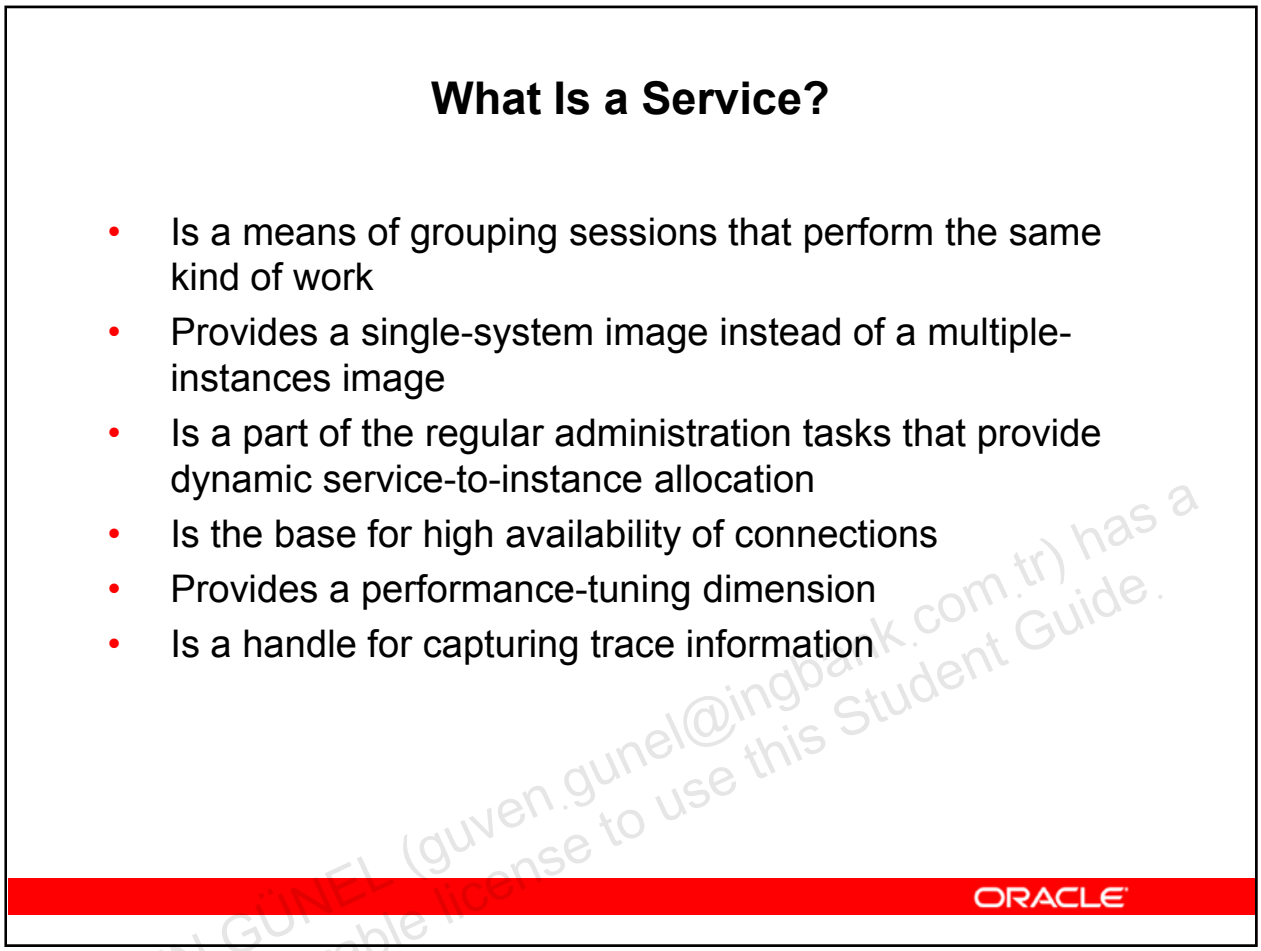


Location for Diagnostic Traces

Starting with Oracle Database 11*g*, Release 1, Automatic Diagnostic Repository (ADR) is a file-based repository for database diagnostic data such as traces, incident dumps, packages, the alert log, Health Monitor reports, core dumps, and so on. The traditional …_DUMP_DEST initialization parameters are ignored. The ADR root directory is known as the ADR base. Its location is set by the DIAGNOSTIC_DEST initialization parameter. In the slide, this location is denoted by $ADR_HOME. However, there is no official environment variable called ADR_HOME. The table shown in the slide describes the different classes of trace data and dumps that reside both in Oracle Database 10*g* (and earlier releases) and in Oracle Database 11*g*. With Oracle Database 11*g*, there is no distinction between foreground and background trace files. Both types of files go into the $ADR_HOME/trace directory. You can use V$DIAG_INFO to list some important ADR locations.

All nonincident traces are stored inside the TRACE subdirectory. Starting with Oracle Database 11*g,* critical error information is dumped into the corresponding process trace files instead of incident dumps. Incident dumps are placed in files separated from the normal process trace files.

Note: The main difference between a trace and a dump is that a trace is a continuous output, such as when SQL tracing is turned on, and a dump is a one-time output in response to an event, such as an incident. Also, a core dump is a binary memory dump that is port specific.



What Is a Service?

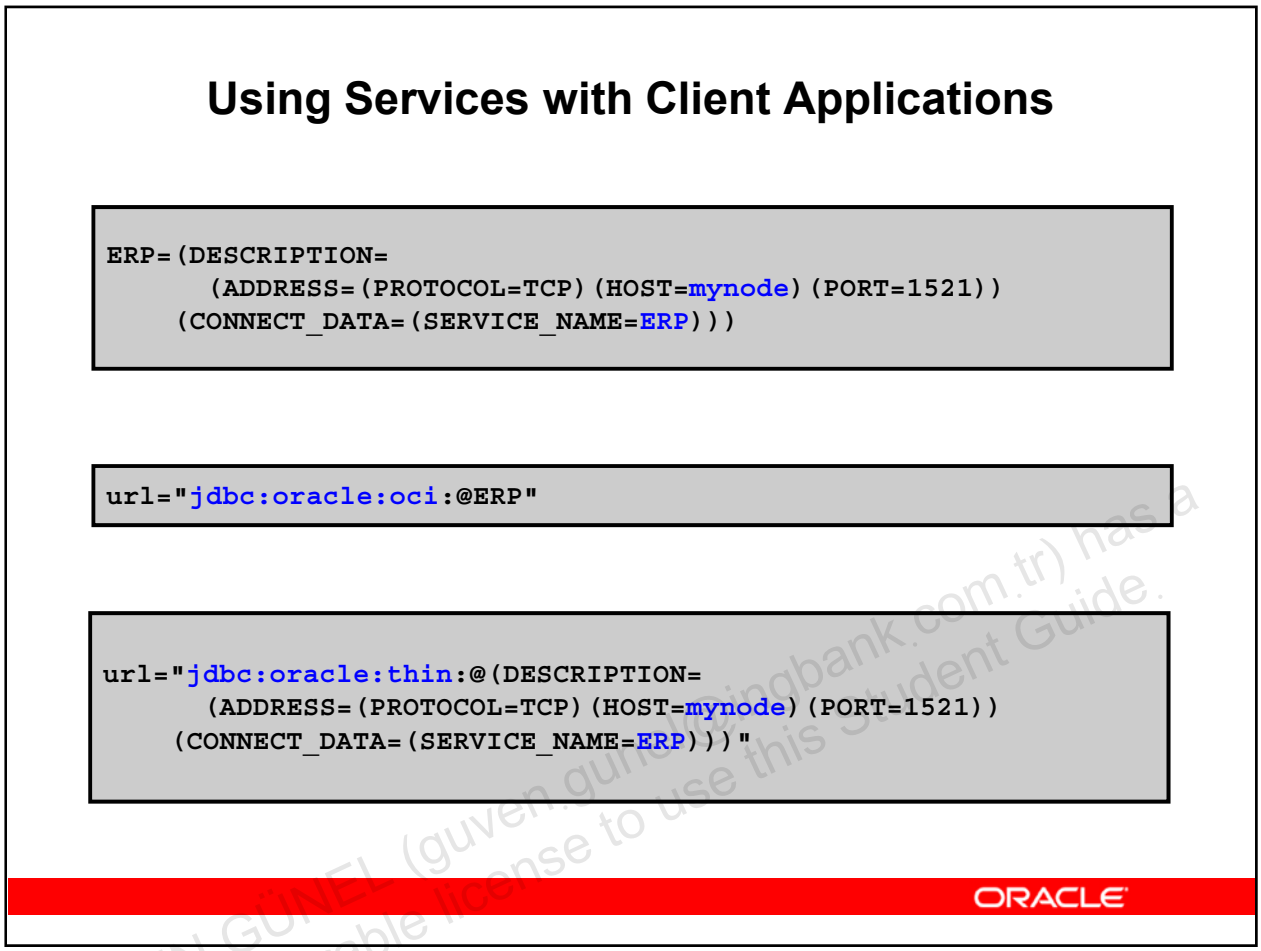
The concept of a service was first introduced in Oracle8*i* as a means for the listener to perform connection load balancing between nodes and instances of a cluster. However, the concept, definition, and implementation of services have been dramatically expanded. A service organizes work execution within the database to make it more manageable, measurable, tunable, and recoverable. A service is a grouping of related tasks within the database with common functionality, quality expectations, and priority relative to other services. A service provides a single-system image for managing competing applications that run within a single instance and across multiple instances and databases.

Services can be configured, administered, enabled, disabled, and measured as a single entity using standard interfaces, Enterprise Manager, and `SRVCTL`,.

Services provide availability. Following outages, a service is recovered quickly and automatically at surviving instances.

Services provide an additional dimension to performance tuning. With services, workloads are visible and measurable. Tuning by “service and SQL” replaces tuning by “session and SQL” in the majority of systems where sessions are anonymous and shared.

From a tracing point of view, a service provides a handle that permits capturing trace information by service name regardless of the session.



Using Services with Client Applications

A service name is used by any client connecting to the database server. That service name is automatically applied to the client actions. Applications can be grouped by services by simply using a different service name for each application to connect.

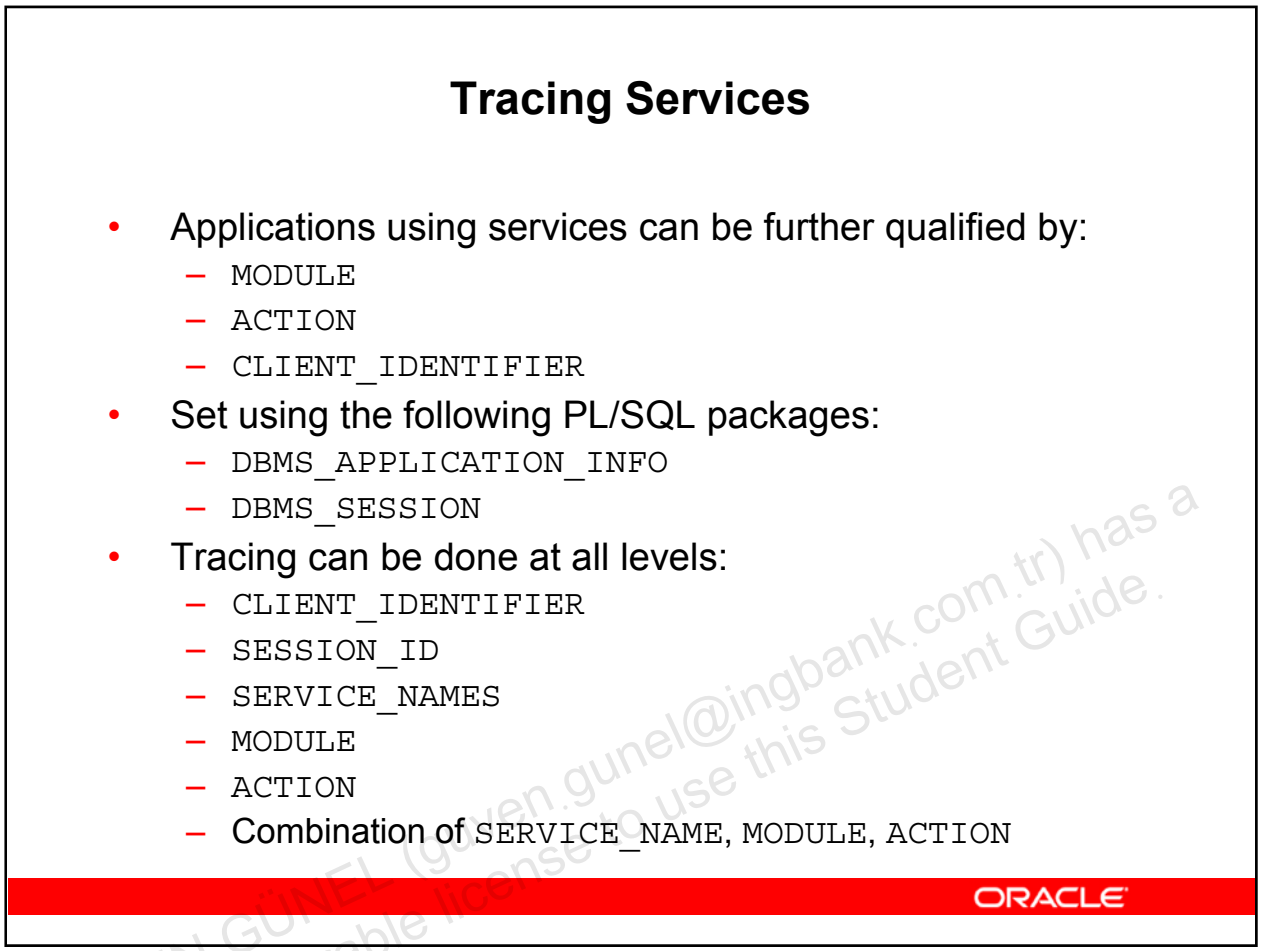
Applications and middle-tier connection pools select a service by using the Transparent Network Substrate (TNS) connection descriptor.

The selected service must match the service that has been created.

The first example in the slide shows the TNS connect descriptor that can be used to access the ERP service.

The second example shows the thick Java Database Connectivity (JDBC) connection description using the previously defined TNS connect descriptor.

The third example shows the thin JDBC connection description using the same TNS connect descriptor.



Tracing Services

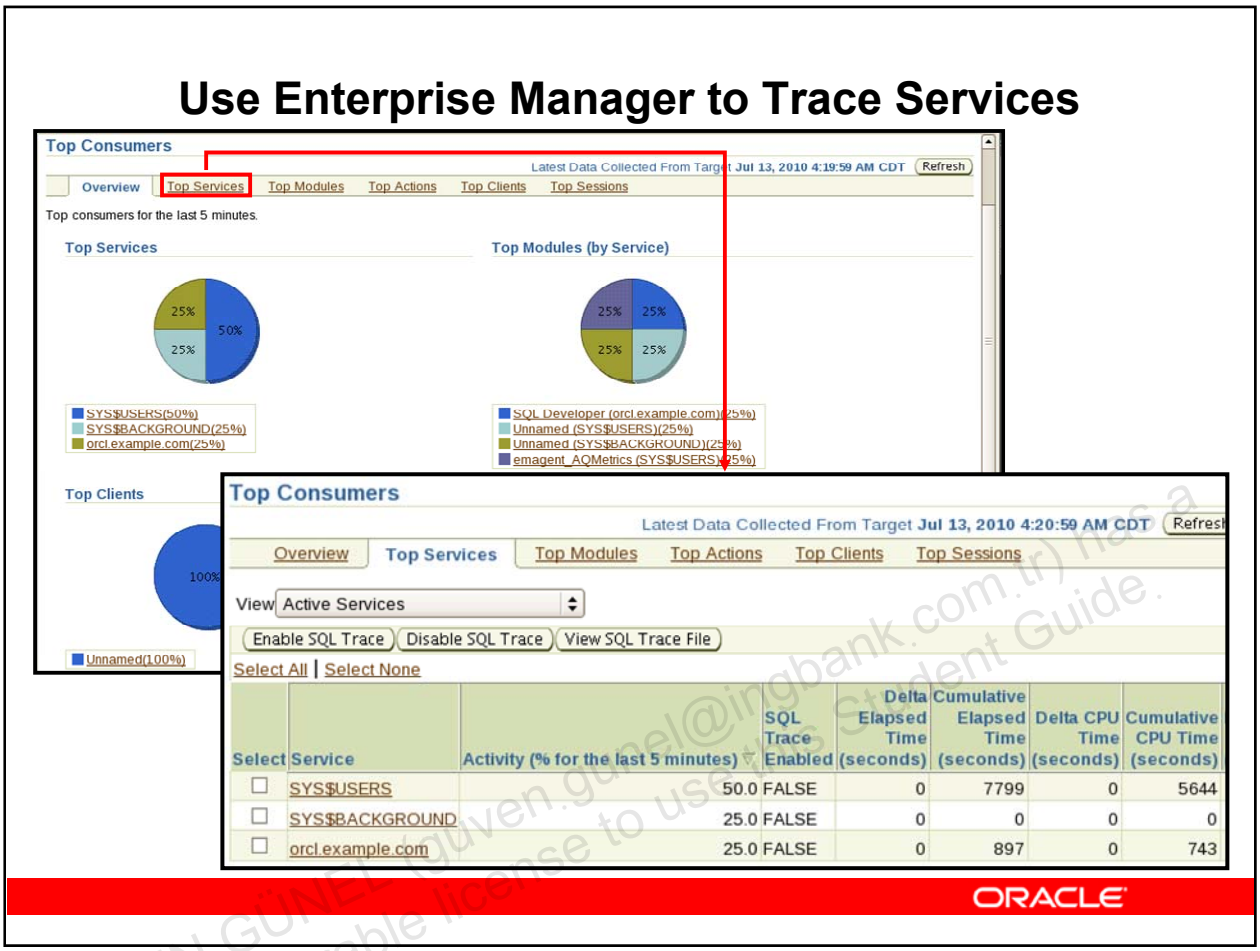
An application can qualify a service by MODULE and ACTION names to identify the important transactions within the service. This enables you to locate the poorly performing transactions for categorized workloads. This is important when you monitor performance in systems using connection pools or transaction processing monitors. For these systems, the sessions are shared, which makes accountability difficult. SERVICE_NAME, MODULE, ACTION, CLIENT_IDENTIFIER, and SESSION_ID are actual columns in V$SESSION. SERVICE_NAME is set automatically at login time based on the connect descriptor, and SESSION_ID is automatically set by the database when a session is created. MODULE and ACTION names are set by the application by using the DBMS_APPLICATION_INFO PL/SQL package or special Oracle Call Interface (OCI) calls. MODULE should be set to a name that is recognizable by the user for the program that currently executes. Likewise, ACTION should be set to a specific action or task that a user performs within a module (for example, entering a new customer). CLIENT_IDENTIFIER can be set using the DBMS_SESSION.SET_IDENTIFIER procedure.

The traditional method of tracing each session produces trace files with SQL commands that may contain the trace information for multiple end users or applications. Unless all database sessions are being traced, some information from the end user sessions may be missed. This results in a hit-or-miss approach to diagnose problematic SQL.

With the criteria that you provide (SERVICE_NAME, MODULE, or ACTION), specific trace information is captured in a set of trace files and combined into a single output trace file. This enables you to produce trace files that contain SQL that is relevant to a specific workload. It is also possible to do the same for CLIENT_IDs and SESSION_IDs.

Note: DBA_ENABLED_TRACES displays information about enabled traces.

Use Enterprise Manager to Trace Services



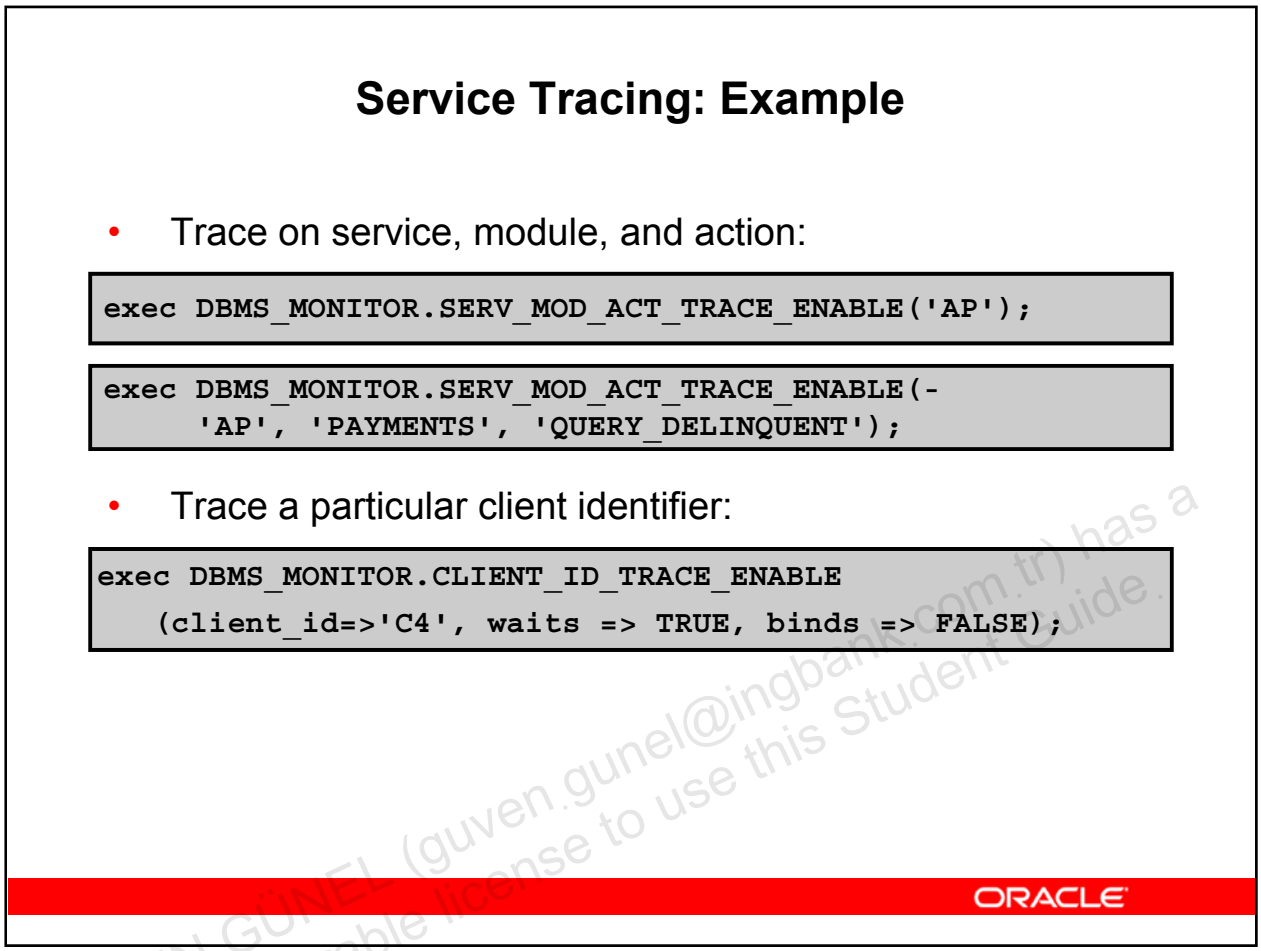
Use Enterprise Manager to Trace Services

On the Performance page, you can click the Top Consumers link. The Top Consumers page is displayed.

The Top Consumers page has several tabs for displaying your database as a single-system image. The Overview tabbed page contains four pie charts: Top Clients, Top Services, Top Modules, and Top Actions. Each chart provides a different perspective about the top resource consumers in your database.

The Top Services tabbed page displays performance-related information for the services that are defined in your database. On this page, you can enable or disable tracing at the service level.

Service Tracing: Example



Service Tracing: Example

In the first code box, all sessions that log in under the `AP` service are traced. A trace file is created for each session that uses the service, regardless of the module and action. You can also enable tracing for specific tasks within a service. This is illustrated in the second example, where all sessions of the `AP` service that execute the `QUERY_DELINQUENT` action within the `PAYMENTS` module are traced.

Tracing by service, module, and action enable you to focus your tuning efforts on specific SQL, rather than sifting through trace files with SQL from different programs. Only the SQL statements that are identified with this `MODULE` and `ACTION` are recorded in the trace file. With this feature, relevant wait events for a specific action can be identified.

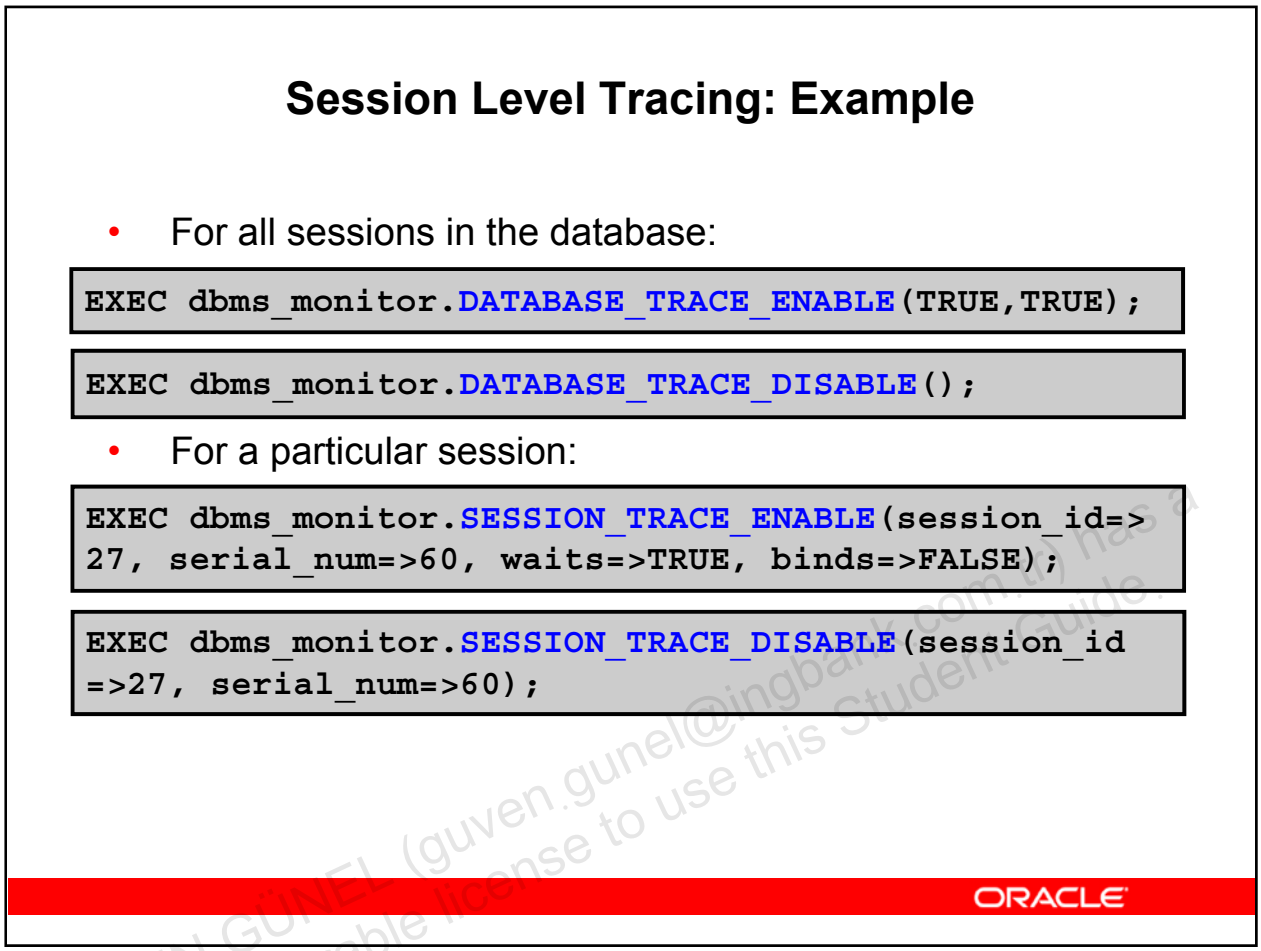
You can also start tracing for a particular client identifier as shown by the third example. In this example, `C4` is the client identifier for which SQL tracing is to be enabled. The `TRUE` argument specifies that wait information is present in the trace file. The `FALSE` argument specifies that bind information is not present in the trace file.

Although not shown in the slide, you can use the `CLIENT_ID_TRACE_DISABLE` procedure to disable tracing globally for the database for a given client identifier. To disable tracing, for the previous example, execute the following command:

```
EXECUTE DBMS_MONITOR.CLIENT_ID_TRACE_DISABLE(client_id => 'C4');
```

Note: `CLIENT_IDENTIFIER` can be set using the `DBMS_SESSION.SET_IDENTIFIER` procedure.

Session Level Tracing: Example



Session Level Tracing: Example

You can use tracing to debug performance problems. Trace-enabling procedures have been implemented as part of the `DBMS_MONITOR` package. These procedures enable tracing globally for a database.

You can use the `DATABASE_TRACE_ENABLE` procedure to enable session level SQL tracing instance-wide. The procedure has the following parameters:

- `WAITS`: Specifies whether wait information is to be traced
- `BINDS`: Specifies whether bind information is to be traced
- `INSTANCE_NAME`: Specifies the instance for which tracing is to be enabled. Omitting `INSTANCE_NAME` means that the session-level tracing is enabled for the whole database.

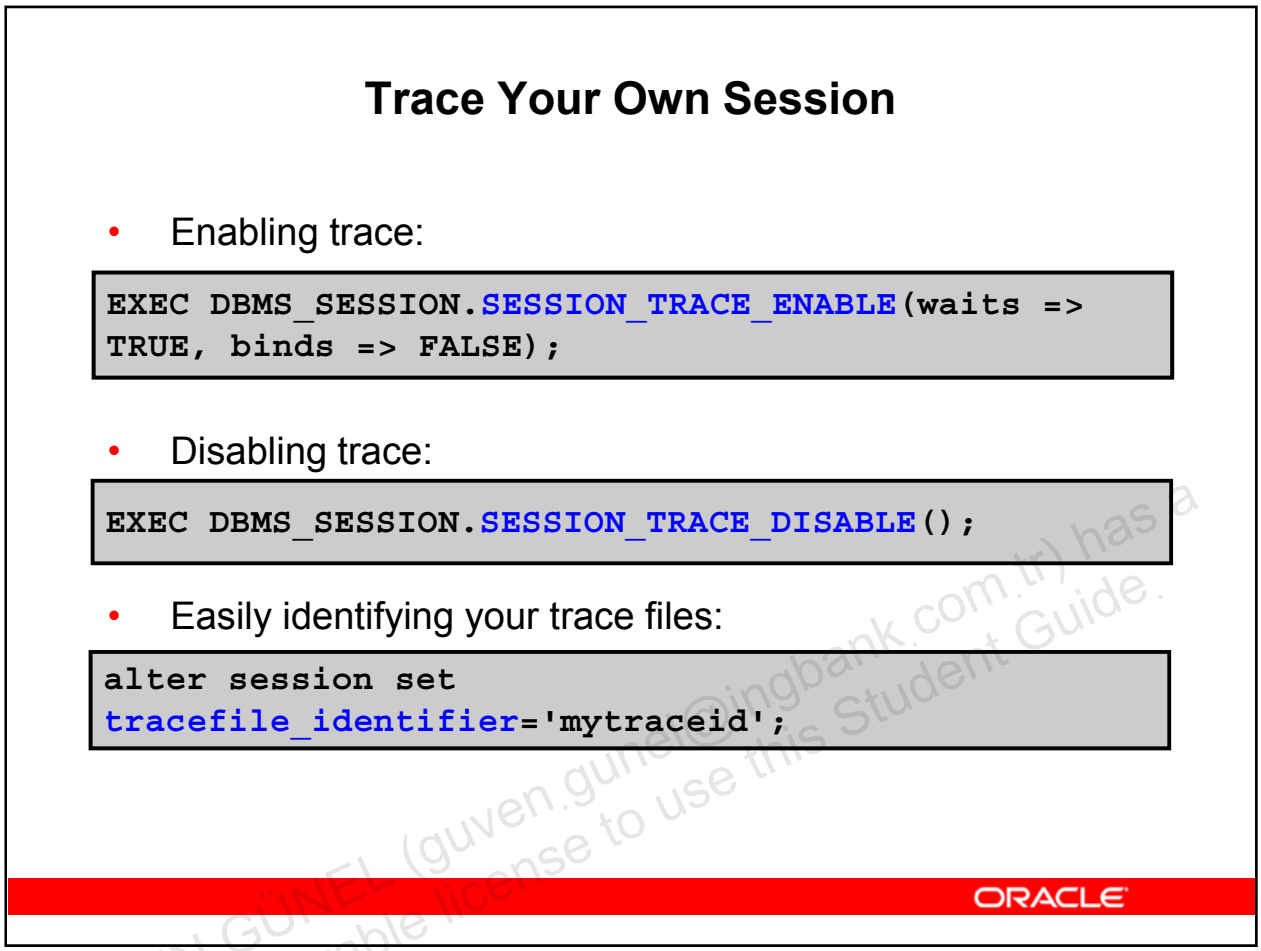
Use the `DATABASE_TRACE_DISABLE` procedure to disable SQL tracing for the whole database or a specific instance.

Similarly, you can use the `SESSION_TRACE_ENABLE` procedure to enable tracing for a given database session identifier on the local instance. The `SID` and `SERIAL#` information can be found from `V$SESSION`.

Use the `SESSION_TRACE_DISABLE` procedure to disable the trace for a given database session identifier and serial number.

Note: SQL Trace involves some overhead, so you usually do not want to enable SQL Trace at the instance level.

Trace Your Own Session



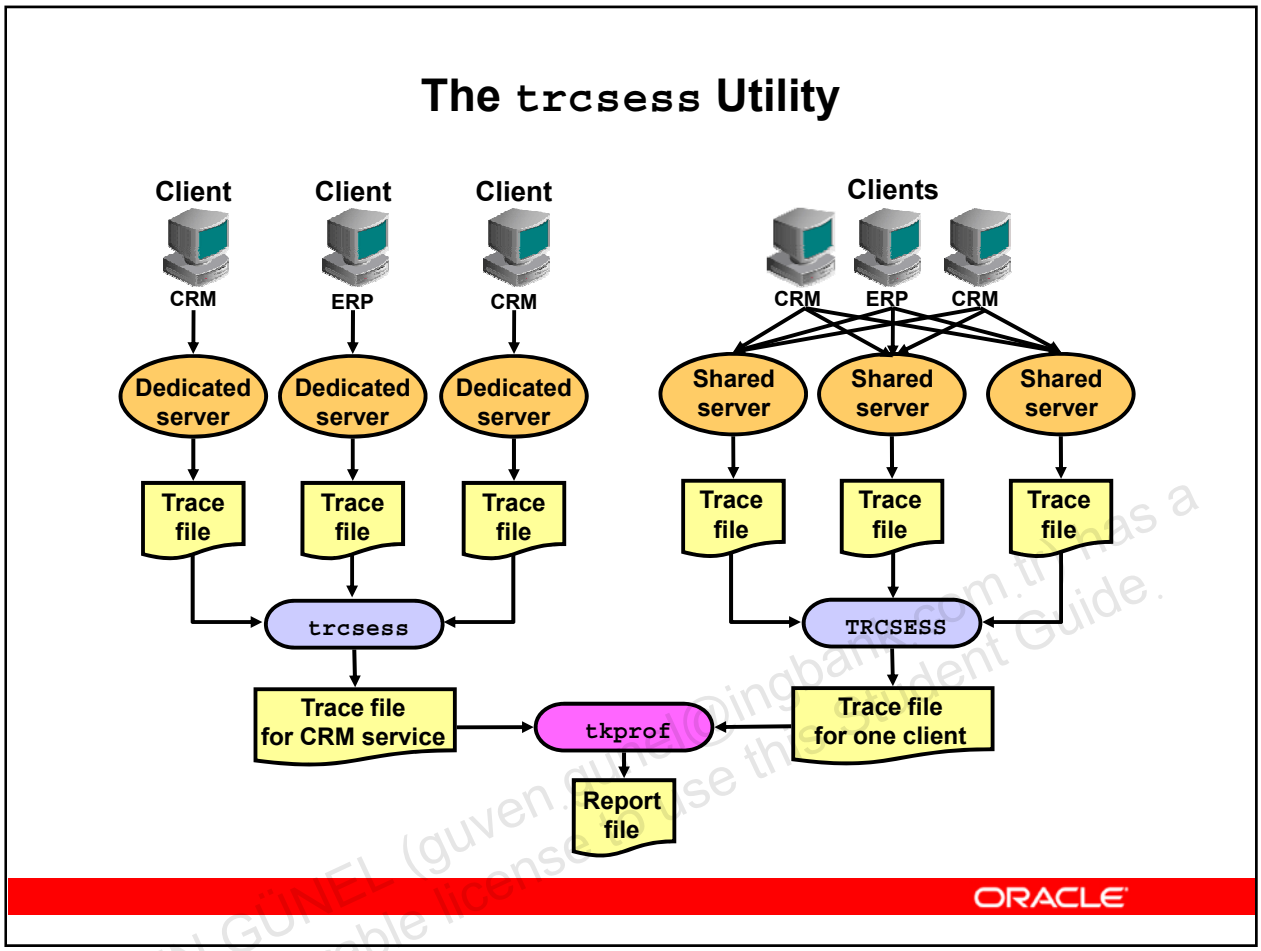
Trace Your Own Session

Although the DBMS_MONITOR package can be invoked only by a user with the DBA role, any user can enable SQL tracing for his or her own session by using the DBMS_SESSION package. The SESSION_TRACE_ENABLE procedure can be invoked by any user to enable session level SQL tracing for his or her own session. An example is shown in the slide.

You can then use the DBMS_SESSION.SESSION_TRACE_DISABLE procedure to stop dumping to your trace file.

The TRACEFILE_IDENTIFIER initialization parameter specifies a custom identifier that becomes part of the Oracle trace file name. You can use such a custom identifier to identify a trace file simply from its name and without opening it or view its contents. Each time this parameter is dynamically modified at the session level, the next trace dump written to a trace file will have the new parameter value embedded in its name. This parameter can only be used to change the name of the foreground process trace file; the background processes continue to have their trace files named in the regular format. For foreground processes, the TRACEID column of the V$PROCESS view contains the current value of this parameter. When this parameter value is set, the trace file name has the following format: sid_ora_pid_traceid.trc.

The trcsess Utility



The trcsess Utility

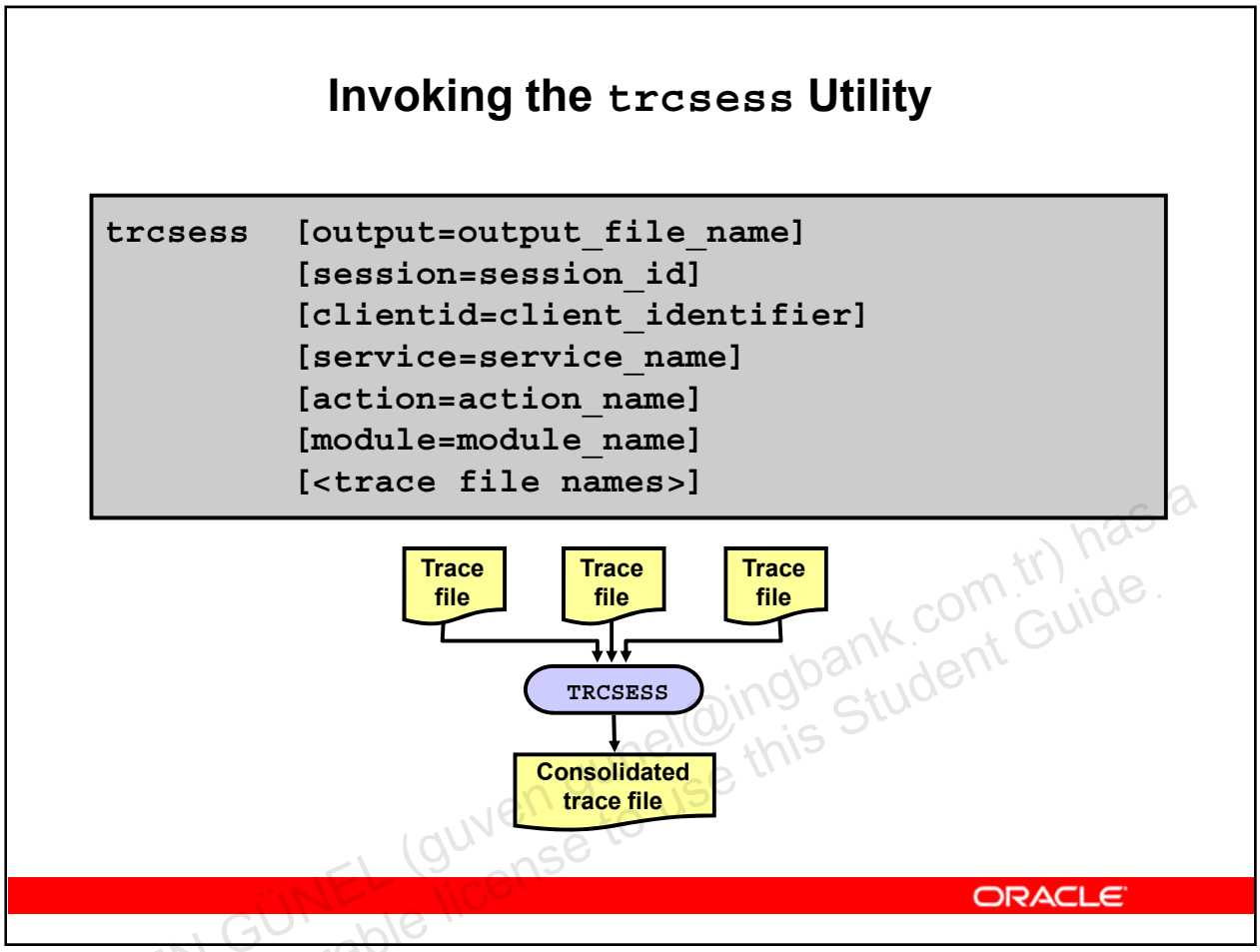
The `trcsess` utility consolidates trace output from selected trace files on the basis of several criteria: session ID, client identifier, service name, action name, and module name. After `trcsess` merges the trace information into a single output file, the output file can be processed by `tkprof`.

When using the `DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE` procedure, tracing information is present in multiple trace files and you must use the `trcsess` tool to collect it into a single file.

The `trcsess` utility is useful for consolidating the tracing of a particular session or service for performance or debugging purposes.

Tracing a specific session is usually not a problem in the dedicated server model because a single dedicated process serves a session during its lifetime. All the trace information for the session can be seen from the trace file belonging to the dedicated server that serves it. However, tracing a service might become a complex task even in the dedicated server model.

Moreover, in a shared-server configuration, a user session is serviced by different processes from time-to-time. The trace pertaining to the user session is scattered across different trace files belonging to different processes. This makes it difficult to get a complete picture of the life cycle of a session.



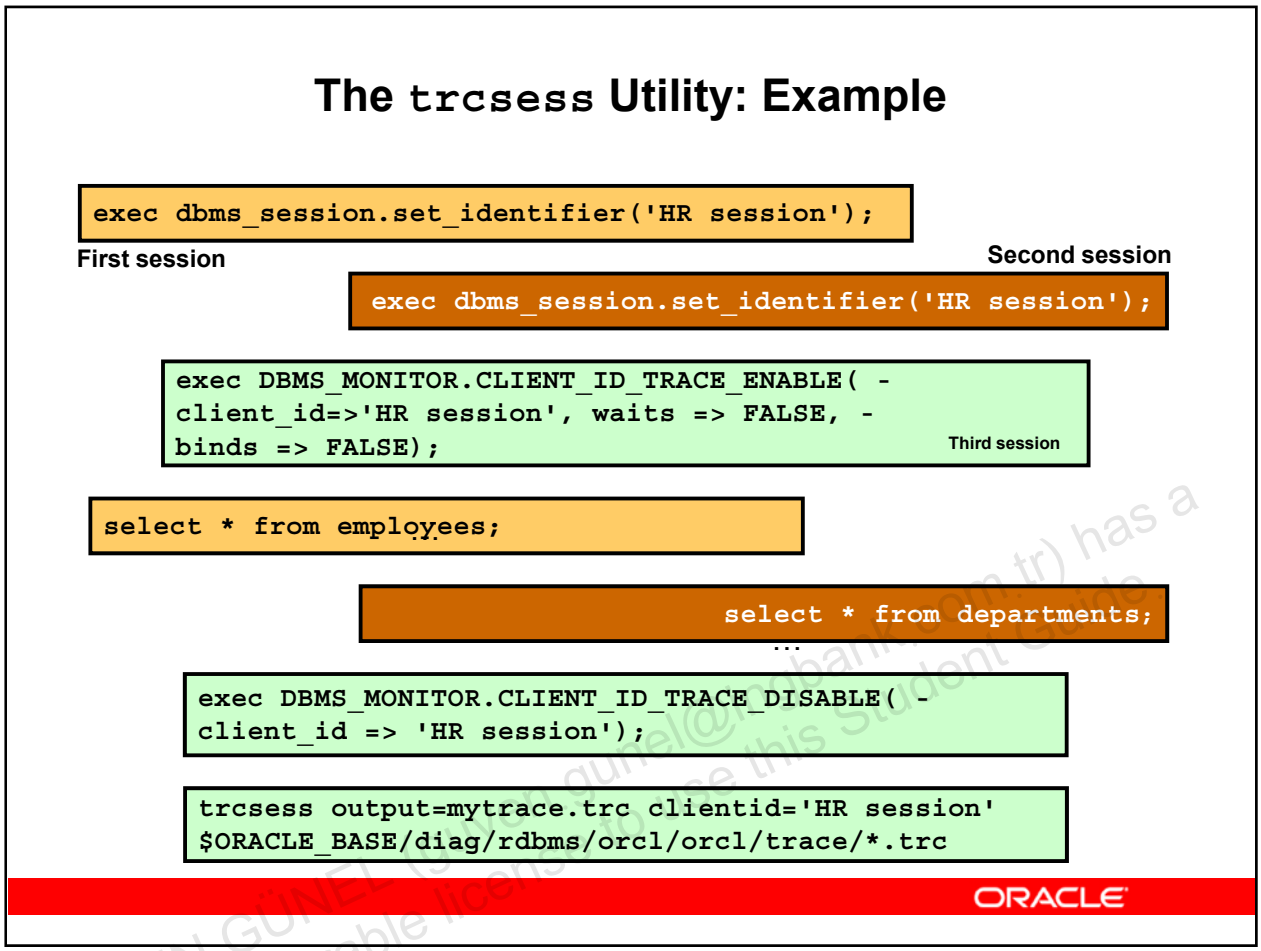
Invoking the `trcsess` Utility

The syntax for the `trcsess` utility is shown in the slide, where:

- `output` specifies the file where the output is generated. If this option is not specified, standard output is used for the 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 given client identifier.
- `service` consolidates the trace information for the given service name.
- `action` consolidates the trace information for the given action name.
- `module` consolidates the trace information for the given module name.
- `<trace file names>` is a list of all the trace file names, separated by spaces, in which `trcsess` should look for trace information. The wildcard character "*" can be used to specify the trace file names. If trace files are not specified, all the files in the current directory are taken as input to `trcsess`. You can find trace files in ADR.

Note: One of the `session`, `clientid`, `service`, `action`, or `module` options must be specified. If there is more than one option specified, the trace files, which satisfy all the criteria specified are consolidated into the output file.

The trcsess Utility: Example



The trcsess Utility: Example

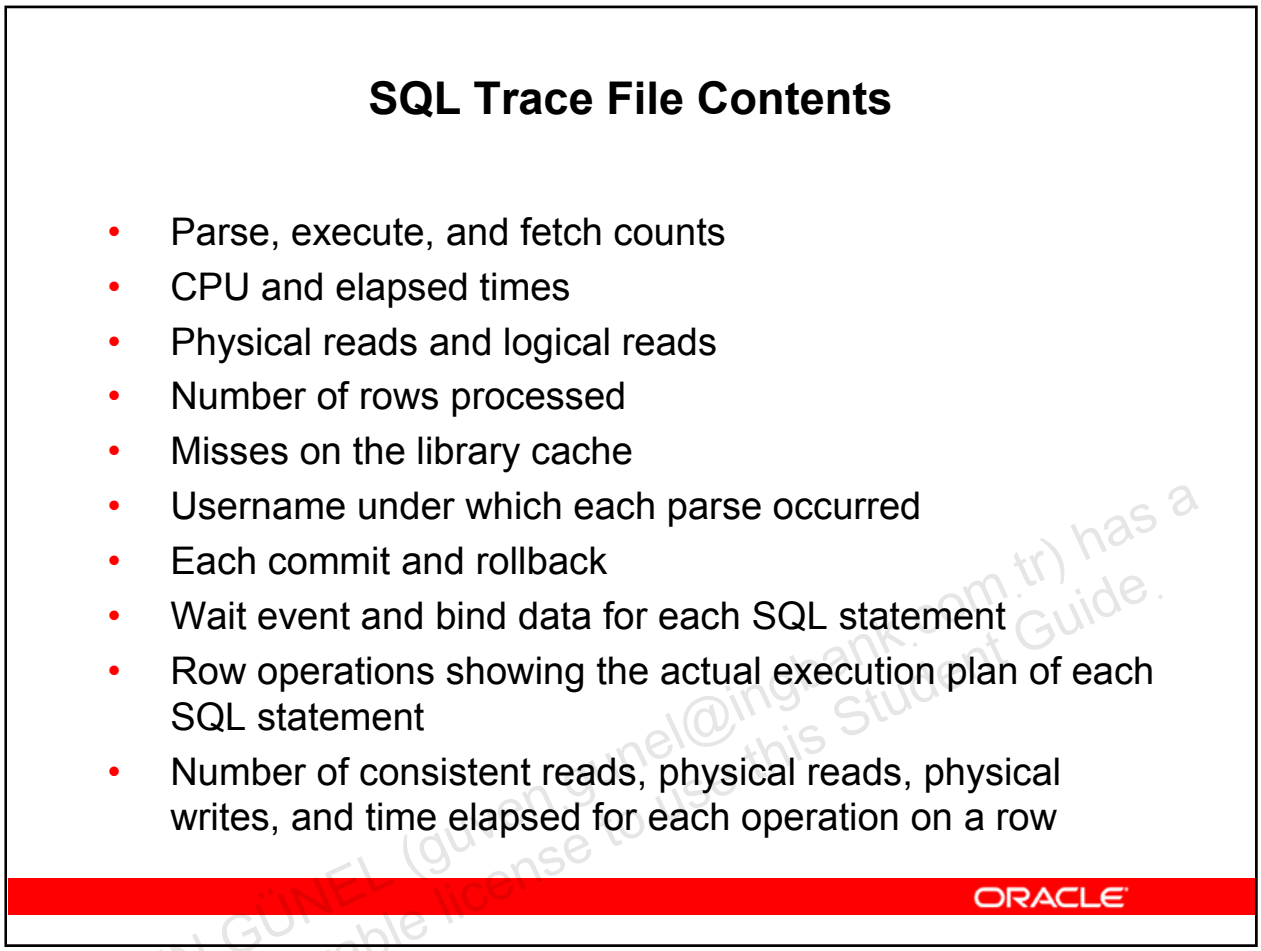
The example in the slide illustrates a possible use of the `trcsess` utility. The example assumes that you have three different sessions: Two sessions that are traced (left and right), and one session (center) that enables or disables tracing and concatenates trace information from the previous two sessions.

The first and second session set their client identifier to the `'HR session'` value. This is done using the `DBMS_SESSION` package. Then, the third session enables tracing for these two sessions using the `DBMS_MONITOR` package.

At that point, two new trace files are generated in ADR; one for each session that is identified with the `'HR session'` client identifier.

Each traced session now executes its SQL statements. Every statement generates trace information in its own trace file in ADR.

Then, the third session stops trace generation using the `DBMS_MONITOR` package, and consolidates trace information for the `'HR session'` client identifier in the `mytrace.trc` file. The example assumes that all trace files are generated in the `$ORACLE_BASE/diag/rdbms/orcl/orcl/trace` directory, which is the default in most cases.



SQL Trace File Contents

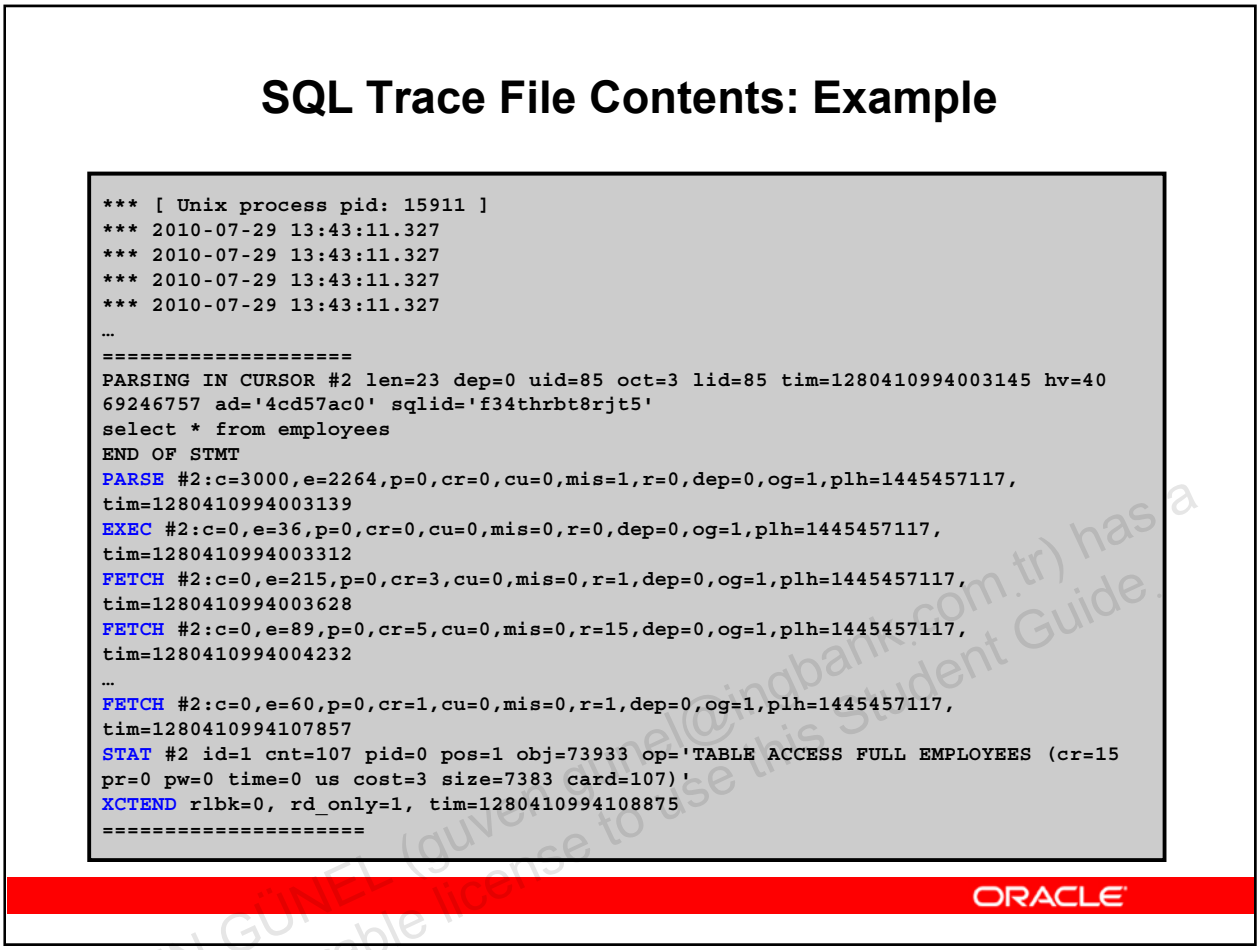
As seen already, a SQL trace file provides performance information on individual SQL statements. It generates the following statistics for each statement:

- Parse, execute, and fetch counts
- CPU and elapsed times
- Physical reads and logical reads
- Number of rows processed
- Misses on the library cache
- Username 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, 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. This is possible only when the `STATISTICS_LEVEL` initialization parameter is set to `ALL`.

Note: Using the SQL Trace facility can have a severe performance impact and may result in increased system overhead, excessive CPU usage, and inadequate disk space.



SQL Trace File Contents: Example

There are multiple types of trace files that can be generated by the Oracle Database. The one that is referred to in this lesson is generally called a SQL trace file. The slide shows you a sample output from the `mytrace.trc` SQL trace file generated by the previous example.

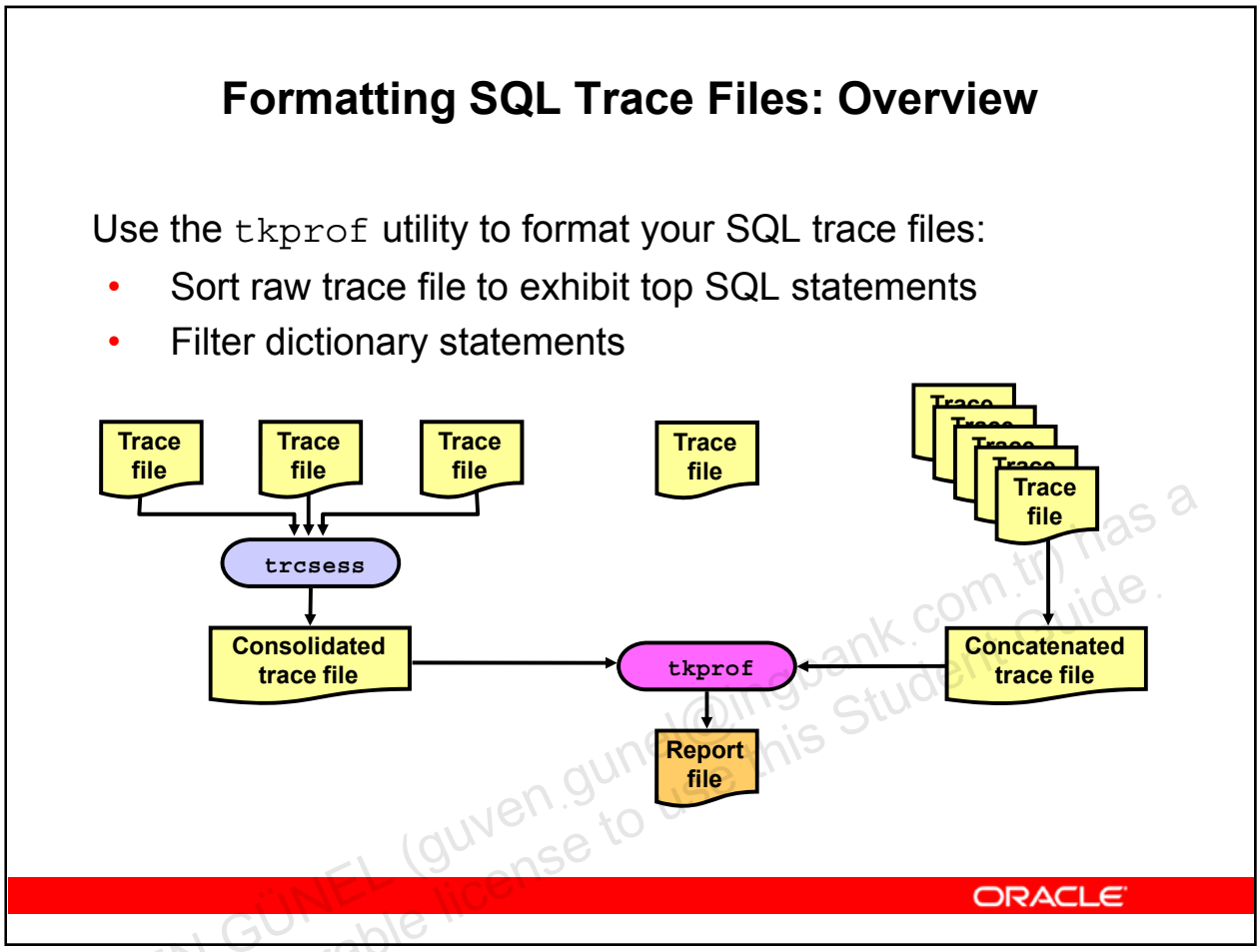
In this type of trace file, you can find (for each statement that was traced) the statement itself, with some corresponding cursor details. You can see statistic details for each phase of the statement's execution: `PARSE`, `EXEC`, and `FETCH`. As you can see, you can have multiple `FETCH` for one `EXEC` depending on the number of rows returned by your query.

Last part of the trace is the execution plan with some cumulated statistics for each row source.

Depending on the way you enabled tracing, you can also obtain information about wait events and bind variables in the generated trace files.

Generally, you do not try to interpret the trace file itself. This is because you do not get an overall idea of what your sessions did. For example, one session could have executed the same statement multiple times at different moments. The corresponding traces are then scattered across the entire trace file, which makes them hard to find.

Instead, you use another tool, such as `tkprof` to interpret the contents of the raw trace information.



Formatting SQL Trace Files: Overview

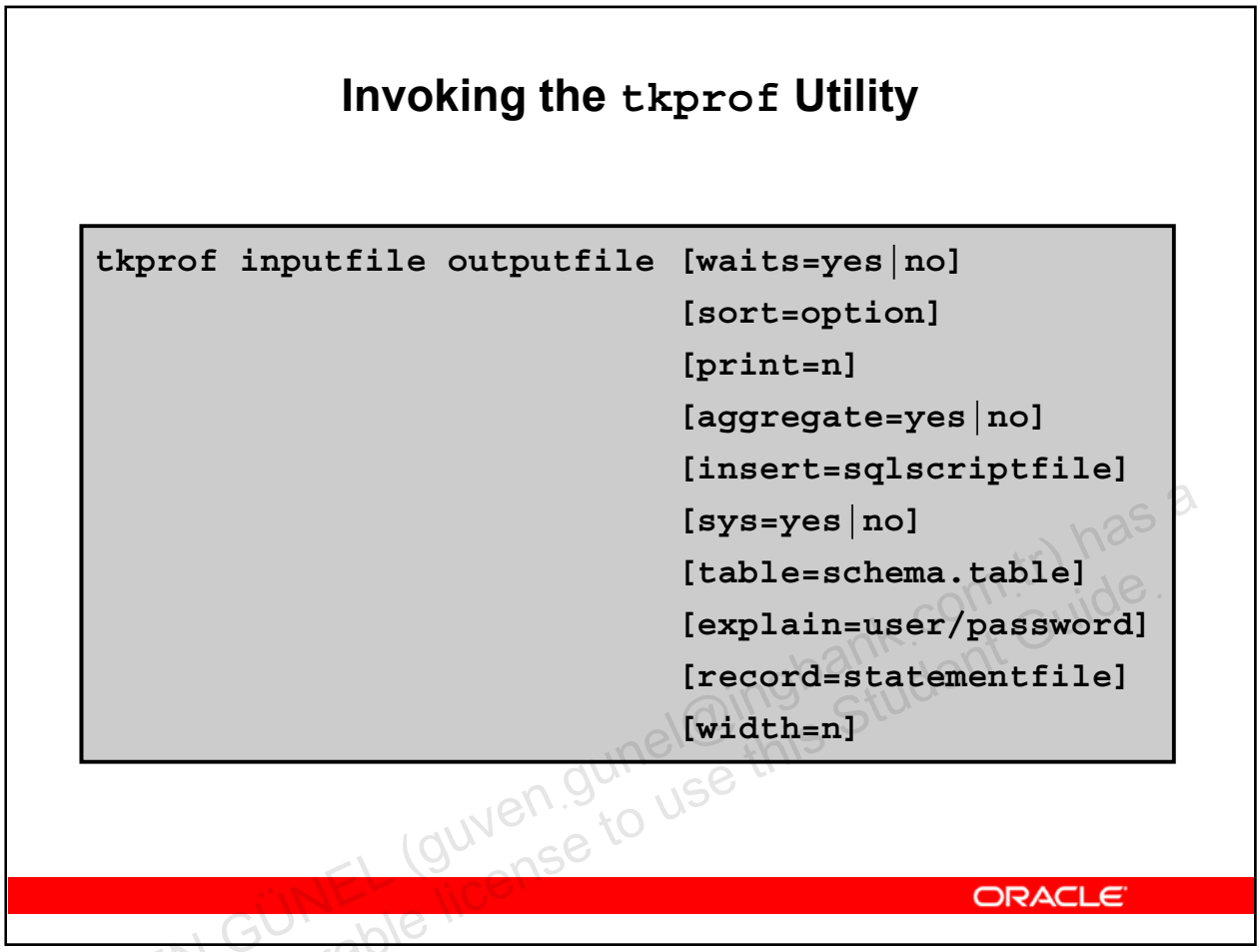
The `tkprof` utility parses SQL trace files to produce more readable output. Remember that all the information in `tkprof` is available from the raw trace file. There is a huge number of sort options that you can invoke with `tkprof` at the command prompt. A useful starting point is the `fchela` sort option, which orders the output by elapsed time fetching. The resultant file contains the most time-consuming SQL statement at the start of the file. Another useful parameter is `SYS=NO`. This can be used to prevent SQL statements run as the `SYS` user from being displayed. This can make the output file much shorter and easier to manage.

After a number of SQL trace files have been generated, you can perform any of the following:

- Run `tkprof` on each individual trace file, producing a number of 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`s and `ROLLBACK`s that are recorded in the trace file.

Note: Set the `TIMED_STATISTICS` parameter to `TRUE` when tracing sessions because no time-based comparisons can be made without this. `TRUE` is the default value with Oracle Database 11*g*.



Invoking the tkprof Utility

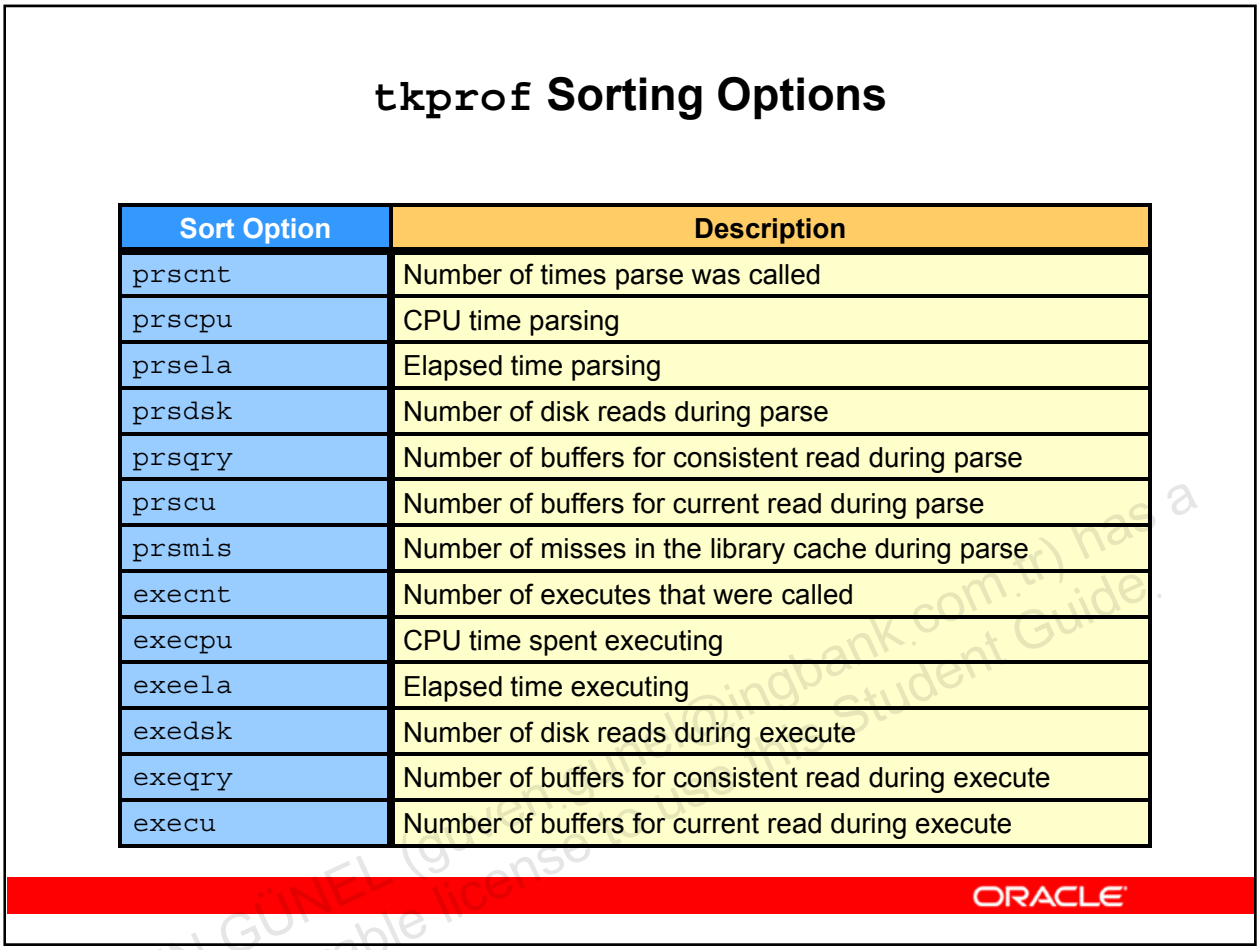
When you enter the `tkprof` command without any arguments, it generates a usage message together with a description of all `tkprof` options. The various arguments are shown in the slide:

- **`inputfile`:** Specifies the SQL trace input file
- **`outputfile`:** Specifies the file to which `tkprof` writes its formatted output
- **`waits`:** Specifies whether to record the summary for any wait events found in the trace file. Values are `YES` or `NO`. The default is `YES`.
- **`sorts`:** Sorts traced SQL statements in the descending order of specified sort option before listing them into the output file. If more than one option is specified, the output is sorted in the descending order by the sum of the values specified in the sort options. If you omit this parameter, `tkprof` lists statements into the output file in the order of first use.
- **`print`:** Lists only the first integer sorted SQL statements from the output file. If you omit this parameter, `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 set to NO, tkprof does not aggregate multiple users of the same SQL text.
- **insert:** Creates a SQL script to store the trace file statistics in the database. tkprof creates this script with the name you specify for sqlscriptfile. 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 SYS user, 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:** 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 already exists, 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, 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 already exist, the user must also be able to issue the CREATE TABLE and DROP TABLE statements. This option allows multiple individuals to run tkprof concurrently with the same user in the EXPLAIN value. These individuals can specify different TABLE values and avoid destructively interfering with each other's processing on the temporary plan table. If you use the EXPLAIN parameter without the TABLE parameter, tkprof uses the PROF$PLAN_TABLE table in the schema of the user specified by the EXPLAIN parameter. If you use the TABLE parameter without the EXPLAIN parameter, tkprof ignores the TABLE parameter. If no plan table exists, tkprof creates the PROF$PLAN_TABLE 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 determines execution plans by issuing the EXPLAIN PLAN statement after connecting to the system 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.
- **record:** Creates a SQL script with the specified file name statementfile with all the nonrecursive SQL statements in the trace file. This can be used 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.

The input and output files are the only required arguments.

tkprof Sorting Options



tkprof Sorting Options

The table lists all the sort options you can use with the sort argument of `tkprof`.

tkprof Sorting Options

`tkprof` Sorting Options

Sort Option	Description
`exerow`	Number of rows processed during execute
`exemis`	Number of library cache misses during execute
`fchcnt`	Number of times fetch was called
`fchcpu`	CPU time spent fetching
`fchela`	Elapsed time fetching
`fchdsk`	Number of disk reads during fetch
`fchqry`	Number of buffers for consistent read during fetch
`fchcu`	Number of buffers for current read during fetch
`fchrow`	Number of rows fetched
`userid`	User ID of user that parsed the cursor

Output of the tkprof Command

Output of the `tkprof` Command

- Text of the SQL statement
- Trace statistics (for statement and recursive calls) separated into three SQL processing steps:

PARSE	Translates the SQL statement into an execution plan
EXECUTE	Executes the statement (This step modifies the data for the `INSERT`, `UPDATE`, and `DELETE` statements.)
FETCH	Retrieves the rows returned by a query (Fetches are performed only for the `SELECT` statements.)

ORACLE

Output of the tkprof Command

The tkprof output file lists the statistics for a SQL statement by the SQL processing step. The step for each row that contains statistics is identified by the value of the call column.

PARSE — This step translates the SQL statement into an execution plan and includes checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.

EXECUTE — This step is the actual execution of the statement by the Oracle server. For the INSERT, UPDATE, and DELETE statements, this step modifies the data (including sorts when needed). For the SELECT statements, this step identifies the selected rows.

FETCH — This step retrieves rows returned by a query and sorts them when needed. Fetches are performed only for the SELECT statements.

Note: The PARSE value includes both hard and soft parses. A hard parse refers to the development of the execution plan (including optimization); it is subsequently stored in the library cache. A soft parse means that a SQL statement is sent for parsing to the database, but the database finds it in the library cache and only needs to verify things, such as access rights. Hard parses can be expensive, particularly due to the optimization. A soft parse is mostly expensive in terms of library cache activity 25

Output of the `tkprof` Command

There are seven categories of trace statistics:

Count	Number of times the procedure was executed
CPU	Number of seconds to process
Elapsed	Total number of seconds to execute
Disk	Number of physical blocks read
Query	Number of logical buffers read for consistent read
Current	Number of logical buffers read in current mode
Rows	Number of rows processed by the fetch or execute

ORACLE

Output of the `tkprof` Command (continued)

The output is explained on the following page.

Sample output is as follows:

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ---
Parse        1      0.03       0.06          0          0          0      0
Execute      1      0.06       0.30          1          3          0      0
Fetch        2      0.00       0.46          0          0          0      1
------- ------  -------- ---------- ---------- ---------- ----------  ---
total        4      0.09       0.83          1          3          0      1
```

Next to the `CALL` column, `tkprof` displays the following statistics for each statement:

- **`Count`:** Number of times a statement was parsed, executed, or fetched (Check this column for values greater than 1 before interpreting the statistics in the other columns. Unless the `AGGREGATE = NO` option is used, `tkprof` aggregates identical statement executions into one summary table.)
- **`CPU`:** Total CPU time in seconds for all parse, execute, or fetch calls
- **`Elapsed`:** Total elapsed time in seconds for all parse, execute, or fetch calls

- **`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 (Buffers are usually retrieved in consistent mode for queries.)
- **`Current`:** Total number of buffers retrieved in current mode (Buffers typically are retrieved in current mode for data manipulation language statements. However, segment header blocks are always retrieved in current mode.)
- **`Rows`:** Total 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 the `UPDATE`, `DELETE`, and `INSERT` statements, the number of rows processed appears for the execute step.)

Note

- `DISK` is equivalent to `physical reads` from `v$sysstat` or `AUTOTRACE`.
- `QUERY` is equivalent to `consistent gets` from `v$sysstat` or `AUTOTRACE`.
- `CURRENT` is equivalent to `db block gets` from `v$sysstat` or `AUTOTRACE`.

Output of the `tkprof` Command

The `tkprof` output also includes the following:

- Recursive SQL statements
- Library cache misses
- Parsing user ID
- Execution plan
- Optimizer mode or hint
- Row source operation

```
...
Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 85

Rows     Row Source Operation
-------  ---------------------------------------------------
      5  TABLE ACCESS BY INDEX ROWID EMPLOYEES (cr=4 pr=1 pw=0 time=0 us …
      5   INDEX RANGE SCAN EMP_NAME_IX (cr=2 pr=1 pw=0 time=80 us cost=1 …
…
```

ORACLE

Output of the `tkprof` Command (continued)

Recursive Calls

To execute a SQL statement issued by a user, the Oracle server must occasionally issue additional statements. Such statements are called *recursive SQL statements*. For example, if you insert a row in a table that does not have enough space to hold that row, the Oracle server makes recursive calls to allocate the space dynamically. Recursive calls are also generated when data dictionary information is not available in the data dictionary cache and must be retrieved from disk.

If recursive calls occur while the SQL Trace facility is enabled, `tkprof` marks them clearly as recursive SQL statements in the output file. You can suppress the listing of recursive calls in the output file by setting the `SYS=NO` command-line parameter. Note that the statistics for recursive SQL statements are always included in the listing for the SQL statement that caused the recursive call.

Library Cache Misses

`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.

Row Source Operations

These provide the number of rows processed for each operation executed on the rows and additional row source information, such as physical reads and writes; cr = consistent reads, w = physical writes, r = physical reads, time = time (in microseconds).

Parsing User ID

This is the ID of the last user to parse the statement.

Row Source Operation

The row source operation shows the data sources for execution of the SQL statement. This is included only if the cursor has been closed during tracing. If the row source operation does not appear in the trace file, you may then want to view the output of the `EXPLAIN PLAN`.

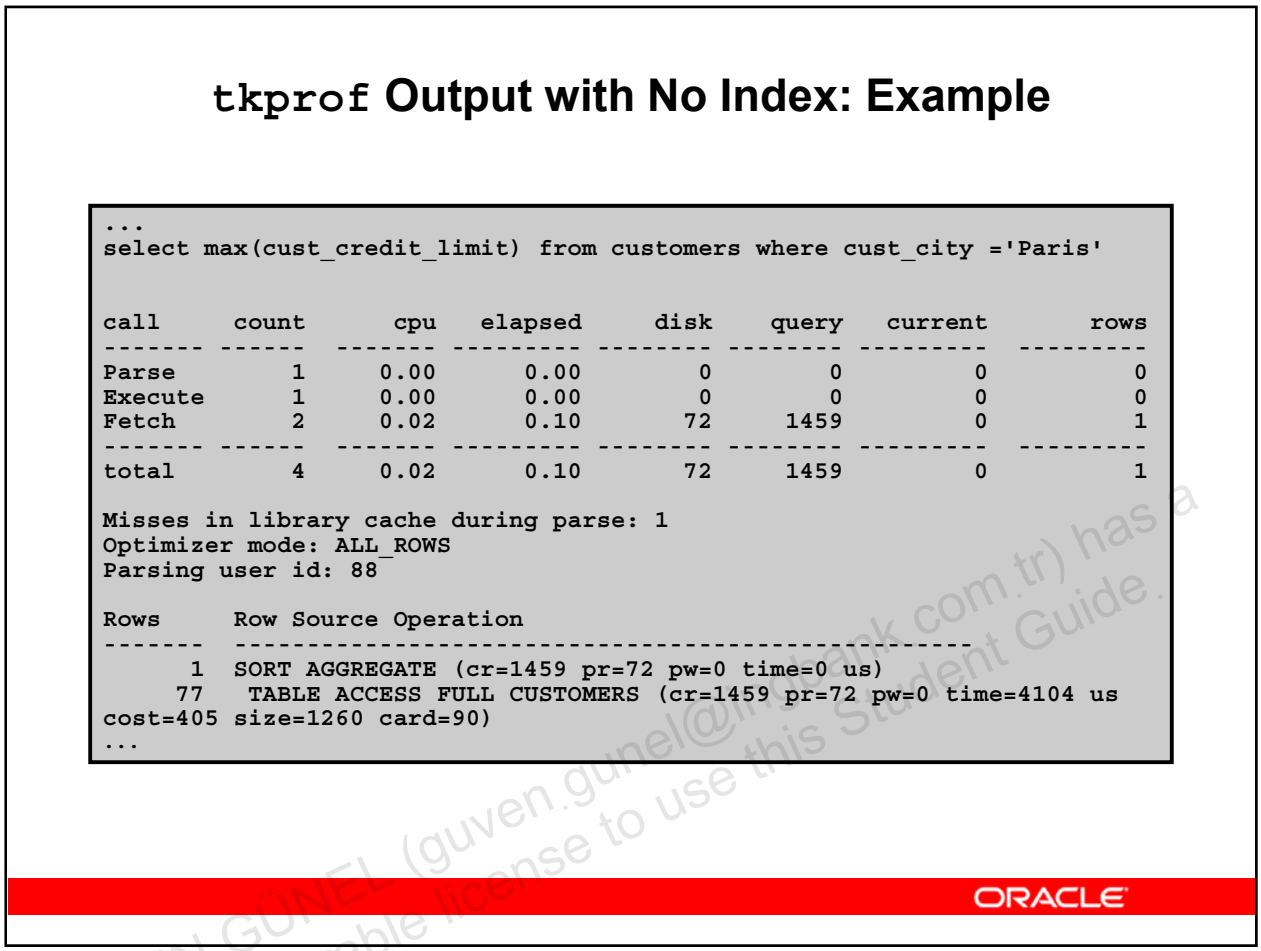
Execution Plan

If you specify the `EXPLAIN` parameter on the `tkprof` command line, `tkprof` uses the `EXPLAIN PLAN` command to generate the execution plan of each SQL statement traced. `tkprof` also displays the number of rows processed by each step of the execution plan.

Note: Be aware that the execution plan is generated at the time that the `tkprof` command is run and not at the time the trace file was produced. This could make a difference if, for example, an index has been created or dropped since tracing the statements.

Optimizer Mode or Hint

This indicates the optimizer hint that is used during the execution of the statement. If there is no hint, it shows the optimizer mode that is used.

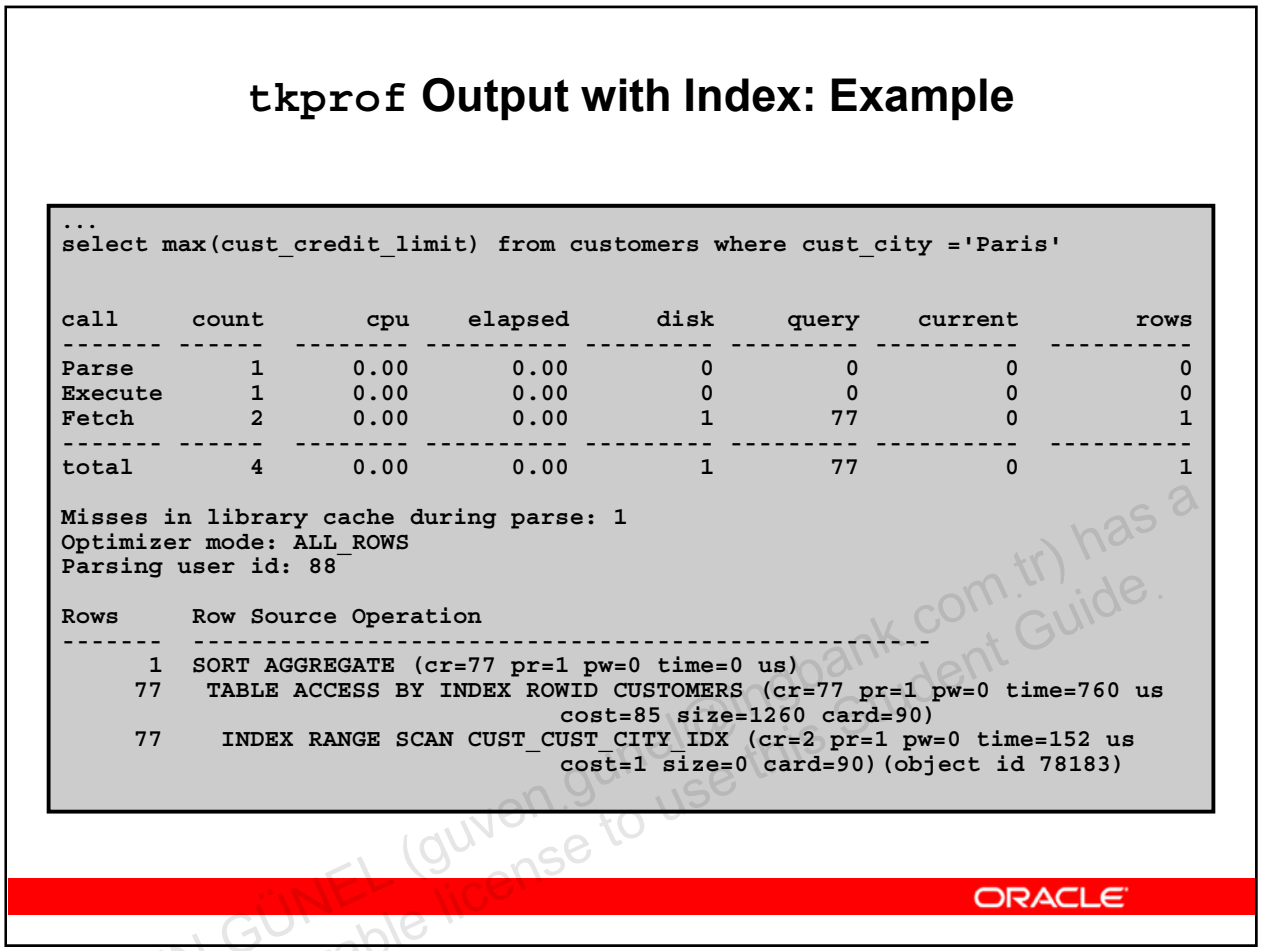


tkprof Output with No Index: Example

The example in the slide shows that the aggregation of results across several executions (rows) is being fetched from the CUSTOMERS table. It requires 0.12 second of CPU fetch time. The statement is executed through a full table scan of the CUSTOMERS table, as you can see in the row source operation of the output.

The statement must be optimized.

Note: If CPU or elapsed values are 0, timed_statistics is not set.



tkprof Output with Index: Example

The results shown in the slide indicate that CPU time was reduced to 0.01 second when an index was created on the CUST_CITY column. These results may have been achieved because the statement uses the index to retrieve the data. Additionally, because this example reexecutes the same statement, most of the data blocks are already in memory. You can achieve significant improvements in performance by indexing sensibly. Identify areas for potential improvement using the SQL Trace facility.

Note: Indexes should not be built unless required. Indexes do slow down the processing of the INSERT, UPDATE, and DELETE commands because references to rows must be added, changed, or removed. Unused indexes should be removed. However, instead of processing all the application SQL through EXPLAIN PLAN, you can use index monitoring to identify and remove any indexes that are not used.

Quiz

Which command would you use to create a trace file of your SQL*Plus session in a dedicated server environment?

a. `alter session set tracefile_identifier='mytraceid';`

b. `EXEC DBMS_SESSION.SESSION_TRACE_ENABLE(waits => TRUE, binds => FALSE);`

c. `trcsess output=mytrace.trc clientid='HR session' $ORACLE_BASE/diag/rdbms/orcl/orcl/trace/*.trc`

d. `tkprof *mytrace*.trc mytrace.txt SYS=NO`

ORACLE

Answer: b

Quiz

The _____ utility formats the trace file into a readable format.

a. `trcsess`
b. `tkprof`
c. SQL Developer
d. SQL*Plus Autotrace

Answer: b, c

Quiz

In an environment with an applications server that uses a connection pool, you will use _____ to identify which trace files need to be combined to get an overall trace of the application.

a. `trcsess`
b. `tkprof`
c. SQL Developer
d. DBMS_APPLICATION_INFO

Answer: d

Summary

In this lesson, you should have learned how to:

- Configure the SQL Trace facility to collect session statistics
- Use the `trcsess` utility to consolidate SQL trace files
- Format trace files using the `tkprof` utility
- Interpret the output of the `tkprof` command

ORACLE

Practice 5: Overview

This practice covers the following topics:

- Creating a service
- Tracing your application using services
- Interpreting trace information using `trcsess` and `tkprof`

ORACLE