

# SQL Tuning for DBAs: *Reactive Performance Tuning*

## Stalking the wily Resource Hog

Tim Gorman

Principal - Evergreen Database Technologies, Inc (<http://www.evergreen-database.com>)

### Abstract

Every database administrator's favorite way to start a day is to hear the cry:

*The database is slow! Fix it!*

It is as if people believe that an Oracle database administrator has special *init.ora* parameters like:

```
_OVERCOME_POOR_DESIGN = TRUE
```

or:

```
_DISABLE_BAD_PERFORMANCE = TRUE
```

Most DBAs do not have a magic bullet. There are some *init.ora* parameters that can be tweaked to increase the size of different parts of the Oracle instance's System Global Area (SGA), such as the Buffer Cache, the Shared Pool, and the Redo Log Buffer. Making each of these data structures larger can improve system performance, due to the effects of caching I/O in shared memory. But there is quite clearly a *point of diminishing returns* with any solution involving caching, and at some point database performance would still be poor even if the entire database were cached in the SGA.

Every computer, no matter how large or small, has only so many operations that it can perform in a certain slice of time. In other words, no matter how many MIPS (to randomly choose a common metric) are at your disposal, there is still a finite amount of them. If the application programs have not been optimised, then each program will consume an enormous number of MIPS, relative to what they should be consuming. However, large-scale computer systems are capable of performing such prodigious quantities of work that, when a small number of excessively resource-intensive processes are executed, nobody notices that they were slow. If response time is 2 seconds instead of 0.02 seconds, most human beings will not find fault. This is the classic situation encountered during development; the quantity of data tested against is small and the overall load on the system is light or non-existent. As a result, the

operation does not appear to be slow or, if it does, it may not be considered worth the effort to improve it since this particular operation may not be executed often.

However, everything changes when this code hits a heavily-loaded production system. All of a sudden, instead of running in an environment that had tons of MIPS to spare, there is no longer any *headroom*. The task is now executing in an environment where any resource-intensive process can overwhelm the scant remaining MIPS on a system.

In such a situation as this, system administrators and database administrators should not be looking for ways to further increase the capacity of the machine, the operating system, or the database instance. Taking this approach is likely to yield only incremental benefits at best, since the problem is being *accommodated* instead of being *fixed*. It is not a lack of capacity that is causing the problem, rather it is a resource-intensive process, a *resource hog*, that is using too many resources. Thus, increasing resource capacity only temporarily postpones the crisis. Just like the use of appeasement in international diplomacy, attempting to tune subsystems or the database instance in an effort to increase capacity only encourages the beast to demand more.

Thus, it should be understood that system administrators and database administrators can only do so much to a system in order to correct intermittent performance problems. If system capacity is not reasonable, then certainly it should be improved. Rather than looking for fault in the system or in the database instance, critical eyes should first be turned to the database application, to determine whether it is simply performing *too much work* in order to accomplish its tasks.

What is *reasonable system capacity*? What is *too much work*? All of these phrases are subjective, dependent on your particular environment. I won't attempt to quantify them here.

Rather, I will argue that if you are not currently using tools, tips, and techniques similar to those described in this paper, then there is a good chance that your system is suffering extensive performance problems due to *resource hogs*. Your Oracle database application would be far better served by killing them instead of searching for ways to accommodate them.

This paper provides a methodology for Oracle database administrators (DBAs) to detect such *resource hogs*. Once they have been identified, then they must be prioritized so that scant resources are not wasted on operations that, although inefficient, are not doing a great deal of damage to overall system performance. Another way to describe this prioritization is *triage*, the medical term whereby the most seriously injured patients (who are not fatally wounded) are treated first.

Last, when only the worst few problems are identified and prioritized, then this paper will describe the tools, tips, and techniques for either fixing the SQL statement outright or decisively determining how it should be fixed.

### What does the problem *feel* like?

When system performance problems occur, there are usually two different kinds of symptoms:

1. poor performance accompanied by surprisingly low CPU consumption
2. poor performance accompanied by incredibly high CPU consumption

In the former case (#1), it is obvious that there is some kind of *bottleneck* which is strangling system performance. This bottleneck can be occurring anywhere: in the way the operating system allocates work among multiple CPUs, in the I/O subsystem, in the latches and locks that control access to SGA data structures, in the design of the application where all activity is being passed through a single sequential process. Problems such as these in Oracle-based systems can be detected using the *Session Wait* interface, which will not be covered by this paper.

In the latter case (#2), system performance problems are undoubtedly due to the fact that the system is simply being hammered with more work than it can handle at one time. It is being asked to do too many things at once, and as a result, most processes are having to wait. As mentioned earlier, a common response to this situation is to assume that nothing can be done but to increase resource capacity, but more

impressive and permanent gains can be made by identifying and eliminating the most resource-intensive processes, also known as *resource hogs*. Finding these processes in an Oracle-based system involves using the *SQL Area* interface. This is the primary focus of this paper.

Of course, in real life the problem is often a *combination* of these two scenarios:

3. poor performance accompanied by moderate to high CPU consumption

In this situation, CPU consumption may go through *peaks* and *valleys*. At times it is excessively high, and it is clear that the poor performance is due to that. At other times, the CPU consumption drops to moderate or reasonable levels, yet overall performance does not improve. Or, system administrators or DBAs may simply feel that the system is not working too hard to justify the poor performance. Sure, it is busy, but it doesn't seem *that* busy, in their subjective opinion.

It is still probably best to start the treatment of this illness as if it were a resource issue (#2), not a *blockage* or *bottleneck* (#1), at least initially. Because certain low-level operations in the Oracle RDBMS are *single-threaded* (such as shared memory access while updating memory structures), the lack of CPU resources due to high consumption can exaggerate contention to the point where it becomes a problem. Once the *resource hogs* are dealt with, it would then be more productive to check to see if the *blockages* or *bottlenecks* still exist.

All in all, unless you have a situation of poor performance with very low CPU consumption, it is best to start looking for *resource hogs*.

### Looking for *resource hogs*

As documented in most tuning classes and books, there are several levels of tuning Oracle database applications:

- business rules
- data model
- SQL statements
- database instance
- CPU, I/O, and memory subsystems

These are listed in order from the highest-level of design to the lowest-level of technical detail. Conventional wisdom dictates that when you are

building a new application you start from the *top-down* and when you are tuning an existing production application you start from the *bottom-up*.

In this paper, I would like to alter that tidy symmetry somewhat, to assert that to tune an existing production application you start in the middle, with the SQL statements, not from the bottom-up. From a start in the middle, you can then decisively determine whether it is more appropriate to move upward (toward tuning the *data model* or the *business rules*) or downward (toward tuning the *database instance* or the *machine's subsystems*) or to simply tune the SQL itself.

### Why start with SQL statements?

Quite frankly, because it's easy and because it's the basis for all processing in Oracle. The V\$SQLAREA view contains information about SQL statements currently cached in the Shared Pool and can be queried using a SQL statement from any utility. The "V\$" views are all based upon the "X\$" tables, which are themselves based on internal memory structures in the Oracle database instance's SGA, not on tables stored in datafiles. Thus, the information in "V\$" views is initiated each time the database instance is started and is discarded when the instance is terminated. The V\$SQLAREA view displays SQL statements that are currently executing in the database instance, as well as frequently used SQL statements that remain cached in the Shared Pool. Since practically all activity in Oracle (excepting certain *direct-path* operations) is performed using SQL statements, this is a wonderful central location for monitoring.

### How to identify the "top 10" resource hogs

The V\$SQLAREA view contains over two dozen columns, including the following which are invaluable for the kind of tuning we need to perform:

- \* SQL\_TEXT
- \* SID
- \* BUFFER\_GETS
- \* DISK\_READS

The column SQL\_TEXT contains the first 1000 characters of the SQL statement itself; if the statement is longer than that, it can be found in its entirety in the views V\$SQLTEXT\_WITH\_NEWLINES or V\$SQLTEXT. All these views are keyed on the column SID, which is the Oracle session ID.

Each row in the V\$SQLAREA view portrays a SQL statement executing for an Oracle session. Thus, there is a *one-to-one* relationship between rows in the V\$SESSION view and the V\$SQLAREA view.

The column BUFFER\_GETS is the total number of times the SQL statement read a database block from the buffer cache in the SGA. Since almost every SQL operation passes through the buffer cache, this value represents the best metric for determining how much *work* is being performed. It is not perfect, as there are many *direct-read* operations in Oracle that completely bypass the buffer cache. So, supplementing this information, the column DISK\_READS is the total number times the SQL statement read database blocks from disk, either to satisfy a logical read or to satisfy a direct-read.

Thus, the formula:

$$(DISK\_READS * 100) + BUFFER\_GETS$$

is a very adequate metric of the *amount of work* being performed by a SQL statement. The weighting factor of 100 is completely arbitrary, but it reflects the fact that DISK\_READS are inherently *more expensive* than BUFFER\_GETS to shared memory.

Querying the V\$SQLAREA view and sorting the output by this formula is an easy way to allow the SQL statements performing the largest *amount of work* to be listed first. This query would look something like:

```
SELECT SQL_TEXT,
       DISK_READS,
       BUFFER_GETS,
       SID,
       HASH_VALUE
FROM   V$SQLAREA
WHERE  DISK_READS > 100000
AND    BUFFER_GETS > 1000000
ORDER BY (DISK_READS * 100) + BUFFER_GETS
DESC;
```

Oracle Consulting's System Performance Group, established in the USA by Cary Millsap and Craig Shallahammer, is largely responsible for popularizing the use of this formula in the field as the basis for identifying resource-intensive processes. The APS7 (Application Performance Suite for Oracle7) toolkit includes this query and the formula, which is referred to as a derived *load factor*.

The query's WHERE clause reduces the amount of data returned by the query, so that inconsequential

SQL statements are not even displayed. These minimum values are, of course, subjective; your system may require higher or lower thresholds in order to allow this query to return only about 5-20 rows.

Having the query return 5-20 rows is important, as it enforces the idea of *triage*. When tuning a running production system beset by *resource hogs*, it is important to remain focused on only the very worst cases. At most, you would only want to glance at the more than a handful of statements as a way of checking to see whether closely-related variations on the top set of worst SQL statements are represented near the top. If several SQL statements are very similar, then they may be suffering from the same problem. If this is so, then it might be beneficial to concentrate on this set, which cumulatively add up as serious problems even if the individual SQL statements comprising the set do not.

#### Patterns to look for

- Rapidly descending load factor values This is a clear indication that the SQL statements detected are indeed accounting for a significant percentage of the total amount of work being performed by the Oracle database instance. If the first five SQL statements listed by this query have a *load factor* orders of magnitude higher than the next fifteen SQL statements, then it should be quite clear that fixing them will have a dramatic impact on overall database performance.
- DISK\_READS close to or equal to BUFFER\_GETS This indicates that most (if not all) of the *gets* or *logical reads* of database blocks are becoming *physical reads* against the disk drives. This generally indicates a full-table scan, which is *usually* not desirable but which *usually* can be quite easy to fix.

#### Patterns to beware of

- Thresholds used in the WHERE clause of the query need to be set to very low values What are *very low values*? Of course, this is subjective, but fewer than 10,000 *physical reads* and 50,000 *logical reads* is not particularly alarming for a production database. Of course, it depends on the application, but many run-of-the-mill SQL statements can easily consume 10,000 `DISK_READS` and not represent a problem of any kind. If the most expensive queries in the system do not exceed such low thresholds, then it could mean that the database instance has simply not

been up and running for a long time yet. Or, it may indicate that the size of the SQL Area in the Shared Pool in the Oracle SGA is too small; as a result, only currently executing SQL statements are being stored in the Shared Pool, and all others are being swapped out via the Shared Pool's *least-recently-used* (LRU) algorithm. If you do not think that the Shared Pool is too small, then it might be that the culprit for resource consumption might not be an Oracle process at all. Carefully examine the operating system utilities to determine whether a non-Oracle process is consuming excessive amounts of system resources. Also, check the Oracle *Session Wait* interface (not covered in this paper) to determine whether some form of contention is affecting performance inside the Oracle processes, while a non-Oracle process is actually making CPU resources scarce.

- Load factor very similar for most of the top 5-20 SQL statements This may be yet another indication that the Shared Pool may not be large enough, as only currently executing SQL statements are reporting statistics, and no SQL statement is staying in the Shared Pool for very long. Another explanation for this ambiguous result is, again, that resource-intensive processes are not at the heart of the performance problems. Consider looking at operating-system utilities for non-Oracle resource consumers or check the Oracle *Session Wait* interface for contention problems (not covered in this paper).
- Bind variables not being used in SQL statements; lots of similar SQL statements varying only by hard-coded data values This is a classic predicament, and indicates a problem that may need to be resolved before attempting to tune the SQL statements in question. If an application is not using *bind variables* when it should, then each use of the SQL statement appears to the Oracle RDBMS as a completely separate statement. As a result, the caching mechanism of the Shared Pool's SQL Area is being entirely circumvented, causing massive amounts of resource contention in the Shared Pool. When there are high levels of concurrently active users in this predicament (i.e. 100-300 concurrently active users or more), the contention resulting from not using bind variables can be as detrimental as if the SQL statements were not using indexes! Whenever this situation is detected in an online transaction processing (OLTP) environment, please consider the use of

bind variables to be as important as the use of indexing.

### Prioritization and *triage*

Based on the query of V\$SQLAREA, it should be easy to identify a small number (3-5) of the very worst SQL statements. Each such statement should be made into a discrete task for a SQL tuning person. It is important to focus on this small number of tasks, fix them, and then re-evaluate for the next “top 5” or “top 10” all over again. Just like the medical technique of *triage*, this is the fastest method of using a small number of resources (i.e. SQL tuning people) to address a crisis in a complex environment. Focus on only the very worst problems, fix them just enough so that they are no longer a problem, but don’t linger over them attempting to perfect them, and then re-assess.

### Tools for tuning SQL statements

Quite briefly, the best tools are SQL trace and TKPROF with EXPLAIN PLAN.

#### EXPLAIN PLAN

The EXPLAIN PLAN utility is quite well documented in the standard Oracle RDBMS documents *Oracle7 Server Tuning* and *Oracle8 Server Tuning*, as well as additional texts such as O’Reilly and Associate’s book *Oracle Performance Tuning*.

The EXPLAIN PLAN utility executes the Oracle Optimizer against a SQL statement and inserts the *execution plan* for the statement into another table usually referred to as the *plan table*. The plan table is, by default, named PLAN\_TABLE, and the script to create it can be found in a file named *utlxplan.sql* in a sub-directory named *rdms/admin* under the ORACLE\_HOME directory. On UNIX, the file might be named \$ORACLE\_HOME/rdms/admin/utlxplan.sql and on Windows NT the file might be named C:\ORANT\RDBMS $nn$ \ADMIN\UTLXPLAN.SQL, where *nn* is the version of the Oracle RDBMS, such as 73 (for RDBMS 7.3).

EXPLAIN PLAN is designed to interpret exactly what the SQL statement will do, but it provides no information to help decide whether this execution plan is a good idea, or not. As a result, EXPLAIN PLAN is very limited as a tuning tool.

### The SQL Trace utility

Another set of tools which addresses this deficiency is the SQL Tracing utility, with it’s companion reporting formatter TKPROF.

The SQL Trace utility was originally written as a debugging facility for Oracle processes. When enabled, the Oracle server process creates a “flat file” in standard text, into which a record of all activity by the Oracle process is placed. This *trace file* can grow to be quite large, and it’s size is restricted by the value of the MAX\_DUMP\_FILE\_SIZE initialization parameter, set by the DBA. Trace files are located in the operating-system directory indicated by the USER\_DUMP\_DEST initialization parameter, and are usually named “*sid\_ora\_nnnnn.trc*”, where *sid* is the ORACLE\_SID string and *nnnnn* is the operating-system process ID of the Oracle server process. Each entry in the trace file records a specific operation performed while the Oracle server process is processing a SQL statement. These entries begin with the following phrases:

#### PARSING IN CURSOR #*nn*

Parsing a SQL statement for cursor *nn*; the full text of the SQL statement is included in this entry

#### PARSE #*nn*:

Re-parsing the SQL statement for cursor *nn*.

#### EXEC #*nn*:

Executing the SQL statement for cursor *nn*. For DDL statements (such as CREATE, ALTER, and DROP) and DML statements (such as INSERT, UPDATE, and DELETE), this is the final operation of the SQL statement.

#### FETCH #*nn*:

Fetching rows of data for the SQL statement for cursor *nn*. This step is valid only for queries (i.e. SELECT statements).

#### WAIT #*nn*:

The SQL statement being executed by cursor *nn* is waiting for a particular Oracle event to complete. This information mirrors the

V\$SESSION\_WAIT and  
V\$SYSTEM\_EVENT views.

**BINDS #nn:**

Displaying the data values passed to the SQL statement for cursor *nn*.

**PARSE ERROR #nn:**

The following error was encountered while parsing the SQL statement for cursor *nn*.

**UNMAP #nn:**

Freeing up any temporary structures (such as TEMPORARY segments used during sorting) used by the SQL statement for cursor *nn*.

**STAT #nn:**

Row count statistics for lines in the EXPLAIN PLAN associated with the SQL statement for cursor *nn*.

**XCTEND #nn:**

Indicates either a COMMIT or ROLLBACK to end the transaction, associated with cursor *nn*.

For the purposes of performance tuning, the contents of this trace file are too voluminous to be of any use. It is one of the ironies of the information age that as our abilities to provide information has skyrocketed, our abilities to understand and assimilate that information has not changed very much. So, while the SQL Trace utility provides all the information necessary to understand how a SQL statement is working in real-life, we use a report formatting utility named TKPROF to make sense of it all.

**Using the SQL Trace file just for debugging, not tuning...**

One fact that is frequently overlooked: this trace file can be used for application debugging. Remember what information is stored here: a record of every operation performed by every SQL statement in an Oracle session. The information is recorded in the same order in which everything was executed. Information from *recursive* operations such as stored procedure calls, packaged procedure calls, database trigger firings, and even internal Oracle recursive calls to the Oracle data dictionary are recorded.

This kind of information can be priceless if you are trying to gain insight on exactly what is going on behind the scenes. The problem is that the information in this file is very cryptic and concise. However, a great deal can be learned when you understand that much of the same information in this file shows up in the TKPROF report. Mapping data items in the raw SQL Trace file to data items on the TKPROF report is not difficult at all.

**TKPROF report formatting utility**

The TKPROF utility reads a SQL Trace *trace file* as input and produces another “flat file” in standard text as output. It is a very simple utility, summarizing the tons of information provided in the trace file so that it can be understood for performance tuning.

The problem with TKPROF is that people generally do not use it to its full advantage, and then assume (from the disappointing output) that both SQL Trace and TKPROF have little or no benefit.

TKPROF has only one mandatory command-line parameter: the name of the SQL Trace *trace file*. If you do not provide the name of a file into which the tuning report can be output, then TKPROF will prompt you interactively. Many people tend to provide only this minimal amount of information.

The resulting report is only marginally useful. The SQL statements and some of their accompanying statistics are duly output, but the order in which the SQL statements are printed in the report roughly mirrors the order in which they were executed. This means that if an Oracle session executed dozens or hundreds of SQL statements, then it may be necessary for a person reading the report to wade through dozens or hundreds of pages in order to find a SQL statement whose statistics show it to be a problem.

To combat this problem, the TKPROF utility has two command-line parameters, SORT and PRINT. The SORT parameter allows you to specify the specific *performance statistics* by which you would like the resulting reported sorted. There are twenty-three different statistics by which TKPROF can sort; these can be displayed by running TKPROF without specifying any command-line parameters. In UNIX, this would look as follows:

prscnt number of times parse was called  
prscpu cpu time parsing  
prselc elapsed time parsing  
prsdsk number of disk reads during parse

prsqry number of buffers for consistent read during parse  
prscu number of buffers for current read during parse  
prsmis number of misses in library cache during parse  
execnt number of execute was called  
execpu cpu time spent executing  
exeela elapsed time executing  
exedsk number of disk reads during execute  
exeqry nbr of buffers for consistent read during execute  
execu number of buffers for current read during execute  
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 userid of user that parsed the cursor

The most useful set of statistics pertain to *logical reads*, which in the TKPROF reports are divided into two parts, *consistent gets* and *current gets*. Together, these two statistics represent logical reads, which represent the number of accesses of database blocks in the Oracle buffer cache in the SGA. Because this kind of activity is central to almost everything that Oracle processes do, logical reads represent an excellent *metric* for performance tuning, representing *raw work*. A SQL statement that performs 1,000,000 logical reads is certainly performing much more work than another SQL statement performing 100 logical reads. What is useful about logical reads is that they are not affected by the environment outside of the Oracle database instance. If a SQL statement requires 1,000,000 logical reads on a busy UNIX machine, it will still require 1,000,000 to perform that statement when the machine is lightly loaded or completely unused otherwise. Thus, as a performance metric, logical reads are *isolated* from the environment external to Oracle.

The same is true if the SQL statement is run against an exactly identical database using the identical version of the Oracle RDBMS; 1,000,000 logical reads will still be required. Thus, as a performance metric, logical reads are *portable* across environments.

This combination of characteristics, *isolated* and *portable*, can not be said of any of the other two dozen performance statistics by which TKPROF can sort.

In order to sort the TKPROF report by logical reads, use the following command-line parameter:

```
sort=exeqry,fchqry,execu,fchcu
```

Using this sorting criteria, the resulting TKPROF report will display the most expensive SQL statement first, with less expensive SQL statements following in descending importance.

Now, reading the TKPROF report becomes much easier; the worst SQL statements “float” to the top.

However, the report may still be dozens or hundreds of pages long. As you get used to using TKPROF, you will undoubtedly notice that you will tend to only pay attention to the first couple of SQL statements in the report. This is, again, another form of *triage*; why bother with the SQL statements at the bottom of the TKPROF report that only use a couple of logical reads each?

To keep the report short, you can use TKPROF’s command-line parameter PRINT, to which you specify the number of SQL statements you wish to see in the report. Used in conjunction with the SORT parameter, this will allow TKPROF to produce a report showing only the very worst SQL statements from a session. For example:

```
sort=exeqry,fchqry,execu,fchcu print=10
```

will display the ten worst SQL statements shown in the SQL Trace.

Last, be sure to use EXPLAIN PLAN with TKPROF, so that the execution plan (and the number of data rows processed at each step) can be displayed. One problem here is that the TKPROF command-line parameter to enable this requires an Oracle USERNAME and PASSWORD in clear text on the operating-system command-line. The purpose is allow the TKPROF utility to actually connect to the Oracle RDBMS as the specified username and run the EXPLAIN PLAN utility.

Unfortunately, this can be a serious security breach, as the password being passed in clear text on the operating-system command-line provides anyone on that system with opportunity to steal it. Please be aware of this fact, and use this option accordingly.

With the combination of run-time statistics as well as the execution plan, SQL Trace and TKPROF with

EXPLAIN PLAN provide the decisive empirical data that a SQL tuning person can use to determine conclusively whether a SQL statement is efficient or not. Better yet, this information can be used to compare one version of a statement against another, allowing a tuning person to determine whether one execution plan is superior to another.

This enables the testing of hypotheses, the ability to take a problem statement and *baseline* it. Starting with this baseline, you can methodically test several different hypotheses. The execution plan information allows you to verify whether what you have in mind is taking place. The run-time statistics (especially the logical read statistics) provide conclusive empirical proof of whether the hypothesis is an improvement or not.

### Enabling SQL Trace

There are three ways to enable the SQL Trace utility:

- for every Oracle process in a database instance
- within a single process, by that process
- within a single process, by another process

There are various reasons for using each method, as explained below.

#### Enabling SQL Trace for an entire instance

This is done by setting the *init.ora* initialization parameter `SQL_TRACE` to the value of `TRUE`; the default value is `FALSE`. The database instance must be stopped and restarted to do this.

As a result, every single Oracle process, including the *background* processes such as `SMON` and `PMON`, will create a flat file in the file system. Background process trace files will be created in the directory specified by the initialization parameter `BACKGROUND_DUMP_DEST` and all other processes will create trace files in the directory specified by the initialization parameter `USER_DUMP_DEST`. As you can imagine, in a busy system this can result in quite a lot of disk space being used up. Additionally, you can expect additional CPU consumption as each Oracle process is burdened with the additional overhead of recording every operation performed to a trace file.

As a result, enabling `SQL_TRACE` in this manner is something rarely done in production systems or busy development or test systems.

#### Enabling SQL Trace for a single process, by that same process

This is the most common method of enabling SQL Trace, using the SQL command:

```
ALTER SESSION SET SQL_TRACE = TRUE
```

In interactive tools like `SQL*Plus` and `WorkSheet`, all you need to do is enter the command as a SQL statement.

In other utilities such as `Oracle Forms` or `SQL*Forms`, this may involve a command-line parameter, such as `-s`. Alternatively, tracing can be enabled by executing the `ALTER SESSION` command in a trigger or in a user-exit.

In the Oracle Precompilers, this command can be executed as:

```
EXEC SQL ALTER SESSION SET SQL_TRACE = TRUE
```

In Oracle Call interfaces (OCI), it is executed like any other DDL command.

In PL/SQL, DDL commands such as `ALTER SESSION` cannot be used. The built-in package `DBMS_SESSION` has a procedure named `SET_SQL_TRACE` which takes a single boolean input parameter of `TRUE` or `FALSE`.

When developing applications, it is wise to make sure that SQL Tracing can be enabled conditionally based on a standard method, so that it can be enabled readily under production conditions.

For example, it is common to designate UNIX environment variables to convey information to programs. If an environment variable named `PROG_SQLTRACE` is given the value of `TRUE`, then the following code inside a `PRO*C` program could be used:

```
EXEC SQL CONNECT :username_password;
p_env = getenv("PROG_SQLTRACE");
if (p_env != 0)
{
    if (strcmp(p_env, "TRUE") == 0)
    {
        EXEC SQL ALTER SESSION SET
        SQL_TRACE = TRUE;
    }
}
}
```

There are many ways to accomplish this; this is just one idea for one situation.

### Enabling SQL Trace for a single process, by another process

In many cases, code for enabling SQL Trace at run-time has not been embedded, so ordinarily it would not be possible to use it.

But, there are two methods available for enabling SQL Trace on a running process from another process.

First, there is another procedure in the DBMS\_SESSION package named SET\_SQL\_TRACE\_IN\_SESSION, which takes three input parameters. The first two are SID and SERIAL#, which uniquely identify Oracle sessions. These two values comprise the primary key of the V\$SESSION view. The third input parameter is the boolean value of TRUE or FALSE.

The other method is available only to DBAs. From Oracle RDBMS 7.3 and above, the Server Manager utility has a set of commands called ORADEBUG. Prior to 7.3, special-purpose utilities named *oradbx* (for UNIX) and *orambx* (for VMS) were required to run ORADEBUG commands.

In Server Manager, the following commands could be used to enable SQL Trace in the Oracle process whose operating-system process ID was *nnnnn*:

```
SVRMGR> oradebug setospid nnnnn
SVRMGR> oradebug event 10046 trace name
context forever, level 0
```

When using *oradbx* or *orambx*, use:

```
oradbx> debug nnnnn
oradbx> event 10046 trace name context
forever, level 0
```

When enabling SQL Trace in this way, please be aware that the resulting trace file may not have caught everything done by the process being traced. The explanation is simple: quite a lot may have happened before you enabled tracing.

### Summary

It is possible to identify the most resource-intensive processes in an Oracle database instance, using the V\$SQLAREA view.

It is then possible to tune those *resource hogs* using SQL Trace and TKPROF. These tuning efforts may result in:

- the creation of indexes to better support the SQL statement

- modifications to the text of the SQL statement to provide a more efficient execution plan
- the realization that the SQL statement itself cannot be tuned any better

Given such a realization as the third item, one is left with choices.

If the execution plan is the most efficient possible but performance is not suitable, then either the *data model* does not optimally support the SQL statement or the *underlying database instance or operating-system* needs to be tuned.

In the former case, changes to the *data model* may in turn prompt examinations of the *business requirements* that underlie the SQL statement itself. It is possible that business requirement may force changes to the data model, such as the creation of *summary tables*.

It is also possible that the SQL statement itself is inappropriate for satisfying the business requirement. This might result in the SQL statement being re-engineered or eliminated.

In the latter case, when it is realized that the SQL statement is tuned efficiently and effectively uses the data model while fulfilling its business requirement, it might then become necessary to improve overall database instance performance, or overall system performance. This may or may not involve increasing system resource capacity, such as additional CPU capacity.

In closing, start with the SQL statements. If they are inefficient, all other tuning efforts are doomed to fail or simply cost too much.

Finally, monitor for *resource hogs* frequently. Don't let them accumulate unnoticed, so they surprise you at moments of crisis. This is as much a part of the DBAs job as monitoring space and performing backups. Make sure that once they are identified, they get resolved; don't *toss them over the fence* for someone else to clear up.

**Good luck, and good hunting!**