



CHAPTER 2

Your Performance Toolkit

2 Effective Oracle by Design



In this chapter, we'll look at all of the tools I use on a recurring basis. These are the tools I use every day in order to test ideas, debug processes, tune algorithms, and so on. It is a fairly short list and incorporates only one graphical user interface (GUI) tool. All of the other tools are predominantly character mode, text-based tools. In today's world, nearly everyone is using a GUI of some sort, be it the Macintosh, Linux, some Unix variant, or one of the many different Microsoft Windows versions. So, why is a tool like SQL*Plus, a command-line interface to the database, still relevant? For the same reason telnet (or more likely ssh, the secure shell) is useful. In a mobile environment, where you move from computer to computer, your networking capabilities and set of installed software do not remain constant. I've seen people sit down at someone else's computer and not be able to work because the editor they used wasn't installed, or the GUI tool they used to connect to the database wasn't there. But, in most cases, the standard command-line tools are there. If you can use them, you'll be able to work on almost any computer.

So, do I use any graphical tools for the database? The answer is very few. I use JDeveloper occasionally. I use Oracle Enterprise Manager (OEM) infrequently. But that is the extent of it. I would say that 99% of my interaction with Oracle in an ad-hoc fashion is via the good, old command-line SQL*Plus tool.

For application tuning, I use my favorite tools to make sure the application is as fast and as scalable as possible. First, I'll tune it in single-user mode, using EXPLAIN PLAN, AUTOTRACE, TKPROF, and/or Runstats. Then I'll tune the application in multiuser mode, using Statspack.

This chapter covers my tools of choice:

- SQL*Plus
- EXPLAIN PLAN
- AUTOTRACE
- TKPROF (hands down, my favorite)
- Runstats (a homegrown tool)
- Statspack
- DBMS_PROFILER
- JDeveloper (it's not just for Java anymore)

I'll describe the main uses of each tool, explain how to set it up, and offer tips for interpreting its output.

SQL*Plus

SQL*Plus is ubiquitous, always available, and always the same. If you can operate SQL*Plus on your Windows machine, you can do it on Unix, Linux, and even the mainframe, without any training. I choose it as my primary means of "talking" to the database for the same reason I choose my editor of choice, vi: It is simple, powerful, and here. (Well, for Windows I need to carry a copy of vi in my pocket on a floppy disk, but at least it fits!)

Another major reason for using SQL*Plus as your tool of choice for testing is that it is reproducible. It is a known quantity and operates the same from release to release. If you want to prove a point, you cannot use TOAD (a common GUI), for example, for the simple reason that most people do not use TOAD. It's the same with virtually any other tool out there. Consider SQL*Plus your baseline tool, because it's something everyone has. If you use it constantly, you'll have the ability to sit down and be productive on anyone's workstation immediately.

These days, there is even iSQL*Plus. If you have a browser and can connect to the server hosting iSQL*Plus, you can interactively query and work with your database using just that. I make frequent use of iSQL*Plus from home to test on different versions of Oracle when answering questions submitted on the asktom web site. On my desktop machine at home, I have only the latest version of the database. When I want to test something on any earlier version (going all of the way back to version 7.0—ten releases), I need to connect to my servers at work. I use a broadband satellite connection, and as anyone can tell you, doing client server (SQL*Net or telnet) operations over a satellite with a two-second turnaround time is painful (the satellite I use is about 22,000 miles from my house and every network request transmitted from my computer takes about two seconds to go up and come back to me). Using the web browser metaphor, whereby I submit a script in one message and receive all of the output in another message, turns a one-hour back-and-forth process that would have happened with client server techniques, into a two-second experience. So, over dial-up or other slow or high-latency connections, iSQL*Plus is a lifesaver. (And there is no way I could use a traditional client/server GUI tool in this environment.)

So, what do I use SQL*Plus for mainly?

- **AUTOTRACE** This is very simple method to get an execution plan for a query, to see statistics such as logical I/Os for a statement, and so on. (AUTOTRACE is covered in detail later in this chapter.)
- **As a scripting tool** Some people use shell scripts to automate SQL*Plus; I use SQL*Plus to automate scripts. I can write a script in SQL*Plus to automate an export operation, for example, that can be used on *any* platform. I don't need to recode for Windows simply because I used Unix in the first place.

Set Up SQL*Plus

The setup for SQL*Plus is amazingly easy. In fact, it should already be done. Every client software installation has it, and every server installation has it. That is why if you were to limit yourself to exactly one tool, SQL*Plus would be it. No matter where you go, it is there.

On Windows, there are two versions of SQL*Plus: a GUI one (the sqlplusw.exe program) and a DOS-based character-mode one (the sqlplus.exe program). The character-mode SQL*Plus is 100% compatible with SQL*Plus on every other platform on which Oracle is delivered. The GUI SQL*Plus, which offers no real functional benefit over the character mode—after all, it is a character-mode tool running in a window—is different enough to be confusing and isn't as flexible as the command-line version. Additionally, it is already officially deprecated in the next release of Oracle, so it won't be around for long. In the end, it is your choice, but you should give the character-mode sqlplus.exe a chance. Hey, at least you can pick the colors easily in a DOS window!

Customize the SQL*Plus Environment

SQL*Plus has the ability to run a script automatically upon startup. This script can be used to customize your SQL*Plus environment and set up certain variables.



NOTE

*SQL*Plus can also run a `glogin.sql` (global login.sql) script, which can contain site-wide default settings. The use of the `glogin.sql` script is somewhat dated, however. It was more useful when dozens of people might share a single computer. Today, with most people having a desktop machine with SQL*Plus on it, the `glogin.sql` script's use is somewhat obviated.*

I'll share my `login.sql` script with you. This should give you a good idea of what you can do with such a script.



```
REM turn off the terminal output - make it so SQLPlus does not
REM print out anything when we log in
set termout off
```

```
REM default your editor here.  SQLPlus has many individual settings
REM This is one of the most important ones
define _editor=vi
```

```
REM serveroutput controls whether your DBMS_OUTPUT.PUT_LINE calls
REM go into the bit bucket (serveroutput off) or get displayed
REM on screen.  I always want serveroutput set on and as big
REM as possible - this does that.  The format wrapped elements
REM causes SQLPlus to preserve leading whitespace - very useful
set serveroutput on size 1000000 format wrapped
```

```
REM Here I set some default column widths for commonly queried
REM columns - columns I find myself setting frequently, day after day
column object_name format a30
column segment_name format a30
column file_name format a40
column name format a30
column file_name format a30
column what format a30 word_wrapped
column plan_plus_exp format a100
```

```
REM by default, a spool file is a fixed width file with lots of
REM trailing blanks.  Trimspool removes these trailing blanks
REM making the spool file significantly smaller
set trimspool on
```

```
REM LONG controls how much of a LONG or CLOB sqlplus displays
REM by default.  It defaults to 80 characters which in general
REM is far too small.  I use the first 5000 characters by default
```

```
Set long 5000
```

```
REM This sets the default width at which sqlplus wraps output.
REM I use a telnet client that can go upto 131 characters wide -
REM hence this is my preferred setting.
```

```
set linesize 131
```

```
REM SQLplus will print column headings every N lines of output
REM this defaults to 14 lines. I find that they just clutter my
REM screen so this setting effectively disables them for all
REM intents and purposes - except for the first page of course
```

```
set pagesize 9999
```

```
REM here is how I set my signature prompt in sqlplus to
REM username@database> I use the NEW_VALUE concept to format
REM a nice prompt string that defaults to IDLE (useful for those
REM of you that use sqlplus to startup their databases - the
REM prompt will default to idle> if your database isn't started)
```

```
define gname=idle
column global_name new_value gname
select lower(user) || '@' ||
substr( global_name, 1, decode( dot,
0, length(global_name),
dot-1) ) global_name
from (select global_name, instr(global_name, '.') dot
from global_name );
set sqlprompt '&gname> '
```

```
REM and lastly, we'll put termout back on so sqlplus prints
```

```
REM to the screen
```

```
set termout on
```

Use @CONNECT

Once you start using the login.sql script, probably the first thing you'll discover is that SQL*Plus runs it *once*, on startup. If you reconnect, the prompt doesn't change, and some settings are reset. The setting I noticed right away was serveroutput. Consider the following:

```
$ sqlplus /
```

```
SQL*Plus: Release 9.2.0.1.0 - Production on Sun Dec 15 14:16:54 2002
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
Connected to:
```

```
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
```

```
ops$tkyte@ORA920> show serveroutput
```

```
serveroutput ON size 1000000 format WRAPPED
```

```
ops$tkyte@ORA920> connect /
```

```
Connected.
```

6 Effective Oracle by Design

```
ops$tkyte@ORA920> show serveroutput
serveroutput OFF
ops$tkyte@ORA920>
```

Since I rely on the prompt being correct (so I know who I'm logged in as and what database I'm in) and on DBMS_OUTPUT output being shown, this is disappointing. My solution is to use @CONNECT, rather than CONNECT.

```
ops$tkyte@ORA920> @connect /
ops$tkyte@ORA920> show serveroutput
serveroutput ON size 1000000 format WRAPPED
ops$tkyte@ORA920>
```

@CONNECT is a very simple script that just reruns the login.sql script for me.

```
set termout off
connect &1
@login
```

NOTE

*In order to connect with SYSDBA, you will need to enter **@connect "/as sysdba"** with quotes to get the AS SYSDBA text part of the &1 substitution variable.*

In this fashion, I always am working in the environment I expect to be in.

Use SQLPATH

SQLPATH is the name of an environment variable you can set to tell SQL*Plus where to look for scripts. By using this environment variable, you can put your login.sql, connect.sql, and other scripts in a single directory, and SQL*Plus will find them for you. SQL*Plus will look in the current directory, and then it will look in the directory specified in the SQLPATH environment variable. This setting works on every platform that supports environment variables (such as Unix, Linux, Macintosh, and even Windows).

Read the Documentation!

SQL*Plus is well-documented. All of the settings are available for you to see in the SQL*Plus documentation.

The most important chapter in the documentation, the one I refer to frequently, is the "SQL*Plus Command Reference" chapter. Here, you will learn about all of the SET options you have (there are a *lot* of them). It is amazing how many SQL*Plus questions I answer by supplying a URL for this particular chapter.

EXPLAIN PLAN

EXPLAIN PLAN is the SQL command that you may use to get Oracle to tell you what the query plan for a given SQL query would be if you executed it *right now*. It's important to grasp the

point that it is the plan that gets used if you were to execute the query in the current session, with the current settings.

EXPLAIN PLAN cannot tell you what plan was *actually* used to run a given query in the past, because that query execution could have taken place in a session with very different settings. For example, a query run in a session with a large sort area size may well use a different plan than the same query in a session with a small sort area size. (As you'll see in this chapter, Oracle9i does provide some ways to view the actual plan that was used for a query when it was executed.)

Set Up for EXPLAIN PLAN

Setup for EXPLAIN PLAN involves several scripts in \$ORACLE_HOME/rdbms/admin:

- utlxplan.sql (for UTiLility eXplain PLAN table), which contains a CREATE TABLE statement for a table named PLAN_TABLE. This is the table in which EXPLAIN PLAN places the query plan.
- utlxplp.sql (for UTiLility eXplain PLAN Parallel), which displays the contents of the plan table, including information specific to parallel-query plans.
- utlxpls.sql (for UTiLility eXplain PLAN Serial), which displays the contents of the plan table for normal, serial (nonparallel) plans.

In Oracle9i Release 2, you also need to be aware of an important package, named DBMS_XPLAN. This is a new supplied package that makes it very easy to query the plan table.

To set up for EXPLAIN PLAN, first create the plan table itself:

```
ops$tkyte@ORA920> @?/rdbms/admin/utlxplan
Table created.
```

```
ops$tkyte@ORA920> desc plan_table
```

Name	Null?	Type
STATEMENT_ID		VARCHAR2 (30)
TIMESTAMP		DATE
REMARKS		VARCHAR2 (80)
OPERATION		VARCHAR2 (30)
... some columns removed ...		
ACCESS_PREDICATES		VARCHAR2 (4000)
FILTER_PREDICATES		VARCHAR2 (4000)

If you would like to make EXPLAIN PLAN available to the world out of the box, without setup, create a schema called UTILS or TOOLS. This schema would be granted CREATE SESSION and CREATE TABLE privileges. Create the plan table in this schema as a GLOBAL TEMPORARY TABLE with the "ON COMMIT PRESERVE ROWS" option and grant the ALL privilege on PLAN_TABLE to public. A DBA could then create a public synonym PLAN_TABLE for UTILS.PLAN_TABLE. Now, each developer can "share" the same plan table. All the developers would want to take care to issue a DELETE FROM PLAN_TABLE command or "TRUNCATE PLAN_TABLE" before using EXPLAIN PLAN. This will not cause concurrency issues; they will

8 Effective Oracle by Design

not see other developers' plans; they will not see your plans; and there will be no blocking/waiting issues.

Use EXPLAIN PLAN

Now, you are ready to explain a query into your plan table. The format of the EXPLAIN PLAN command is simply:

```
explain plan
  [set statement_id = 'text']
  [into [owner.]table_name]
for statement;
```

The text in brackets is optional, and we won't be using it in our examples. The `statement_id` allows you to store multiple plans in the plan table. The `owner.table_name`, allows you to use a table other than `PLAN_TABLE` if you prefer.

Here, we'll create a table to test with:

```
ops$tkyte@ORA920> CREATE TABLE t
2  (
3    collection_year int,
4    data            varchar2(25)
5  )
6  PARTITION BY RANGE (COLLECTION_YEAR) (
7    PARTITION PART_99 VALUES LESS THAN (2000),
8    PARTITION PART_00 VALUES LESS THAN (2001),
9    PARTITION PART_01 VALUES LESS THAN (2002),
10   PARTITION PART_02 VALUES LESS THAN (2003),
11   PARTITION the_rest VALUES LESS THAN (MAXVALUE)
12 )
13 ;
Table created.
```

This creates a very simple partitioned table. I have chosen a partitioned table to show why EXPLAIN PLAN is still relevant, even though AUTOTRACE will seem to be much easier to use. (Soon, we'll compare the EXPLAIN PLAN output to AUTOTRACE's output, to demonstrate the difference.)

We start with a delete operation against the plan table, to clear out any preexisting rows, and then continue the EXPLAIN PLAN statement itself:

```
ops$tkyte@ORA920> delete from plan_table;
2 rows deleted.

ops$tkyte@ORA920> explain plan for
2  select * from t where collection_year = 2002;
Explained.
```

That is it. Now, we are ready to run `utlxpls` (serial plan output). The results are as follows:


```
ops$tkyte@ORA920> @?/rdbms/admin/utlxlpls
```

```
PLAN_TABLE_OUTPUT
```

```
-----
```

Id	Operation	Name	Rows	Bytes	Cost	Pstart	Pstop
0	SELECT STATEMENT		1	27	2		
* 1	TABLE ACCESS FULL	T	1	27	2	4	4

```
-----
```

```
Predicate Information (identified by operation id):
```

```
-----
```

```
1 - filter("T"."COLLECTION_YEAR"=2002)
```

```
Note: cpu costing is off
```

```
14 rows selected.
```

That shows how Oracle will evaluate the plan. In a nutshell, a full-table scan will be performed on the table T. We can also see the cost of performing this step (COST=2), the expected number of rows to be returned, and how many bytes of output would be returned. The optimizer is “guessing” this information, since we did not analyze the table. We can also see that only one partition is being accessed, as shown in the Pstart and Pstop columns. So, while this is a full-table scan, partition elimination is making it possible for us to skip reading the bulk of the table.

Note that only Oracle9i Release 2 or later will have output exactly what is shown here. In prior releases, you will not see the predicate information at the end of the report. This is new information printed by the DBMS_XPLAN supplied package, which is not available prior to Oracle9i Release 2. This additional information makes it easy to see exactly what portions of the predicate Oracle is applying at each step of the query plan.

An AUTOTRACE Comparison

Even though we haven’t talked about AUTOTRACE yet, we’ll jump ahead a bit to make a point about the benefits of EXPLAIN PLAN. We’ll compare the utlxlpls output with the output provided by AUTOTRACE:

```
ops$tkyte@ORA920> set autotrace traceonly explain
ops$tkyte@ORA920> select * from t where collection_year = 2002;
```

```
Execution Plan
```

```
-----
```

```
0          SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=27)
1    0      TABLE ACCESS (FULL) OF 'T' (Cost=2 Card=1 Bytes=27)
```

```
ops$tkyte@ORA920> set autotrace off
```

10 Effective Oracle by Design

Now, while getting AUTOTRACE to show us a plan is easy, we can see that a very important piece of information is missing here! The partition elimination information is not displayed. With the EXPLAIN PLAN scripts, we get that additional, important information (important when using partitioning, that is).

In summary, you should consider using EXPLAIN PLAN and DBMS_XPLAN in Oracle9i Release 2 to view query plans if the query plan is what you are interested in reviewing. Use AUTOTRACE to view statistics only. DBMS_XPLAN shows information relevant to the query, based on the query. AUTOTRACE always shows the same information, regardless of the query type.

How to Read a Query Plan

Often, I am asked this question: "How exactly do we read a query plan?" Here, I will present my approach to reading the plan. We'll take a look at a query plan resulting from a query against the SCOTT/TIGER tables. Note that I have added primary keys to the EMP and DEPT tables, so they are indexed.

```
scott@ORA920> delete from plan_table;  
7 rows deleted.
```

```
scott@ORA920> explain plan for  
  2 select ename, dname, grade  
  3    from emp, dept, salgrade  
  4   where emp.deptno = dept.deptno  
  5     and emp.sal between salgrade.losal and salgrade.hisal  
  6 /  
Explained.
```

```
scott@ORA920> @?/rdbms/admin/utlxpls
```

PLAN_TABLE_OUTPUT

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT				
1	NESTED LOOPS				
2	NESTED LOOPS				
3	TABLE ACCESS FULL	SALGRADE			
* 4	TABLE ACCESS FULL	EMP			
5	TABLE ACCESS BY INDEX ROWID	DEPT			
* 6	INDEX UNIQUE SCAN	DEPT_PK			

Predicate Information (identified by operation id):

```
4 - filter("EMP"."SAL"<="SALGRADE"."HISAL" AND  
          "EMP"."SAL">="SALGRADE"."LOSAL")  
6 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

Note: rule based optimization

21 rows selected.

How can we figure out what happens first, second, and so on? How does that plan actually get evaluated? First, I will show you the pseudo code for evaluation of the plan, and then we will discuss how I arrived at this conclusion.

```

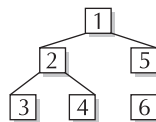
For salgrade in (select * from salgrade)
  Loop
    For emp in ( select * from emp )
      Loop
        If ( emp.sal between salgrade.losal and salgrade.hisal )
          Then
            Select * into dept_rec
              From dept
              Where dept.deptno = emp.deptno;

            OUTPUT RECORD with fields from salgrade,emp,dept
          End if;
        End loop;
      End loop;
    End loop;
  End loop;

```

The way I read this plan is by turning it into a graph of sorts—an evaluation tree. In order to do this, you need to understand access paths. For detailed information on all of the access paths available to Oracle, see the *Oracle Performance and Tuning Guide*. There are quite a few access paths, and the descriptions in the guide are quite comprehensive.

To build the tree, we can start from the top, with step 1, which will be our root node in the tree. Next, we need to find the things that feed this root node. That will be accomplished in steps 2 and 5, which are at the same level of indentation, because they feed into step 1. Continuing, we can see that steps 3 and 4 feed step 2, and that step 6 feeds step 5. Putting it together iteratively, we can draw this evaluation tree:



Reading the tree, we see that in order to get step 1, we need steps 2 and 5; step 2 comes first. In order to get step 2, we need steps 3 and 4; step 3 comes first. That is how we arrive at the pseudo code:

```

For salgrade in (select * from salgrade)
  Loop
    For emp in ( select * from emp )
      Loop

```

12 Effective Oracle by Design

The full scan of the SALGRADE table is step 3. The full scan of the EMP table is step 4. Step 2 is a nested loop, which is roughly equivalent to two FOR loops. Once we evaluate step 2 like that, we can look at step 5. Step 5 runs step 6 first. Step 6 is the index-scan step. We are taking the output of step 2 and using that to feed this part of the query plan. So, the output from step 2 is used to perform an index scan. Then that index scan output is used to access the DEPT table by ROWID. That result is the output of step 1, our result set.

Now, to make this interesting, we will run an equivalent query, but we will mix up the order of the tables in the FROM clause this time. Since I am using the rule-based optimizer (RBO), this will affect the generated query plan. (This is just one reason why you don't want to use the RBO; we will cover some more reasons in the "Understand the AUTOTRACE Output" section later in this chapter.) The RBO is sensitive to the order of tables in the FROM clause and will use the order in which we enter them to choose a "driving table" for the query if none of the predicates do so (in the event of a "tie," the RBO will look at the order the developer typed in table names to pick what table to use first)! We will use the same logic to build its query plan tree and evaluate how it processes the query.

```
scott@ORA920> delete from plan_table;
7 rows deleted.
```

```
scott@ORA920> explain plan for
 2  select ename, dname, grade
 3    from salgrade, dept, emp
 4   where emp.deptno = dept.deptno
 5     and emp.sal between salgrade.losal and salgrade.hisal
 6   /
Explained.
```

```
scott@ORA920> @?/rdbms/admin/utlxlpls
```

PLAN_TABLE_OUTPUT

Id	Operation	Name	Rows	Bytes

0	SELECT STATEMENT			
1	NESTED LOOPS			
2	NESTED LOOPS			
3	TABLE ACCESS FULL	EMP		
4	TABLE ACCESS BY INDEX ROWID	DEPT		
* 5	INDEX UNIQUE SCAN	DEPT_PK		
* 6	TABLE ACCESS FULL	SALGRADE		

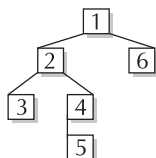
Predicate Information (identified by operation id):

```
-----
 5 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
 6 - filter("EMP"."SAL"<="SALGRADE"."HISAL" AND
           "EMP"."SAL">="SALGRADE"."LOSAL")
```

Note: rule based optimization

21 rows selected.

Here, we see that steps 2 and 6 feed step 1, steps 3 and 4 feed step 2, and step 5 feeds step 4. The evaluation tree looks like this:



So, starting with steps 3 and 4, the pseudo code logic here is

```

For emp in ( select * from emp )
Loop
  -- using the index
  Select * from dept where dept.deptno = emp.deptno

  For salgrade in (select * from salgrade )
  Loop
    If ( emp.sal between salgrade.losal and salgrade.hisal )
    Then
      OUTPUT RECORD;
    End if;
  End loop
End loop;

```

And that is it. If you draw a graphical tree, and then read it from the bottom up, left to right, you will get a good understanding of the flow of the data.

Avoid the EXPLAIN PLAN Trap

EXPLAIN PLAN is a way to get the query plan for a given SQL query if you were to execute it *right now*, in your current environment. It will not show you necessarily what plan was used yesterday to run that query or what plan would be used if another session were to execute it in the future.

In releases prior to Oracle9i Release 1, capturing the actual plan was very hard to do if all you had was the query and wanted to see the plan (you needed to try to set up a session that was exactly the same as the session that ran the query). In Oracle9i, it is easy to see the actual plan for a query that has been executed. This information is in the new V\$SQL_PLAN view.

TIP

Using SQL_TRACE, you can capture the actual plan used at runtime. However, it requires that tracing was enabled when the query was executed. This technique is described in the section on TKPROF, later in this chapter.

14 Effective Oracle by Design

As an example, we will run the same query in different environments in such a way that the environmental settings would have a material effect on the query plan. To set up for this example, we'll create and index a table T:

```
ops$tkyte@ORA920> create table t
2 as
3 select * from all_objects;
Table created.

ops$tkyte@ORA920> alter table t add constraint t_pk primary key(object_id);
Table altered.

ops$tkyte@ORA920> begin
2     dbms_stats.gather_table_stats
3     ( user, 'T',
4       method_opt => 'for all columns size AUTO',
5       cascade => TRUE );
6 end;
7 /
```

PL/SQL procedure successfully completed.

NOTE

The DBMS_STATS options I used are for Oracle9i Release 1 and up. They are roughly equivalent to analyze table T compute statistics for table for all indexes for all indexed columns. That command may be used in earlier releases of Oracle. It gathers the largest amount of statistics and is rather computationally expensive. Also, you may need to change 32000 in the following example to some smaller number if your ALL_OBJECTS table is smaller than mine is. My maximum OBJECT_ID is about 50,000. Adjust the values as needed to reproduce the effects shown.

Now, in a session where the application has changed the optimizer_index_cost_adj, a setting that has a great influence on the optimizer and the plans it will choose, a user executed:

```
ops$tkyte@ORA920> alter session set optimizer_index_cost_adj = 10;
Session altered.

ops$tkyte@ORA920> select * from t t1 where object_id > 32000;
128 rows selected.
```

Suppose that we are interested in how that query executes for tuning purposes. So, we do an EXPLAIN PLAN on it in our *other* session, where this session setting was not made:

```
ops$tkyte@ORA920> delete from plan_table;
8 rows deleted.
```

```
ops$tkyte@ORA920> explain plan for
  2 select * from t t2 where object_id > 32000;
Explained.
```

```
ops$tkyte@ORA920> set echo off
```

```
PLAN_TABLE_OUTPUT
```

Id	Operation	Name	Rows	Bytes	Cost	

0	SELECT STATEMENT		235	22560	42	
* 1	TABLE ACCESS FULL	T	235	22560	42	

```
Predicate Information (identified by operation id):
-----
```

```
1 - filter("T2"."OBJECT_ID">32000)
```

```
Note: cpu costing is off
```

```
14 rows selected.
```

Apparently, that query is using a full-table scan. Or is it? Using the information from V\$SQL_PLAN, we can tell for sure. Instead of using the EXPLAIN PLAN command, we'll just populate PLAN_TABLE with the information for this query right from the V\$SQL_PLAN table:

```
ops$tkyte@ORA920> delete from plan_table
  2 /
2 rows deleted.
```

```
ops$tkyte@ORA920> insert into plan_table
  2 ( STATEMENT_ID, TIMESTAMP, REMARKS, OPERATION,
  3   OPTIONS, OBJECT_NODE, OBJECT_OWNER, OBJECT_NAME,
  4   OPTIMIZER, SEARCH_COLUMNS, ID, PARENT_ID,
  5   POSITION, COST, CARDINALITY, BYTES, OTHER_TAG,
  6   PARTITION_START, PARTITION_STOP, PARTITION_ID,
  7   OTHER, DISTRIBUTION, CPU_COST,
  8   IO_COST, TEMP_SPACE )
  9 select rawtohex(address)||'_'||child_number,
 10        sysdate, null, operation, options,
 11        object_node, object_owner, object_name,
 12        optimizer, search_columns, id, parent_id,
 13        position, cost, cardinality, bytes, other_tag,
 14        partition_start, partition_stop, partition_id,
 15        other, distribution, cpu_cost, io_cost,
 16        temp_space
 17 from v$sql_plan
 18 where (address,child_number) in
 19        ( select address, child_number
```

16 Effective Oracle by Design

```
20          from v$sql
21       where sql_text =
22             'select * from t t1 where object_id > 32000'
23             and child_number = 0 )
24 /
```

3 rows created.

```
ops$tkyte@ORA920> set echo off
@?/rdbms/admin/utlxpls
```

PLAN_TABLE_OUTPUT

Id	Operation	Name	Rows	Bytes	Cost (%CPU)		

0	SELECT STATEMENT						
1	TABLE ACCESS BY INDEX ROWID	T	235	22560	21	(5)	
2	INDEX RANGE SCAN	T_PK	235		2	(0)	


8 rows selected.

Here, we see that the real plan was an index-range scan, not a full-table scan. It is not that EXPLAIN PLAN lied to us. Rather, it is that environmental differences can have a profound effect on a query plan. They could be as subtle as the simple change of a full-table scan to an index-range scan, like this, or as far-reaching as actually querying different objects (for example, table T is A.T when I query it, but it is B.T when you query it).

To be sure that you are looking at the real, live plan in Oracle9i, grab it from the V\$SQL_PLAN dynamic view. Alternatively, if you have access to a SQL_TRACE trace file, you may find the actual plan used in there, after using TKPROF to format it. That technique is discussed in the “TKPROF” section later in this chapter.


Use DBMS_XPLAN and V\$SQL_PLAN

If you edit the script utlxpls.sql in Oracle9i Release 2, you’ll discover it is effectively one-line long:

```
 select plan_table_output
from table( dbms_xplan.display( 'plan_table',null,'serial'))
```

If you edit that same script in Oracle9i Release 1 or before, you’ll find a huge query. DBMS_XPLAN.DISPLAY is a better method for querying and displaying the plan output. It is a function that simply returns a collection, which is a procedurally formatted EXPLAIN PLAN output, including the supplemental information at the bottom of the report (new in Oracle9i Release 2). This is a side effect of using the new DBMS_XPLAN package.

So, if you do not have access to the utlxpls.sql script, the simple query shown here will perform the same function. In fact, the DBMS_XPLAN package is so good at adjusting its output based on the inputs that you do not even need to supply the inputs as utlxpls.sql does. This simple line suffices:

```
 select * from table(dbms_xplan.display)
```


Using this feature coupled with the V\$SQL_PLAN dynamic performance view, you can easily dump the query plans for already executed statements, directly from the database.

In the previous section, I demonstrated how you can use an INSERT into the PLAN_TABLE and then run utlxpls or utlxplp to see the plan. In Oracle9i Release 2, using DBMS_XPLAN and a view you can create, it becomes even easier. If you use a schema that has been granted SELECT privileges on SYS.V_\$SQL_PLAN directly, you'll be able to create this view:

```
ops$tkyte@ORA920> create or replace view dynamic_plan_table
2  as
3  select
4    rawtohex(address) || '_' || child_number statement_id,
5    sysdate timestamp, operation, options, object_node,
6    object_owner, object_name, 0 object_instance,
7    optimizer, search_columns, id, parent_id, position,
8    cost, cardinality, bytes, other_tag, partition_start,
9    partition_stop, partition_id, other, distribution,
10   cpu_cost, io_cost, temp_space, access_predicates,
11   filter_predicates
12   from v$sql_plan;
```

View created.

Now, you can query any plan from the database with a single query:

```
ops$tkyte@ORA920> select plan_table_output
2    from TABLE( dbms_xplan.display
3                  ( 'dynamic_plan_table',
4                    (select rawtohex(address) || '_' || child_number x
5                      from v$sql
6 where sql_text='select * from t t1 where object_id > 32000' ),
7                  'serial' ) )
8  /
```

PLAN_TABLE_OUTPUT

Id	Operation	Name	Rows	Bytes	Cst(%CPU)	

0	SELECT STATEMENT					
1	TABLE ACCESS BY INDEX ROWID	T	291	27936	25 (0)	
* 2	INDEX RANGE SCAN	T_PK	291		2 (0)	

Predicate Information (identified by operation id):

```
-----
2 - access("OBJECT_ID">32000)
```

13 rows selected.

The emphasized text in the code is a query that gets the `STATEMENT_ID`. In this query, you can use whatever values you want to identify the exact query plan you wanted to review. The use of this technique, querying the `V$` table rather than inserting the contents of `V$SQL_PLAN` into a “real table,” is appropriate if you’re generating the explain plan for this query once. Access to `V$` tables can be quite expensive latch wise on a busy system. So, if you plan on running the explain plan for a given statement many times over, copying the information to a temporary working table would be preferred.

AUTOTRACE

A close cousin of the `EXPLAIN PLAN` is `AUTOTRACE`, which is a rather nifty feature of `SQL*Plus`. `EXPLAIN PLAN` shows you what the database will do when asked to run the query. `AUTOTRACE` gives you a look at how much work it actually took to perform your query, providing some important statistics regarding its actual execution. One of the nice things about `AUTOTRACE` is that it should be fully accessible to each and every developer, all of the time. `TKPROF` (covered later in this chapter) is a great tool for tuning, but it relies on access to trace files, which may not be available in all environments.

I use `AUTOTRACE` as my first-line tuning tool. Given the query, representative inputs (binds) to the query, and access to `AUTOTRACE`, I generally have the tuning tools I need. Occasionally, I need to dig a little deeper with `TKPROF`, but most of the time, `AUTOTRACE` and `SQL*Plus` are sufficient.

“I have a poorly performing query. Can you help me?”

Give me the query in a `SQL` script, using bind variables where your application uses bind variables. Give me access to your database and make sure `AUTOTRACE` is on.

It is important when reviewing plans and tuning queries to emulate what your application does. I make sure to mention the bind variables, so that I’m not given the query `select * from some_table where column = 55` to tune, when the application actually executes `select * from some_table where column = :bind_variable`. You cannot tune a query with literals and expect a query that contains bind variables to have the same performance characteristics.

Set Up AUTOTRACE

`AUTOTRACE`’s elegance is in its simplicity. Once the DBA sets up `AUTOTRACE`, anyone can use it. My preferred method for setting up `AUTOTRACE` is as follows:

1. Issue `cd $ORACLE_HOME/rdbms/admin.`
2. Log in to `SQL*Plus` as someone with `CREATE TABLE` and `CREATE PUBLIC SYNONYM` privileges (for example, as a DBA).
3. Make `PLAN_TABLE` universally available (as described earlier in the section about `EXPLAIN PLAN`).
4. Exit `SQL*Plus` and issue `cd $ORACLE_HOME/sqlplus/admin.`

5. Log in to SQL*Plus as SYSDBA (`sqlplus "/ as sysdba"`).
6. Run `SQL> @plustrace`.
7. Run `SQL> grant plustrace to public`.

By making it public, you let anyone trace using SQL*Plus. That way, everyone, without exception, can use AUTOTRACE. After all, we never want to give developers an excuse *not* to be tuning their code! (But if you want, you can replace public with some user.)

Use AUTOTRACE

Now that the installation is complete, you are ready to start using AUTOTRACE. AUTOTRACE generates a report after any SQL DML statements (such as INSERT, UPDATE, DELETE, SELECT, and MERGE). You can control this report via the following SET commands in SQL*Plus:

- `SET AUTOTRACE OFF` No AUTOTRACE report is generated. This is the default. Queries are run as normal.
- `SET AUTOTRACE ON EXPLAIN` The query is run as normal, and the AUTOTRACE report shows only the optimizer execution path.
- `SET AUTOTRACE ON STATISTICS` The query is run as normal, and the AUTOTRACE report shows only the SQL statement execution statistics.
- `SET AUTOTRACE ON` The query execution takes place, and the AUTOTRACE report includes both the optimizer execution path and the SQL statement execution statistics.
- `SET AUTOTRACE TRACEONLY` Like `SET AUTOTRACE ON`, but suppresses the printing of the user's query output, if any. This is useful for tuning a query that returns a large result set to the client. Rather than waiting for 1,000 rows of the output to be printed and scrolled on the screen (an operation that typically takes more time than actually executing the query itself), you can suppress this display.
- `SET AUTOTRACE TRACEONLY STATISTICS` Like `SET AUTOTRACE TRACEONLY`, but suppresses the display of the query plan. It shows only the execution statistics.
- `SET AUTOTRACE TRACEONLY EXPLAIN` Like `SET AUTOTRACE TRACEONLY`, but suppresses the display of the execution statistics, showing only the query plan. In addition, for SELECT statements, this setting does not actually *execute* the query. It only parses and explains the query. INSERT, UPDATE, DELETE, and MERGE statements *are* executed using this mode; only SELECT statements are handled differently.

These are just some of the options for the AUTOTRACE command. Refer to the SQL*Plus guide for information about all of the options.

Format the AUTOTRACE Output

Now that AUTOTRACE is installed and you know how to enable it, let's look at the output it produces. We will display the plan and the runtime statistics. Since we don't care about seeing the data itself, we will suppress that. We will use the SCOTT/TIGER EMP and DEPT tables, and `SET AUTOTRACE TRACEONLY` in order to execute the query.

20 Effective Oracle by Design

```
scott@ORA920> set autotrace traceonly
scott@ORA920> select *
      2      from emp FULL OUTER JOIN dept
      3              on (emp.deptno = dept.deptno)
      4      /
15 rows selected.
```

Execution Plan

```
-----
0  SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=328 Bytes=38376)
1 0  VIEW (Cost=10 Card=328 Bytes=38376)
2 1   UNION-ALL
3 2     HASH JOIN (OUTER) (Cost=5 Card=327 Bytes=34335)
4 3       TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=327 Bytes=28449)
5 3       TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=72)
6 2     HASH JOIN (ANTI) (Cost=5 Card=1 Bytes=31)
7 6       TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=72)
8 6       TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=327 Bytes=4251)
```

This is probably my most used format of the AUTOTRACE command, translating to “show me the statistics, but hold the data.” So, what we can see is the query plan. We can see that a full outer join apparently takes a lot of work! It does an outer join of EMP to DEPT and then UNION ALLS that with an anti-join of DEPT to EMP. That is the definition of a full outer join: Give me every row from both tables, regardless of whether or not it has a match in the other table.

You do have some amount of control over the formatting of this report. The defaults (found in \$ORACLE_HOME/sqlplus/admin/glogin.sql) are as follows:

```
COLUMN id_plus_exp FORMAT 990 HEADING i
COLUMN parent_id_plus_exp FORMAT 990 HEADING p
COLUMN plan_plus_exp FORMAT a60
COLUMN object_node_plus_exp FORMAT a8
COLUMN other_tag_plus_exp FORMAT a29
COLUMN other_plus_exp FORMAT a44
```

The ID_PLUS_EXP and PARENT_ID_PLUS_EXP columns are the first two numbers you see in the EXPLAIN PLAN output above. The PLAN_PLUS_EXP column is perhaps the most important one. It is the textual description of the plan step itself; for example, TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=72). I find the default of 60 characters wide to be too small for most uses, so I set it to 100 in my login.sql file.

The last three settings control the output information displayed for parallel query plans. The easiest way to see which columns they affect is to run a parallel query with SET AUTOTRACE TRACEONLY EXPLAIN and turn them off, one by one. You'll clearly see what disappears from the report, so you'll know what they control. Here is a simple script to do that (assuming your system is set up to allow for parallel queries):

```
set echo on

set autotrace traceonly explain
select /*+ parallel( emp 2 ) */ * from emp
```

```

/
COLUMN object_node_plus_exp NOPRINT
/
COLUMN other_tag_plus_exp NOPRINT
/
COLUMN other_plus_exp NOPRINT
/
set autotrace off

```

Understand the AUTOTRACE Output

There are two possible parts to an AUTOTRACE output: the query plan report and the statistics. Looking first at the query plan, we see output like this:

Execution Plan

```

-----
0  SELECT STATEMENT Optimizer=CHOOSE (Cost=10 Card=328 Bytes=38376)
1 0  VIEW (Cost=10 Card=328 Bytes=38376)
2 1    UNION-ALL
3 2      HASH JOIN (OUTER) (Cost=5 Card=327 Bytes=34335)
4 3        TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=327 Bytes=28449)
5 3        TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=72)
6 2      HASH JOIN (ANTI) (Cost=5 Card=1 Bytes=31)
7 6        TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=72)
8 6        TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=327 Bytes=4251)

```

This shows the output of a query executed using the cost-based optimizer (CBO). You can tell that the CBO was used by the presence of the information at the end of the query plan steps in parentheses: the Cost, Card, and Bytes information. In this query plan output, the CBO information represents the following:

- **Cost** The cost assigned to each step of the query plan by the CBO. The CBO works by generating many different execution paths/plans for the same query and assigns a cost to each and every one. The query plan with the lowest cost wins. In the full outer join example, we can see the total cost for this query is 10.
- **Card** Card is short for *Cardinality*. It is the estimated number of rows that will flow out of a given query plan step. In the full outer join example, we can see the optimizer expects there to be 327 rows in EMP and 4 rows in DEPT.
- **Bytes** The size in bytes of the data the CBO expects each step of the plan to return. This is dependent on the number of rows (Card) and the estimated width of the rows.

If the Cost, Card, and Bytes information is not present, that is a clear indicator the query was executed using the RBO. Here is an example that shows the difference between using the RBO and the CBO:

```

scott@ORA920> set autotrace traceonly explain
scott@ORA920> select * from dual;

```

22 Effective Oracle by Design

Execution Plan

```
-----  
0          SELECT STATEMENT Optimizer=CHOOSE  
1    0      TABLE ACCESS (FULL) OF 'DUAL'
```

```
scott@ORA920> select /*+ FIRST_ROWS */ * from dual;
```

Execution Plan

```
-----  
0          SELECT STATEMENT Optimizer=HINT: FIRST_ROWS  
          (Cost=11 Card=8168 Bytes=16336)  
1    0      TABLE ACCESS (FULL) OF 'DUAL' (Cost=11 Card=8168 Bytes=16336)
```

We can see that the first query against DUAL used the RBO, because it does not show the Cost, Card, and Bytes information. The RBO uses a set of rules to optimize a query. It does not care about the size of objects (numbers of rows or amount of data in bytes). It only cares about the structures in the database (indexes, clusters, tables, and so on). Therefore, it does not use or report the Cost, Card, and Bytes information.

Continuing with the next section of the AUTOTRACE report, we see the runtime statistics:

Statistics

```
-----  
0 recursive calls  
0 db block gets  
13 consistent gets  
0 physical reads  
0 redo size  
1542 bytes sent via SQL*Net to client  
499 bytes received via SQL*Net from client  
2 SQL*Net roundtrips to/from client  
0 sorts (memory)  
0 sorts (disk)  
15 rows processed
```

```
scott@ORA920> set autotrace off
```

Table 2-1 briefly explains what each of these items means.

Now, let's look at these statistics in detail and see what they can tell us about our queries.

What Are You Looking for in AUTOTRACE Output?

Now that you have seen how to get AUTOTRACE going, how to customize the look and feel of the report and what is reported, the question remains: What exactly are you looking for? Generally, you are looking at the statistics. Let's go over the statistics and the information they relay.

Statistics Returned	Meaning
Recursive calls	Number of SQL statements executed in order to execute your SQL statement.
Db block gets	Total number of blocks read from the buffer cache in current mode.
Consistent gets	Number of times a consistent read was requested for a block in the buffer cache. Consistent reads may require read asides to the undo (rollback) information, and these reads to the undo will be counted here as well.
Physical reads	Number of physical reads from the datafiles into the buffer cache.
Redo size	Total amount of redo generated in bytes during the execution of this statement.
Bytes sent via SQL*Net to client	Total number of bytes sent to the client from the server.
Bytes received via SQL*Net from client	Total number of bytes received from the client.
SQL*Net roundtrips to/from client	Total number of SQL*Net messages sent to and received from the client. This includes round-trips for fetches from a multiple-row result set.
Sorts (memory)	Sorts done in the user's session memory (sort area). Controlled via the <code>sort_area_size</code> database parameter.
Sorts (disk)	Sorts that use the disk (temporary tablespace) because the sort exceeded the user's sort area size.
Rows processed	Rows processed by modifications or returned from a SELECT statement.

TABLE 2-1. *AUTOTRACE Report Runtime Statistics***Recursive Calls**

The recursive calls statistic refers to the SQL run on your behalf as a side effect of some other SQL statement. For example, if you execute an insert that fires a trigger that runs a query, that will be recursive SQL. You will see recursive SQL from many other operations, such parsing a query, requesting additional space, working with temporary space, and so on.

24 Effective Oracle by Design

A high number of recursive calls for repeated executions (to remove parsing and other first-time phenomena from consideration) is something to look at, to see if you can reduce or remove calls. If it can be avoided, it should be. It indicates additional work, perhaps unnecessary additional work, being done in the background. Here, we will look at some of the most frequent causes of a high number of recursive calls and some solutions.

Hard Parses If the recursive calls number is initially high, I may run the query again and see if this statistic remains high. If it doesn't, that would indicate the recursive SQL was due to a hard parse. Consider the following example:

```
ops$tkyte@ORA920> alter system flush shared_pool;
System altered.

ops$tkyte@ORA920> set autotrace traceonly statistics;
ops$tkyte@ORA920> select * from scott.emp;
14 rows selected.
```

Statistics

```
-----
531 recursive calls
  0 db block gets
 99 consistent gets
  2 physical reads
  0 redo size
1315 bytes sent via SQL*Net to client
 499 bytes received via SQL*Net from client
  2 SQL*Net roundtrips to/from client
 11 sorts (memory)
  0 sorts (disk)
 14 rows processed
```

```
ops$tkyte@ORA920> select * from scott.emp;

14 rows selected.
```

Statistics

```
-----
  0 recursive calls
  0 db block gets
  4 consistent gets
  0 physical reads
  0 redo size
1315 bytes sent via SQL*Net to client
 499 bytes received via SQL*Net from client
  2 SQL*Net roundtrips to/from client
  0 sorts (memory)
  0 sorts (disk)
 14 rows processed
```



```
ops$tkyte@ORA920> set autotrace off
```

As you can see, in this case, the recursive SQL was 100% due to parsing the query for the first time. Oracle needed to execute many queries (since we flushed the shared pool, its cache of this sort of information) in order to figure out the objects being accessed, permissions, and the like. The number of recursive SQL calls went from hundreds to zero, and the number of logical I/Os (consistent gets) dropped dramatically as well. This was all in reaction to not having to hard parse that query the second time around.

PL/SQL Function Calls If the recursive SQL calls remain high, you need to dig deeper to determine why. One reason might be that you are calling a PL/SQL function from SQL, and this function executes many SQL statements itself, or refers to pseudo columns such as USER that implicitly use SQL. All of the SQL executed in the PL/SQL function counts as recursive SQL. Here is an example:

```
ops$tkyte@ORA920> create or replace function some_function return number
2 as
3   l_user      varchar2(30) default user;
4   l_cnt       number;
5 begin
6     select count(*) into l_cnt from dual;
7     return l_cnt;
8 end;
9 /
Function created.
```

The emphasized code will be counted as recursive SQL when we run this. The following is the output after running the query once (to get it parsed):

```
ops$tkyte@ORA920> set autotrace traceonly statistics;
ops$tkyte@ORA920> select ename, some_function
2   from scott.emp
3   /

14 rows selected.
```

Statistics

```
-----
28 recursive calls
...
14 rows processed
```

```
ops$tkyte@ORA920> set autotrace off
```

As you can see, there were 28 recursive calls, or 2 for each row queried.

26 Effective Oracle by Design

A possible solution is to fold the PL/SQL routine right into the query itself; for example, by using a complex CASE statement or by selecting a SELECT. The above query could have been written simply as:

```
ops$tkyte@ORA920> select ename, (select count(*) from dual)
      2  from scott.emp;
```

That will not incur any recursive SQL calls, and it performs the same operation.

As for the USER local variable, I would recommend setting that once per session (using a PL/SQL package), rather than referring to the USER pseudo column throughout the code. Every time you declare a variable and default it to USER, that will be a recursive SQL call. It's better to have a package global variable that is defaulted to USER and just reference that instead.

Side Effects from Modifications Recursive calls may also occur when you are doing a modification and many side effects (triggers, function-based indexes, and so on) are happening. Take the following, for example:

```
ops$tkyte@ORA920> create table t ( x int );
Table created.
```

```
ops$tkyte@ORA920> create trigger t_trigger before insert on t for each row
      2  begin
      3      for x in ( select *
      4                  from dual
      5                  where :new.x > (select count(*) from emp))
      6      loop
      7          raise_application_error( -20001, 'check failed' );
      8      end loop;
      9  end;
     10  /
Trigger created.
```

```
ops$tkyte@ORA920> insert into t select 1 from all_users;
38 rows created.
```

```
ops$tkyte@ORA920> set autotrace traceonly statistics
```

```
ops$tkyte@ORA920> insert into t select 1 from all_users;
38 rows created.
```

Statistics

```
-----
          39  recursive calls
```

...

```
          38  rows processed
```

```
ops$tkyte@ORA920> set autotrace off
```

Here, firing the trigger, and then running that query for each row processed by the trigger, generated all of the recursive SQL calls. Generally, this is not something you can avoid (because you would need to remove the trigger entirely), but you can minimize it by writing an efficient trigger—avoiding as much recursive SQL as possible and moving SQL out of a trigger into a package (covered in detail in Chapter 5).

Space Requests You may see large recursive SQL operations performed to satisfy requests for space, due to disk sorts or as the result of large modifications to a table that require it to extend. This is generally not a problem with locally managed tablespaces, where space is managed as bitmaps in the datafile headers. It can happen with dictionary-managed tablespaces, where space is managed in database tables much like your own data is managed.

Consider this example performed using a dictionary-managed tablespace. We'll start by creating a table with small extents (every extent will be 64KB).

```
ops$tkyte@ORA817DEV> create tablespace testing
  2 datafile '/tmp/testing.dbf' size 1m reuse
  3 autoextend on next 1m
  4 extent management dictionary
  5 /
Tablespace created.

ops$tkyte@ORA817DEV> create table t
  2 storage( initial 64k next 64k pctincrease 0 )
  3 tablespace testing
  4 as
  5 select * from all_objects where 1=0;
Table created.
```

Next, we'll insert a couple of rows into this table. The INSERT statement is carefully crafted using bind variables so that subsequent executes of that query will be soft parsed, so the recursive SQL we are attempting to measure will not include parse-related recursive statements. Consider this step a priming of the pump to get started, to warm up the engine:

```
ops$tkyte@ORA817DEV> variable n number;
ops$tkyte@ORA817DEV> exec :n := 5;
PL/SQL procedure successfully completed.
ops$tkyte@ORA817DEV> insert into t
  2 select * from all_objects where rownum < :n;
  5 rows created.
```

And now we are ready to perform a mass insert into this table. Setting the bind variable to a large number and simply reexecuting that same SQL INSERT statement:

```
ops$tkyte@ORA817DEV> set autotrace traceonly statistics;

ops$tkyte@ORA817DEV> exec :n := 99999
PL/SQL procedure successfully completed.
```

28 Effective Oracle by Design

```
ops$tkyte@ORA817DEV> insert into t
  2 select * from all_objects where rownum < :n;
23698 rows created.
```

Statistics

```
-----
      2910 recursive calls
2441  db block gets
...
      23698 rows processed
```

```
ops$tkyte@ORA817DEV> set autotrace off
```

That is a lot of recursive SQL for that insert operation. No triggers or PL/SQL function calls are involved. This was all due to space management. So, can we decrease that easily? Sure, by using locally managed tablespaces:

```
ops$tkyte@ORA817DEV> create tablespace testing_lmt
  2 datafile '/tmp/testing_lmt.dbf' size 1m reuse
  3 autoextend on next 1m
  4 extent management local
  5 uniform size 64k
  6 /
Tablespace created.
```

```
ops$tkyte@ORA817DEV> drop table t;
Table dropped.
```

```
ops$tkyte@ORA817DEV> create table t
  2 tablespace testing_lmt
  3 as
  4 select * from all_objects where 1=0;
Table created.
```

That emulates our dictionary-managed table example exactly. The table T will have 64KB extents. Now, let's repeat the same insert test.

```
ops$tkyte@ORA817DEV> variable n number;
ops$tkyte@ORA817DEV> exec :n := 5;
PL/SQL procedure successfully completed.
```

```
ops$tkyte@ORA817DEV> insert into t
  2 select * from all_objects where rownum < :n;
4 rows created.
```

```
ops$tkyte@ORA817DEV> set autotrace traceonly statistics;
ops$tkyte@ORA817DEV> exec :n := 99999
PL/SQL procedure successfully completed.
```

```
ops$tkyte@ORA817DEV> insert into t
  2 select * from all_objects where rownum < :n;
23698 rows created.
```

```
Statistics
-----
```

```
800 recursive calls
```

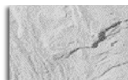
```
2501 db block gets
```

```
...
```

```
23698 rows processed
```

```
ops$tkyte@ORA817DEV> set autotrace off
```

This is much better. What are the 800 remaining recursive SQL queries? By using SQL_TRACE and TKPROF to analyze that (as explained in the section on TKPROF, later in this chapter), we would see that these were due to tablespace quota management—a series of SELECT and UPDATE statements to manage the quota information. This recursive SQL is truly unavoidable. Oracle will always be checking our quotas as we require more space. Still, we might be able to minimize the recursive SQL by using system-allocated extents or by using a larger uniform extent size to reduce the number of extents needed to hold this much data.



NOTE

If you are testing this on 9iR2, you may discover that you cannot run this example. If your system tablespace was created as a locally managed tablespace, you cannot create any dictionary managed tablespaces at all. This is why my previous example was executed in my Oracle8i release 3 (8.1.7) database, as noted in my SQL prompt!

Recursive SQL Wrap-up Keep in mind that recursive SQL is something to be avoided *if possible*, but not anything to lose sleep over if it is not avoidable. Don't blindly try to get this to zero, because it is not possible or practical all of the time. If you see hundreds or thousands of recursive SQL calls, check it out. Find out what is causing it (SQL_TRACE will help greatly here), understand why it happens, and work on fixing the cause if possible.

Db Block Gets and Consistent Gets

Blocks may be retrieved and used by Oracle in one of two ways: current or consistent. A current mode get is a retrieval of a block as it exists right now. You will see these most frequently during modification statements, which must update only the latest copy of the block. Consistent Gets are retrieval of blocks from the buffer cache in “read consistent” mode and may include read asides to UNDO (rollback segments). A query will generally perform “consistent gets.”

These are the most important parts of the AUTOTRACE report. They represent your logical I/Os—the number of times you had to latch a buffer in order to inspect it. If you recall from Chapter 1, a *latch* is just another name for a lock; a latch is a serialization device. The less we latch, the better. In general, the less logical I/O we can do, the better.

Query Tuning But how do we decrease the logical I/Os? In many cases, achieving this requires letting go of old myths, in particular, the myth that if your query isn't using indexes, the optimizer *must* be doing the wrong thing.

"I have created two tables:

```
create table I1(n number primary key, v varchar2(10));
create table I2(n number primary key, v varchar2(10));
and a map table
create table MAP
(n number primary key,
 i1 number referencing I1(n),
 i2 number referencing I2(n));
create unique index IDX_MAP on MAP(i1, i2)
```

Now, when I take the EXPLAIN PLAN for the query:

```
select *
  from i1, map, i2
 where i1.n = map.i1
    and i2.n = map.i2
    and i1.v = 'x'
    and i2.v = 'y';
```

I see the plan as:

```
Execution Plan
-----
0          SELECT STATEMENT Optimizer=CHOOSE
1    0      NESTED LOOPS
2      1        NESTED LOOPS
3        2          TABLE ACCESS (FULL) OF 'MAP'
4        2          TABLE ACCESS (BY INDEX ROWID) OF 'I2'
5          4            INDEX (UNIQUE SCAN) OF 'SYS_C00683648' (UNIQUE)
6        1          TABLE ACCESS (BY INDEX ROWID) OF 'I1'
7          6            INDEX (UNIQUE SCAN) OF 'SYS_C00683647' (UNIQUE)
```

Is there any way to avoid the full-table scan on the MAP table? Whatever I try, one table is always going for a full scan. What should I do to avoid a full scan in such a case?"

My response was simple. I started with telling him to repeat the following:

Full scans are not always evil; indexes are not always good.

Say this over and over until he believed it. Then I asked him to look at his query and to tell me how a full scan could be avoided. Using the existing data structures, what plan could a human come up with that does not involve a full-table or index scan? I do not see any possible plan myself.

Additionally, given the existing structures, the indexes that are being used are actually deadly here. I can tell by the AUTOTRACE output that he is using the RBO (because there is no Cost, Card, and Bytes information). The plan the RBO came up with is a really bad plan in general. The CBO would be smarter and stop using the indexes. So, one solution is to simply analyze the tables and use a plan that avoids the indexes altogether!

Here, I'll show you a simple example. We'll join two tables together. Before we run the query, we'll use AUTOTRACE to see the plans that would be generated and try to outguess the optimizer with hints (in the erroneous belief that if the optimizer avoids an index, it has done the wrong thing).

```
ops$tkyte@ORA920> insert into i1
  2 select rownum, rpad(' ',10,' ') from all_objects;
30020 rows created.
```

```
ops$tkyte@ORA920> insert into i2
  2 select rownum, rpad(' ',10,' ') from all_objects;
30020 rows created.
```

```
ops$tkyte@ORA920> insert into map
  2 select rownum, rownum, rownum from all_objects;
30020 rows created.
```

```
ops$tkyte@ORA920> set autotrace traceonly
```

```
ops$tkyte@ORA920> select *
  2   from
  3       i1,
  4       map,
  5       i2
  6   where      i1.n = map.i1
  7   and i2.n = map.i2
  8   and i1.v = 'x'
  9   and i2.v = 'y';
```

```
no rows selected
```

Execution Plan

```
-----
 0      SELECT STATEMENT Optimizer=CHOOSE
 1      0      NESTED LOOPS
 2      1      NESTED LOOPS
 3      2      TABLE ACCESS (FULL) OF 'MAP'
 4      2      TABLE ACCESS (BY INDEX ROWID) OF 'I2'
 5      4      INDEX (UNIQUE SCAN) OF 'SYS_C003755' (UNIQUE)
 6      1      TABLE ACCESS (BY INDEX ROWID) OF 'I1'
 7      6      INDEX (UNIQUE SCAN) OF 'SYS_C003754' (UNIQUE)
```

Statistics

```
-----
```

32 Effective Oracle by Design

```

0 recursive calls
0 db block gets
60127 consistent gets
0 physical reads
60 redo size
513 bytes sent via SQL*Net to client
368 bytes received via SQL*Net from client
1 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
0 rows processed
```

At this point, we can see that the performance of this query is poor, with more than 60,000 logical I/Os. For a query that ultimately returns no data, this is very high. Now, let's give the CBO an opportunity.

```
ops$tkyte@ORA920> analyze table i1 compute statistics;
Table analyzed.
```

```
ops$tkyte@ORA920> analyze table i2 compute statistics;
Table analyzed.
```

```
ops$tkyte@ORA920> analyze table map compute statistics;
Table analyzed.
```

```
ops$tkyte@ORA920> select *
2   from
3       i1,
4       map,
5       i2
6   where    i1.n = map.i1
7   and i2.n = map.i2
8   and i1.v = 'x'
9   and i2.v = 'y';
no rows selected
```

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=21 Card=1 Bytes=40)
1    0      NESTED LOOPS (Cost=21 Card=1 Bytes=40)
2    1        HASH JOIN (Cost=20 Card=1 Bytes=26)
3    2          TABLE ACCESS (FULL) OF 'I1' (Cost=10 Card=1 Bytes=14)
4    2          TABLE ACCESS (FULL) OF 'MAP' (Cost=9 Card=30020 Bytes=360240)
5    1          TABLE ACCESS (BY INDEX ROWID) OF 'I2' (Cost=1 Card=1 Bytes=14)
6    5            INDEX (UNIQUE SCAN) OF 'SYS_C003755' (UNIQUE)
```

Statistics

```
-----
0 recursive calls
```



```

          0 db block gets
          92 consistent gets
0 physical reads
          0 redo size
        513 bytes sent via SQL*Net to client
        368 bytes received via SQL*Net from client
          1 SQL*Net roundtrips to/from client
          0 sorts (memory)
          0 sorts (disk)
          0 rows processed

```

```
ops$tkyte@ORA920> set autotrace off
```

As you can see, by avoiding the use of an index, we increased the performance and resource use of this query by many orders of magnitude. This is a great start.

Now, let's consider the predicates `i1.v = value` and `i2.v = value`. Perhaps creating an index on `i1.v` or `i2.v` would be helpful.

```
ops$tkyte@ORA920> create index i1_idx on i1(v);
Index created.
```

```
ops$tkyte@ORA920> analyze table i1 compute statistics;
Table analyzed.
```

```
ops$tkyte@ORA920> set autotrace traceonly
```

```
ops$tkyte@ORA920> select *
  2   from
  3       i1,
  4       map,
  5       i2
  6   where    i1.n = map.i1
  7   and i2.n = map.i2
  8   and i1.v = 'x'
  9   and i2.v = 'y';

```

```
no rows selected
```

Execution Plan

```

-----
 0      SELECT STATEMENT Optimizer=CHOOSE (Cost=13 Card=1 Bytes=40)
 1    0      NESTED LOOPS (Cost=13 Card=1 Bytes=40)
 2    1        HASH JOIN (Cost=12 Card=1 Bytes=26)
 3    2          TABLE ACCESS (BY INDEX ROWID) OF 'I1'
                (Cost=2 Card=1 Bytes=14)
 4    3            INDEX (RANGE SCAN) OF 'I1_IDX' (NON-UNIQUE)
                (Cost=1 Card=1)
 5    2          TABLE ACCESS (FULL) OF 'MAP'
                (Cost=9 Card=30020 Bytes=360240)

```

34 Effective Oracle by Design

```
6      1      TABLE ACCESS (BY INDEX ROWID) OF 'I2' (Cost=1 Card=1 Bytes=14)
7      6      INDEX (UNIQUE SCAN) OF 'SYS_C003755' (UNIQUE)
```

Statistics

```
-----
          0 recursive calls
          0 db block gets
          2 consistent gets
0 physical reads
          0 redo size
        513 bytes sent via SQL*Net to client
       368 bytes received via SQL*Net from client
          1 SQL*Net roundtrips to/from client
          0 sorts (memory)
          0 sorts (disk)
          0 rows processed
```

```
ops$tkyte@ORA920> set autotrace off
```

This just helps prove that indexes are not always best and full scans are not always to be avoided. The solution to this problem is to use the CBO and to properly index the data structures according to our data-retrieval needs.

In general, the major way to reduce db block gets and consistent gets is via query tuning. However, to be successful, you must keep an open mind and have a good understanding of the available access paths. You must have a good grasp of the SQL language, including the entire suite of functionality, so you can understand the difference between NOT IN and NOT EXISTS, when WHERE EXISTS would be appropriate, and when WHERE IN is a better choice. One of the best ways to discover all of this is through simple tests, such as the ones I've been demonstrating.

Array Size Effects The array size is the number of rows fetched (or sent, in the case of inserts, updates, and deletes) by the server at a time. It can have a dramatic effect on performance.

To demonstrate the effects of array size, we'll run the same query a couple of times and look at just the consistent get differences between runs:

```
ops$tkyte@ORA920> create table t as select * from all_objects;
Table created.
```

```
ops$tkyte@ORA920> set autotrace traceonly statistics;
```

```
ops$tkyte@ORA920> set arraysize 2
ops$tkyte@ORA920> select * from t;
29352 rows selected.
```

Statistics

```
-----
14889 consistent gets
```

Note how one half of 29,352 (rows fetched) is very close to 14,889, the number of consistent gets. Every row we fetched from the server actually caused it to send two rows back. So, for every two rows of data, we needed to do a logical I/O to get the data. Oracle got a block, took two rows from it, and sent it to SQL*Plus. Then SQL*Plus asked for the next two rows, and Oracle got that block again or got the next block, if we had already fetched the data, and returned the next two rows, and so on.

Next, let's increase the array size:

```
ops$tkyte@ORA920> set arraysize 5
ops$tkyte@ORA920> select * from t;
29352 rows selected.
```

Statistics

```
-----
          6173  consistent gets
...
```

Now, 29,352 divided by 5 is about 5,871, and that would be the least amount of consistent gets we would be able to achieve (the actual observed number of consistent gets is slightly higher). All that means is sometimes in order to get two rows, we needed to get two blocks: we got the last row from one block and the first row from the next block.

Let's increase the array size again:

```
ops$tkyte@ORA920> set arraysize 10
ops$tkyte@ORA920> select * from t;
29352 rows selected.
```

Statistics

```
-----
          3285  consistent gets
...
```

```
ops$tkyte@ORA920> set arraysize 15
ops$tkyte@ORA920> select * from t;
29352 rows selected.
```

Statistics

```
-----
          2333  consistent gets
...
```

```
ops$tkyte@ORA920> set arraysize 100
ops$tkyte@ORA920> select * from t;
29352 rows selected.
```

Statistics

```
-----
           693  consistent gets
```

...

```
ops$tkyte@ORA920> set arraysize 5000
ops$tkyte@ORA920> select * from t;
29352 rows selected.
```

```
Statistics
```

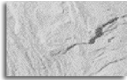
```
-----
          410  consistent gets
```

...

```
ops$tkyte@ORA920> set autotrace off
```

As you can see, as the array size goes up, the number of consistent gets goes down. So, does that mean you should set your array size to 5,000, as in this last test? Absolutely *not*.

If you notice, the overall number of consistent gets has not dropped dramatically between array sizes of 100 and 5,000. However, the amount of RAM needed on the client and server has gone up with the increased array size. The client must now be able to cache 5,000 rows. Not only that, but it makes our performance look choppy: The server works really hard and fast to get 5,000 rows, then the client works really hard and fast to process 5,000 rows, then the server, then the client, and so on. It would be better to have more of a stream of information flowing: Ask for 100 rows, get 100 rows, ask for 100, process 100, and so on. That way, both the client and server are more or less continuously processing data, rather than the processing occurring in small bursts.



NOTE

A common criticism of PL/SQL is its slow performance. However, this is not a PL/SQL problem per se, but it's due to the fact that people typically code PL/SQL in a very row-at-a-time oriented fashion. Given that even native dynamic SQL can now be bulk-based from Oracle9i, there is no longer any reason for this approach. We'll investigate this further in Chapter 9.

I've found empirically that somewhere between 100 and 500 is a nice array size. Diminishing marginal returns kick in shortly after that range. Performance will actually decrease with larger array sizes.

Virtually every programming environment I've come across—from Pro*C, to OCI, to Java/JDBC, and even VB/ODBC—allows you to tweak the array size. Refer to the documentation for your environment for specific instructions on handling array size.

Physical Reads

The physical reads statistic is a measure of how much real I/O, or physical I/O, your query performed. A physical read of table or index data places the block into the buffer cache. Then we perform a logical I/O to retrieve the block. Hence, most physical reads are immediately followed by a logical I/O!

There are two major types of common physical I/Os:

- **Reading your data in from datafiles** Doing I/O to the datafiles to retrieve index and table data. These operations will be followed immediately by a logical I/O to the cache.

- **Direct reads from TEMP** This is in response to a sort area or hash area not being large enough to support the entire sort/hash in memory. Oracle is forced to swap out some of the data to TEMP and read it back in later. These physical reads bypass the buffer cache and will not incur a logical I/O.

There is not a lot we can do about the first type of I/O, the reading of data in from disk. After all, if it is not in the buffer cache, it has to get there somehow. If you run a small query—a query that performs hundreds of logical I/Os (consistent gets)—and you repeatedly observe physical I/Os being performed, that could be an indication your buffer cache is on the small side (there isn't sufficient space to cache even the results of your small query with hundreds of logical I/Os). Generally, the number of physical reads should go down for most queries after you run the query once.

Many people think that they must flush the buffer cache before tuning, to emulate the real world. I feel quite the opposite. If you tune with logical I/Os (consistent gets) in mind, the physical I/Os will take care of themselves over time (assuming you ensure an even distribution of I/O across disks and such). Flushing the buffer cache during tuning is as artificial as running the query a couple of times and using the last results. Flushing the buffer cache gets rid of tons of information that rightly would be there in the real world when you run the query.

I suggest running the query twice. If the logical I/Os are small as a proportion of your buffer cache, yet you are still seeing physical reads, that most likely indicates direct reads from your temporary tablespace. We will take a look at fixing that problem here in this section. Later, when we talk about TKPROF and SQL_TRACE, we'll go into details about finding where physical I/Os are taking place.

To detect direct reads from TEMP, first disable automatic process global area (PGA) management by the server and physically set the sort area size and hash area size. If you are using Oracle9i and have `WORKAREA_SIZE_POLICY = AUTO`, the settings for the hash area size and sort area size, among others, are ignored (and this example won't work!). In this example, I've set the hash area size artificially low, in order to have the optimizer consistently choose a sort merge join to exercise the sort area size. I'll show the query plan for the first execution, but chop it out for the subsequent ones.

NOTE

ALTER SESSION SET WORKAREA_SIZE_POLICY is a new feature of Oracle9i. This example will work in Oracle8i and earlier without issuing that command.

```
ops$tkyte@ORA920> alter session set workarea_size_policy = manual;
Session altered.
```

```
ops$tkyte@ORA920> alter session set hash_area_size = 1024;
Session altered.
```

```
ops$tkyte@ORA920> create table t as select * from all_objects;
Table created.
```

```
ops$tkyte@ORA920> analyze table t compute statistics
2 for table for columns object_id;
Table analyzed.
```

38 Effective Oracle by Design

That sets up our test. Now, we'll try a big sort with 100KB, 1MB, and 10MB sort areas to see what happens in each case.

```
ops$tkyte@ORA920> set autotrace traceonly
ops$tkyte@ORA920> alter session set sort_area_size = 102400;
Session altered.
```

```
ops$tkyte@ORA920> select *
   2   from t t1, t t2
   3   where t1.object_id = t2.object_id
   4   /
29366 rows selected.
```

[results snipped out - second run after parsing and warming up the cache is relevant]

```
ops$tkyte@ORA920> /
29366 rows selected.
```

Execution Plan

```
-----
0   SELECT STATEMENT Optimizer=CHOOSE (Cost=4264 Card=29366 Bytes=5638272)
1  0   MERGE JOIN (Cost=4264 Card=29366 Bytes=5638272)
2    1     SORT (JOIN) (Cost=2132 Card=29366 Bytes=2819136)
3     2      TABLE ACCESS (FULL) OF 'T' (Cost=42 Card=29366 Bytes=2819136)
4    1     SORT (JOIN) (Cost=2132 Card=29366 Bytes=2819136)
5     4      TABLE ACCESS (FULL) OF 'T' (Cost=42 Card=29366 Bytes=2819136)
```

Statistics

```
-----
          0   recursive calls
         216   db block gets
810 consistent gets
4486 physical reads
0   redo size
 2474401   bytes sent via SQL*Net to client
   22026   bytes received via SQL*Net from client
    1959   SQL*Net roundtrips to/from client
          0   sorts (memory)
2 sorts (disk)
29366   rows processed
```

Here, we can see that physical reads far exceed logical I/Os. That is one indicator that we had to swap to disk. The 2 sorts (disk) statistic helps back that up, but do not rely exclusively on that statistic to point this out. Other operations that use temporary space, such as a hash join, don't report sorts to disk, because they didn't sort! If the physical reads do not disappear on the second run, even though there are only hundreds of logical I/Os (you would expect that data to

be in the cache), or the physical reads outnumber the logical I/Os, you should suspect that the allocated memory was exceeded.

So, what happens when we increase the sort area size for this process?

```
ops$tkyte@ORA920> alter session set sort_area_size = 1024000;
Session altered.
```

```
ops$tkyte@ORA920> select *
2   from t t1, t t2
3   where t1.object_id = t2.object_id
4   /
29366 rows selected.
```

Statistics

```
-----
0   recursive calls
12  db block gets
810 consistent gets
1222 physical reads
0   redo size
2474401 bytes sent via SQL*Net to client
22026  bytes received via SQL*Net from client
1959   SQL*Net roundtrips to/from client
0   sorts (memory)
2 sorts (disk)
29366 rows processed
```

The physical reads went down. We were able to store ten times the data in RAM. We did much less swapping in and out to TEMP.

Let's make the sort area bigger and see what happens.

```
ops$tkyte@ORA920> alter session set sort_area_size = 10240000;
Session altered.
```

```
ops$tkyte@ORA920> select *
2   from t t1, t t2
3   where t1.object_id = t2.object_id
4   /
29366 rows selected.
```

Statistics

```
-----
0   recursive calls
0   db block gets
810 consistent gets
0 physical reads
0   redo size
2474401 bytes sent via SQL*Net to client
22026  bytes received via SQL*Net from client
```

```

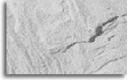
1959 SQL*Net roundtrips to/from client
2  sorts (memory)
0  sorts (disk)
29366 rows processed

ops$tkyte@ORA920> set autotrace off

```

The need for physical reads went away entirely.

Some sites run with an absurdly small sort area size (hash area size and others). They do this out of the mistaken belief that, by setting the sort area size, they are permanently allocating that much memory per session. The sort area size settings control how much memory will be dynamically allocated at runtime in order to satisfy a sort request. After the sort is completed, this memory will be shrunk back to the sort area retained size. After the query result set is exhausted, this memory will be released entirely. The default settings are far too small, and even a modest increase would benefit many sites. Their overall performance may be enhanced simply by increasing this parameter. The optimizer will recognize the sort area size is larger and generate query plans that take advantage of it.



NOTE

This is not to say that you should rush and up your sort_area_size (refer back to Chapter 1 and test, test, test). Also note that in most cases you would be implicitly bumping up the hash area size as well.

In Oracle9i and above, using the automated work area size policy, the sort area size is not an issue. Under this policy, the DBA tells Oracle how much dynamic memory it is allowed to use from the operating system for processing (separate from the SGA memory). Then Oracle will set sort areas, hash areas, and the like automatically, using 10MB when appropriate, 1MB when that is better, and 100KB when that makes sense. In fact, the amount used can vary from statement to statement in your session as the load goes up and down over time. The decision on the amount of memory to use is made dynamically for every single statement your session performs.

Redo Size

The redo size statistic shows how much redo your statement generated when executed. This is most useful when judging the efficiency of large bulk operations. This comes into play most often with direct path inserts or CREATE TABLE AS SELECT (CTAS) statements. The amount of redo generated by MERGE, UPDATE, or DELETE statements will, in many cases, be outside of our control. If you perform these operations with as few indexes enabled as possible, you can reduce the amount of redo generated, but you will have to ultimately rebuild those disabled indexes. This is practical only in specialized environments when you are performing large data loads in general, such as a data warehouse.

Bulk Loads Let's look at a common problem: How can you load a large number of rows into an existing table efficiently? If you are talking about a couple of thousand rows in a very large table, a straight insert is probably the right way to go. If you are talking about tens, or hundreds of thousands, or even millions of rows of data in a very large table, other methods will prevail.

“In development, our bulk inserts work very fast and generate very little redo. When we move into production, our INSERT /*+ append */ statements work much slower and generate many gigabytes of redo log, filling up our archive log destination. What is going wrong?”

It is not that anything is going wrong. It is that many people run the development and test machines in NOARCHIVELOG mode and production in ARCHIVELOG mode. This difference can be crucial, and is yet another reason why your test environment should mirror your production environment. You want to catch these differences in your testing and development phase, not in production!

Let’s look at an example of a bulk load of data into a table. Since everyone has the ALL_OBJECTS view available, we’ll use that for demonstration purposes. In real life, we would be loading from an external table (in Oracle9i and above), or we would be processing a file loaded by the SQL*Loader tool to load (in Oracle8i and earlier). We’ll begin by creating a table to load into:

```
ops$tkyte@ORA920> create table big_table
2 as
3 select *
4   from all_objects
5  where 1=0;
Table created.
```

That just created a table that is structurally the same as ALL_OBJECTS but has no data. So, let’s fill it up:

```
ops$tkyte@ORA920> set autotrace on statistics;

ops$tkyte@ORA920> insert into big_table select * from all_objects;
29368 rows created.

Statistics
-----
...
3277944 redo size

ops$tkyte@ORA920> insert /*+ APPEND */ into big_table select * from
all_objects;
29368 rows created.

Statistics
-----
...
3328820 redo size

ops$tkyte@ORA920> commit;
Commit complete.
```

42 Effective Oracle by Design

This is something that confuses many Oracle developers and DBAs. They think that second insert operation should not have generated any redo. We used the syntax for the direct-path insert (just like a direct-path load in SQL*Loader), so it should bypass all of the redo generation. But that is not the case. A direct-path insert will only bypass redo generation in two cases:

- You are using a NOARCHIVELOG mode database (my database is in ARCHIVELOG mode).
- You are performing the operation on a table marked as NOLOGGING.

The insert `/*+ APPEND */` will *minimize* redo generation in all cases, as it minimizes the amount of UNDO generated. The redo that would otherwise be generated for the UNDO information is not created, but ultimately the logging mode of the target table is what will dictate whether redo is generated for the table or not. So, let's put the table into NOLOGGING mode and retry the operation:

```
ops$tkyte@ORA920> alter table big_table nologging;
Table altered.
```

```
ops$tkyte@ORA920> insert into big_table
  2  select * from all_objects;
29368 rows created.
```

Statistics

...

3239724 redo size

```
ops$tkyte@ORA920> insert /*+ APPEND */ into big_table
  2  select * from all_objects;
29368 rows created.
```

Statistics

...

7536 redo size


```
ops$tkyte@ORA920> commit;
Commit complete.
```

Already, we can see how AUTOTRACE is useful in showing how similar commands might perform very differently. Using just a normal insert, we generated over 3.2MB of redo log. The second direct-path insert generated virtually no redo.

The fact that the normal insert generated redo is a point of some confusion as well. Just setting a table to NOLOGGING does not mean that all redo is prevented against this table. Only certain explicit, bulk operations such as this direct-path insert, will not be logged in the normal fashion. All other operations—such as insert, update, delete, and merge—will be logged as normal.

So, does that mean we should set all of our tables to NOLOGGING and use `/*+ APPEND */` on all inserts? No, not a chance.

First, realize that this option works on only INSERT as SELECT statements (bulk inserts). It does *not* work with INSERT VALUES statements. I've seen many pieces of code with:

```
 insert /*+ append */ into t values ( ... )
```

When I ask about it, the developer says, “Well, I didn’t need that logged, so I turned it off.” That just proves that the developer doesn’t understand what this option does at all! So, using this on an INSERT with a VALUES clause only makes you look bad (it has no other effect!). By all means, if you can convert your code into SET AT A TIME logic and use /*+ APPEND */ on very large bulk operations, after coordinating with the DBA, do so. But don’t even attempt to use it on a single row.


An important reason to avoid the NOLOGGING setting is its obvious result: This operation will not be logged. While that might sound excellent to some developers, that should send a shudder down a DBA’s spine. You have this table loaded, and people starting doing modifications against it. The next day, the disk crashes. “No problem,” says the DBA, “I’ll just restore from the backup two days ago.” She rolls forward and finds this table you loaded yesterday wasn’t recovered! All of the work of the last day is lost! Why? Because you prevented the redo from being generated. It is not in the archive logs, and it cannot be recovered. This underscores an important point: When using NOLOGGING operations, the development and DBA teams must be in close contact. The DBA needs to know to schedule a backup of the affected datafiles as soon as possible afterwards, to prevent data loss.

Here are a few other reasons not to set NOLOGGING and use /*+ APPEND */ on all inserts:

- The direct-path insert writes data above the high-water mark for the table, ignoring any free space on the free lists, just as a direct-path load will. If you delete many of the rows in a table and then INSERT /*+ APPEND */ into it, you will not be reusing any of that space.
- You must commit after a successful direct-path insert before reading from that table in that transaction. This could put commits into your transactions where they don’t really belong.
- Only one session at a time can direct-path insert into a table. All other modifications are blocked. This operation serializes (but you can do parallel direct-path inserts from a single session).

Again, this option should be used with care and thought. You need coordination among the people using it and the DBAs, so they can schedule that backup right after this occurs.

Redo and Index Operations What happens if you are doing INSERT /*+ APPEND */ as SELECT, with the table set NOLOGGING, but you still have a lot of redo and archives? What could be wrong? The answer is that you have indexes on the table, and the indexes cannot just be appended to; they must be merged into. Since you need to merge data into them, you need redo in order to recover from instance failure (if your system crashed in the middle of the index operations, you would end up with a corrupt index structure). Hence, redo is generated for the index operations. We can see this readily by continuing on with this example:

```
 ops$tkyte@ORA920> create index big_table_idx on
2 big_table(owner,object_type,object_name);
Index created.
```

44 Effective Oracle by Design

```
ops$tkyte@ORA920> insert /*+ APPEND */ into big_table
  2  select * from all_objects;
29369 rows created.
```

Statistics

...

18020324 redo size

```
ops$tkyte@ORA920> commit;
Commit complete.
```

Wow, what a difference an index can make! Indexes are complex data structures, so maintaining them can be expensive. Here, we know that the table data should generate about 3.2MB of redo, but this single index generated 18MB of redo!

If this were a bulk load into a data warehouse or data mart, I would something similar to the following:

1. Set the indexes to the UNUSABLE state (not dropping them, just setting them as UNUSABLE for now).
2. Set the session to skip UNUSUABLE indexes and do the bulk load.
3. Re-enable the indexes.

Here are the steps you can take in SQL*Plus to see this (SQLLDR would be similar, but you would use the `skip_index_maintenance` parameter instead of the first two ALTER commands):

```
ops$tkyte@ORA920> alter index big_table_idx unusable;
Index altered.
```

```
ops$tkyte@ORA920> alter session set skip_unusable_indexes=true;
Session altered.
```

```
ops$tkyte@ORA920> insert /*+ APPEND */ into big_table
  2  select * from all_objects;
29369 rows created.
```

Statistics

...

7588 redo size

```
ops$tkyte@ORA920> alter index big_table_idx rebuild nologging;
Index altered.
```

Now, all we need to do is schedule a backup of the affected datafiles, and we are finished with our load!

Why not just drop the indexes and re-create them? I've seen far too many systems where the DROP INDEX command worked, but the subsequent CREATE INDEX command failed for some reason. No one noticed they just lost an index, and performance went down the tubes.



TIP

If nothing changed, but after your last data load, performance has been really bad, check that you aren't missing your index. If this is the case, just create the index, and your performance should return to normal.

If you don't ever drop the index, but set it to UNUSABLE instead, you can never lose the index. If the command to re-enable the index fails, when the users run queries that need that index, they will get an error message, which, of course, they will report immediately. So, rather than having the system suffer from mysterious performance issues for hours or days, the DBA will get a call early on and fix the problem.

SQL*Net Statistics

There are three pieces to the SQL*Net statistics:

- How much you sent to the server (bytes received via SQL*Net from client)
- How much the server sent to you (bytes sent via SQL*Net to client)
- How many round-trips were made (SQL*Net roundtrips to/from client)

The ones you have the most control over are the last two. You want to minimize the data sent to you. The way you control that is by selecting *only* the columns that are relevant to you. Frequently, I see people coding `select * from table`, and then using only one or two columns (out of thirty or forty column). Not only does this flood the network with a ton of unnecessary data, it consumes additional RAM, and it can radically affect the efficiency of your query plans. To avoid such problem, select only those columns that are needed to solve your problem—no more, no less.

The number of round-trips is tunable when you are processing a SELECT command. To see this, let's reuse the earlier example that showed the effects of array size on the consistent gets statistics. By just providing the array size and number of round-trips, we can see the measurable impact the array size has on the SQL*Net roundtrips to/from client statistic:

```
ops$tkyte@ORA920> set arraysize 2
ops$tkyte@ORA920> select * from t;
29369 rows selected.

...
14686  SQL*Net roundtrips to/from client

ops$tkyte@ORA920> set arraysize 5
...
5875  SQL*Net roundtrips to/from client
```

46 Effective Oracle by Design

```
ops$tkyte@ORA920> set arraysize 10
...
      2938  SQL*Net roundtrips to/from client

ops$tkyte@ORA920> set arraysize 15
...
      1959  SQL*Net roundtrips to/from client

ops$tkyte@ORA920> set arraysize 100
...
      295   SQL*Net roundtrips to/from client

ops$tkyte@ORA920> set arraysize 5000
...
      7     SQL*Net roundtrips to/from client
```

In general, the fewer round-trips, the better—to a point! Generally, an array fetch size of between 100 and 500 results in the best overall performance/memory usage.

Sorts and Rows Processed

The final three statistics deal with sorts and row processing:

- Sorts (memory) shows how many sorts you did entirely in memory (no swapping to disks).
- Sorts (disk) shows how many sorts you did that required some temporary disk space.
- Rows processed shows how many rows were affected (either returned from a SELECT statement or modified due to an INSERT, UPDATE, DELETE or MERGE statement).

Our goal is to reduce the number of sorts (disk). As demonstrated earlier in the section about the physical reads statistic, one method of accomplishing this is to set a sort area size that is more appropriate for what you are doing; that is, in practice, most sort area size settings are too small. Another method is plain, old-fashioned query tuning. Determine if there is an alternative yet equivalent method that you can use to reduce the size of the sort you need to perform, or even to remove the sort entirely.

Note however, that a “sorts to disk” of zero *does not mean you are bypassing TEMP altogether*. It is entirely possible for the sort to be done in memory but have the results of the sort written to disk. This will most commonly happen when the `sort_area_retained_size` (the amount of sorted data to retain in memory after sorting) is set smaller than the `sort_area_size`.

This completes the discussion of the AUTOTRACE statistics. As you’ve seen, AUTOTRACE is a fairly powerful tool. It is a large step above EXPLAIN PLAN, which can only show you what plan would be used, not what actually happens when you use that plan. However, it is a step below TKPROF, the tool we’ll look at next.

TKPROF

Oracle has the ability to turn on a fairly low-level tracing capability for us (generally done via the ALTER SESSION command). Once tracing is enabled, Oracle will record all of the SQL and top-level PL/SQL calls our application makes to a trace file on the server (the database server

machine; never the client machine). This trace file will not only have our SQL and PL/SQL calls, but it will also contain our timing information, possibly information on wait events (what is slowing us down), how many logical I/Os and physical I/Os we performed, the CPU and wall clock (elapsed) timings, numbers of rows processed, query plans with row counts, and more. This trace file by itself is fairly hard to read. What we really want to do is generate a report from this trace file in a user-friendly format. That is TKPROF's only goal in life: to turn a trace file into something we can use easily.

As I've already mentioned several times, TKPROF is my favorite tuning tool. It's also one that seems to be widely overlooked by many people. Sometimes, this is out of ignorance—they didn't know it existed. Many times this is out of prevention—the DBA prevents developers from having access to this tool by preventing access to the necessary trace files. Here, I'll remove the ignorance and also provide a method to allow access to the trace files in a manner that might squelch the qualms of the DBA. First, we'll start with how to enable and run TKPROF.



NOTE

*You may occasionally need to go into a trace file and get a piece of detail that the TKPROF tool doesn't expose, but for the most part, you can ignore much of its contents. If you are interested in the guts of a trace file (what exactly is in there), see Chapter 10 of my book, *Expert One on One Oracle*. Also refer to the chapter on this utility in the *Oracle9i Performance Tuning Guide and Reference*.*

Enable TKPROF

Before getting ready to run TKPROF, you need to understand how to enable it (methods for turning on SQL_TRACE). I keep a small script called trace.sql for this purpose. It simply consists of these two commands:



```
alter session set timed_statistics=true;
alter session set events '10046 trace name context forever, level 12';
```

In the event the system setting of timed statistics is not enabled, you will want to enable ALTER SESSION SET TIMED_STATISTICS=TRUE at the session level at the very least. Without timed statistics, a TKPROF report is virtually useless. You will not see CPU times, and you will not see elapsed times. In short, you won't be able to see where the bottlenecks are! Don't bother tracing unless you set this as well.

After setting timed statistics, you can set tracing on by using ALTER SESSION SET SQL_TRACE=TRUE, or its close cousin ALTER SESSION SET EVENTS '10046 TRACE NAME CONTEXT FOREVER, LEVEL <N>', where <N> can be:

- 1 to enable the standard SQL_TRACE facility (same as SQL_TRACE=TRUE)
- 4 to enable SQL_TRACE and also capture bind variable values in the trace file
- 8 to enable SQL_TRACE and also capture wait events into the trace file
- 12 to enable standard SQL_TRACE and also capture bind variables and wait events

As you can see in my script, I use ALTER SESSION SET EVENTS '10046 TRACE NAME CONTEXT FOREVER, LEVEL 12'. This is especially valuable in Oracle9i, where the TKPROF utility includes wait events in the report (earlier releases did not). TKPROF won't show you bind variable values; for that, you need the trace file itself. Note that the inclusion of wait events in the trace file may significantly increase its over size! You may have to check your max_dump_file_size init.ora parameter to ensure it is large enough to accommodate this additional information.

Run TKPROF

To begin, we'll use a quick test to demonstrate what TKPROF provides and how to access the trace files on the server. We'll run the script to enable tracing, and then run some queries.

Generate the Trace File

In this example, we'll use BIG_TABLE, a table created from ALL_OBJECTS and duplicated over and over to have quite a few rows.

```
ops$tkyte@ORA920> alter session set timed_statistics=true;
Session altered.

ops$tkyte@ORA920> alter session set events
  2 '10046 trace name context forever, level 12';
Session altered.

ops$tkyte@ORA920> select count(*) from big_table
  2 /

COUNT(*)
-----
176210

ops$tkyte@ORA920> select *
  2   from big_table
  3   where owner = 'SYS'
  4     and object_type = 'PACKAGE'
  5     and object_name like 'F%'
  6 /
...
6 rows selected.

ops$tkyte@ORA920>
```

Now that we've generated some activity and created a trace file, we are ready to run some queries to help us with running TKPROF.

Get the Trace Filename

To run TKPROF, we need to know the name of the trace file. One of the following two queries will be useful in doing so. The first is for Windows, using the standard 8.3 filename convention found in Oracle8i and before. The second is for Unix and Windows in Oracle9i and up (they

fixed the filenames on that platform). It may need slight modifications for different Unix variants (this version works on Solaris and Linux).

```
select c.value || '\ORA' || to_char(a.spid,'fm00000') || '.trc'
  from v$process a, v$session b, v$parameter c
 where a.addr = b.paddr
       and b.audsid = sys_context('userenv','sessionid')
       and c.name = 'user_dump_dest'
/

select rtrim(c.value,'/') || '/' || d.instance_name ||
       '_ora_' || ltrim(to_char(a.spid)) || '.trc'
  from v$process a, v$session b, v$parameter c, v$instance d
 where a.addr = b.paddr
       and b.audsid = sys_context( 'userenv', 'sessionid')
       and c.name = 'user_dump_dest'
/
```

On my Linux server, for example, running that query before I exited SQL*Plus told me:

```
ops$tkyte@ORA920> select rtrim(c.value,'/') || '/' || d.instance_name ||
2          '_ora_' || ltrim(to_char(a.spid)) || '.trc'
3      from v$process a, v$session b, v$parameter c, v$instance d
4  where a.addr = b.paddr
5         and b.audsid = sys_context( 'userenv', 'sessionid')
6         and c.name = 'user_dump_dest'
7  /

RTRIM(C.VALUE,'/') || '/' || D.INSTANCE_NAME || '_ORA_' || LTRIM(TO_CHAR
-----/usr/oracle/ora920/ad
min/ora920/udump/ora920_ora_14246.trc

ops$tkyte@ORA920> exit
Disconnected from Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
```

After running these queries, exit SQL*Plus (or whatever tool you are using). You need to do this to close the trace file completely and have all of the information available in the trace file.

Create the TKPROF Report

Now that we have the name of the trace file, we are ready to run TKPROF. Issue the following command:

```
$ tkprof /usr/oracle/...../ora920/udump/ora920_ora_14246.trc tk.prof
```

This creates a text file, tk.prof, in our current working directory.

Read a TKPROF Report

When we open the tk.prf text file, we find output similar to this:

```
select count(*) from big_table

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      2    0.22      0.52       2433       2442         0         1
-----
total      4    0.22      0.52       2433       2442         0         1

Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 147

Rows      Row Source Operation
-----
      1  SORT AGGREGATE
176210  TABLE ACCESS FULL BIG_TABLE
```

And in Oracle9i, we find this nice section included:

```
Elapsed times include waiting on following events:
Event waited on          Times    Max. Wait    Total Waited
-----
SQL*Net message to client        2          0.00          0.00
db file sequential read           1          0.00          0.00
db file scattered read        163          0.07          0.40
SQL*Net message from client      2          0.00          0.00
*****
```

We'll take a look at this piece by piece, starting with the top.

The Query and Execution Statistics

The report begins with the original text of the query that was processed by the database:

```
select count(*) from big_table
```

Next is a tabular report that shows vital execution statistics for each phase of the query itself. We see the three main phases of the query:

- **Parse** This phase is where Oracle finds the query in the shared pool (soft parse) or creates a new plan for the query (hard parse).
- **Execute** This phase is the work done by Oracle on the OPEN or EXECUTE statement of the query. For a SELECT statement, this will be empty in many cases. For an UPDATE statement, this will be where all of the work is done.

- **Fetch** For a SELECT statement, this phase will be where most of the work is done and visible. For an UPDATE statement, it will show no work (you don't fetch from an update operation).

Every statement processed will have these three phases.

Across the top of the report, we find the headings:

- **Count** How many times this phase of the query was performed. In a properly written application, all of your SQL will have a Parse count of 1 and an Execute count of 1 or more. You do not want to parse more than once, if at all possible.
- **CPU** The amount of CPU time spent on this phase of the statement in thousands of seconds.
- **Elapsed** The amount of wall clock time spent on this phase. When the elapsed time is much larger than the CPU time, that means we spent some amount of time waiting for something. In Oracle9i with TKPROF, is it easy to see what that wait was for. At the bottom of the report, we see the wait was for "db file scattered read," which means we waited for physical I/O to complete.
- **Disk** How many physical I/Os were performed during this phase of the query. In this example, when we fetched rows from the table, we performed 2,433 physical disk reads.
- **Query** How many logical I/O were performed to retrieve consistent mode blocks. Those are blocks that may have been reconstructed from the rollback segments, so we would see them as they existed when our query began. Generally, all physical I/Os result in a logical I/O. In most cases, you will find that your logical I/Os outnumber the physical I/Os. However, as we saw earlier in the AUTOTRACE section, direct reads and writes for temporary space violate that rule, and you may have physical I/Os that do not translate into logical I/Os.
- **Current** How many logical I/Os were performed to retrieve blocks as of right now. You will most frequently see these during modification DML operations, such as updates and deletes. There, a block must be retrieved in current mode in order to process the modification, as opposed to when you query a table and Oracle retrieves the block as of the time the query began.
- **Rows** The number of rows processed or affected by that phase. During a modification, you will see the Rows value in the Execute phase. During a SELECT query, this value appears in the Fetch phase.

A question that frequently comes up with regards to TKPROF and the report is, how could output such as the following be produced:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	14755	12.77	12.60	4	29511	856828	14755
Fetch	0	0.00	0.00	0	0	0	0
total	14756	12.77	12.60	4	29511	856828	14755

How can CPU time be larger then elapsed time? This discrepancy is due to the way timings are collected and attempts to time very fast operations or lots of operations. For example, suppose you were using a stopwatch that measured only down to the second. You timed 50 events. Each event seemed to have taken two seconds according to the stopwatch. That means you had 100 seconds of time elapsed during these events right? Well, probably not. Suppose each event really took 2.99 seconds then you really had almost 150 seconds of time there.

Taking it a step further, suppose the stopwatch was continuously running. So, when the event started, you would look at the stopwatch and then when the event finished, you would look again and subtract the two numbers. This is closer to what happens with timing on a computer, you look at the system “watch”, do something, and look again. The delta represents the timing of the event. Now, we’ll perform the same timing test as mentioned previously. Again, each event appears to have taken 2 seconds, but they may have taken just 1.01! How so? Well, when the event started, the clock was really at 2.99, but you only saw “2” (the granularity of the stopwatch). When the event completed, the stopwatch reported 4 (and the real time was 4.00). The delta as you see it is 2, but the “real” delta is actually 1.01.

Over time these discrepancies can accumulate in the aggregate total. The rule of averages would have them effectively cancel each other out more or less but over time, a small error can creep in. That is the cause of the discrepancy—where the elapsed time is less then the CPU time. At the lowest level, Oracle is gathering statistics for timing at either the millisecond or microsecond level. And, further, it may time some events using one clock and other events using a different one—this is unavoidable as the timing information is gathered from the operating system, using its API’s. In this example, we executed a statement 14,755 times, meaning the average CPU time to execute that statement was 0.00865469 seconds. If we were to run this test over and over, we would find that the timings for CPU and elapsed are more or less the same. In general, this error is not so large as to send you looking down the wrong path, but it can be confusing the first time you see it.

Query Environment

The next section of the TKPROF report tells us something about the environment the query ran in. In this case, we see:

```
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 147
```

The misses in library cache of zero tells us this query was soft parsed—Oracle found the query in the shared pool already (I ran this query at least once before). The optimizer goal simply displays the setting of the optimizer goal when this query was parsed. In this case, it is CHOOSE, meaning either the RBO or CBO could be used. The parsing user ID shows the schema that was in effect when this query was parsed. If I query:

```
ops$tkyte@ORA920> select * from all_users where user_id = 147;
```

USERNAME	USER_ID	CREATED
OPS\$TKYTE	147	24-DEC-02

I can see that it was my account.

The Query Plan

The next section of the TKPROF report is the query plan that was actually used when this query ran. This query plan is not generated by EXPLAIN PLAN or AUTOTRACE; rather, this query plan was written into the trace file at runtime. It is the “true” plan.

To show you why this information is relevant, I’ll execute these SQL commands first:

```
ops$tkyte@ORA920> truncate table big_table;
Table truncated.

ops$tkyte@ORA920> alter table big_table add constraint big_table_pk
  2 primary key(object_id);
Table altered.

ops$tkyte@ORA920> analyze table big_table compute statistics;
Table analyzed.
```

Then I’ll rerun TKPROF with another argument: the EXPLAIN argument.

```
$ tkprof ora920_ora_14246.trc /tmp/tk.prf explain=/>
...
Rows      Row Source Operation
-----
1          SORT AGGREGATE
176210     TABLE ACCESS FULL BIG_TABLE

Rows      Execution Plan
-----
0  SELECT STATEMENT  GOAL: CHOOSE
1  SORT (AGGREGATE)
176210  INDEX  GOAL: ANALYZED (FULL SCAN) OF 'BIG_TABLE_IDX'
        (NON-UNIQUE)
```

That’s strange—there are two query plans now. The first one is the true one, the one that was actually used. The second one is the plan that would be used if we ran the query right now. It’s different because we added that primary key and analyzed the table. This is one reason why you might not want to use the EXPLAIN= option of TKPROF: It may not show you the actual plan that was used at runtime. Other settings that affect the optimizer, such as `sort_area_size` and `db_file_multiblock_read_count`, would have the same effect. If the session that ran the query had different settings from the default, EXPLAIN might show you the wrong information, as it did in this example.

The query plan TKPROF provides shows not only the query plan, but as an added bonus: the number of rows that flowed out of each step. Here, we can tell that 176,210 rows flowed out of the TABLE ACCESS FULL BIG_TABLE step into the SORT AGGREGATE, and that 1 row flowed out of that step. When tuning a query, this information can be extremely useful. It can help you identify the problematic parts of a SQL statement that you might be able to tune.

54 Effective Oracle by Design

As an added bonus to this section, if we are lucky enough to have Oracle9i version 9.2.0.2 and above, we might see output like this:

Rows	Row Source Operation
-----	-----
1	SORT AGGREGATE (cr=2300 r=2209 w=0 time=1062862 us)
176210	TABLE ACCESS FULL OBJ#(30635) (cr=2300 r=2209 w=0 time=361127 us)

Now, we see even more detail. Not only do we get the plan and the number of rows flowing through each step of the plan, but we also see logical I/O, physical I/O, and timing information. (This 9.2.0.2 example was performed on a different server, so these statistics are slightly different than the ones shown earlier.) These are represented as follows:

- cr is consistent mode reads, showing consistent gets (logical I/O)
- r is physical reads
- w is physical writes
- time is elapsed time in millionths of a second; us stands for microsecond

Wait Events

The last portion of this TKPROF report shows the wait events. This information is available in Oracle9i Release 1 and up when you use the SET EVENT syntax to enable tracing. In this case, the report for our query was:

Event waited on	Times Waited	Max. Wait	Total Waited
-----	-----	-----	-----
SQL*Net message to client	2	0.00	0.00
db file sequential read	1	0.00	0.00
db file scattered read	163	0.07	0.40
SQL*Net message from client	2	0.00	0.00

This report clearly identified the “big” wait event for us: 4/10 second was spent waiting for I/O (we waited 163 times for that).

You can use that information in order to help you tune your queries. For example, if this query were performing poorly, we would know to concentrate on reducing or speeding up the I/O here. One approach in this case would be to add that primary key, as I did, and analyze the table. The query plan (as shown above) would become a faster index full scan (faster because the index is much smaller than the table, so there is less I/O).

TKPROF for the Masses

Now that you’ve seen TKPROF and what it can do, how can you make it available to everyone? You know that in order for TKPROF to work, you need access to the trace files. This is the sticky point in many cases. The trace files can contain sensitive information, so you might not want developer 1 looking at developer 2’s trace file. In a production system, you don’t want anyone looking at the trace files except for the DBA staff.

I would argue that in a production system, you would be tracing just to get a dump (in order to diagnose an issue) and that would be done with the help of a DBA. However, in a development

and test environment, you need a more scalable solution. In other words, the developers should be able to access these trace files without needing to ask a DBA to get the files for them. They need the trace file now, they need it fast, and they need it often. In order to get access to the trace file, they typically also need physical access to the server (to access the `user_dump_dest` directory).

Typical Access Methods

There are a few traditional approaches to providing access to trace files:

- You could use the undocumented `init.ora` parameter `_TRACE_FILES_PUBLIC=TRUE`.

Do not use this on a live production system, because it will make all trace files publicly accessible, which could be a security issue. On a test or development machine, this is generally acceptable. You have a limited universe of developers who access this machine and, in general, it is okay for them to see their trace files. This makes the trace files readable by the world (usually, only the Oracle account and the DBA group can read these trace files!). It does not solve the problem of physical access, however. The developers still need some sort of access to the file system, which can be a show-stopper. Solutions include exporting the `user_dump_dest` directory as a read-only file system and allowing developers to mount it, or allowing telnet access to the server itself.

- Set up a cron job to run every N minutes to move trace files to a public directory.

I see this one frequently, but have never figured out why it is popular. As compared to the first method, it is harder in that you need to script it, you still need access to the server's file system, and it adds no security or anything useful. It impairs the access to the files, since the developers must wait N minutes for their trace files. In short, while it works, it certainly isn't a method I would suggest using.

- Supply a `setuid` type program (Unix) that lets the developers copy trace files.

Again, for the same reasons as the previous method, this is an overly complicated solution to a simple problem. I do not recommend it.

An Alternative Solution to Provide Worry-free Access

I am going to offer another alternative to providing trace file access that will work nicely and that I've used with great success. You do not need to set any undocumented `init.ora` parameters. You can make it so the developers can access *only* their trace files. You can remove the need to provide access to the physical server file system. It accomplishes the goal of providing immediate access to the trace files, while at the same time removing many of the concerns of doing so. You can do all of this with a fairly simple PL/SQL set of routines.

What we will do involves many steps, it's like we are creating an "application" and in fact, we are. The first step will be to create a schema with the minimal set of privileges necessary. This schema will be used to allow users to view trace files as if they were database tables (they can select `*` from "tracefile" in effect). In order to do this, we'll write a little PL/SQL to read a trace file and make that PL/SQL callable from SQL. We'll also use a LOGOFF trigger in order to capture the trace files that are generated into a database table, along with the "owner" of that trace file so that developers will only see their trace files and not just any trace file. And lastly, we'll develop a SQLPlus script that makes invoking TKPROF against these "database tables that are trace files" as easy as possible.

Create a Schema First, we'll create a schema that will be used to provide access to the trace files owned by that user, *as if each trace file were a database table*. The end user will be able to select * from their_trace_file. Using SPOOL in SQL*Plus, they can save this trace file locally and not need to have access to the server at all. The user we need will be created with at least these privileges:

```
create user trace_files identified by trace_files
default tablespace users quota unlimited on users;

grant create any directory, /* to read user dump dest */
create session, /* to log on in the first place */
create table, /* used to hold users -> trace files */
create view, /* used so users can see what traces they have */
create procedure, /* create the func that gets the trace data */
create trigger, /* to capture trace file names upon logoff */
create type, /* to create a type to be returned by function */
administer database trigger /* to create the logoff trigger */
to trace_files;

/* these are needed to find the trace file name */
grant select on v_$process to trace_files;
grant select on v_$session to trace_files;
grant select on v_$instance to trace_files;
```

Create a View and Table Next, we'll create a view that returns the name of the trace file for the current session. As discussed in "Get the Trace Filename" earlier in this chapter, this view may need to be customized for your operating system:

```
create view session_trace_file_name
as
select d.instance_name || '_ora_' || ltrim(to_char(a.spid)) ||
       '.trc' filename
   from v_$process a, v_$session b, v_$instance d
  where a.addr = b.paddr
        and b.audsid = sys_context( 'userenv', 'sessionid')
/
```

NOTE

Oracle8i Release 2 (version 8.1.6) added a new init.ora parameter TRACEFILE_IDENTIFIER that is settable at the session level that may be used to more easily identify tracefiles. When set, the tracefile_identifier string will be appended to the tracefile name. For that reason, you might also consider joining to v_\$parameter in the above view to pick up that part of the name if you choose to use it.

And then create a table to hold the mapping of usernames to filenames. We'll also keep a TIMESTAMP (use DATE in Oracle8i and before) to see when the trace file session ended. The view is what end users will use to see which trace files they have available:


```

create table avail_trace_files
( username   varchar2(30) default user,
  filename   varchar2(512),
  dt         timestamp default systimestamp,
  constraint avail_trace_files_pk primary key(username,filename)
)
organization index
/
create view user_avail_trace_files
as
select * from avail_trace_files where username = user
/
grant select on user_avail_trace_files to public
/

```

Create a Trigger Now, we'll use a LOGOFF trigger to capture the name of the trace file and the current username, if a trace file actually exists. We'll use a BFILE to achieve this.

NOTE

You will need to use the correct directory name for your udump_dir directory. The one in the example is mine. Yours may be different!

```

create or replace directory UDUMP_DIR
as '/usr/oracle/ora920/OraHome1/admin/ora920/udump'
/

create or replace trigger capture_trace_files
before logoff on database
begin
  for x in ( select * from session_trace_file_name )
  loop
    if ( dbms_lob.fileexists(
      bfilename('UDUMP_DIR', x.filename ) ) = 1 )
    then
      insert into avail_trace_files (filename)
      values (x.filename);
    end if;
  end loop;
end;
/

```

Add the Function Next, we need the function that will read and stream the trace data back to the end user. For this, we will use a pipelined PL/SQL function. In order to do that, we'll need a simple collection type:

```

create or replace type vcArray as table of varchar2(4000)
/

```

And then we add the function itself. It begins by issuing a SELECT against USER_AVAIL_TRACE_FILES to make sure that the requested file is available for the currently logged-in user. That SELECT INTO will raise NO_DATA_FOUND, which when propagated back to the calling SELECT will just appear as “No data found.” If that query succeeds, we’ll go onto read the trace file a line at a time and return it to the caller.

```

create or replace
function trace_file_contents( p_filename in varchar2 )
return vcArray
pipelined
as
    l_bfile          bfile := bfilename('UDUMP_DIR',p_filename);
    l_last           number := 1;
    l_current         number;
begin
    select rownum into l_current
    from user_avail_trace_files
    where filename = p_filename;
    dbms_lob.fileopen( l_bfile );
    loop
        l_current := dbms_lob.instr( l_bfile, '0A', l_last, 1 );
        exit when (nvl(l_current,0) = 0);
        pipe row(
            utl_raw.cast_to_varchar2(
                dbms_lob.substr( l_bfile, l_current-l_last+1,
                                l_last ) )
        );
        l_last := l_current+1;
    end loop;
    dbms_lob.fileclose(l_bfile);
    return;
end;
/
grant execute on vcArray to public
/
grant execute on trace_file_contents to public
/

```

NOTE

The pipelined keyword is new with Oracle9i Release 1 and up. In prior releases, you would not use pipelined and pipe row(). Instead, you would declare a local variable of type vcArray, populate it element by element, and return that at the end of the routine. See http://asktom.oracle.com/~tkyte/tkprof_forall.html for an Oracle8/8i-specific solution. Beware, large tracefiles will consume large amounts of UGA/PGA memory using this technique. The url (http://asktom.oracle.com/~tkyte/tkprof_forall.html) discusses another alternative implementation using global temporary tables instead (the procedure fills a global temporary table instead of an in memory table).

That is it. Let's test this solution.

Test the Access At this point, we'll create a minimally privileged developer account and see if this works.

```

trace_files@ORA920> @connect "/" as sysdba"

sys@ORA920> create user developer identified by developer;
User created.

sys@ORA920> grant create session,
      2      alter session
      3  to developer;
Grant succeeded.

sys@ORA920> @connect developer/developer

developer@ORA920> select * from trace_files.user_avail_trace_files;
no rows selected

developer@ORA920> @connect developer/developer

developer@ORA920> select * from trace_files.user_avail_trace_files;
no rows selected

```

That logoff (by reconnecting, we logged off and logged back in) put nothing in there. We did not generate a trace file in that session. Now, let's try this:

```

developer@ORA920> alter session set sql_trace=true;
Session altered.

developer@ORA920> @connect developer/developer

developer@ORA920> column filename new_val f
developer@ORA920> select * from trace_files.user_avail_trace_files;

```

USERNAME	FILENAME	DT
DEVELOPER	ora920_ora_14973.trc	29-DEC-02 04.31.05.241607 PM

By simply generating a trace file, we have a row in there now. Let's continue:

```

developer@ORA920> select *
      2  from TABLE( trace_files.trace_file_contents( '&f' ) );

COLUMN_VALUE
-----
/usr/oracle/OraHome1/admin/ora920/udump/ora920_ora_14973.trc
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options

```

60 Effective Oracle by Design

```
JServer Release 9.2.0.1.0 - Production
ORACLE_HOME = /usr/oracle/ora920/OraHome1
System name:      Linux
Node name:        tkyte-pc-isdn.us.oracle.com
Release:          2.4.18-14
Version:          #1 Wed Sep 4 13:35:50 EDT 2002
Machine:          i686
Instance name:    ora920
Redo thread mounted by this instance: 1
Oracle process number: 14
Unix process pid: 14973, image: oracle@tkyte-pc-isdn.us.oracle.com

*** SESSION ID: (9.389) 2002-12-29 16:31:05.154
APPNAME mod='SQL*Plus' mh=3669949024 act='' ah=4029777240
=====
PARSING IN CURSOR #1 len=32 dep=0 uid=237 oct=42 lid=237 tim=1016794399565448
hv=1197935484 ad='5399abdc'
alter session set sql_trace=true
END OF STMT
EXEC #1:c=0,e=127,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1016794399564994
=====
... <snipped out for brevity> ...
XCTEND rlbk=0, rd_only=1
STAT #12 id=1 cnt=0 pid=0 pos=1 obj=0 op='UPDATE '
STAT #12 id=2 cnt=0 pid=1 pos=1 obj=5992 op='TABLE ACCESS FULL
WM$WORKSPACES_TABLE '
STAT #11 id=1 cnt=4 pid=0 pos=1 obj=222 op='TABLE ACCESS FULL DUAL '

128 rows selected.
```

Now, just to show that the security works, let's log in as another user and try to query the same trace file:

```
developer@ORA920> @connect /

ops$tkyte@ORA920> select *
  2 from TABLE( trace_files.trace_file_contents
  3              ( ' ora920_ora_14973.trc ' ) );

no rows selected

ops$tkyte@ORA920>
```

Automating the Access So, now that we have this capability to access trace files, how can we use it? We could use a simple script, like this (called tklast.sql perhaps):

```
column filename new_val f
select filename
  from trace_files.user_avail_trace_files
 where dt = ( select max(dt)
```

```

        from trace_files.user_avail_trace_files
    )
/
set termout off
set heading off
set feedback off
set embedded on
set linesize 4000
set trimspool on
set verify off
spool &f
select * from TABLE( trace_files.trace_file_contents( '&f' ) );
spool off
set verify on
set feedback on
set heading on
set termout on
host tkprof &f tk.prf
edit tk.prf

```

This is all you would need to automate this process. It would find the last trace file for your username, retrieve it, and run TKPROF against it to format it.

We've looked at some tools Oracle provides out of the box, now we'll take a look at a benchmarking tool I've developed in order to test alternative approaches to a given problem—to see which approach is better. This tool is Runstats.

Runstats

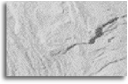
Runstats is a tool I developed to compare methods and show which is superior. We used Runstats in several examples in Chapter 1, where I introduced it as a small-time benchmarking tool. You can download a copy today from otn.oracle.com.

Runstats simply measures three key things:

- **Wall clock or elapsed time** This is useful to know, but not the most important piece of information.
- **System statistics** This shows, side by side, how many times each approach did something (such as parse calls, for example) and the difference between the two.
- **Latching** This is the key output of this report.

As I've noted previously, latches are a type of lightweight lock. Locks are serialization devices. Serialization devices inhibit concurrency. Things that inhibit concurrency are less scalable, can support fewer users, require more resources. Our goal is always to build applications that have the potential to scale—ones that can service 1 user as well as 1,000 or 10,000. The less latching we incur in our approaches, the better off we are. I might choose an approach that takes longer to run on the wall clock but that uses 10% of the latches. I know that the approach that uses fewer latches will scale infinitely better than the approach that uses more latches.

Runstats is best used in isolation; that is, on a single-user database. We will be measuring statistics and latching (locking) activity that result from our approaches. We do not wish for other sessions to contribute to the systems load or latching while this is going on. A small test database is perfect for these sorts of tests. I frequently use my desktop PC or laptop for example.



NOTE

I believe all developers should have a test bed database they control to try ideas on, without needing to ask a DBA to do something all of the time. Developers definitely should have a database on their desktop, given that the licensing for the personal developer version is simply “use it to develop and test with, do not deploy, and you can just have it.” This way, there is nothing to lose! Also, I’ve taken some informal polls at conferences and seminars. Virtually every DBA out there started as a developer! The experience and training developers could get by having their own database—being able to see how it really works—pays for itself in the long run by the increased experience.

Set Up Runstats

In order to use Runstats, you need to set up access to several V\$ tables, create a table to hold the statistics, and create a Runstats package.

Create a View for V\$ Table Access

You will need access to three V\$ tables (those magic, dynamic performance tables): V\$STATNAME, V\$MYSTAT, and V\$LATCH. Here is a view I use:

```
create or replace view stats
as select 'STAT...' || a.name name, b.value
   from v$statname a, v$mystat b
   where a.statistic# = b.statistic#
union all
select 'LATCH.' || name, gets
   from v$latch;
```

You can either have SELECT on V\$STATNAME, V\$MYSTAT, and V\$LATCH granted directly to you (that way you can create the view yourself) or you can have someone that does have SELECT on those objects create the view for you and grant SELECT privileges to you.

Create a Temporary Table

Once you have that set up, all you need is a small table to collect the statistics:

```
create global temporary table run_stats
( runid varchar2(15),
  name varchar2(80),
  value int )
on commit preserve rows;
```

Create a Runstats Package

Last, you need to create the package that is Runstats. It contains three simple API calls:

- RS_START (Runstats Start) to be called at the beginning of a Runstats test
- RS_MIDDLE to be called in the middle, as you might have guessed
- RS_STOP to finish off and print the report

The specification is as follows:

```
ops$tkyte@ORA920> create or replace package runstats_pkg
2  as
3      procedure rs_start;
4      procedure rs_middle;
5      procedure rs_stop( p_difference_threshold in number default 0 );
6  end;
7  /
```

Package created.

The parameter `p_difference_threshold` is used to control the amount of data printed at the end. Runstats collects statistics and latching information for each run, and then prints a report of how much of a resource each test (each approach) used and the difference between them. You can use this input parameter to see only the statistics and latches that had a difference greater than this number. By default this is zero, and you see all of the outputs.

Next, we'll look at the package body procedure by procedure. The package begins with some global variables. These will we used to record the elapsed times for our runs:

```
ops$tkyte@ORA920> create or replace package body runstats_pkg
2  as
3
4  g_start number;
5  g_run1   number;
6  g_run2   number;
7
```

Next is the RS_START routine. This will simply clear out our statistics holding table and then populate it with the "before" statistics and latches. It will then capture the current timer value, a clock of sorts that we can use to compute elapsed times in hundredths of seconds:

```
8  procedure rs_start
9  is
10 begin
11     delete from run_stats;
12
13     insert into run_stats
14     select 'before', stats.* from stats;
15
```

64 Effective Oracle by Design

```
16      g_start := dbms_utility.get_time;
17  end;
18
```

Next is the RS_MIDDLE routine. This procedure simply records the elapsed time for the first run of our test in G_RUN1. Then it inserts the current set of statistics and latches. If we were to subtract these values from the ones we saved previously in RS_START, we could discover how many latches the first method used, how many cursors (a statistic) it used, and so on.

Last, it records the start time for our next run:

```
19 procedure rs_middle
20 is
21 begin
22     g_run1 := (dbms_utility.get_time-g_start);
23
24     insert into run_stats
25     select 'after 1', stats.* from stats;
26     g_start := dbms_utility.get_time;
27
28 end;
29
30 procedure rs_stop(p_difference_threshold in number default 0)
31 is
32 begin
33     g_run2 := (dbms_utility.get_time-g_start);
34
35     dbms_output.put_line
36     ( 'Run1 ran in ' || g_run1 || ' hsecs' );
37     dbms_output.put_line
38     ( 'Run2 ran in ' || g_run2 || ' hsecs' );
39     dbms_output.put_line
40     ( 'run 1 ran in ' || round(g_run1/g_run2*100,2) ||
41     '% of the time' );
42     dbms_output.put_line( chr(9) );
43
44     insert into run_stats
45     select 'after 2', stats.* from stats;
46
47     dbms_output.put_line
48     ( rpad( 'Name', 30 ) || lpad( 'Run1', 10 ) ||
49     lpad( 'Run2', 10 ) || lpad( 'Diff', 10 ) );
50
51     for x in
52     ( select rpad( a.name, 30 ) ||
53       to_char( b.value-a.value, '9,999,999' ) ||
54       to_char( c.value-b.value, '9,999,999' ) ||
55       to_char( ( (c.value-b.value)-(b.value-a.value)), '9,999,999' ) data
```



```

56         from run_stats a, run_stats b, run_stats c
57     where a.name = b.name
58         and b.name = c.name
59         and a.runid = 'before'
60         and b.runid = 'after 1'
61         and c.runid = 'after 2'
62         and (c.value-a.value) > 0
63         and abs( (c.value-b.value) - (b.value-a.value) )
64             > p_difference_threshold
65     order by abs( (c.value-b.value)-(b.value-a.value))
66 ) loop
67     dbms_output.put_line( x.data );
68 end loop;
69
70     dbms_output.put_line( chr(9) );
71 dbms_output.put_line
72 ( 'Run1 latches total versus runs -- difference and pct' );
73 dbms_output.put_line
74 ( lpad( 'Run1', 10 ) || lpad( 'Run2', 10 ) ||
75   lpad( 'Diff', 10 ) || lpad( 'Pct', 8 ) );
76
77 for x in
78 ( select to_char( run1, '9,999,999' ) ||
79       to_char( run2, '9,999,999' ) ||
80       to_char( diff, '9,999,999' ) ||
81       to_char( round( run1/run2*100,2 ), '999.99' ) || '%' data
82   from ( select sum(b.value-a.value) run1, sum(c.value-b.value) run2,
83              sum( (c.value-b.value)-(b.value-a.value)) diff
84         from run_stats a, run_stats b, run_stats c
85        where a.name = b.name
86              and b.name = c.name
87              and a.runid = 'before'
88              and b.runid = 'after 1'
89              and c.runid = 'after 2'
90              and a.name like 'LATCH%'
91        )
92 ) loop
93     dbms_output.put_line( x.data );
94 end loop;
95 end;
96
97 end;
98 /

```


Package body created.

And now you are ready to use Runstats.

Use Runstats


In order to see Runstats in action, we'll demonstrate what happens when applications over parse. I see this frequently with JDBC- and ODBC-style applications, in which the developer will code a routine in the following fashion and then call this routine dozens or hundreds or more times per connection.

```

 Procedure do_insert
    Open cursor
    Parse statement
    Bind variables (hopefully they do this!)
    Execute statement
    Close cursor
  
```

When what they should be doing is more like this:


```

 Procedure do_insert
    If (first time) the
        Open cursor
        Parse statement
        First time = FALSE
    End if
    Bind variables
    Execute statement
  
```

The developers should parse the statement *once* during their connection, and then bind and execute it over and over again. *Parsing* of a statement is really just another name for *compiling*. Just as you would never compile a routine, call it, and discard the compiled code, only to recompile the routine later, you don't want to lose the parsed cursor either!

We will use Runstats twice. The first time, we will show a quick fix to the code that is in production via a server setting—something to help us while the developers revise the code. The second will compare the quick-fix code to the real thing. So, we'll start with these two demonstration routines:

```

 create or replace package demo_pkg
as
    procedure parse_bind_execute_close( p_input in varchar2 );
    procedure bind_execute( p_input in varchar2 );
end;
/
  
```

As its name implies, one will always parse, bind, execute, and close a cursor. The other will be the correctly coded routine that will simply parse once and then bind/execute repeatedly. We will use DBMS_SQL as the implementation. DBMS_SQL is similar in nature to JDBC and ODBC, as far as preparing and executing statements, so the analogy here is a good one.

```

create or replace package body demo_pkg
as

  g_first_time boolean := TRUE;
  g_cursor      number;

  procedure parse_bind_execute_close( p_input in varchar2 )
  as
    l_cursor      number;
    l_output       varchar2(4000);
    l_status       number;
  begin
    l_cursor := dbms_sql.open_cursor;
    dbms_sql.parse( l_cursor,
                    'select * from dual where dummy = :x',
                    dbms_sql.native );
    dbms_sql.bind_variable( l_cursor, ':x', p_input );
    dbms_sql.define_column( l_cursor, 1, l_output, 4000 );
    l_status := dbms_sql.execute( l_cursor );
    if ( dbms_sql.fetch_rows( l_cursor ) <= 0 )
    then
      l_output := null;
    else
      dbms_sql.column_value( l_cursor, 1, l_output );
    end if;
    dbms_sql.close_cursor( l_cursor );
  end parse_bind_execute_close;

  procedure bind_execute( p_input in varchar2 )
  as
    l_output       varchar2(4000);
    l_status       number;
  begin
    if ( g_first_time )
    then
      g_cursor := dbms_sql.open_cursor;
      dbms_sql.parse( g_cursor,
                      'select * from dual where dummy = :x',
                      dbms_sql.native );
      dbms_sql.define_column( g_cursor, 1, l_output, 4000 );
      g_first_time := FALSE;
    end if;

    dbms_sql.bind_variable( g_cursor, ':x', p_input );
    l_status := dbms_sql.execute( g_cursor );
    if ( dbms_sql.fetch_rows( g_cursor ) <= 0 )

```

```

    then
        l_output := null;
    else
        dbms_sql.column_value( g_cursor, 1, l_output );
    end if;
end bind_execute;

end;
/

```

The Quick Fix Test

Now, we are ready to measure our first hypothesis: If we use a quick fix, we may get some more scalability out of our system. The quick fix is `SESSION_CACHED_CURSORS`, a parameter that controls whether Oracle will silently cache cursors in the background, much as it does with static SQL in PL/SQL (a key reason why coding database applications in PL/SQL is an excellent idea: so much of the work is done for you). We'll run a statement 1,000 times without and then with cursor caching enabled.

```

ops$tkyte@ORA920> begin
2     runstats_pkg.rs_start;
3     execute immediate
4     'alter session set session_cached_cursors=0';
5     for i in 1 .. 1000
6     loop
7         demo_pkg.parse_bind_execute_close( 'Y' );
8     end loop;
9     runstats_pkg.rs_middle;
10    execute immediate
11    'alter session set session_cached_cursors=100';
12    for i in 1 .. 1000
13    loop
14        demo_pkg.parse_bind_execute_close( 'Y' );
15    end loop;
16    runstats_pkg.rs_stop(500);
17 end;
18 /
Run1 ran in 160 hsecs
Run2 ran in 146 hsecs
run 1 ran in 109.59% of the time

```

Interestingly, the code that is executed with cursor caching is only marginally faster, although many people expect it to be blindingly fast. The really important facts to see from this are in the second part of the Runstats report. This contains the statistics and latching information:

STAT...session cursor cache hi	0	999	999
LATCH.library cache pin	18,022	16,016	-2,006
LATCH.shared pool	11,031	9,003	-2,028
LATCH.library cache pin alloca	10,038	6,008	-4,030
LATCH.library cache	25,062	20,022	-5,040

```

STAT...session uga memory          65,464          0    -65,464
STAT...session pga memory          65,536          0    -65,536
STAT...session pga memory max         0        65,536    65,536

```

Run1 latches total versus runs -- difference and pct

```

Run1      Run2      Diff      Pct
73,393    60,428    -12,965  121.46%

```

PL/SQL procedure successfully completed.

Here, we removed 20% of the latching on the library cache and shared pool by simply flipping a switch.

“Tom, these are the top-five wait events from a Statspack report. What would you recommend?”

Event	Waits	Wait (cs)	%Total
library cache load lock	3,561	182,952	11.58
db file sequential read	168,481	154,051	9.75
log file sync	31,884	102,048	6.46
log file parallel write	54,402	59,165	3.75

Basically, I looked at this and discovered tons of parsing and reparsing going on. I suggested flipping this `SESSION_CACHED_CURSORS` switch.

“We set the `SESSION_CACHED_CURSOR` to 100 and repeated the same test (after shutdown/restart). Latch contention on the library cache is reduced (approximately 50%).”

The Best Practice Test

Now, to complete our example, we need to compare the `PARSE_BIND_EXECUTE_CLOSE` routine to the `BIND_EXECUTE` routine. How does the code that was fixed with a temporary bandage fare when compared to the code that is implemented using best practices? With Runstats, we’ll compare the two routines:


```

ops$tkyte@ORA920> begin
2     execute immediate
3     'alter session set session_cached_cursors=100';
4     runstats_pkg.rs_start;
5     for i in 1 .. 1000
6     loop
7         demo_pkg.parse_bind_execute_close( 'Y' );
8     end loop;
9     runstats_pkg.rs_middle;
10    for i in 1 .. 1000
11    loop
12        demo_pkg.bind_execute( 'Y' );

```

```
13      end loop;
14      runstats_pkg.rs_stop(500);
15  end;
16  /
Run1 ran in 114 hsecs
Run2 ran in 75 hsecs
run 1 ran in 152% of the time
```

There, we can see the runtime difference is much more striking. However, the really good stuff is yet to come:



Name	Run1	Run2	Diff
STAT...session cursor cache hi	999	1	-998
STAT...opened cursors cumulati	1,001	2	-999
STAT...parse count (total)	1,001	2	-999
STAT...recursive calls	6,002	3,004	-2,998
LATCH.library cache pin alloca	6,012	10	-6,002
LATCH.shared pool	9,001	1,010	-7,991
LATCH.library cache pin	16,014	2,024	-13,990
LATCH.library cache	20,028	2,032	-17,996
STAT...session uga memory	65,464	0	-65,464
STAT...session pga memory	65,536	0	-65,536
STAT...session pga memory max	0	65,536	65,536

```
Run1 latches total versus runs -- difference and pct
Run1      Run2      Diff      Pct
60,254    14,472    -45,782  416.35%
```

PL/SQL procedure successfully completed.

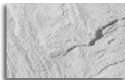
You can see that the BIND_EXECUTE routine use one-fourth the amount of latching than the PARSE_BIND_EXECUTE_CLOSE routine used. That could very well be the difference between discovering your application scales and finding your application doesn't work.

Statspack

Statspack is a tool that was first officially introduced with version 8.1.6 of the database. It is designed as a replacement for UTLBSTAT (begin stats) and UTLESTAT (end stats—generate a report). At the end of the day, Statspack is basically just a textual report that describes how your database instance has been performing. It has the ability to keep a long-running archive of these reports (actually the base data that can be used to generate a report, stored in database tables), as well as utilities to selectively purge old data or just remove all of it. Its primary use is for tuning the database after the application has been tuned.

NOTE

Statspack could be made to work with versions 8.1.5 and 8.0.6 of the database with a little effort. See <http://www.oracle.com/oramag/oracle/00-Mar/index.html?o20tun.html> for details on accomplishing that. That article also provides a good overview of this tool.



Set Up Statspack

Statspack is designed to be installed when connected as SYSDBA (connect *username/password* as sysdba). In order to install Statspack, you must be able to perform that operation. In many installations, this will be a task that must be done by a DBA or administrator.

Once you have the ability to connect as SYSDBA (or, in Oracle8i and earlier, connect *internal*), installing Statspack is trivial. You simply run statscre.sql in Oracle8.1.6 or spcreate.sql in Oracle8i Release 3 (8.1.7) and later. These scripts are found in \$ORACLE_HOME/rdbms/admin. Also in that directory is a text file spdoc.txt that you might want to read through before installing Statspack. It contains any last-minute notes and describes new feature enhancements.

When you are connected as SYSDBA via SQL*Plus, a Statspack install looks something like this:

```
[tkyte@tkyte-pc-isdn admin]$ sqlplus "/ as sysdba"
SQL*Plus: Release 9.2.0.1.0 - Production on Sat Jan 11 13:37:23 2003
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Connected to:
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production
```

```
sys@ORA920> @spcreate
... Installing Required Packages
```

NOTE

This install will run dbmspool.sql to create the DBMS_SHARED_POOL package and dbmsjob.sql to create the DBMS_JOB package. On most systems, DBMS_JOB is already installed, and on many systems, DBMS_SHARED_POOL is as well. If you are installing Statspack into a used system, you may not want these packages to be re-created, because people may be using them. You can edit spcusr.sql and comment out the installation of these two packages.

You'll need to know three pieces of information before running the spcreate.sql script:

- The password you want to assign to the PERFSTAT user that will be created
- The default tablespace PERFSTAT will use to hold the statistics tables it created (you cannot use the SYSTEM table). You may want to consider creating a special tablespace just for statspack collection so you can more easily monitor and manage the storage used by this tool.
- The temporary tablespace PERFSTAT should use (again, this cannot be SYSTEM)

The script will prompt you for this information as it executes. In the event you make a mistake or inadvertently cancel the installation, you should use spdrop.sql (in Oracle8.1.7 and up) or statsdrp.sql (in Oracle8.1.6 and earlier) to remove the user and installed views, prior to attempting another installation of Statspack.

The Statspack installation will create some .lis files (you will see their names during the installation). You should review these for any possible errors that might have occurred. They should install cleanly, however, as long as you supplied valid tablespace names (and didn't already have a user PERFSTAT).

Use Statspack

The Statspack report provides the following information:

- Instance information, such as the database ID and name, versions, operating system hostname, and the like
- Snapshot information, such as the beginning and end times and duration of the snapshot—the collection points—as it is now (the Statspack report)
- Database cache size information for the buffer cache, log buffer, and shared pool
- Overall load statistics, by second and by transaction, such as the amount of redo generated, number of transactions, statements executed, and so on
- Efficiency percentages, also known as hit ratios, such as library cache hits, buffer hits, soft parse percentages, and so on
- Shared pool utilization showing memory usage over the observed period of time
- Top five timed events, what you have been waiting for/waiting on (perhaps the most important part of the report!)
- A report of all wait events in the system during the observed period of time
- Various top-SQL reports, such as the SQL with the most buffer gets (those that do the most logical I/O), the SQL with the most physical reads, the most executed SQL, the most frequently parsed SQL, and so on
- A statistics report showing all of the various counters for the observed period of time, such as how many enqueue waits there were (locks that caused a wait), how many physical reads, how many disk sorts, and so on
- I/O reports by tablespace and file
- Buffer pool statistics and buffer pool advisor
- PGA advisor
- Undo information
- Latching information
- Detailed SGA breakdown and utilization report
- A listing of the init.ora parameters in use

Some of these reports are version dependent, for example, the PGA advisor is a new 9i feature that you won't see in 8i, the functionality just did not exist in that release. Additionally,

the reports you see will be a function of the level of snapshot you take as well as the reporting options you use. But, as you can see, this is potentially a ton of information. A small report might be 20 printed pages long. Fortunately, as we'll see in a bit, there are a few vital points to review, and they might send you to look at details in some section, but it shouldn't take you more than a couple of minutes to skim and absorb the contents of a Statspack report.

What People Do Wrong with Statspack

The single most common misuse of Statspack I see is the “more is better” approach. People have sent me Statspack reports that span hours or even days. The times between the snapshots (the collection points) should, in general, be measured in minutes, not hours and never days.

The Statspack reports I like are from 15-minute intervals during a busy or peak time, when the performance is at its worst. That provides a very focused look at what was going wrong at that exact moment in time. The problem with a very large Statspack snapshot window, where the time between the two snapshots is measured in hours, is that the events that caused serious performance issues for 20 minutes during peak processing don't look so bad when they're spread out over an 8-hour window. It would be like saying, “Hey, I hit 32 red lights on my last 8-hour car ride.” That doesn't sound too bad—4 an hour; must be a lot of city driving. But if I told you, “Hey, I hit 6 red lights in 15 minutes during lunch,” then we have identified a serious bottleneck in your travels.

It's also true with Statspack that measuring things over too long of a period tends to level them out over time. Nothing will stand out and strike you as being wrong. So, when taking snapshots, schedule them about 15 to 30 minutes (maximum) apart. You might wait 3 or 4 hours between these two observations, but you should always do them in pairs and within minutes of each other.

Another common mistake with Statspack is to gather snapshots only when there is a problem. That is fine to a point, but how much better would it be to have a Statspack report from when things were going good to compare with when things are bad. A simple Statspack report that shows a tremendous increase in physical I/O activity or table scans (long tables) could help you track down that missing index. Or, if you see your soft parse percentage value went from 99% to 70%, you know that someone introduced a new feature into the system that isn't using bind variables (and is killing you). Having a history of the good times is just as important as having a history of the bad; you need both.

“When looking through the Statspack report, I found that events direct path read and direct path write are always on the top two of the Top 5 Wait Events section, as follows:

Top 5 Wait Events

~~~~~

| Event                   | Times | Wait (cs) | Wait% |
|-------------------------|-------|-----------|-------|
| direct path read        | 576   | 292       | 66.42 |
| direct path write       | 2,314 | 61        | 13.86 |
| log file parallel write | 19    | 15        | 3.43  |
| log file sync           | 16    | 13        | 3.09  |

What could be the problem? How can I fix it?”

Well, the observation period here was about 30 minutes. There were 2.92 seconds of wait on direct path reads. Big deal! In this case, it would not be worth his time and energy to pursue this one further. Realize that there will always be some wait events, somewhere. I remember in my physics class in college, we always did our calculations assuming a “frictionless surface” my freshman year. Trying to achieve zero waits is like trying to find that frictionless surface. It doesn’t exist (as I was to find out in my second-year physics class, when it got a lot harder).

Statspack at a Glance

What if you have this 20-page Statspack report and you want to figure out if everything is okay. Here, I’ll review what I look for in the report, section by section. I’ll use an actual Statspack report from one of my internal systems. We’ll see if anything needs further investigation (turns out it did!).

Statspack Report Header

This is the standard header for a Statspack report:

```
STATSPACK report for

DB Name          DB Id      Instance Inst Num Release      Cluster Host
-----
ORA9I            2272536868 ora9i          1 9.2.0.1.0    NO      aria

              Snap Id      Snap Time      Sessions Curs/Sess Comment
              -----
Begin Snap:      1 30-Dec-02 09:58:58    67,254      3.0
End Snap:        2 30-Dec-02 10:14:52    67,260      3.0
Elapsed:         15.90 (mins)

Cache Sizes (end)
~~~~~
 Buffer Cache: 96M Std Block Size: 8K
 Shared Pool Size: 112M Log Buffer: 512K
```

Note that this section may appear slightly different depending on your version of Oracle. For example, the Curs/Sess column, which shows the number of open cursors per session, is new with Oracle9i (an 8i Statspack report would not show this data).

Here, the item I am most interested in is the elapsed time. I want that to be large enough to be meaningful, but small enough to be relevant. I am partial to numbers between 15 and 30 minutes; with longer times, we begin to lose the needle in the haystack, as noted in the previous section.

Statspack Load Profile

Next, we see the load profile:

```
Load Profile
~~~~~
              Per Second      Per Transaction
              -----
      Redo size:      75,733.92      20,737.70
      Logical reads:  1,535.11      420.35
```

```

          Block changes:          449.56          123.10
          Physical reads:         562.99          154.16
          Physical writes:        62.53           17.12
          User calls:             8.04            2.20
          Parses:                 56.43           15.45
Hard parses:                 0.38              0.10
Sorts:          11.63              3.19
          Logons:                 0.30            0.08
Executes:                   94.21              25.80
Transactions:              3.65


% Blocks changed per Read:  29.29    Recursive Call %:    97.14
Rollback per transaction %:  9.41     Rows per Sort:    752.04

```

Here, I'm interested in a variety of things, but if I'm looking at a "health check," three items are important: the Hard parses (we want very few of them), Executes (how many statements we are executing per second/transaction), and Transactions (how many transactions per second we process). This gives an overall view of the load on the server. In this case, we are looking at a very good hard parse number and a fairly light system load—just three to four transactions per second.

### Statspack Instance Efficiency Percentage

Next, we move onto the Instance Efficiency Percentages section, which includes perhaps the only ratios I look at in any detail:

 Instance Efficiency Percentages (Target 100%)

```

~~~~~
 Buffer Nowait %: 100.00 Redo NoWait %: 100.00
 Buffer Hit %: 94.94 In-memory Sort %: 98.50
Library Hit %: 99.81 Soft Parse %: 99.33
Execute to Parse %: 40.10 Latch Hit %: 99.91
Parse CPU to Parse Elapsed %: 86.12 % Non-Parse CPU: 94.69

Shared Pool Statistics Begin End

 Memory Usage %: 86.95 85.16
 % SQL with executions>1: 66.38 67.40
 % Memory for SQL w/exec>1: 67.28 69.68

```

The three in italics are the ones I zero in on right away: Library Hit, Soft Parse %, and Execute to Parse. All of these have to do with how well the shared pool is being utilized. Time after time, I find this to be the area of greatest payback, where we can achieve some real gains in performance. Here, in this report, I am quite pleased with the Library Hit and the Soft Parse % values. If the Library Hit ratio was low, it could be indicative of a shared pool that is too small, or just as likely, that the system did not make correct use of bind variables in the application. It would be an indicator to look at issues such as those.

The Soft Parse % value is one of the most important (if not the only important) ratio in the database. For your typical system, it should be as near 100% as possible. You quite simply do not

hard parse after the database has been up for a while in your typical transactional/general-purpose database. The way you achieve that is with bind variables!

In a data warehouse, we would like to generally see the Soft Parse ratio lower. We don't necessarily want to use bind variables in a data warehouse. This is because they typically use materialized views, histograms, and other things that are easily thwarted by bind variables. In a regular system like this, we are doing many executions per second, and hard parsing is something to be avoided. In a data warehouse, we may have many seconds between executions, so hard parsing is not evil; in fact, it is good in those environments.

Now, just so you know, the Soft Parse % is computed using this function:

```
round(100*(1-:hprs/:prse),2)
```

In this function, :hprs is the number of hard parses, and :prse is the total parse count. As you tune your system to reduce or eliminate hard parsing so that :hprs goes to zero, this ratio gets closer and closer to 100%. That is your goal in most cases.

The next ratio of interest is the Execute to Parse, which is computed using this function:

```
round(100*(1-:prse/:exe),2)
```

where :prse is the number of parses and :exe is the number of executions.

The number in this report, 40%, looks pretty bad. In this case, it is wholly unavoidable. The system we are looking at does only web-based applications using MOD\_PLSQL exclusively. In such a system (this applies even to J2EE applications with a connection pool as well), it is not unusual to find a lower than optimal execute-to-parse ratio. This is due solely to the fact that each web page is an entire database session. The sessions are very short, so there is not much chance for reuse of cursors here. We don't generally have the same statement executing 1,000 times per page; hence, the execute-to-parse ratio will be low. Now, if this system were designed for serving web pages by day and batch processing by night, we would expect our execute-to-parse ratios to be excellent at night and bad during the day.

The moral to this story is to look at these ratios and look at how the system operates. Then, using that knowledge, determine if the ratio is okay given the conditions. If I just said that the execute-to-parse ratio for your system should be 95% or better, that would be unachievable in many web-based systems. That doesn't mean you shouldn't try! If you have a routine that will be executed many times to generate a page, you should definitely parse once per page and execute it over and over, closing the cursor if necessary before your connection is returned to the connection pool. (In MOD\_PLSQL, this is all done automatically for you; you do not need to worry about caching the parsed statement, reusing it, and closing it.)

Statspack Top 5 Timed Events

Moving on, we get to the Top 5 Timed Events section (in Oracle9i Release 2 and up) or Top 5 Wait Events (in Oracle9i Release 1 and earlier).

|                    |       |          |          |
|--------------------|-------|----------|----------|
| Top 5 Timed Events |       |          |          |
| ~~~~~              |       |          |          |
|                    |       |          | % Total  |
| Event              | Waits | Time (s) | Ela Time |
| -----              |       |          |          |
| CPU time           |       | 508      | 37.39    |


|                        |        |     |       |
|------------------------|--------|-----|-------|
| direct path write      | 5,168  | 279 | 20.50 |
| db file scattered read | 38,554 | 270 | 19.85 |
| log file sync          | 2,610  | 88  | 6.50  |
| direct path read       | 2,702  | 86  | 6.29  |
| -----                  |        |     |       |

This section is among the most important and relevant sections in the Statspack report. Here is where you find out what events (typically wait events) are consuming the most time.

In Oracle9i Release 2, this section is renamed and includes a new event: CPU time. CPU time is not really a wait event (hence, the new name), but rather the sum of the CPU used by this session, or the amount of CPU time used during the snapshot window. In a heavily loaded system, if the CPU time event is the biggest event, that could point to some CPU-intensive processing (for example, forcing the use of an index when a full scan should have used), which could be the cause of the bottleneck. This example is from a four-CPU machine, so the fact that I used 508 out of 3,600 CPU seconds is okay (15 minutes × 60 seconds × 4 CPUs).

Next, we see that the direct path write value, which isn't good. This wait event will be generated while waiting for writes to TEMP space generally (direct loads, Parallel DML (PDML) such as parallel updates, and uncached LOBs in version 8.1.6 and before (in 8.1.7 there is a separate wait event for lobs) will generate this as well). When I saw this, I dug a little further to see what was causing that number. As it turned out, our PGA\_AGGREGATE\_TARGET parameter was set far too low. We had it at 25MB instead of 250MB, which would be much more appropriate on our system. I reset it (online) and ran the Statspack report again. This time, the direct path write wait was virtually removed from the report. We had 120 waits for direct path writes, totaling three-thousandth second (as opposed to 279 seconds).

Next is the db file scattered read wait value, another high number in this example. That generally happens during a full scan of a table. I used the Statspack report to help me identify the query in question and fixed it. Then I regenerated the Statspack report, for the following updated Top 5 Timed Events report:

Top 5 Timed Events

| ~~~~~                       |       |          | % Total  |
|-----------------------------|-------|----------|----------|
| Event                       | Waits | Time (s) | Ela Time |
| -----                       |       |          |          |
| CPU time                    |       | 592      | 87.62    |
| log file sync               | 3,607 | 31       | 4.62     |
| log file parallel write     | 8,054 | 21       | 3.14     |
| db file sequential read     | 2,844 | 10       | 1.51     |
| control file parallel write | 293   | 5        | .79      |

The db file scattered read waits disappeared from the report. So far, so good.

**The Statspack Load Profile Revisited**

Let's take a peek at the Load Profile section in the new Statspack report I ran for a 15-minute window after making just those two changes mentioned in the previous section.

Load Profile

| ~~~~~ | Per Second | Per Transaction |
|-------|------------|-----------------|
|-------|------------|-----------------|

|                             |          |                         |
|-----------------------------|----------|-------------------------|
|                             | -----    | -----                   |
| Redo size:                  | 9,564.85 | 1,367.28                |
| Logical reads:              | 1,025.17 | 146.55                  |
| Block changes:              | 60.80    | 8.69                    |
| Physical reads:             | 8.49     | 1.21                    |
| Physical writes:            | 1.32     | 0.19                    |
| User calls:                 | 11.89    | 1.70                    |
| Parses:                     | 122.11   | 17.46                   |
| Hard parses:                | 0.52     | 0.07                    |
| Sorts:                      | 23.46    | 3.35                    |
| Logons:                     | 0.43     | 0.06                    |
| Executes:                   | 156.05   | 22.31                   |
| Transactions:               | 7.00     |                         |
|                             |          |                         |
| % Blocks changed per Read:  | 5.93     | Recursive Call %: 97.77 |
| Rollback per transaction %: | 0.00     | Rows per Sort: 6.34     |

We were doing two times the transactions at this point, but we reduced the number of logical reads/transaction greatly (by fixing that one bad query). Physical reads are virtually nonexistent. We are doing two times the work, with a fraction of the waits.

I'm glad I went through this exercise now. Writing this section of the book actually paid off!

### Statspack Report Summary

The first page of the Statspack report told me almost everything I needed to know. Sure, I skipped ahead frequently. For example, when I saw the high number of direct path read waits, I went to the File I/O stats section of the Statspack to confirm my suspicion that TEMP was the culprit. Given the excessively high writes against my temporary tablespace during the period under observation, I was able to confirm that as the problem. Knowing that the major cause of that would be too small of a sort area, I skipped down to the init.ora section and found we were using the PGA\_AGGREGATE\_TARGET parameter, and that the setting of 25MB was far too small for my server. Increasing that to 250MB solved that problem. I changed that one thing, and then repeated the 15-minute Statspack report, using about the same load to confirm the fix.

Then I zeroed in on the high number of db file scattered reads. For that, I used the Top SQL reports to find the SQL with the most reads. I got lucky and spotted the query that was erroneously doing a full scan of a big table. Interestingly, the query had a hint in it (something I am very much against, see Chapter 6 on the CBO for details on that topic). The developers of the application built their system on a small database (they did not test to load; they violated one of my rules). The hinted query worked great on their toy system, but on my machine with well over 40,000 named accounts—well, let's just say it wasn't such a good idea. But those hints stayed in there and were obeyed, causing the query to do the wrong thing in production. Fixing that fixed the other major issue.

Now, I just need to look at how I can speed up my log disks, and I'm finished! That is how I distill a 20-page Statspack report down to something I can use from start to finish in a couple of minutes. Use the first page or two in depth, and then refer to the remaining material as needed.

## DBMS\_PROFILER

I often speak to groups of people. On many occasions, I've taken impromptu polls, asking, "How many people have used the source code profiler built into the database?" Generally, no hands go up. "How many people were actually aware of the fact that there is a source code profiler in the database?" Again, zero, or close to it, hands go up.

A source code profiler like DBMS\_PROFILER can, in minutes, pinpoint that section of code that deserves your undivided attention for an afternoon of tuning. Without it, you could spend a week trying to figure out where you begin to look. My goal here is not to teach you how to use DBMS\_PROFILER, but rather to make you aware of its existence.

## Why You Want to Use the Profiler

Given my experience as a C programmer, I've found that a source code profiler is an invaluable tool for two main tasks:

- Testing your code, to ensure that you have 100% code coverage in your test cases
- Tuning your algorithms, by finding that "low-hanging fruit"—code that if tuned to death would give you the biggest payoff

I don't see how you can do this testing and tuning without using a profiler!

"Within a large loop, if we want to commit every 1,000 records (million or billion records), which is more efficient: using `mod()` then `commit`, or to set up a counter, such as `counter := counter + 1, if count ... then commit; counter := 0;`. For example, in the following:

```
1).....
 START LOOP

 cnt := cnt + 1;
 IF (cnt%1000) = 0 THEN <= using mod() function
 commit;
 END IF;

 END LOOP;

2).....
 START LOOP

 cnt := cnt + 1;
```

```

 IF cnt = 1000 THEN <= no mod() function
 commit;
 cnt := 0;
 END IF;

 END LOOP;


```

Is 1 or 2 better?"

The fastest is to bite the bullet, configure sufficient rollback, and do the update in a single UPDATE statement. The second fastest is to bite the bullet, configure sufficient rollback, and do the transaction in a single loop without commits. The commit is what will slow you down. You'll have to consider how you'll restart this process as well. Watch out for the ora-01555, which will inevitably get you if you are updating the table you are reading as well.

DBMS\_PROFILER is an easy way to determine the best procedural algorithm to use. As an example, I created the following procedures and ran them:

```

create or replace procedure do_mod
as
 cnt number := 0;
begin
 dbms_profiler.start_profiler('mod');
 for i in 1 .. 500000
 loop
 cnt := cnt + 1;
 if (mod(cnt,1000) = 0)
 then
 commit;
 end if;
 end loop;
 dbms_profiler.stop_profiler;
end;
/

create or replace procedure no_mod
as
 cnt number := 0;
begin
 dbms_profiler.start_profiler('no mod');
 for i in 1 .. 500000
 loop
 cnt := cnt + 1;
 if (cnt = 1000)
 then

```



```
 commit;
 cnt := 0;
 end if;
end loop;
dbms_profiler.stop_profiler;
end;
/

exec do_mod
exec no_mod
```

Now, after running DBMS\_PROFILER, it reports:

```
=====
Total time

GRAND_TOTAL

 11.41

=====
Total time spent on each run

RUNID RUN_COMMENT SECS
----- -----
 1 mod 8.18
 2 no mod 3.23

=====
Percentage of time in each module, for each run separately

RUNID RUN_COMMENT UNIT_OWNER UNIT_NAME SECS PERCEN
----- -----
 1 mod OPS$TKYTE DO_MOD 8.18 100.0
 2 no mod OPS$TKYTE NO_MOD 3.23 100.0
```

So already, this shows that the MOD function takes longer. Just to prove that it is MOD causing the difference, we can drill down further.

```
=====
Lines taking more than 1% of the total time, each run separate

RUNID HSECS PCT OWNER UNIT_NA LIN TEXT
----- -----
 1 550.06 48.2 OPS$TKYTE DO_MOD 9 if (mod(cnt,1000) =
 0)
 1 135.22 11.9 OPS$TKYTE DO_MOD 8 cnt := cnt + 1;
```

```

2 107.71 9.4 OPS$TKYTE NO_MOD 8 cnt := cnt + 1;
2 104.34 9.1 OPS$TKYTE NO_MOD 9 if (cnt = 1000)
1 67.95 6.0 OPS$TKYTE DO_MOD 6 for i in 1 .. 500000
2 64.66 5.7 OPS$TKYTE NO_MOD 12 cnt := 0;
1 64.64 5.7 OPS$TKYTE DO_MOD 11 commit;
2 44.99 3.9 OPS$TKYTE NO_MOD 6 for i in 1 .. 500000

```

8 rows selected.

That clinches it. Using MOD took about 5.5 seconds. Doing `if ( cnt=1000 )` took 1 second, plus the time to do `cnt := 0` gives a grand total of about 1.5 seconds. So, we are looking at 5.5 seconds for mod, and 1.5 seconds for `cnt=1000; cnt:=0;`.

## Profiler Resources

The following are some of the resources I recommend for getting information about setting up, installing, using, and interpreting the output of DBMS\_PROFILER:

- *Expert One on One Oracle*, which has a section on setting up the profiler and using it in the appendix, “Necessary Supplied Packages.”
- *Oracle Supplied Packages and Types Guide*, which has a section on how and why to use the profiler, documentation on the tables used to store the profiling information, security concerns/considerations, and documentation for each of the subprograms in the package.
- [www.google.com](http://www.google.com), where you can search the Web for DBMS\_PROFILER and find some articles. Also, search the Usenet news groups there as well. You’ll find quite a few threads on it (many written by me.)

The source code profiler in Oracle is not much different from any source code profiler you might have used with a 3GL language. Well, that is only partially true. The one advantage this profiler has is that the data is stored in the database, in tables. This means that you can create your own custom reports above and beyond what is already there!

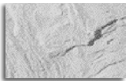
## JDeveloper (and Debugging)

It’s not just for Java anymore. JDeveloper now supports a much broader range of functionality. For me, it has become the Java, PL/SQL, SQL, XML, HTML, and so on development tool of choice.

One of the coolest features of JDeveloper is the fact that it is now written entirely in Java. The same distribution you would run on a Macintosh is the one you would use on Linux, Solaris, HP/UX, or even Windows. That means that the day your application is ready on one platform, it is ready on all platforms.

### NOTE

*In the past, there was a PL/SQL development tool called Procedure Builder, which was limited in scope and functionality. For example, trying to use it on Linux was pretty hard. Given that JDeveloper now runs the same on all platforms, I have the same functionality on the operating system I choose as you do on the one you want to use.*



The most exciting feature in JDeveloper in regard to PL/SQL support is its built-in capability to not only create, edit, and compile PL/SQL, but to interactively debug it as well. The same features one would expect from a C or Java debugger—breakpoints, watch variables, step over, step into, and so on—are fully available with PL/SQL running in an Oracle8i or later database. This is something that will save hours of development time. Not only do you get that nice syntax highlighting that immediately shows you the location of an unclosed quote that prevents your code from compiling, but you can debug it as well. Figure 2-1 shows an example of what you might expect to see.

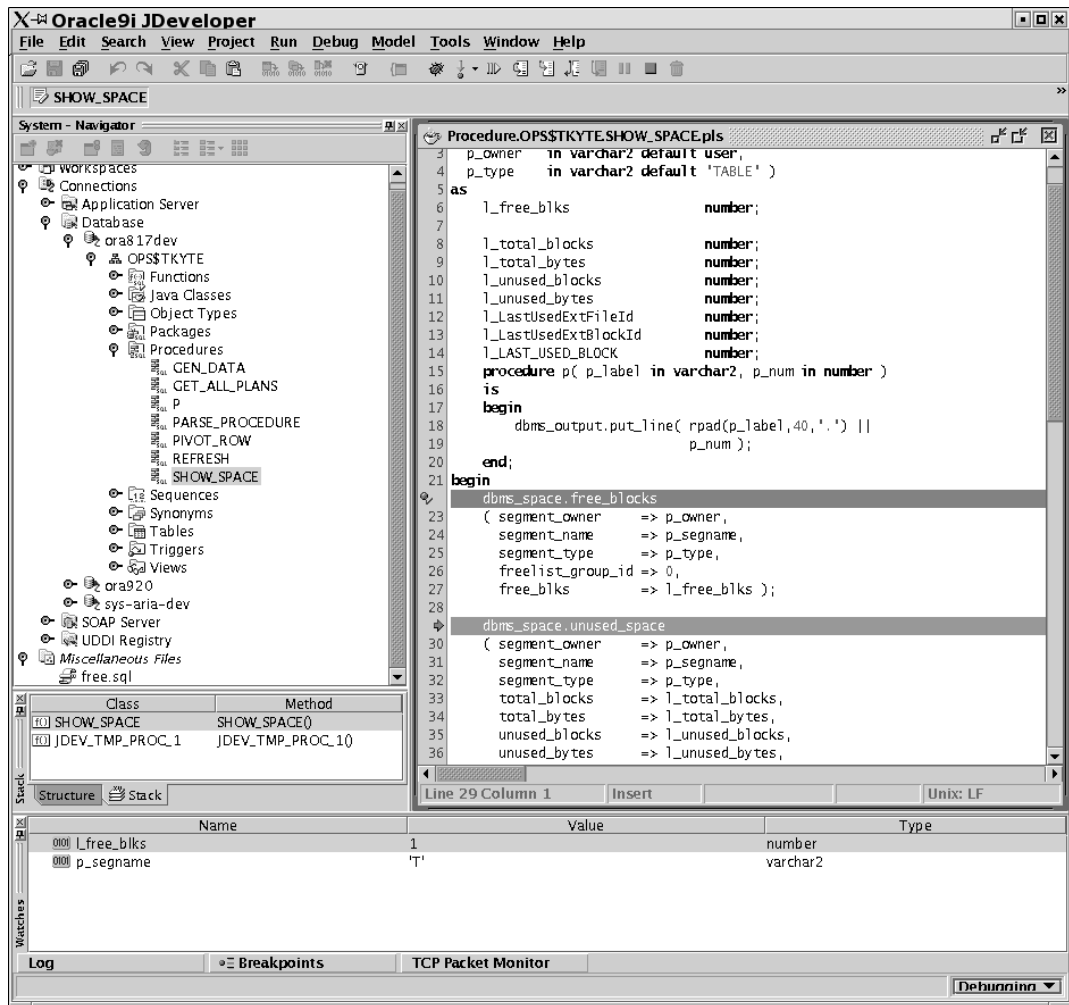
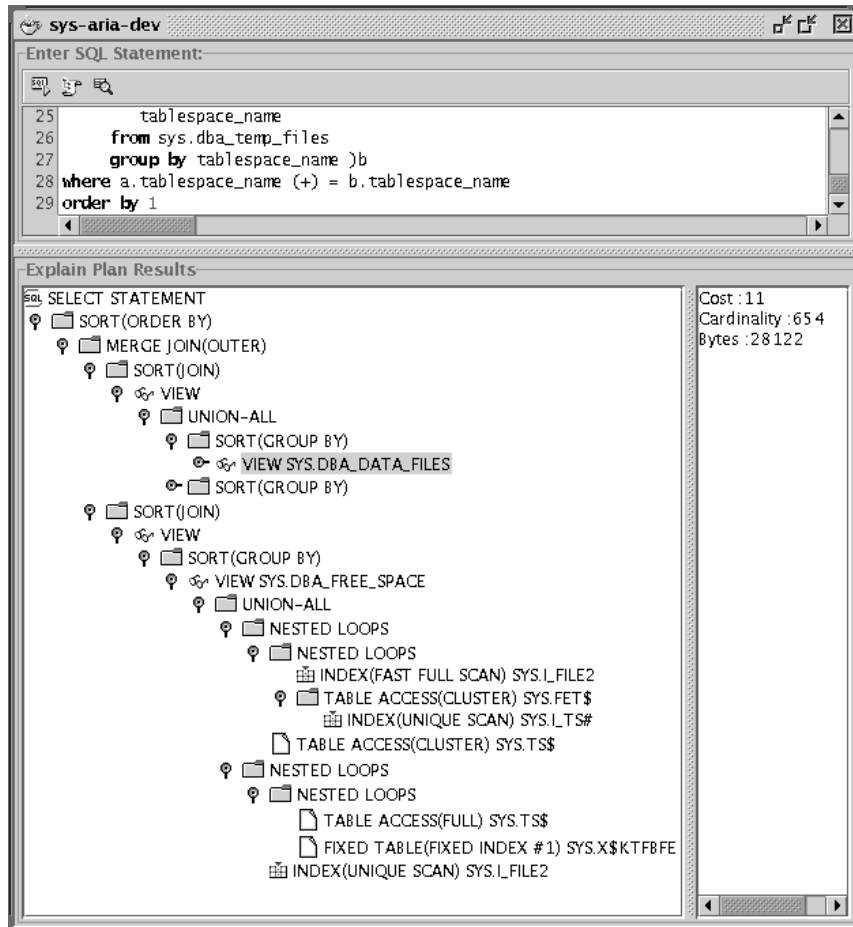
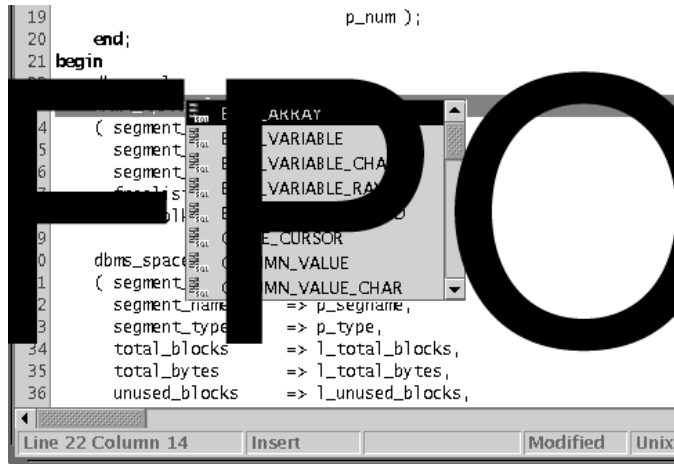


FIGURE 2-1. JDeveloper Main Screen

In Figure 2-1, you can see the navigator on the left, which shows all of the database objects you can work with in JDeveloper now. Additionally, in this example, I'm in a debug session. I had a breakpoint set on the first line of a subroutine and am currently executing line 29 of the code, after stepping over the call to DBMS\_SPACE.FREE\_BLOCKS. At the very bottom of the screen, you can see some watch variables and their values, which change as the procedure modifies them. If I were to run this procedure to completion, any output it generates in the way of DBMS\_OUTPUT calls would be displayed as well.



**FIGURE 2-2.** Graphical Explain Plan



**FIGURE 2-3.** *PL/SQL Code Insight*

In this chapter, we've talked about using EXPLAIN PLAN, AUTOTRACE, and the like. JDeveloper has similar facilities built into it. Figure 2-2 shows an example of a very neat graphical query plan.

What is nice about this is that the information is collapsible, allowing you to easily read a multipage plan by expanding or collapsing entire nodes of the plan. In Figure 2-2, for example, the highlighted node is currently collapsed. If I were to expand it, that node would take up the entire screen. Off to the right side, you can see the details of each step of the plan.

In addition to niceties like the debugger, code-sensitive highlighting for PL/SQL, JDeveloper offers *code insight*. Using this feature, it pops up a list of available routines when you type in a package name or partial name, as shown in Figure 2-3, and you just select from that list.

You also get the SQL Worksheet, which is a nice GUI replacement for SQL\*Plus, as shown in Figure 2-4. Although it is not as powerful as SQL\*Plus, it does have some handy features, such as a spreadsheet widget to view results a page at a time, the ability to paginate up and down, SQL history recall, and so on.

So, if you have been ignoring JDeveloper for the last couple of years because it was a Java development environment, it is time to look again. This is a tool I find myself using more and more frequently when doing development, or even when I just need to run some queries, inspect table structures, and so on.

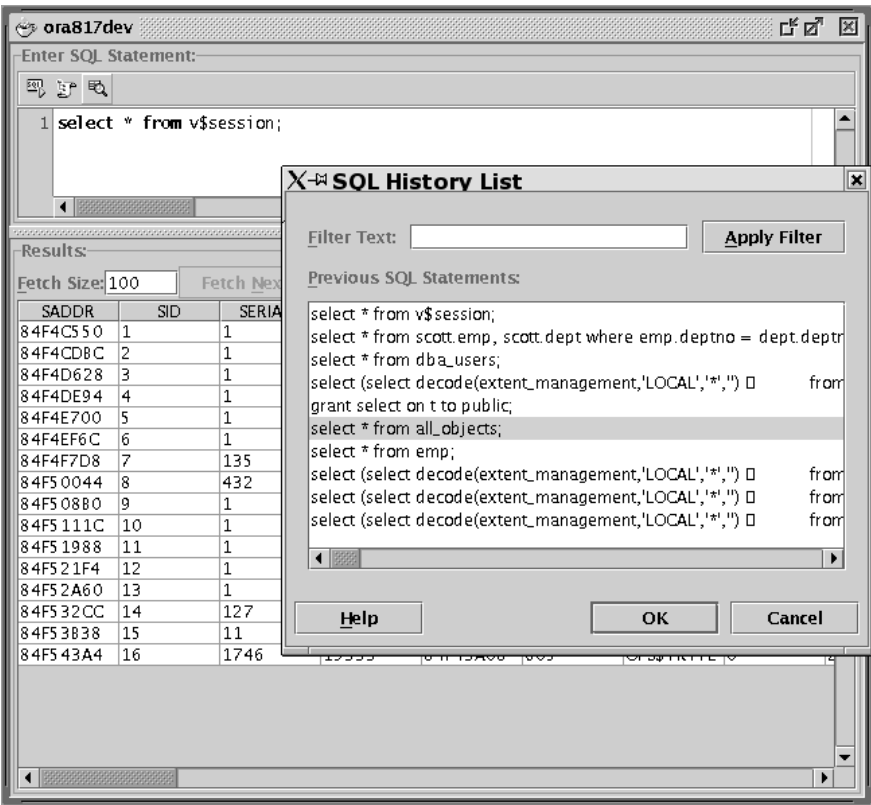


FIGURE 2-4. SQL Worksheet

# Summary

In this chapter, I’ve presented an overview of the tools I use and why I use them. Additionally, we explored how to use many of them. We’ve seen how to read AUTOTRACE and Statspack reports, for example. All of the tools, except for JDeveloper, come with the base database. This means that you have access to them, regardless of what version or flavor of Oracle you bought (Enterprise, Standard, or Personal).

Now that we’ve looked at some of the tools, let’s take a look at the database architecture and how it will affect you. The next chapter covers some architectural topics regarding the database such as shared server versus dedicated server, clustering, partitioning, and the like. You’ll see as we progress through this book that we’ll constantly be coming back to the tools we just reviewed in this chapter over and over.