

SELECT[®] Online

The Journal of the International Oracle Users Group

Select Journal
Home

Regular Columns

Past Issues

General Information

IOUG Home



Select Magazine - November 2000

Volume 7, No. 2

Use EXPLAIN PLAN and TKPROF To Tune Your Applications

By Roger Schrag

Developers and DBAs use the EXPLAIN PLAN and TKPROF functions built into the Oracle 8i server to get the best performance out of their applications. Explore here how to invoke these tools both from the command line and from graphical development tools, and how to read and interpret Oracle 8i execution plans and TKPROF reports.

An Overview of EXPLAIN PLAN and TKPROF

In this section, we'll take a high-level look at the EXPLAIN PLAN and TKPROF facilities: what they are, prerequisites for using them and how to invoke them. We will also look at how these facilities help you tune your applications.

Execution Plans and the EXPLAIN PLAN Statement

Before the database server can execute a SQL statement, Oracle must first parse the statement and develop an execution plan. The execution plan is a task list of sorts that decomposes a potentially complex SQL operation into a series of basic data-access operations. For example, a query against the DEPT table might have an execution plan that consists of an index lookup on the DEPTNO index, followed by a table access by ROWID.

The EXPLAIN PLAN statement allows you to submit an SQL statement to Oracle and have the database prepare the execution plan for the statement without actually executing it. The execution plan is made available to you in the form of rows inserted into a special table called a plan table. You may query the rows in the plan table using ordinary SELECT statements in order to see the steps of the execution plan for the statement you explained. You may keep multiple execution plans in the plan table by assigning each a unique statement_id. Or you may choose to delete the rows from the plan table after you are finished looking at the execution plan. You can also roll back an EXPLAIN PLAN statement in order to remove the execution plan from the plan table.

The EXPLAIN PLAN statement runs very quickly, even if the statement being explained is a query that might run for hours. This is because the statement is simply parsed and its execution plan saved into the plan table. The actual statement is never executed by EXPLAIN PLAN. Along these same lines, if the statement being explained includes bind variables, the variables never need to actually be bound. The values that would be bound are not relevant since the statement is not actually executed.

You don't need any special system privileges in order to use the EXPLAIN PLAN statement. However, you do need to have INSERT privileges on the plan table, and you must have sufficient privileges to execute the statement you are trying to explain. The one difference is that in order to explain a statement that involves views, you must have privileges on all of the tables that make up the view. If you don't, you'll get an "ORA-01039: insufficient privileges on underlying objects of the view" error.

The columns that make up the plan table are as follows:

Name	Null?	Type
STATEMENT_ID		VARCHAR2(30)
TIMESTAMP		DATE
REMARKS		VARCHAR2(80)
OPERATION		VARCHAR2(30)
OPTIONS		VARCHAR2(30)
OBJECT_NODE		VARCHAR2(128)
OBJECT_OWNER		VARCHAR2(30)
OBJECT_NAME		VARCHAR2(30)
OBJECT_INSTANCE		NUMBER(38)
OBJECT_TYPE		VARCHAR2(30)
OPTIMIZER		VARCHAR2(255)
SEARCH_COLUMNS		NUMBER
ID		NUMBER(38)
PARENT_ID		NUMBER(38)
POSITION		NUMBER(38)
COST		NUMBER(38)
CARDINALITY		NUMBER(38)
BYTES		NUMBER(38)
OTHER_TAG		VARCHAR2(255)
PARTITION_START		VARCHAR2(255)
PARTITION_STOP		VARCHAR2(255)
PARTITION_ID		NUMBER(38)
OTHER		LONG
DISTRIBUTION		VARCHAR2(30)

There are other ways to view execution plans besides issuing the EXPLAIN PLAN statement and querying the plan table. SQL*Plus can automatically display an execution plan after each statement is executed. Also, there are many GUI tools available that allow you to click on a SQL statement in the shared pool and view its execution plan, in addition, TKPROF can optionally include execution plans in its reports as well.

Trace Files and the TKPROF Utility

TKPROF is a utility that you invoke at the operating system level in order to analyze SQL trace files and generate reports that present the trace information in a readable form. Although the details of how you invoke TKPROF vary from one platform to the next, Oracle Corporation provides TKPROF with all releases of the database and the basic functionality is the same on all platforms.

The term trace file may be a bit confusing. More recent releases of the database offer a product called Oracle Trace Collection Services. Also, Net8 is capable of generating trace files. SQL trace files are entirely different. SQL trace is a facility that you enable or disable for individual database sessions or for the entire instance as a whole. When SQL trace is enabled for a database session, the Oracle server process handling that session writes detailed information about all database calls and operations to a trace file. Special database events may be set in order to cause Oracle to write even more specific information—such as the values of bind variables—into the trace file.

SQL trace files are text files that, strictly speaking, are readable by humans. However, they are extremely verbose, repetitive and cryptic. For example, if an application opens a cursor and fetches 1,000 rows from the cursor one row at a time, there will be more than 1,000 separate entries in the trace file.

TKPROF is a program that you invoke at the operating system command prompt in order to reformat the trace file into a format that is much easier to comprehend. Each SQL statement is displayed in the report, along with counts of how many times it was parsed, executed, and fetched. CPU time, elapsed time, logical reads, physical reads and rows processed also are reported, along with information about recursion level and misses in the library cache. TKPROF also can optionally include the execution plan for each SQL statement in the report, along with counts of how many rows were processed at each step of the execution plan.

The SQL statements can be listed in a TKPROF report in the order of how much resource they use, if desired. Also, recursive SQL statements issued by the SYS user to manage the data dictionary can be included or excluded, and TKPROF can write SQL statements from the traced session into a spool file.

How EXPLAIN PLAN and TKPROF Aid in the Application Tuning Process

EXPLAIN PLAN and TKPROF are valuable tools in the tuning process. Tuning at the application level typically yields the most dramatic results, and these two tools can help with the tuning in many different ways.

EXPLAIN PLAN and TKPROF allow you to proactively tune an application while it is in development. It is relatively easy to enable SQL trace, run an application in a test environment, run TKPROF on the trace file and review the output to determine if application or schema changes are needed. EXPLAIN PLAN is handy for evaluating individual SQL statements.

By reviewing execution plans, you also can validate the scalability of an application. If the database operations are dependent upon full table scans of tables that could grow quite large, then there may be scalability problems ahead. On the other hand, if you access large tables via selective indexes, then scalability may not be a problem.

EXPLAIN PLAN and TKPROF also may be used in an existing production environment in order to zero in on resource intensive operations and get insights into how the code may be optimized. TKPROF can further be used to quantify the resources required by specific database operations or application functions.

EXPLAIN PLAN also is handy for estimating resource requirements in advance. Suppose you have an ad hoc reporting request against a very large database. Running queries through EXPLAIN PLAN will let you determine in advance if the queries are feasible or if they will be resource intensive and will take unacceptably long to run.

Generating Execution Plans and TKPROF Reports

In this section, we will discuss the details of how to generate execution plans (both with the EXPLAIN PLAN statement and other methods) and how to generate SQL trace files and create TKPROF reports.

Using the EXPLAIN PLAN Statement

Before you can use the EXPLAIN PLAN statement, you must have INSERT privileges on a plan table. The plan table can have any name you like, but the names and data types of the columns are not flexible. You will find a script called utlxplan.sql in \$ORACLE_HOME/rdbms/admin that creates a plan table with the name plan_table in the local schema. If you use this script to create your plan table, you can be assured that the table will have the right definition for use with EXPLAIN PLAN.

Once you have access to a plan table, you are ready to run the EXPLAIN PLAN statement. The syntax is as follows:

```
EXPLAIN PLAN [SET STATEMENT_ID = <string in single quotes>]
```

```
[INTO <plan table name]
```

```
FOR <SQL statement;
```

If you do not specify the INTO clause, then Oracle assumes the name of the plan table is plan_table. You can use the SET clause to assign a name to the execution plan. This is useful if you want to be able to have multiple execution plans stored in the plan table at once—giving each execution plan a distinct name enables you to determine which rows in the plan table belong to which execution plan.

The EXPLAIN PLAN statement runs quickly because all Oracle has to do is parse the SQL statement being explained and store the execution plan in the plan table. The SQL statement can include bind variables, although the variables will not get bound and the values of the bind variables will be irrelevant.

If you issue the EXPLAIN PLAN statement from SQL*Plus, you will get back the feedback message "Explained." At this point the execution plan for the explained SQL statement has been inserted into the plan table, and you can now query the plan table to examine the execution plan.

Execution plans are a hierarchical arrangement of simple data access operations. Because of the hierarchy, you need to use a CONNECT BY clause in your query from the plan table. Using the LPAD function, you can cause the output to be formatted in such a way that the indenting helps you traverse the hierarchy. There are many different ways to format the data retrieved from the plan table. No one query is the best, because the plan table holds a lot of detailed information. Different DBAs will find various aspects more useful in different situations.

A simple SQL*Plus script to retrieve an execution plan from the plan table is as follows:

```
REM
REM explain.sql
REM
SET VERIFY OFF
SET PAGESIZE 100
ACCEPT stmt_id CHAR PROMPT "Enter statement_id: "
COL id          FORMAT 999
COL parent_id   FORMAT 999 HEADING "PARENT"
COL operation    FORMAT a35 TRUNCATE
COL object_name FORMAT a30
SELECT      id, parent_id, LPAD (' ', LEVEL - 1) || operation || ' ' ||
           options operation, object_name
FROM        plan_table
WHERE       statement_id = '&stmt_id'
START WITH id = 0
AND         statement_id = '&stmt_id'
CONNECT BY PRIOR
           id = parent_id
AND         statement_id = '&stmt_id';
```

I have a simple query that we will use in a few examples. We'll call this "the invoice item query." The query is as follows:

```

SELECT a.customer_name, a.customer_number, b.invoice_number,
       b.invoice_type, b.invoice_date, b.total_amount, c.
line_number,
       c.part_number, c.quantity, c.unit_cost
FROM   customers a, invoices b, invoice_items c
WHERE  c.invoice_id = :b1
AND    c.line_number = :b2
AND    b.invoice_id = c.invoice_id
AND    a.customer_id = b.customer_id;

```

The explain.sql SQL*Plus script above displays the execution plan for the invoice item query as follows:

ID	PARENT	OPERATION	OBJECT_NAME
0		SELECT STATEMENT	
1	0	NESTED LOOPS	
2	1	NESTED LOOPS	
3	2	TABLE ACCESS BY INDEX ROWID	INVOICE_ITEMS
4	3	INDEX UNIQUE SCAN	INVOICE_ITEMS_PK
5	2	TABLE ACCESS BY INDEX ROWID	INVOICES
6	5	INDEX UNIQUE SCAN	INVOICES_PK
7	1	TABLE ACCESS BY INDEX ROWID	CUSTOMERS
8	7	INDEX UNIQUE SCAN	CUSTOMERS_PK

The execution plan shows that Oracle is using nested loop joins to join three tables, and that accesses from all three tables are by unique index lookup. This is probably a very efficient query. We will look at how to read execution plans in greater detail in a later section.

The explain.sql script for displaying an execution plan is very basic in that it does not display a lot of the information contained in the plan table. Things left off of the display include optimizer estimated cost, cardinality, partition information (only relevant when accessing partitioned tables) and parallelism information (only relevant when executing parallel queries or parallel DML).

If you are using Oracle 8.1.5 or later, you can find two plan query scripts in \$ORACLE_HOME/rdbms/admin. utlxpls.sql is intended for displaying execution plans of statements that do not involve parallel processing, while utlxplp.sql shows additional information pertaining to parallel processing. The output of the latter script is more confusing, so only use it when parallel query or DML come into play. The output from utlxpls.sql for the invoice item query is as follows:

Plan Table							
Operation	Name	Rows	Bytes	Cost	Pstart	Pstop	
SELECT STATEMENT		1	39	4			
NESTED LOOPS		1	39	4			
NESTED LOOPS		1	27	3			
TABLE ACCESS BY INDEX R	INVOICE_I	1	15	2			
INDEX UNIQUE SCAN	INVOICE_I	2		1			
TABLE ACCESS BY INDEX R	INVOICES	2	24	1			
INDEX UNIQUE SCAN	INVOICES_	2					
TABLE ACCESS BY INDEX RO	CUSTOMERS	100	1K	1			
INDEX UNIQUE SCAN	CUSTOMERS	100					

When you no longer need an execution plan, you should delete it from the plan table. You can do this by rolling back the EXPLAIN PLAN statement (if you have not committed yet) or by deleting rows from the plan table. If you have multiple execution plans in the plan table, then you should delete selectively by statement_id. Note that if you explain two SQL statements and assign both the same statement_id, you will get an ugly Cartesian product when you query the plan table!

The Autotrace Feature of SQL*Plus

SQL*Plus has an autotrace feature that allows you to automatically display execution plans and helpful statistics for each statement executed in a SQL*Plus session without having to use the EXPLAIN PLAN statement or query the plan table. You turn this feature on and off with the following SQL*Plus command:

```
SET AUTOTRACE OFF|ON|TRACEONLY [EXPLAIN] [STATISTICS]
```

When you turn on autotrace in SQL*Plus, the default behavior is for SQL*Plus to execute each statement and display the results in the normal fashion, followed by an execution plan listing and a listing of various server-side resources used to execute the statement. By using the TRACEONLY keyword, you can have SQL*Plus suppress the query results. By using the EXPLAIN or STATISTICS keywords, you can have SQL*Plus display just the execution plan without the resource statistics or just the statistics without the execution plan.

In order to have SQL*Plus display execution plans, you must have privileges on a plan table by the name of plan_table. In order to have SQL*Plus display the resource statistics, you must have SELECT privileges on v\$sesstat, v\$statname and v\$session. There is a script in \$ORACLE_HOME/sqlplus/admin called plustrce.sql that creates a role with these three privileges in it, but this script is not run automatically by the Oracle installer.

The autotrace feature of SQL*Plus makes it extremely easy to generate and view execution plans, with resource statistics as an added bonus. One key drawback, however, is that the statement being explained must actually be executed by the database server before SQL*Plus will display the execution plan. This makes the tool unusable in the situation where you would like to predict how long an operation might take to complete.

A sample output from SQL*Plus for the invoice item query is as follows:

```

Execution Plan
-----

0 SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=1 Bytes=39)
1 0 NESTED LOOPS (Cost=4 Card=1 Bytes=39)
2 1 NESTED LOOPS (Cost=3 Card=1 Bytes=27)
3 2 TABLE ACCESS (BY INDEX ROWID) OF 'INVOICE_ITEMS' (Cost=2 Card=1 Bytes=15)
4 3 INDEX (UNIQUE SCAN) OF 'INVOICE_ITEMS_PK' (UNIQUE) (Cost=1 Card=2)
5 2 TABLE ACCESS (BY INDEX ROWID) OF 'INVOICES' (Cost=1 Card=2 Bytes=24)
6 5 INDEX (UNIQUE SCAN) OF 'INVOICES_PK' (UNIQUE)
7 1 TABLE ACCESS (BY INDEX ROWID) OF 'CUSTOMERS' (Cost=1 Card=100 Bytes=1200)
8 7 INDEX (UNIQUE SCAN) OF 'CUSTOMERS_PK' (UNIQUE)

Statistics
-----

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

```

Although we haven't discussed how to read an execution plan yet, you can see that the output from SQL*Plus provides the same basic information, with several additional details in the form of estimates from the query optimizer.

Using GUI Tools to View Execution Plans

There are many GUI tools available that allow you to view execution plans for SQL statements that you specify or for statements already sitting in the shared pool of the database instance. Any comprehensive database management tool will offer this capability, but there are several free tools available for download on the Internet that have this feature as well.

One tool in particular that I really like is TOAD (Tool for Oracle Application Developers). Although TOAD was originally developed as a free tool, Quest Software now owns TOAD, and it is available in both a free version (limited functionality) and an enhanced version that may be purchased (full feature set). You may download TOAD from Quest Software at www.toadsoft.com/downld.html. TOAD has lots of handy features. The one relevant to us here is the ability to click on any SQL

statement in the shared pool and instantly view its execution plan.

As with the EXPLAIN PLAN statement and the autotrace facility in SQL*Plus, you will need to have access to a plan table. Here is TOAD's rendition of the execution plan for the invoice item query we've been using:

You can see that the information displayed is almost identical to that from the autotrace facility in SQL*Plus. One nice feature of TOAD's execution plan viewer is that you can collapse and expand the individual operations that make up the execution plan. Also, the vertical and horizontal lines connecting different steps help you keep track of the nesting and which child operations go with which parent operations in the hierarchy. The benefits of these features become more apparent when working with extremely complicated execution plans.

Unfortunately, when looking at execution plans for SQL statements that involve database links or parallelism, TOAD leaves out critical information that is present in the plan table and is reported by the autotrace feature of SQL*Plus. Perhaps this deficiency only exists in the free version of TOAD; I would like to think that if you pay for the full version of TOAD, you will get complete execution plans.

Generating a SQL Trace File

SQL trace may be enabled at the instance or session level. To enable SQL trace at the instance level, add the following parameter setting to the instance parameter file and restart the database instance:

```
sql_trace = true
```

When an Oracle instance starts up with the above parameter setting, every database session will run in SQL trace mode, meaning that all SQL operations for every database session will be written to trace files. Even the daemon processes like PMON and SMON will be traced! In practice, enabling SQL trace at the instance level is usually not very useful. It can be overpowering, sort of like using a fire hose to pour yourself a glass of water.

It is more typical to enable SQL trace in a specific session. You can turn SQL trace on and off as desired to trace only the operations that you wish to trace. If you have access to the database session you wish to trace, then use the ALTER SESSION statement as follows to enable and disable SQL trace:

```
ALTER SESSION SET sql_trace = TRUE|FALSE;
```

This technique works well if you have access to the application source code and can add in ALTER SESSION statements at will. It also works well when the application runs from SQL*Plus and you can execute ALTER SESSION statements at the SQL*Plus prompt before invoking the application.

In situations where you cannot invoke an ALTER SESSION command from the session you wish to trace—as with prepackaged applications, for example—you can connect to the database as a DBA user and invoke the dbms_system built-in package in order to turn on or off SQL trace in another session. You do this by querying v\$session to find the SID and serial number of the session you wish to trace and then invoking the dbms_system package with a command of the form:

```
EXECUTE SYS.dbms_system.set_sql_trace_in_session (<SID, <serial#, TRUE|FALSE);
```

When you enable SQL trace in a session for the first time, the Oracle server process handling that session will create a trace file in the directory on the database server designated by the user_dump_dest initialization parameter. As the server is called by the application to perform database operations, the server process will append to the trace file.

Note that tracing a database session that is using multi-threaded server (MTS) is a bit complicated because each database request from the application could get picked up by a different server process. In this situation, each server process will create a trace file containing trace information about the operations performed by that process only. This means that you will potentially have to combine multiple trace files together to get the full picture of how the application interacted with the database. Furthermore, if multiple sessions are being traced at once, it will be hard to tell which operations in the trace file belong to which session. For these reasons, you should use dedicated server mode when tracing a database session with SQL trace.

SQL trace files contain detailed timing information. By default, Oracle does not track timing, so all timing figures in trace files will show as zero. If you would like to see legitimate timing information, then you need to enable timed statistics. You can do this at the instance level by setting the following parameter in the instance parameter file and restarting the instance:

```
timed_statistics = true
```

You also can dynamically enable or disable timed statistics collection at either the instance or the session level with the following commands:

```
ALTER SYSTEM SET timed_statistics = TRUE|FALSE;
```

```
ALTER SESSION SET timed_statistics = TRUE|FALSE;
```

There is no known way to enable timed statistics collection for an individual session from another session (akin to the SYS.dbms_system.set_sql_trace_in_session built-in).

There is very high overhead associated with enabling SQL trace. Some DBAs believe the performance penalty could be more than 25 percent. Another concern is that enabling SQL trace causes the generation of potentially large trace files. For these reasons, you should use SQL trace sparingly. Only trace what you need to trace and think very carefully before enabling SQL trace at the instance level.

On the other hand, there is little, if any, measurable performance penalty in enabling timed statistics collection. Many DBAs run production databases with timed statistics collection enabled at the system level so that various system statistics (more than just SQL trace files) will include detailed timing information. Note that Oracle 8.1.5 had some serious memory corruption bugs associated with enabling timed statistics collection at the instance level, but these seem to have been fixed in Oracle 8.1.6.

On Unix platforms, Oracle will typically set permissions so that only the oracle user and members of the dba Unix group can read the trace files. If you want anybody with a Unix login to be able to read the trace files, then you should set the following undocumented (but supported) initialization parameter in the parameter file:

```
_trace_files_public = true
```

If you trace a database session that makes a large number of calls to the database server, the trace file can get quite large. The initialization parameter max_dump_file_size allows you to set a maximum trace file size. On Unix platforms, this parameter is specified in units of 512 byte blocks. Thus a setting of 10240 will limit trace files to 5 Mb apiece. When a SQL trace file reaches the maximum size, the database server process stops writing trace information to the trace file. On Unix platforms, there will be no limit on trace file size if you do not explicitly set the max_dump_file_size parameter.

If you are tracing a session and realize that the trace file is about to reach the limit set by max_dump_file_size, you can eliminate the limit dynamically so that you don't lose trace information. To do this, query the PID column in v\$process to find the Oracle PID of the process writing the trace file. Then execute the following statements in SQL*Plus:

```
CONNECT / AS SYSDBA
```

```
ORADEBUG SETORAPID <pid
```

```
ORADEBUG UNLIMIT
```

Running TKPROF on a SQL Trace File

Before you can use TKPROF, you need to generate a trace file and locate it. Oracle writes trace files on the database server to the directory specified by the user_dump_dest initialization parameter. (Daemon processes such as PMON write their trace files to the directory specified by background_dump_dest.) On Unix platforms, the trace file will have a name that incorporates the operating system PID of the server process writing the trace file.

If there are a lot of trace files in the user_dump_dest directory, it could be tricky to find the one you want. One tactic is to examine the timestamps on the files. Another technique is to embed a comment in a SQL statement in the application that will make its way into the trace file. An example of this is as follows:

```
ALTER SESSION /* Module glpost.c */ SET sql_trace = TRUE;
```

Because TKPROF is a utility you invoke from the operating system and not from within a database session, there will naturally be some variation in the user interface from one operating system platform to another. On Unix platforms, you run TKPROF from the operating system prompt with a syntax as follows:

```
tkprof <trace file <output file [explain=<username/password] [sys=n] \
[insert=<filename] [record=<filename] [sort=<keyword]
```

If you invoke TKPROF with no arguments at all, you will get a help screen listing all of the options. This is especially helpful because TKPROF offers many sort capabilities, but you select the desired sort by specifying a cryptic keyword. The help screen identifies all of the sort keywords.

In its simplest form, you run TKPROF specifying the name of a SQL trace file and an output filename. TKPROF will read the trace file and generate a report file with the output filename you specified. TKPROF will not connect to the database, and the report will not include execution plans for the SQL statements. SQL statements that were executed by the SYS user recursively (to dynamically allocate an extent in a dictionary-managed tablespace, for example) will be included in the report, and the statements will appear in the report approximately in the order in which they were executed in the database session that was traced.

If you include the explain keyword, TKPROF will connect to the database and execute an EXPLAIN PLAN statement for each SQL statement found in the trace file. The execution plan results will be included in the report file. As we will see later, TKPROF merges valuable information from the trace file into the execution plan display, making this just about the most valuable way to display an execution plan. Note that the username you specify when running TKPROF should be the same as the username connected in the database session that was traced. You do not need to have a plan table in order to use the explain keyword—TKPROF will create and drop its own plan table if needed.

If you specify sys=n, TKPROF will exclude from the report SQL statements initiated by Oracle as the SYS user. This will make your report look tidier because it will only contain statements actually issued by your application. The theory is that Oracle internal SQL has already been fully optimized by the kernel developers at Oracle Corporation, so you should not have to deal with it. However, using sys=n will exclude potentially valuable information from the TKPROF report. Suppose the SGA is not properly sized on the instance and Oracle is spending a lot of time resolving dictionary cache misses. This would manifest itself in lots of time spent on recursive SQL statements initiated by the SYS user. Using sys=n would exclude this information from the report.

If you specify the insert keyword, TKPROF will generate a SQL script in addition to the regular report. This SQL script creates a table called tkprof_table and inserts one row for each SQL statement displayed on the report. The row will contain the text of the SQL statement traced and all of the statistics displayed in the report. You could use this feature to effectively load the TKPROF report into the database and use SQL to analyze and manipulate the statistics. I've never needed to use this feature, but I suppose it could be helpful in some situations.

If you specify the record keyword, TKPROF will generate another type of SQL script in addition to the regular report. This SQL script will contain a copy of each SQL statement issued by the application while tracing was enabled. You could get this same information from the TKPROF report itself, but this way could save some cutting and pasting.

The sort keyword is extremely useful. Typically, a TKPROF report may include hundreds of SQL statements, but you may only be interested in a few resource intensive queries.

The sort keyword allows you to order the listing of the SQL statements so that you don't have to scan the entire file looking for resource hogs. In some ways, the sort feature is too powerful for its own good. For example, you cannot sort statements by CPU time consumed—instead you sort by CPU time spent parsing, CPU time spent executing or CPU time spent fetching.

A sample TKPROF report for the invoice item query we've been using so far is as follows:

```
TKPROF: Release 8.1.6.1.0 - Production on Wed Aug 9 19:06:36 2000
(c) Copyright 1999 Oracle Corporation. All rights reserved.
Trace file: example.trc
Sort options: default
*****
count = number of times OCI procedure was executed
```

```

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

ALTER SESSION /* TKPROF example */ SET sql_trace = TRUE

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

total 1 0.00 0.00 0 0 0 0

Misses in library cache during parse: 0
Misses in library cache during execute: 1

Optimizer goal: CHOOSE

Parsing user id: 34 (RSCHRAG)
*****

ALTER SESSION SET timed_statistics = TRUE

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 0 0.00 0.00 0 0 0 0
-----

total 2 0.00 0.00 0 0 0 0

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: 34 (RSCHRAG)
*****

SELECT a.customer_name, a.customer_number, b.invoice_number,

```

```

b.invoice_type, b.invoice_date, b.total_amount, c.line_number,
c.part_number, c.quantity, c.unit_cost
FROM customers a, invoices b, invoice_items c
WHERE c.invoice_id = :b1
AND c.line_number = :b2
AND b.invoice_id = c.invoice_id
AND a.customer_id = b.customer_id
call count cpu elapsed disk query current rows

```

```
-----
```

```
Parse 1 0.05 0.02 0 0 0 0
```

```
Execute 1 0.00 0.00 0 0 0 0
```

```
Fetch 2 0.00 0.00 8 8 0 1
```

```
-----
```

```
total 4 0.05 0.02 8 8 0 1
```

```
Misses in library cache during parse: 1
```

```
Optimizer goal: CHOOSE
```

```
Parsing user id: 34 (RSCHRAG)
```

```
Rows Row Source Operation
```

```
-----
```

```
1 NESTED LOOPS
```

```
1 NESTED LOOPS
```

```
1 TABLE ACCESS BY INDEX ROWID INVOICE_ITEMS
```

```
1 INDEX UNIQUE SCAN (object id 21892)
```

```
1 TABLE ACCESS BY INDEX ROWID INVOICES
```

```
1 INDEX UNIQUE SCAN (object id 21889)
```

```
1 TABLE ACCESS BY INDEX ROWID CUSTOMERS
```

```
1 INDEX UNIQUE SCAN (object id 21887)
```

```
Rows Execution Plan
```

```
-----
```

```
0 SELECT STATEMENT GOAL: CHOOSE
```

```
1 NESTED LOOPS
```

```
1 NESTED LOOPS
```

```
1 TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF
```

```

' INVOICE_ITEMS '

1 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'INVOICE_ITEMS_PK'

(UNIQUE)

1 TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF

' INVOICES '

1 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'INVOICES_PK'

(UNIQUE)

1 TABLE ACCESS GOAL: ANALYZED (BY INDEX ROWID) OF 'CUSTOMERS'

1 INDEX GOAL: ANALYZED (UNIQUE SCAN) OF 'CUSTOMERS_PK'

(UNIQUE)

*****
ALTER SESSION SET sql_trace = FALSE

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 0 0.00 0.00 0 0 0 0

-----

total 2 0.00 0.00 0 0 0 0

Misses in library cache during parse: 1

Optimizer goal: CHOOSE

Parsing user id: 34 (RSCHRAG)

*****

OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call count cpu elapsed disk query current rows

-----

Parse 3 0.05 0.02 0 0 0 0

Execute 4 0.00 0.00 0 0 0 0

Fetch 2 0.00 0.00 8 8 0 1

-----

total 9 0.05 0.02 8 8 0 1

Misses in library cache during parse: 3

```

```

Misses in library cache during execute: 1

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call count cpu elapsed disk query current rows
-----
Parse 24 0.02 0.04 1 0 1 0
Execute 62 0.01 0.05 0 0 0 0
Fetch 126 0.02 0.02 6 198 0 100
-----

total 212 0.05 0.11 7 198 1 100

Misses in library cache during parse: 11

  4 user SQL statements in session.

 24 internal SQL statements in session.

 28 SQL statements in session.

 1 statement EXPLAINed in this session.

*****

Trace file: example.trc

Trace file compatibility: 8.00.04

Sort options: default

 1 session in tracefile.

 4 user SQL statements in trace file.

 24 internal SQL statements in trace file.

 28 SQL statements in trace file.

 15 unique SQL statements in trace file.

 1 SQL statements EXPLAINed using schema:

RSCHRAG.prof$plan_table

Default table was used.

Table was created.

Table was dropped.

381 lines in trace file.

```

You can see that there is a lot going on in a TKPROF report. We will talk about how to read the report and interpret the different statistics in the next section.

Interpreting Execution Plans and TKPROF Reports

In this section, we will discuss how to read and interpret execution plans and TKPROF reports. While generating an execution plan listing or creating a TKPROF report file is usually a straightforward process, analyzing the data and reaching the correct conclusions can be more of an art. We'll look at lots of examples along the way.

Understanding Execution Plans

An execution plan is a hierarchical structure somewhat like an inverted tree. The SQL statement being examined can be thought of as the root of the tree. This will be the first line on an execution plan listing, the line that is least indented. This statement can be thought of as the result of one or more subordinate operations. Each of these subordinate operations can possibly be decomposed further. This decomposition process continues repeatedly until eventually even the most complex SQL statement is broken down into a set of basic data access operations.

Consider the following simple query and execution plan:

```

SELECT customer_id, customer_number, customer_name
FROM customers
WHERE UPPER (customer_name) LIKE 'ACME%'
ORDER BY customer_name;

  ID PARENT OPERATION OBJECT_NAME
-----
0 SELECT STATEMENT
1 0 SORT ORDER BY
2 1 TABLE ACCESS FULL CUSTOMERS

```

The root operation—that which we explained—is a SELECT statement. The output of the statement will be the results of a sort operation (for the purposes of satisfying the ORDER BY clause). The input to the sort will be the results of a full table scan of the customers' table. Stated more clearly, the database server will execute this query by checking every row in the customers' table for a criteria match and sorting the results. Perhaps the developer expected Oracle to use an index on the customer_name column to avoid a full table scan, but the use of the UPPER function defeated the index. (A function-based index could be deployed to make this query more efficient.)

Consider the following query and execution plan:

```

SELECT a.customer_name, b.invoice_number, b.invoice_date
FROM customers a, invoices b
WHERE b.invoice_date TRUNC (SYSDATE - 1)
AND a.customer_id = b.customer_id;

  ID PARENT OPERATION OBJECT_NAME
-----
0 SELECT STATEMENT
1 0 NESTED LOOPS
2 1 TABLE ACCESS BY INDEX ROWID INVOICES

```

```

3 2 INDEX RANGE SCAN INVOICES_DATE
4 1 TABLE ACCESS BY INDEX ROWID CUSTOMERS
5 4 INDEX UNIQUE SCAN CUSTOMERS_PK

```

Again, the root operation is a SELECT statement. This time, the SELECT statement gets its input from the results of a nested loops join operation. The nested loops operation takes as input the results of accesses to the invoices and customers' tables. (You can tell from the indenting that accesses to both tables feed directly into the nested loops operation.) The invoices table is accessed by a range scan of the invoices_date index, while the customers table is accessed by a unique scan of the customers_pk index.

In plainer language, here is how Oracle will execute this query: Oracle will perform a range scan on the invoices_date index to find the ROWIDs of all rows in the invoices table that have an invoice date matching the query criteria. For each ROWID found, Oracle will fetch the corresponding row from the invoices table, look up the customer_id from the invoices record in the customers_pk index, and use the ROWID found in the customers_pk index entry to fetch the correct customer record. This, in effect, joins the rows fetched from the invoices table with their corresponding matches in the customers table. The results of the nested loops join operation are returned as the query results.

Consider the following query and execution plan:

```

SELECT a.customer_name, COUNT (DISTINCT b.invoice_id)      "Open Invoices",
       COUNT (c.invoice_id) "Open Invoice Items"
FROM customers a, invoices b, invoice_items c
WHERE b.invoice_status = 'OPEN'
AND a.customer_id = b.customer_id
AND c.invoice_id (+) = b.invoice_id
GROUP BY a.customer_name;

ID PARENT OPERATION OBJECT_NAME
-----
0 SELECT STATEMENT
1 0 SORT GROUP BY
2 1 NESTED LOOPS OUTER
3 2 HASH JOIN
4 3 TABLE ACCESS BY INDEX ROWID INVOICES
5 4 INDEX RANGE SCAN INVOICES_STATUS
6 3 TABLE ACCESS FULL CUSTOMERS
7 2 INDEX RANGE SCAN INVOICE_ITEMS_PK

```

This execution plan is more complex than the previous two, and here you can start to get a feel for the way in which

complex operations get broken down into simpler subordinate operations. To execute this query, the database server will do the following: First Oracle will perform a range scan on the invoices_status index to get the ROWIDs of all rows in the invoices table with the desired status. For each ROWID found, the record from the invoices table will be fetched.

This set of invoice records will be set aside for a moment while the focus turns to the customers' table. Here, Oracle will fetch all customers' records with a full table scan. To perform a hash join between the invoices and customers; tables, Oracle will build a hash from the customer records and use the invoice records to probe the customer hash.

Next, a nested loops join will be performed between the results of the hash join and the invoice_items_pk index. For each row resulting from the hash join, Oracle will perform a unique scan of the invoice_items_pk index to find index entries for matching invoice items. Note that Oracle gets everything it needs from the index and doesn't even need to access the invoice_items table at all. Also note that the nested loops operation is an outer join. A sort operation for the purposes of grouping is performed on the results of the nested loops operation in order to complete the SELECT statement.

It is interesting to note that Oracle chose to use a hash join and a full-table scan on the customers table instead of the more traditional nested-loops join. In this database, there are many invoices and a relatively small number of customers, making a full-table scan of the customers' table less expensive than repeated index lookups on the customers_pk index. But suppose the customers' table was enormous and the relative number of invoices was quite small. In that scenario, a nested-loops join might be better than a hash join. Examining the execution plan allows you to see which join method Oracle is using. You could then apply optimizer hints to coerce Oracle to use alternate methods and compare the performance.

You may wonder how I got that whole detailed explanation out of the eight-line execution plan listing shown above. Did I read anything into the execution plan? No! It's all there! Understanding the standard inputs and outputs of each type of operation and coupling this with the indenting is key to reading an execution plan.

A nested-loops join operation always takes two inputs: For every row coming from the first input, the second input is executed once to find matching rows. A hash-join operation also takes two inputs: The second input is read completely once and used to build a hash. For each row coming from the first input, one probe is performed against this hash. Sorting operations, meanwhile, take in one input. When the entire input has been read, the rows are sorted and output in the desired order.

Now let's look at a query with a more complicated execution plan:

```

SELECT customer_name
FROM customers a
WHERE EXISTS
(
SELECT 1
FROM invoices_view b
WHERE b.customer_id = a.customer_id
AND number_of_lines > 100
)
ORDER BY customer_name;

ID PARENT OPERATION OBJECT_NAME
-----
0 SELECT STATEMENT
1 0 SORT ORDER BY
2 1 FILTER

```

```

3 2 TABLE ACCESS FULL CUSTOMERS

4 2 VIEW INVOICES_VIEW

5 4 FILTER

6 5 SORT GROUP BY

7 6 NESTED LOOPS

8 7 TABLE ACCESS BY INDEX ROWID INVOICES

9 8 INDEX RANGE SCAN INVOICES_CUSTOMER_ID

10 7 INDEX RANGE SCAN INVOICE_ITEMS_PK

```

This execution plan is somewhat complex because the query includes a subquery that the optimizer could not rewrite as a simple join, and a view whose definition could not be merged into the query. The definition of the invoices_view view is as follows:

```

CREATE OR REPLACE VIEW invoices_view
AS
SELECT a.invoice_id, a.customer_id, a.invoice_date,      a.invoice_status,
       a.invoice_number, a.invoice_type, a.total_amount,
       COUNT(*) number_of_lines
FROM invoices a, invoice_items b
WHERE b.invoice_id = a.invoice_id
GROUP BY a.invoice_id, a.customer_id,      a.invoice_date, a.invoice_status,
         a.invoice_number, a.invoice_type, a.total_amount;

```

Here is what this execution plan says: Oracle will execute this query by reading all rows from the customers' table with a full-table scan. For each customer record, the invoices_view view will be assembled as a filter and the relevant contents of the view will be examined to determine whether the customer should be part of the result set or not.

Oracle will assemble the view by performing an index range scan on the invoices_customer_id index and fetching the rows from the invoices table containing one specific customer_id. For each invoice record found, the invoice_items_pk index will be range scanned to get a nested loops join of invoices to their invoice_items records. The results of the join are sorted for grouping, and then groups with 100 or fewer invoice_items records are filtered out.

What is left at the step with ID 4 is a list of invoices for one specific customer that have more than 100 invoice_items records associated. If at least one such invoice exists, then the customer passes the filter at the step with ID 2. Finally, all customer records passing this filter are sorted for correct ordering and the results are complete.

Note that queries involving simple views will not result in a "view" operation in the execution plan. This is because Oracle can often merge a view definition into the query referencing the view so that the table accesses required to implement the view just become part of the regular execution plan. In this example, the GROUP BY clause embedded in the view foiled Oracle's ability to merge the view into the query, making a separate "view" operation necessary in order to execute the query.

Also note that the filter operation can take on a few different forms. In general, a filter operation is where Oracle looks at a set of candidate rows and eliminates some based on certain criteria. These criteria could involve a simple test such as number_of_lines > 100 or an elaborate subquery.

In this example, the filter at step ID 5 takes only one input. Here, Oracle evaluates each row from the input one at a time and either adds the row to the output or discards it as appropriate. Meanwhile, the filter at step ID 2 takes two inputs. When a filter takes two inputs, Oracle reads the rows from the first input one at a time and executes the second input once for each row. Based on the results of the second input, the row from the first input is either added to the output or discarded.

Oracle is able to perform simple filtering operations while performing a full-table scan. Therefore, a separate filter operation will not appear in the execution plan when Oracle performs a full-table scan and throws out rows that don't satisfy a WHERE clause. Filter operations with one input commonly appear in queries with view operations or HAVING clauses, while filter operations with multiple inputs will appear in queries with EXISTS clauses.

An important note about execution plans and subqueries: When a SQL statement involves subqueries, Oracle tries to merge the subquery into the main statement by using a join. If this is not feasible and the subquery does not have any dependencies or references to the main query, then Oracle will treat the subquery as a completely separate statement from the standpoint of developing an execution plan—almost as if two separate SQL statements were sent to the database server. When you generate an execution plan for a statement that includes a fully autonomous subquery, the execution plan may not include the operations for the subquery. In this situation, you need to generate an execution plan for the subquery separately.

Other Columns in the Plan Table

Although the plan table contains 24 columns, so far we have only been using six of them in our execution plan listings. These six will get you very far in the tuning process, but some of the other columns can be mildly interesting at times. Still other columns can be very relevant in specific situations.

The optimizer column in the plan table shows the mode (such as RULE or CHOOSE) used by the optimizer to generate the execution plan. The timestamp column shows the date and time that the execution plan was generated. The remarks column is an 80-byte field where you may put your own comments about each step of the execution plan. You can populate the remarks column by using an ordinary UPDATE statement against the plan table.

The object_owner, object_node and object_instance columns can help you further distinguish the database object involved in the operation. You might look at the object_owner column, for example, if objects in multiple schemas have the same name and you are not sure which one is being referenced in the execution plan. The object_node is relevant in distributed queries or transactions. It indicates the database link name to the object if the object resides in a remote database. The object_instance column is helpful in situations such as a self-join where multiple instances of the same object are used in one SQL statement.

The partition_start, partition_stop and partition_id columns offer additional information when a partitioned table is involved in the execution plan. The distribution column gives information about how the multiple Oracle processes involved in a parallel query or parallel DML operation interact with each other.

The cost, cardinality and bytes columns show estimates made by the cost-based optimizer as to how expensive an operation will be. Remember that the execution plan is inserted into the plan table without actually executing the SQL statement. Therefore, these columns reflect Oracle's estimates and not the actual resources used. While it can be amusing to look at the optimizer's predictions, sometimes you need to take them with a grain of salt. Later we'll see that TKPROF reports can include specific information about actual resources used at each step of the execution plan.

The "other" column in the plan table is a wild card where Oracle can store any sort of textual information about each step of an execution plan. The other_tag column gives an indication of what has been placed in the "other" column. This column will contain valuable information during parallel queries and distributed operations.

Consider the following distributed query and output from the SQL*Plus autotrace facility:

```

SELECT /*+ RULE */
    a.customer_number, a.customer_name, b.contact_id, b.contact_name
FROM customers a, contacts@sales.acme.com b
WHERE UPPER (b.contact_name) = UPPER (a.customer_name)
ORDER BY a.customer_number, b.contact_id;

Execution Plan
-----
0  SELECT STATEMENT Optimizer=HINT: RULE
1  0  SORT (ORDER BY)
2  1  MERGE JOIN
3  2  SORT (JOIN)
4  3  REMOTE* SALES.ACME.COM
5  2  SORT (JOIN)
6  5  TABLE ACCESS (FULL) OF 'CUSTOMERS'
4  SERIAL_FROM_REMOTE SELECT "CONTACT_ID", "CONTACT_NAME" FROM "CONTACTS" "B"

```

In the execution plan hierarchy, the step with ID 4 is displayed as a remote operation through the sales.acme.com database link. At the bottom of the execution plan, you can see the actual SQL statement that the local database server sends to sales.acme.com to perform the remote operation. This information came from the "other" and other_tag columns of the plan table.

Here is how to read this execution plan: Oracle observed a hint and used the RULE optimizer mode in order to develop the execution plan. First, a remote query will be sent to sales.acme.com to fetch the contact_ids and names from a remote table. These fetched rows will be sorted for joining purposes and temporarily set aside. Next, Oracle will fetch all records from the customers table with a full-table scan and sort them for joining purposes. Next, the set of contacts and the set of customers will be joined using the merge-join algorithm. Finally, the results of the merge join will be sorted for proper ordering and the results will be returned.

The merge-join operation always takes two inputs, with the prerequisite that each input has already been sorted on the join column or columns. The merge-join operation reads both inputs in their entirety at one time and outputs the results of the join. Merge joins and hash joins are usually more efficient than nested-loops joins when remote tables are involved, because these types of joins will almost always involve fewer network roundtrips. Hash joins are not supported when rule-based optimization is used. Because of the RULE hint, Oracle chose a merge join.

Reading TKPROF Reports

Every TKPROF report starts with a header that lists the TKPROF version, the date and time the report was generated, the name of the trace file, the sort option used and a brief definition of the column headings in the report. Every report ends with a series of summary statistics. You can see the heading and summary statistics on the sample TKPROF report shown earlier in this article.

The main body of the TKPROF report consists of one entry for each distinct SQL statement that was executed by the database server while SQL trace was enabled. There are a few subtleties at play in the previous sentence. If an application queries the customers' table 50 times, each time specifying a different customer_id as a literal, then there will be 50 separate entries in the TKPROF report. If however, the application specifies the customer_id as a bind variable, then there will be only one entry in the report with an indication that the statement was executed 50 times. Furthermore, the report will also include SQL statements initiated by the database server itself in order to perform so-called "recursive operations" such as manage the data dictionary and dictionary cache.

The entries for each SQL statement in the TKPROF report are separated by a row of asterisks. The first part of each entry lists the SQL statement and statistics pertaining to the parsing, execution and fetching of the SQL statement. Consider the following example:

```

*****
SELECT table_name
FROM user_tables
ORDER BY table_name

call count cpu elapsed          disk query current rows
-----
Parse 1 0.01 0.02 0 0 0          0
Execute 1 0.00 0.00 0 0          0 0
Fetch 14 0.59 0.99 0 33633      0 194
-----
total 16 0.60 1.01 0 33633      0 194

Misses in library cache          during parse: 1

Optimizer goal: CHOOSE

Parsing user id: RSCHRAG          [recursive depth: 0]

```

This may not seem like a useful example because it is simply a query against a dictionary view and does not involve application tables. However, this query actually serves the purpose well from the standpoint of highlighting the elements of a TKPROF report.

Reading across, we see that while SQL trace was enabled, the application called on the database server to parse this statement once. 0.01 CPU seconds over a period of 0.02 elapsed seconds were used on the parse call, although no physical disk I/Os or even any buffer gets were required. (We can infer that all dictionary data required to parse the statement were already in the dictionary cache in the SGA.)

The next line shows that the application called on Oracle to execute the query once, with less than 0.01 seconds of CPU time and elapsed time being used on the execute call. Again, no physical disk I/Os or buffer gets were required. The fact that almost no resources were used on the execute call might seem strange, but it makes perfect sense when you consider that Oracle defers all work on most SELECT statements until the first row is fetched.

The next line indicates that the application performed 14 fetch calls, retrieving a total of 194 rows. The 14 calls used a total of 0.59 CPU seconds and 0.99 seconds of elapsed time. Although no physical disk I/Os were performed, 33,633 buffers were gotten in consistent mode (consistent gets). In other words, there were 33,633 hits in the buffer cache and no misses. I ran this query from SQL*Plus, and we can see here that SQL*Plus uses an array interface to fetch multiple rows on one fetch call. We can also see that, although no disk I/Os were necessary, it took quite a bit of processing to complete this query.

The remaining lines on the first part of the entry for this SQL statement show that there was a miss in the library cache (the SQL statement was not already in the shared pool), the CHOOSE optimizer goal was used to develop the execution plan, and the parsing was performed in the RSCHRAG schema.

Notice the text in square brackets concerning recursive depth. This did not actually appear on the report—I added it for effect. The fact that the report did not mention recursive depth for this statement indicates that it was executed at the top level. In other words, the application issued this statement directly to the database server. When recursion is involved, the TKPROF report will indicate the depth of the recursion next to the parsing user.

There are two primary ways in which recursion occurs. Data dictionary operations can cause recursive SQL operations. When a query references a schema object that is missing from the dictionary cache, a recursive query is executed in order to fetch the object definition into the dictionary cache. For example, a query from a view whose definition is not in the dictionary cache will cause a recursive query against view\$ to be parsed in the SYS schema. Also, dynamic space allocations in dictionary-managed tablespaces will cause recursive updates against uet\$ and fet\$ in the SYS schema.

Use of database triggers and stored procedures can also cause recursion. Suppose an application inserts a row into a table that has a database trigger. When the trigger fires, its statements run at a recursion depth of one. If the trigger invokes a stored procedure, the recursion depth could increase to two. This could continue through any number of levels.

So far we have been looking at the top part of the SQL statement entry in the TKPROF report. The remainder of the entry consists of a row source operation list and optionally an execution plan display. (If the explain keyword was not used when the TKPROF report was generated, then the execution plan display will be omitted.) Consider the following example, which is the rest of the entry shown above:

```

Rows Row Source Operation
-----
194 SORT ORDER BY
194 NESTED LOOPS
195 NESTED LOOPS OUTER
195 NESTED LOOPS OUTER
195 NESTED LOOPS
11146 TABLE ACCESS BY INDEX ROWID OBJ$
11146 INDEX RANGE SCAN (object id 34)
11339 TABLE ACCESS CLUSTER TAB$
12665 INDEX UNIQUE SCAN (object id 3)
33 INDEX UNIQUE SCAN (object id 33)
193 TABLE ACCESS CLUSTER SEG$
387 INDEX UNIQUE SCAN (object id 9)
194 TABLE ACCESS CLUSTER TS$
388 INDEX UNIQUE SCAN (object id 7)

Rows Execution Plan
-----
0 SELECT STATEMENT GOAL: CHOOSE
194 SORT (ORDER BY)
194 NESTED LOOPS
195 NESTED LOOPS (OUTER)
195 NESTED LOOPS (OUTER)
195 NESTED LOOPS

```

```

11146 TABLE ACCESS (BY INDEX ROWID) OF      'OBJ$'
11146 INDEX (RANGE SCAN) OF 'I_OBJ2'      (UNIQUE)
11339 TABLE ACCESS (CLUSTER) OF 'TAB$'
12665 INDEX (UNIQUE SCAN) OF 'I_OBJ#'      (NON-UNIQUE)
33 INDEX (UNIQUE SCAN) OF 'I_OBJ1' (UNIQUE)
193 TABLE ACCESS (CLUSTER) OF 'SEG$'
387 INDEX (UNIQUE SCAN) OF 'I_FILE#_BLOCK#' (NON-UNIQUE)
194 TABLE ACCESS (CLUSTER) OF 'TS$'
INDEX (UNIQUE SCAN) OF 'I_TS#' (NON-UNIQUE)

```

The row source operation listing looks very much like an execution plan. It is based on data collected from the SQL trace file and can be thought of as a "poor man's execution plan". It is close, but not complete.

The execution plan shows the same basic information you could get from the autotrace facility of SQL*Plus or by querying the plan table after an EXPLAIN PLAN statement—with one key difference. The rows column along the left side of the execution plan contains a count of how many rows of data Oracle processed at each step during the execution of the statement. This is not an estimate from the optimizer, but rather actual counts based on the contents of the SQL trace file.

Although the query in this example goes against a dictionary view and is not terribly interesting, you can see that Oracle did a lot of work to get the 194 rows in the result: 11,146 range scans were performed against the i_obj2 index, followed by 11,146 accesses on the obj\$ table. This led to 12,665 non-unique lookups on the i_obj# index, 11,339 accesses on the tab \$ table, and so on.

In situations where it is feasible to actually execute the SQL statement you wish to explain (as opposed to merely parsing it as with the EXPLAIN PLAN statement), I believe TKPROF offers the best execution plan display. GUI tools such as TOAD will give you results with much less effort, but the display you get from TOAD is not 100 percent complete and in certain situations critical information is missing. (Again, my experience is with the free version!) Meanwhile, simple plan table query scripts like my explain.sql presented earlier in this paper or utlxpls.sql display very incomplete information. TKPROF gives the most relevant detail, and the actual row counts on each operation can be very useful in diagnosing performance problems. Autotrace in SQL*Plus gives you most of the information and is easy to use, so I give it a close second place.

TKPROF Reports: More Than Just Execution Plans

The information displayed in a TKPROF report can be extremely valuable in the application tuning process. Of course the execution plan listing will give you insights into how Oracle executes the SQL statements that make up the application, and ways to potentially improve performance. However, the other elements of the TKPROF report can be helpful as well.

Looking at the repetition of SQL statements and the library cache miss statistics, you can determine if the application is making appropriate use of Oracle's shared SQL facility. Are bind variables being used, or is every query a unique statement that must be parsed from scratch?

From the counts of parse, execute and fetch calls, you can see if applications are making appropriate use of Oracle's APIs. Is the application fetching rows one at a time? Is the application reparsing the same cursor thousands of times instead of holding it open and avoiding subsequent parses? Is the application submitting large numbers of simple SQL statements instead of bulking them into PL/SQL blocks or perhaps using array binds?

Looking at the CPU and I/O statistics, you can see which statements consume the most system resources. Could some statements be tuned so as to be less CPU intensive or less I/O intensive? Would shaving just a few buffer gets off of a statement's execution plan have a big impact because the statement gets executed so frequently?

The row counts on the individual operations in an execution plan display can help identify inefficiencies. Are tables being joined in the wrong order, causing large numbers of rows to be joined and eliminated only at the very end? Are large numbers of duplicate rows being fed into sorts for uniqueness when perhaps the duplicates could have been weeded out

earlier on?

TKPROF reports may seem long and complicated, but nothing in the report is without purpose. (Well, okay, the row source operation listing sometimes isn't very useful!) You can learn volumes about how your application interacts with the database server by generating and reading a TKPROF report.

Conclusion

We have discussed how to generate execution plans and TKPROF reports, and how to interpret them. We've walked through several examples in order to clarify the techniques presented. When you have a firm understanding of how the Oracle database server executes your SQL statements and what resources are required each step of the way, you have the ability to find bottlenecks and tune your applications for peak performance. EXPLAIN PLAN and TKPROF give you the information you need for this process.

When is a full-table scan better than an index range scan? When is a nested-loops join better than a hash join? In which order should tables be joined? These are all questions without universal answers. In reality, there are many factors that contribute to determining which join method is better or which join order is optimal.

Additionally, we have reviewed the tools that give you the information you need to make tuning decisions. How to translate an execution plan or TKPROF report into an action plan to achieve better performance is not something that can be taught in one article. You will need to read several papers or books in order to give yourself some background on the subject, and then you will need to try potential solutions in a test environment and evaluate them. If you do enough application tuning, you will develop an intuition for spotting performance problems and potential solutions. This intuition comes from lots of experience, and you can't gain it solely from reading papers or books.

For more information about the EXPLAIN PLAN facility, execution plans in general and TKPROF, consult the Oracle manual entitled Oracle8i Designing and Tuning for Performance. To learn more about application tuning techniques, I suggest you pick up Richard Niemiec's tome on the subject, Oracle Performance Tuning Tips & Techniques, available from Oracle Press.

About the Author

Roger Schrag has been an Oracle DBA and application architect for more than 11 years, starting out at Oracle Corporation on the Oracle Financials development team. He is the founder of Database Specialists, Inc., a consulting group specializing in business solutions based on Oracle technology. You can visit Database Specialists on the web at

www.dbspecialists.com, and you can reach Roger by calling +1.415.344.0500 or via email at rschrag@dbspecialists.com.

[Download Acrobat Reader](#)

Copyright 2003 by the International Oracle Users Group