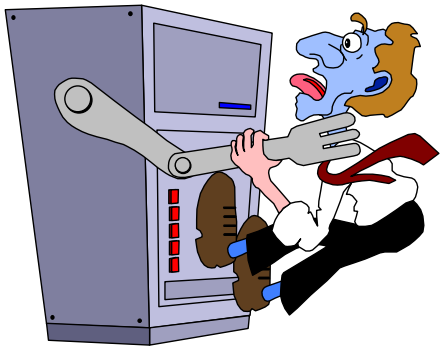
CHAPTER 11

TOPIC OBJECTIVES

- **Tuning SQL Applications**
- **Tuning Memory Allocation**
- **Tuning I/O**
- **Tuning Timing and Sorts**
- **Tuning Contention**

WHY TUNE ORACLE SYSTEMS?

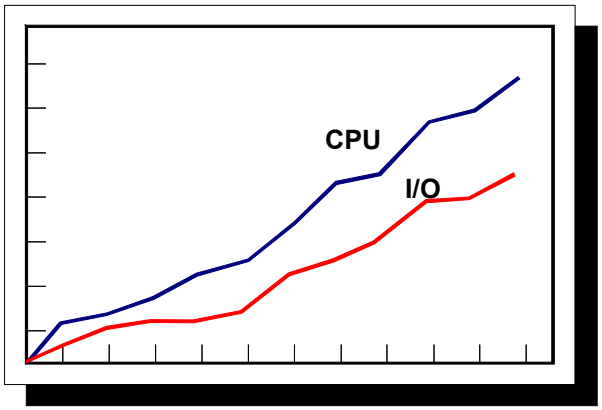


As new applications come on board, we hear constant complaining about how poorly they perform. As applications are written, you should attend to their performance and make it as important a component as the programming itself. There are many short- and long-term benefits to tuning applications:

- Well-tuned applications require less attention down the road.
- Your end users will be happier with the system's performance and its ability to process more data in less time.
- Applications that have been tuned make more efficient use of resources on your computer.

Note: There is only a finite amount of computing power on any computer. Tuning applications can be a time-consuming and frustrating exercise, but using the ideas and guidelines in this chapter, you will get the most bang for your buck.

SQL TUNING STEPS

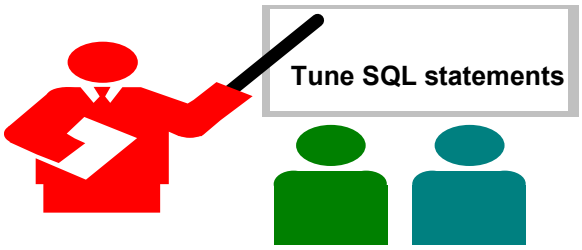


The following techniques should be applied in this order for tuning a database:

- Time SQL Statements and Applications
- Tune Memory Allocation
- Tune I/O
- Time Contention
- Tune Sorts
- Tune Free Lists
- Time Checkpoints
- Tune the Operating System

The focus of tuning should ALWAYS be at the SQL statement level in your applications.

GUIDELINES FOR TUNING



Oracle's Approach to Tuning

For anyone who has taken the Performance Tuning exam for Oracle certification, one of the testable areas dealt with Oracle's Tuning Methodology. Oracle's emphasis on this particular methodology changed when Oracle9i was released. The approach has gone from top-down in 9i to that of following principles in 11g. Neither methodology is absolute as each has its advantages and disadvantages.

Priority	Description
First	Define the problem clearly and then formulate a tuning goal.
Second	Examine the host system and gather Oracle statistics.
Third	Compare the identified problem to the common performance problems identified by Oracle in the Oracle11g Database Performance Methods (Release 1)/Database Performance Planning (Release 2)
Fourth	Use the statistics gathered in the second step to get a conceptual picture of what might be happening on the system.
Fifth	Identify the changes to be made and then implement those changes.
Sixth	Determine whether the objectives identified in step one have been met. If they have, stop tuning. If not, repeat steps five and six until the tuning goal is met.

ORACLE TUNING GUIDE

The performance tuning guide for Oracle11g (Release 2) identifies the overall process as *The Oracle Performance Improvement Method*. The steps have been expanded, but overall, remain the same.

1. Perform the following initial standard checks:

a. Get candid feedback from users. Determine the performance project's scope and subsequent performance goals, as well as performance goals for the future. This process is key in future capacity planning.

b. Get a full set of operating system, database, and application statistics from the system when the performance is both good and bad. If these are not available, then get whatever is available. Missing statistics are analogous to missing evidence at a crime scene: They make detectives work harder and it is more time-consuming.

c. Sanity-check the operating systems of all machines involved with user performance. By sanity-checking the operating system, you look for hardware or operating system resources that are fully utilized. List any over-used resources as symptoms for analysis later. In addition, check that all hardware shows no errors or diagnostics.

2. Check for the top ten most common mistakes with Oracle, and determine if any of these are likely to be the problem. List these as symptoms for later analysis. These are included because they represent the most likely problems. ADDM automatically detects and reports nine of these top ten issues.

3. Build a conceptual model of what is happening on the system using the symptoms as clues to understand what caused the performance problems.

4. Propose a series of remedy actions and the anticipated behavior to the system, then apply them in the order that can benefit the application the most. ADDM produces recommendations each with an expected benefit. A golden rule in performance work is that you only change one thing at a time and then measure the differences. Unfortunately, system downtime requirements might prohibit such a rigorous investigation method. If multiple changes are applied at the same time, then try to ensure that they are isolated so that the effects of each change can be independently validated.

5. Validate that the changes made have had the desired effect, and see if the user's perception of performance has improved. Otherwise, look for more bottlenecks, and continue refining the conceptual model until your understanding of the application becomes more accurate.

6. Repeat the last three steps until performance goals are met or become impossible due to other constraints.

The Change is Part of the Problem

The change from a top-down structured approach to a principle-based "make it stop hurting" one is part of the problem. Gathering statistics is obviously important because how else do you know if you have improved (or worsened) the problem? Still, to some degree with either approach, you are left with the original two questions: what do I look for, and how do I make it better? If the structured approach left you scratching your head, the principled approach only adds to the confusion.

What would help the novice tuner is a list of items or areas to evaluate (configure, diagnose, and tune) in each of the following areas:

- Tuning the Buffer Cache
- Tuning the Redo Log Buffer
- Tuning the Shared Pool Memory
- Tuning the Program Global Area
- Optimizing Data Storage
- Optimizing Tablespaces
- Tuning Undo Segments
- Detecting Lock Contention
- Tuning SQL

These areas pretty much cover the Oracle RDBMS and instance from top to bottom. The remainder of this section will focus on tuning SQL, or more precisely, preventing slow SQL execution. Aren't these the same thing? Mostly yes, but a common approach in development is making a statement perform well enough or fast enough. Each and every statement does not have to be optimal, but some thought has to go into coding them. You do not have the time to optimize hundreds or even thousands of SQL statements, but at the same time, there are guidelines you can follow to avoid common mistakes and bad coding.

17 Tips for Avoiding Problematic Queries

The source of these 17 tips is from Oracle11g Performance Tuning: Optimizing Database Productivity. These tips provide a solid foundation for two outcomes: making a SQL statement perform better, and determining that nothing else can be done in this regard (i.e., you have done all you can with the SQL statement, time to move on to another area).

The 17 tips are listed below.

1. Avoid Cartesian products
2. Avoid full table scans on large tables
3. Use SQL standards and conventions to reduce parsing
4. Lack of indexes on columns contained in the WHERE clause
5. Avoid joining too many tables
6. Monitor V$SESSION_LONGOPS to detect long running operations
7. Use hints as appropriate
8. Use the SHARED_CURSOR parameter
9. Avoid unnecessary sorting
10. Monitor index browning (due to deletions; rebuild as necessary)
11. Use compound indexes with care (Do not repeat columns)
12. Monitor query statistics
13. Use different tablespaces for tables and indexes (reduce I/O contention)
14. Use table partitioning (and local indexes) when appropriate
15. Use literals in the WHERE clause (use bind variables)
16. Keep statistics up to date

That is quite a list and overall is thorough and accurate. Step 13 should be changed to something along the lines of "reduce I/O contention" instead of its currently stated "separate index and table tablespaces" guidance.

There are several relatively easy steps you can take to improve performance. From the user's perspective, one of the most frequently used interfaces with a database involves SQL statements, so getting a handle on them is a good place to start in terms of being able to see an immediate improvement.

Using Bind Variables

On any number of DBA help type of Web sites, a frequently seen bit of advice is to use bind variables, but rarely are the steps or instructions for this step included. Here is a simple way to create and use a bind variable.

```
SQL> variable department_id number
SQL> begin
  2  :department_id := 80;
  3  end;
  4  /

PL/SQL procedure successfully completed.

SQL> print department_id


DEPARTMENT_ID
-------------
           80
```

Now let's make a comparison between querying for employee ID and name with and without the bind variable (with the output turned off using traceonly).

```
SQL> set autotrace traceonly
SQL> select employee_id, first_name, last_name
  2  from employees
  3  where department_id = 80;

34 rows selected.


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=34 Bytes=74
          8)

   1    0   TABLE ACCESS (FULL) OF 'EMPLOYEES' (TABLE) (Cost=3 Card=34
           Bytes=748)




Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         10  consistent gets
          0  physical reads
          0  redo size
       1527  bytes sent via SQL*Net to client
        530  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         34  rows processed
```

USING BIND VARIABLES.

```
SQL> select employee_id, first_name, last_name
  2  from employees
  3  where department_id = :department_id;

34 rows selected.


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=10 Bytes=22
          0)

   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES' (TABLE) (Cost
          =2 Card=10 Bytes=220)

   2    1     INDEX (RANGE SCAN) OF 'EMP_DEPARTMENT_IX' (INDEX) (Cost=
          1 Card=10)




Statistics
----------------------------------------------------------
          8  recursive calls
          0  db block gets
         12  consistent gets
          0  physical reads
          0  redo size
       1527  bytes sent via SQL*Net to client
        530  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         34  rows processed
```

Okay, so the difference isn't that great (the cost went from 3 to 2), but this was a small example (the table only has 107 rows). Is there much of a difference when working with a larger table? Use the SH schema and its SALES table with its 900,000+ rows.

```
SQL> select prod_id, count(prod_id)
  2  from sales
  3  where prod_id > 130
  4  group by prod_id;
```

```
Execution Plan
----------------------------------------------------------
0  SELECT STATEMENT Optimizer=ALL_ROWS (Cost=540 Card=174585 Bytes=698340)
1  0   SORT (GROUP BY) (Cost=540 Card=174585 Bytes=698340)
2  1     PARTITION RANGE (ALL) (Cost=29 Card=174585 Bytes=698340)
3  2       BITMAP CONVERSION (COUNT) (Cost=29 Card=174585 Bytes=698340)
4  3         BITMAP INDEX (FAST FULL SCAN) OF 'SALES_PROD_BIX' (INDEX (BITMAP))
```

QUERY WITH BIND VARIABLES

Same query, but this time using a bind variable.

```
SQL> variable prod_id number
SQL> begin
  2  :prod_id := 130;
  3  end;
  4  /

PL/SQL procedure successfully completed.

SQL> print prod_id


   PROD_ID
----------
       130

SQL> select prod_id, count(prod_id)
  2  from sales
  3  where prod id > :prod id
  4  group by prod_id;
```

```
Execution Plan
----------------------------------------------------------
 0  SELECT STATEMENT Optimizer=ALL_ROWS (Cost=33 Card=72 Bytes=288)
 1  0   SORT (GROUP BY) (Cost=33 Card=72 Bytes=288)
 2  1     PARTITION RANGE (ALL) (Cost=29 Card=45942 Bytes=183768)
 3  2       BITMAP CONVERSION (COUNT) (Cost=29 Card=45942 Bytes=183768)
 4  3         BITMAP INDEX (FAST FULL SCAN) OF 'SALES_PROD_BIX' (INDEX (BITMAP))
```

The cost went from 540 to 33, and that is fairly significant. One of the main benefits is that the query using the bind variable, that is, the work done parsing the query, stays the same each and every time. All you have to do is substitute a new value for the variable.

Overview of the Autotrace Report

In SQL*Plus you can automatically get a report on the execution path used by the SQL optimizer and the statement execution statistics. The report is generated after a successful SQL DML statement, such as SELECT, DELETE, UPDATE or INSERT. It is useful for monitoring and tuning the performance of these DML statements.

Configuring the Autotrace Report

You can control the report by setting the AUTOTRACE system variable. See Table 11-1.

Table 11-1 Autotrace Settings

Autotrace Setting	Result
SET AUTOTRACE OFF	No AUTOTRACE report is generated. This is the default.
SET AUTOTRACE ON EXPLAIN	The AUTOTRACE report shows only the optimizer execution path.
SET AUTOTRACE ON STATISTICS	The AUTOTRACE report shows only the SQL statement execution statistics.
SET AUTOTRACE ON	The AUTOTRACE report includes both the optimizer execution path and the SQL statement execution statistics.
SET AUTOTRACE TRACEONLY	Similar to SET AUTOTRACE ON, but suppresses the printing of the user's query output, if any. If STATISTICS is enabled, query data is still fetched, but not printed.

Setups Required for the Autotrace Report

To use this feature efficiently the SELECT_CATALOG_ROLE must be granted to the user, such as HR. DBA privileges are required to grant the select_catalog_role.

Database Statistics for SQL Statements

The statistics are recorded by the server when your statement executes and indicate the system resources required to execute your statement. The results include the statistics below:

Table 11-2 Database Statistics

Database Statistic Name	Description
recursive calls	Number of recursive calls generated at both the user and system level. Oracle maintains tables used for internal processing. When Oracle needs to make a change to these tables, it internally generates an internal SQL statement, which in turn generates a recursive call.
db block gets	Number of times a CURRENT block was requested.
consistent gets	Number of times a consistent read was requested for a block.
physical reads	Total number of data blocks read from disk. This number equals the value of "physical reads direct" plus all reads into buffer cache.
redo size	Total amount of redo generated in bytes.
bytes sent via SQL*Net to client	Total number of bytes sent to the client from the foreground processes.
bytes received via SQL*Net from client	Total number of bytes received from the client over Oracle Net.
SQL*Net roundtrips to/from client	Total number of Oracle Net messages sent to and received from the client.
sorts (memory)	Number of sort operations that were performed completely in memory and did not require any disk writes.
sorts (disk)	Number of sort operations that required at least one disk write.
rows processed	Number of rows processed during the operation.

Tracing Statements for Performance Statistics and Query Execution Path

If the SQL buffer contains the following statement:

```
SELECT E.LAST_NAME, E.SALARY, J.JOB_TITLE
  FROM EMPLOYEES E, JOBS J
    WHERE E.JOB_ID=J.JOB_ID AND E.SALARY>12000;
```

The statement can be automatically traced when it is run with the following:

```
SET AUTOTRACE ON
/
```

The output is similar to the following:

```
LAST_NAME                      SALARY JOB_TITLE
-------------------------- ---------- -----------------------------------
King                            24000 President
Kochhar                         17000 Administration Vice President
De Haan                         17000 Administration Vice President
Russell                         14000 Sales Manager
Partners                        13500 Sales Manager
Hartstein                       13000 Marketing Manager

6 rows selected.

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES'
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (FULL) OF 'JOBS'
   4    2       INDEX (RANGE SCAN) OF 'EMP_JOB_IX' (NON-UNIQUE)

Statistics
----------------------------------------------------------
          0  recursive calls
          2  db block gets
         34  consistent gets
          0  physical reads
          0  redo size
        …….
          0  sorts (memory)
          0  sorts (disk)
          6  rows processed
```

Tracing Statements Without Displaying Query Data

To trace the same statement without displaying the query data, enter the following:

```
SET AUTOTRACE TRACEONLY
```

This option is useful when you are tuning a large query, but do not want to see the query output.

Monitoring Disk Reads and Buffer Gets

To monitor disk reads and buffer gets, execute the following command:

```
SET AUTOTRACE ON TRACEONLY STATISTICS
```

Example 11-4 Monitoring Disk Reads and Buffer Gets

```
Statistics
----------------------------------------------------------
        70 recursive calls
         0 db block gets
       591 consistent gets
       404 physical reads
         0 redo size
       315 bytes sent via SQL*Net to client
       850 bytes received via SQL*Net from client
         3 SQL*Net roundtrips to/from client
         3 sorts (memory)
         0 sorts (disk)
         0 rows processed
```

If `consistent gets` or `physical reads` is high relative to the amount of data returned, it indicates that the query is expensive and needs to be reviewed for optimization. For example, if you are expecting less than 1,000 rows back and `consistent gets` is 1,000,000 and `physical reads` is 10,000, further optimization is needed.

SYSTEM Variables Influencing SQL*Plus Performance

The following variables can influence SQL*Plus performance.

SET ARRAYSIZE

Sets the number of rows, called a batch, that SQL*Plus will fetch from the database at one time. Valid values are 1 to 5000. A large value increases the efficiency of queries and subqueries that fetch many rows, but requires more memory. Values over approximately 100 provide little added performance. `ARRAYSIZE` has no effect on the results of SQL*Plus operations other than increasing efficiency.

SET DEFINE OFF

Controls whether SQL*Plus parses scripts for substitution variables. If `DEFINE` is `OFF`, SQL*Plus does not parse scripts for substitution variables. If your script does not use substitution variables, setting `DEFINE OFF` may result in some performance gains.

SET FLUSH OFF

Controls when output is sent to the user's display device. `OFF` allows the host operating system to buffer output which may improve performance by reducing the amount of program input and output.

Use `OFF` only when you run a script that does not require user interaction and whose output you do not need to see until the script finishes running.

`SET FLUSH` is not supported in *i*SQL*Plus.

SET SERVEROUTPUT

Controls whether SQL*Plus checks for and displays DBMS output. If `SERVEROUTPUT` is `OFF`, SQL*Plus does not check for DBMS output and does not display output after applicable SQL or PL/SQL statements. Suppressing this output checking and display may result in performance gains.

BARRIERS TO SQL TUNING

The tuning of Oracle SQL is one of the most time consuming, frustrating, and annoying areas of Oracle tuning. There are several factors that make SQL tuning a maddening undertaking.

- Locating the offensive SQL statement - The developer must find the location of the SQL source code in order to make your tuning changes permanent. As we know, SQL source code can exist in a variety of locations, including PL/SQL, C programs, and client-side VB code.

- Resistance from management - SQL tuning is a time-consuming and expensive process, and it is not uncommon for managers to be reluctant to invest in the time required to tune the SQL.

- Tuning with ad hoc SQL generators - Products such as the SAP application dynamically create the SQL inside the SAP ABAP programs, and it is often impossible to modify the SQL source code

- Resistance from SQL programmers - Many programmers are reluctant to admit that they have created a suboptimal SQL statement.

- Tuning nonreusable SQL statements - Many third-party applications generate SQL statements with embedded literal values or array's which are much too large for the data to be processed

- Diminishing marginal returns - As SQL statements are identified and tuned, it becomes harder to locate SQL statements for tuning.

THE PROCESS OF SQL TUNING

Once all the prerequisite tuning has been done to the server, network, disk, instance and objects, the process of SQL tuning can begin. Here is a high level look at the steps of SQL tuning:

- Locate high-use SQL statements - The first step in SQL tuning is locating the frequently executed SQL. This involves using STATSPACK or fishing through the library cache.

- Tune the SQL statement - The tuning of a SQL statement involves generating an execution plan and evaluating alternative execution plans by:

    - Adding Indexes - You can add indexes (especially bitmapped and function-based indexes) to remove unwarranted full-table scans.
    - Changing the optimizer mode - You can try changing the optimizer to rule, all_rows, or first_rows.
    - Adding hints - You can add hints to force a change to the execution plan.

- Make the tuning permanent - Once you have tuned the SQL statement, you must make the change permanent by locating and changing the SQL source or by using optimizer plan stability.

GOALS OF SQL TUNING

Oracle SQL tuning is a complex subject and we will begin with a high-level statement of goals of SQL tuning and get into detail in the following chapters. The goals of SQL tuning are simple:

- Remove unnecessary large-table full-table scans – A unnecessary full-table scans cause a huge amount of unnecessary I/O and can grad down an entire database. The first step is to evaluate the SQL in terms of the number of rows returned. If the query returns less than 40 percent of the tables rows in an ordered table, the query can be tuned to use an index.
- Cache small-table full-table scans - In cases where a full tab le scan is the fastest access method, ensure that a dedicated data buffer is available for the rows. Utilize the KEEP command for the buffer pool.
- Verify optimal index usage - This improves the speed of queries. Oracle sometimes has a choice of indexes, and the tuning of indexes ensures that Oracle is using the proper index.
- Verify optimal join techniques. - Some queries will perform faster with NESTED LOOP joins, others with HASH joins.

These goals may seem deceptively simple, but these tasks make up 90 percent of SQL tuning.

REPORTING ON SQL FROM THE LIBRARY CACHE

Some simple techniques to improve performance by looking at the access.sql or cnumblck.sql script:

- Identifying high-use tables and indexes - See what tables the database accesses the most frequently.

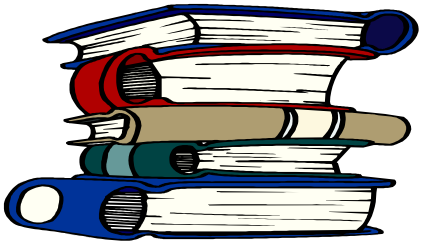
- Identifying tables for caching - You can quickly find small, frequently accessed table for placement in the KEEP pool. I automatically cache tables with 200 blocks when a table has experienced more than 100 full-table scans.

- Identifying tables for row sequencing - You can locate large tables that have frequent index-range scans in order to resequence the rows to reduce I/O.

- Dropping unused indexes - You can reclaim space occupied by unused indexes. Typical databases use no more than a quarter of the indexes available or don’t use them as intended.

- Stopping full-table scans by adding new indexes - Quickly find the full-table scans that you speed up by adding a new index to a table.

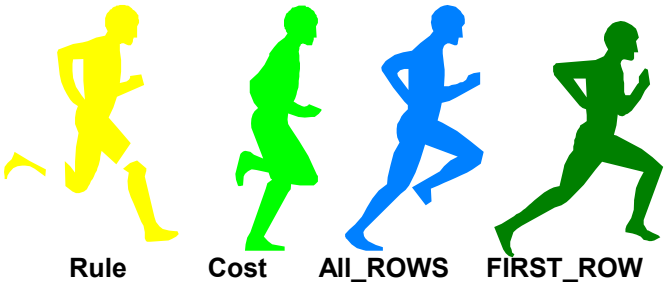
TERMINOLOGY



The following definitions will assist in defining clearly the terminology used in this chapter:

- Disk access is the act of reading information from the database files on disk
- A wait situation arises when a user process is “standing by” while resources or data it requires are being tied up by other user processes.
- An execution plan is the “map” that Oracle builds to get at the data that satisfies a user’s query. The plan is built using statistics about the data that reside in the data dictionary.
- The operation that reads data from and writes data to disk is called I/O (input/output)
- Oracle contains two optimizers, Rule-based and Cost-based. The optimizers create the execution plan.

THE OPTIMIZER



Throughput ----------Response ti

The ORACLE optimizer determines the best way to retrieve the rows based on a particular SQL Statement. It is important to understand how the optimizer works so that you can influence its decisions.

The optimizer can be set using the following commands:

COST: ALTER SESSION SET OPTIMIZER_MODE= ALL_ROWS;
ALTER SESSION SET OPTIMIZER_MODE=FIRST_ROWS;
ALTER SESSION SET OPTIMIZER_MODE=FIRST_ROWS_n;

Statistics are scheduled to be run nightly and on weekends automatically.

Note: The alter session statement sets the optimizer_mode setting for only this session. Then it returns to the database default.

ORACLE OPTIMIZERS

Optimization refers to choosing the most efficient way to execute a SQL statement.

An Oracle optimizer is an application that determines the access plan for a SQL statement. The optimizer evaluates the queries for performance because some queries will perform faster with NESTED LOOP joins, some with HASH joins, while others favor sort-merge joins. It is difficult to predict what join technique will be fastest *a priori*, so many Oracle tuning experts will test-ruin the SQL with each different table join method.

These goals may seem deceptively simple, but these tasks comprise 90 percent of SQL tuning, and they do not require a thorough understanding of the internals of Oracle SQL. Let's begin with an overview of the Oracle SQL optimizers.

Of course, you can tune the SQL all you want, but if you do not feed the optimizer with the correct statistics, the optimizer may not make the correct decisions. It is important to ensure that you have statistics present and that they are current.

Some believe in the practice of running statistics by schedule such as weekly, some believe in just calculating statistics when the data changes, still others believe that you only run statistics to fix a poor access path, and once things are good; do not touch them. It is difficult to say who is correct.

Therefore, the new features in Oracle 11g that tell you when statistics are old and need to be recalculated are extremely helpful. Gone are the days when statistics were calculated weekly (or on whatever schedule), just in case the data changed. Now we know for sure one way or the other. Of course, some will still believe that you should only calculate new statistics if you are having a problem, and once you have decent access paths, leave it alone.

The cost-based optimizer uses three criteria to determine an access path:

- Statistics on tables and indexes
- Version of Oracle
- Initialization parameters - DB_FILE_MULTIBLOCK_READ_COUNT

PROVIDING HINTS FOR THE OPTIMIZER

In general you should use the cost-based optimization (CBO) approach for all new applications. The cost-based approach generally chooses an execution plan that is as good or better than the plan chosen by the rule-based approach, especially for large queries or multiple joins.

Besides using the ALTER SESSION command to set your optimization plan you can also provide hints within your sql statements.

```
ORACLE>  select   /*+  FIRST_ROWS   */  ename, sal
                  from emp     --  Cost based optimizer
                  where deptno = 10;

ORACLE>  select   /*+  use_nl   */ ename, sal,dname,loc
                  from emp,dept     --  use nested loop
                  where deptno > 10;
```

Note: You can set the optimizer for all queries by initializing your INIT.ORA file by putting in the command optimizer_mode = ALL_ROWS

SQL HINTS

How Hints work?

Hints can be used with SELECT, INSERT, UPDATE, and DELETE statements to override the database or session defaults. Hints are defined in comments with the SQL statements.

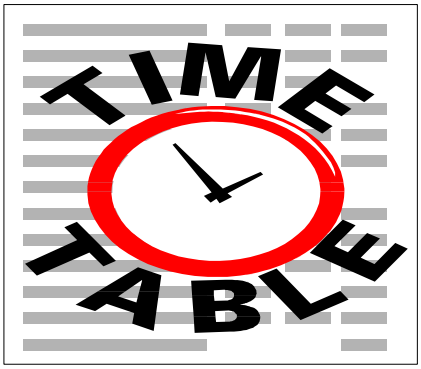
- Try to avoid using hints on views or subqueries. Unexpected plans can be generated when hints are used with views or subqueries.
- The Performance Tuning Reference Manual has information about how to used hints with views and subqueries.

The following table lists some of the more common hints that can be used with SQL statements.

RULE	CHOOSE	FIRST_ROWS
ALL_ROWS	FULL	ROWID
CLUSTER	HASH	HASH_AJ
HASH_SJ	INDEX	INDEX_ASC
INDEX_COMBINE	INDEX_JOIN	INDEX_DESC
INDEX_FFS	NO_INDEX	MERGE_AJ
MERGE_SJ	AND_EQUAL	USE_CONCAT
NO_EXPAND	REWRITE	NOREWRITE

SQL STATEMENT PROCESSING

PARSING



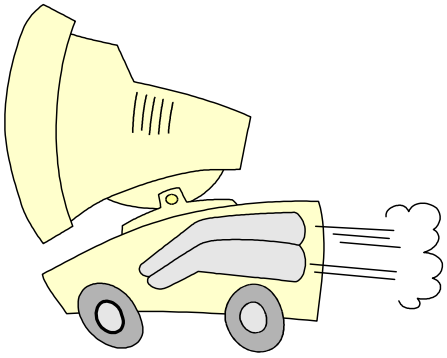
Time and cost intensive

All SQL statements are processed in three phases – parse, execute and fetch, regardless of the tool (e.g. Oracle Forms, Oracle InterOffice) that passed it to Oracle for processing.

Parse is the most time-consuming of the three phases, and the most costly. Tuning sql statements to avoid parsing is natural part of improving performance. Over 75 percent of application tuning can be realized by avoiding the parse phase altogether. By using ready-parsed statements already in the shared pool, you are more than three-quarters of the way there.

You may have to work closely with your database administrator (DBA) to ensure there is adequate space in the shared pool to accommodate an optimal number of ready-parsed SQL statement.

EXECUTE AND FETCH STATEMENTS



EXECUTE THE PLAN AND RETRIEVE THE DATA

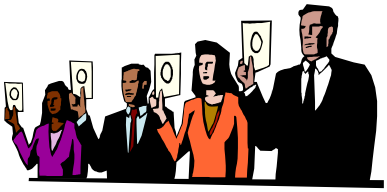
The reads and writes required to process the statement are performed during the execute phase. Oracle now knows how it will get at the data (based on the execution plan determined during the parse phase), and it is armed with all the information necessary to fetch the data that qualifies based on the selection criteria in the SQL statement.

Locks are obtained, as required, if the SQL contains any update or delete operations.

FETCH retrieves all rows that qualified based upon the plan. If the query requires sorting, this is done now. The results are formatted and displayed according to the query's instructions.

NOTE: The golden rule about processing SQL statements is:
PARSE ONCE, EXECUTE MANY TIMES. By reusing parsed statements in the shared pool, you have made the biggest step toward application tuning.

PLAN STABILITY



SQL statements can change paths

SQL statement execution plans can change due to modifications to:

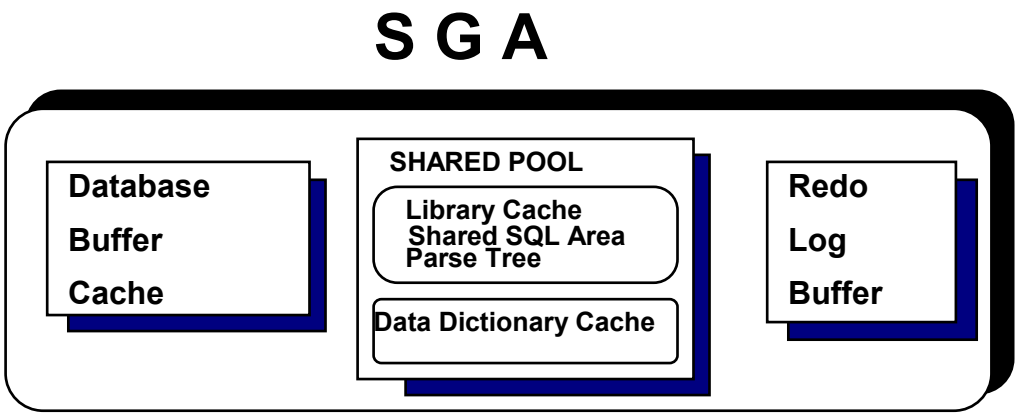
- Optimizer mode settings
- Initialization parameters (DB_FILE_MULTIBLOCK_READ_COUNT)
- Data block storage parameters
- Table Reorganizations
- Schema changes (adding indexes)
- Timeliness of statistics

Plan stability is made possible by keeping execution plans in stored outlines. For a stored outline to work, the SQL statements executed must match the stored SQL statements exactly. The stored outlines rely on hints in the SQL statements to set the execution plan and will remain stable even in the database environment. To create a category (documentation area) otherwise the DEFAULT category is used.
SQL> ALTER SYSTEM SET CREATE_STORED_OUTLINES = MY_STAFF

To create a stored outline:

```
SQL>  CREATE OR REPLACE OUTLINE my_staff_outline
          FOR CATEGORY MY_STAFF
ON
SELECT /*+CLUSTER */
      DEPTNUMB, ID, NAME, SALARY, LOCATION
FROM STAFF S, ORG O
WHERE  DEPTNUMB = DEPT
/
outline created.
```

OPTIMIZING SQL STATEMENTS

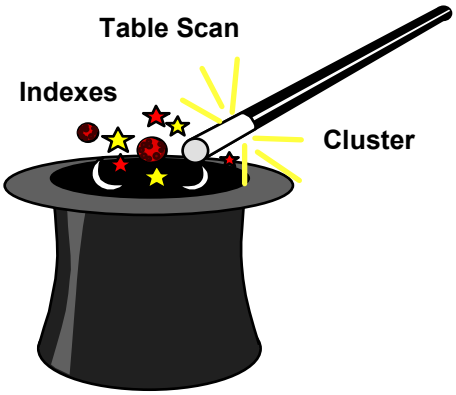


To tune the shared pool in the SGA, modify the INIT.ORA parameter ***shared_pool_size***. The default is 3.5 MB. In order to have a large enough set of rows to look at, review the big.sql or big1.sql script. This will create a table and insert 100,000 rows.

To see what statements have had “executions” (retrieved multiple sets of rows) of more than 4000 and to see the actual statement that executed you can write the following query. See vsqlarea.sql

```
select command_type, substr(sql_text,1,50)  TEXT,
        executions
        from V$SQLAREA
        where executions > 4000
        order by 3;
```

RETRIEVAL METHODS

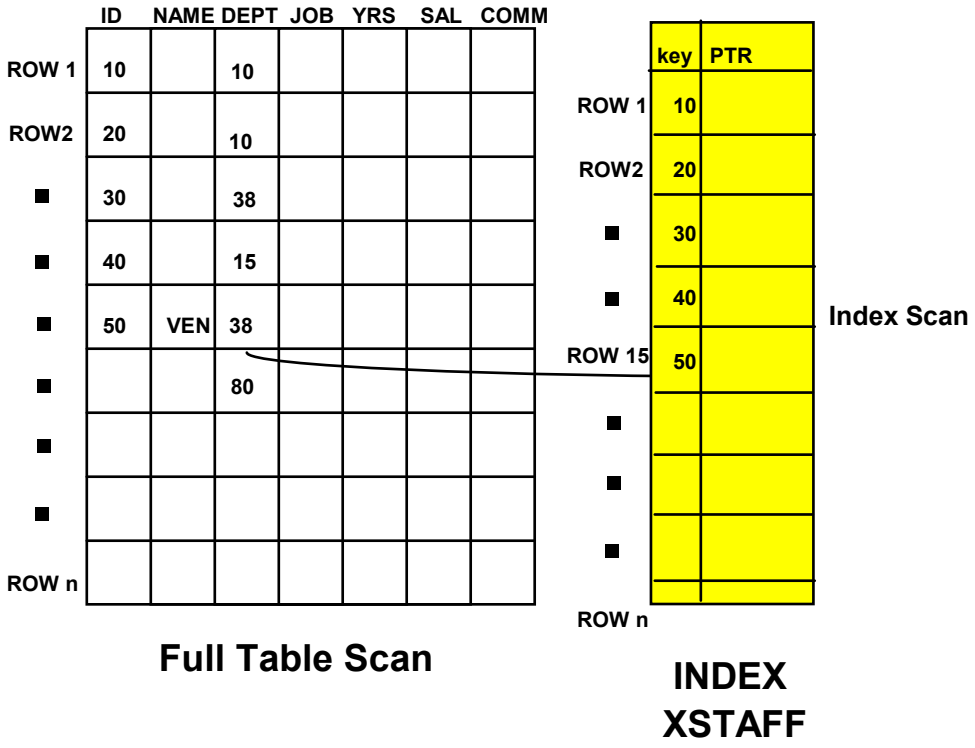


Oracle uses the following methods to retrieve data:

- Row ID
- Index
- Hash Index
- Cluster Index
- Full Table Scan

For large tables, an index search generally improves performance over full table scans. If more than 10-20% of a table will be selected, then a full table scan might be preferable.

TABLE SCANS VERSUS INDEXES



When performing a full table scan, ORACLE reads all of the blocks in the table, and then reads each row in the block once. Multi-block sequential reads may be performed to increase buffering. The number of blocks read is dependent on the INIT.ORA parameter, DB_FILE_MULTIBLOCK_READ_COUNT.

To perform an index search, ORACLE retrieves the rows in index order and then reads the block, regardless of which block contains the rows. Thus, index searches may read a block more than once. Also, since ORACLE uses the index search order, ORACLE may not perform multi-block reads since the following blocks are not known until the index path is traversed. This makes an index search inefficient for any query where a large number of rows is being brought back.

FULL TABLE SCAN OPERATIONS

Bring all rows into the SGA

Full table cans (FTSs) are usually performance killers. If unintentional, they cause too many reads and contribute to other I/O problems. In V$FILESTAT or the statspack report file I/O section, look for many I/O on one tablespace or datafile; this is usually an indicator that you have one or more untuned queries that perform FTSs.

Query V$SYSTAT to determine how many FTSs have taken place in the database since startup.

```
SQL>  Select name, value
        From V$SYSSTAT
        Where NAME LIKE '%table scan%'
```

NAME	VALUE
table scans (short tables)	208
table scans (long tables)	125
……..	
table scan rows gotten	646349
table scan blocks gotten	53034

The values for table scans (long tables) and (short tables) are what we're interested in. Long table scans are the total number of FTSs performed on the tables with more than 5 db_blocks. Indexes should be used on long tables if more than 0 to 20 percent of rows from the table are returned. Short table scans are the number of FTSs performed on tables with less than 5 db_blocks. FTSs are appropriate for these kinds of tables.

USING SQL EFFICIENTLY

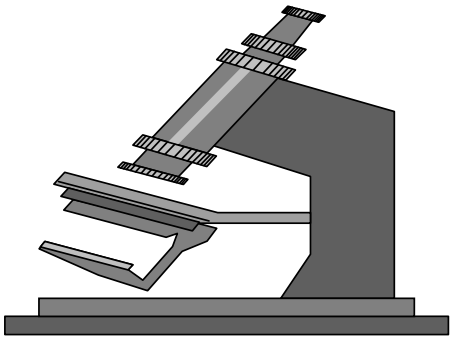
OPTIMIZER CHOICES

To determine whether to use a full table scan or an index, the optimizer closely examines each statement and the available index. An execution plan is adopted at Parse time based on the following criteria:

- Cost-based or rule-based
- Hints
- SQL Syntax and structure
- Where clause predicates or conditions
- Structures and definitions of the database objects in the SQL statement
- Indexes which are available on the database objects

NOTE: The optimizer examines statistics on both table data and index data to establish a ranking rule. Design your applications to take advantage of code stored in the database. Look at all our business processes and centralize common procedures.

ANALYZING TABLES AND INDEXES WITH CBO



For CBO to work, it needs information such as the number of rows in each table, the distribution keys within a table's primary key column(s), and the number of data blocks allocated to and occupied by the table's rows. CBO gets this information through the ANALYZE command.

Before analyze is run, let's take a look at the statistics currently available:

```
ORACLE>select table_name, tablespace_name,
        initial_extent, num_rows,blocks,chain_cnt,
        empty_blocks, avg_row_len,last_analyzed
from user_tables
where table_name = upper('&tablename');
/*  Now save this script as seetable  */
ORACLE> save seetable
```

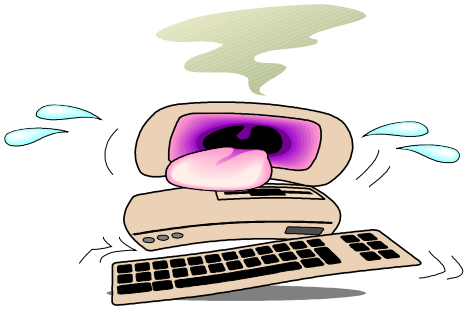
This will show that no statistics have been collected for any tables.
Now lets run the ANALYZE statement on the big table.

```
ORACLE> ANALYZE table big estimate statistics;
```

Now run the seetable script to see what statistics have been collected.

```
ORACLE> @SEETABLE
```

MORE ABOUT THE ANALYZE STATEMENT



Why is my plan so bad?

The ANALYZE statement must be run on a regular basis for any tables with volatile natures (e.g. updates, inserts, and deletes). Additionally, any new indexes created need to be analyzed as well. The analyze command has two different ways of collecting statistics. You can estimate the statistics (take a sampling) or you may compute the statistics (complete table scan). It is recommended that you use estimate for tables and compute for indexes as shown below:

```
analyze table big estimate statistics sample 20 percent;
analyze table big2 estimate statistics sample 10 percent;
analyze index big_bigno compute statistics;
```

To run analyze on a schema (all tables and indexes) use the following PACKAGE:

```
ORACLE>  EXEC DBMS_STATS.GATHER_SCHEMA_STATS
        (OWNNAME=> 'SCOTT');
```

The DBMS_STATS Package gathers more sophisticated statistics for tables, indexes and clusters. One of the added benefits of the DBMS_STATS package is that you can gather statistics in parallel on an object. Here is a brief description of the DBMS_STATS procedures:

- GATHER_TABLE_STATS – Gathers statistics for a specific table

```
Exec dbms_stats.gather_table_stats(ownname=>'SCOTT',tabname=>EMP')
```

- GATHER_INDEX_STATS – Gathers statistics for a specific index
- GATHER_SCHEMA_STATS – Gathers statistics for a specific schema
- GATHER_DATABASE_STATS – Gathers statistics for all objects in the database

COPY STATISTICS BETWEEN DATABASES

Ensure that SQL between databases use the same plan

SQL statements often exhibit unique behavior depending on the database they're run in. For example, DBA's see excellent statement performance in a test environment but then see extremely poor performance in a production environment. You can copy statistics using the DBMS_STATS package and the export/import utilities. Here are the steps:

1. First step to create a statistics table in the appropriate schema.

```
SQL>  EXEC DBMS_STATS.CREATE_STAT_TABLE(OWNNAME= 'JERRY', -
          STATTAB=> 'STAFF_STAT', -
          TBLSPACE=> 'USERS')
```

2. Export the data dictionary statistics for a specific schema to the user-defined table:

```
 SQL>  exec DBMS_STATS.EXPORT_SCHEMA_STATS(OWNNAME=> 'JERRY'
_
          Stattab=>'STAFF_STAT')
```

3. Export the statistics table from the current database, and use the import utility to import the table into the target database.

4. Import the schema statistics into the data dictionary on the target database:

```
SQL>  exec DBMS_STATS.IMPORT_SCHEMA_STATS(OWNNAME=> 'JERRY', -
        STATTAB=>'STAFF_STAT')
```

NOTE: DBA's can use the same process for database, system, table, index, and column statistics; Each has their own DBMS_STATS procedure.

ROWID CONCEPTS

ROWID

Through ROWIDs, Oracle can uniquely identify every row within the database. The ROWID for each row is stored when the data is inserted into the database. Oracle8 uses extended ROWIDs. Older versions used restricted ROWIDs. Oracle's new extended ROWIDs cater to the object side of the database. They allow Oracle to uniquely identify rows within different objects, such as partitioned and nested tables. Extended ROWIDs are made up of four pieces of information and are stored in hexadecimal format. The ROWID is displayed as one large HEX number, which can be split into four parts – A, B, C, and D.

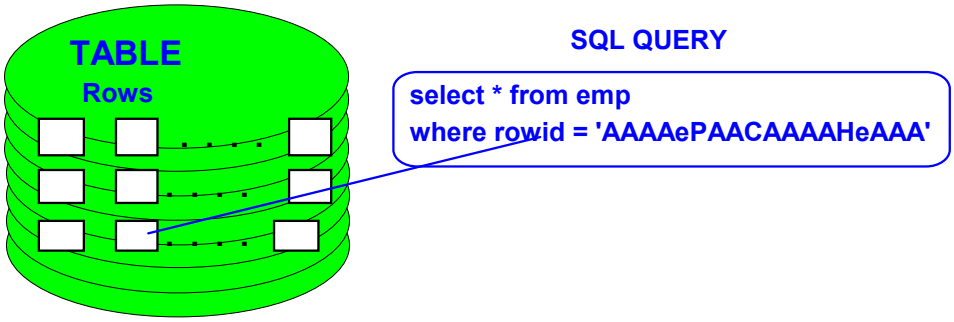
```
SELECT ROWID,
Substr(ROWID,1,6)  "OBJECT",
Substr(ROWID,7,3)  "FILE",
Substr(ROWID,10,6) "BLOCK",
Substr(ROWID,16,3)  "ROW"
from staff
```

ROWID	OBJECT	FILE	BLOCK	ROW
AAAAePAACAAAAHe.AA	AAAAeP	AAC	AAAAHe	AAA
AAAAePAACAAAAHe.AA	AAAAeP	AAC	AAAAHe	AAB
AAAAePAACAAAAHeAAA	AAAAeP	AAC	AAAAHe	AAC
....	...	...	...	
...	...	...	...	
AAAAePAACAAAAHeAAA	AAAAeP	AAC	AAAAHe	AB9

The ROWID describes the row information in hexadecimal format as shown above, which consists of:

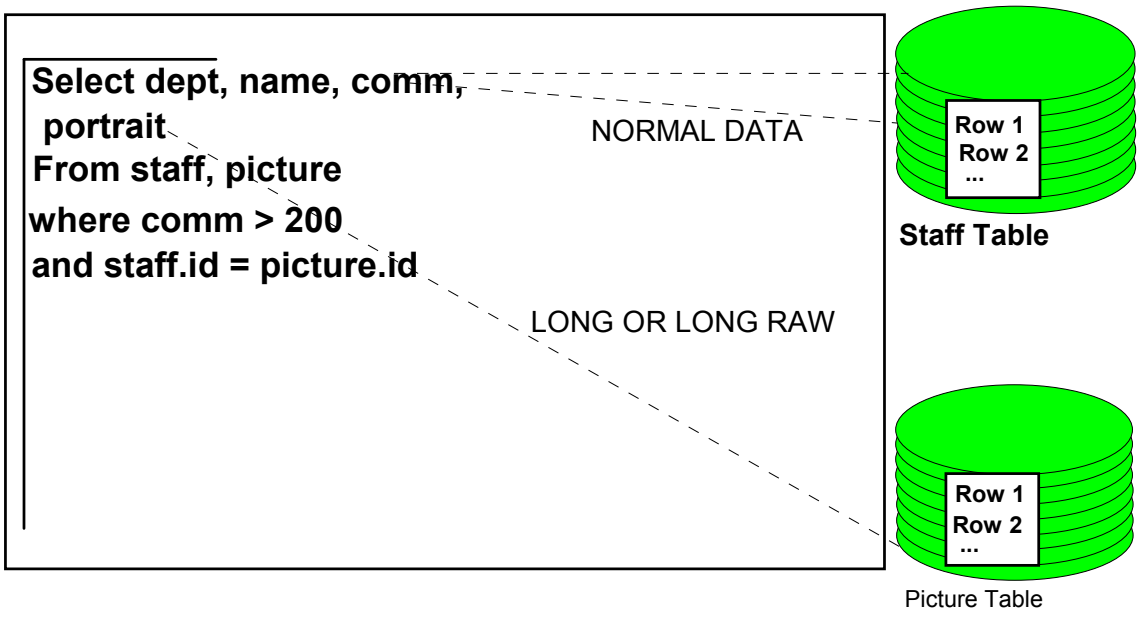
- The SEGMENT number of the object.
- The FILE number (Actual data file)
- The BLOCK within the file
- The ROW in the block

USING ROWIDS



- ROWIDs may be used in select statements
- ROWIDs can change so use with static tables
- You cannot change ROWIDs directly
- You cannot insert using ROWIDs

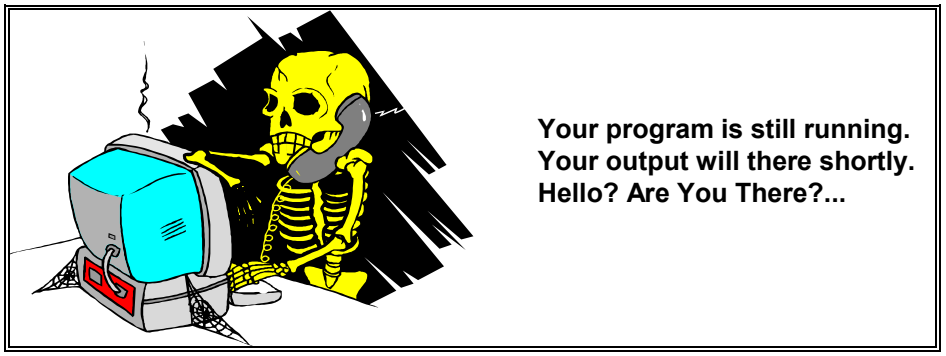
TUNING DATA TYPES



Place each LONG or LONG RAW column in a separate table from all other columns related to it. Use referential integrity constraints to join the two tables together.

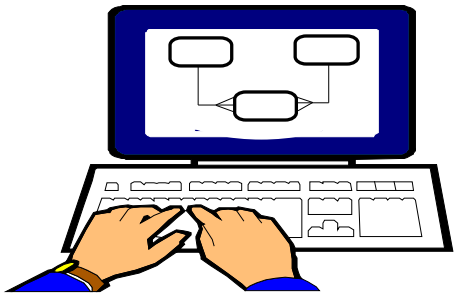
```
ORACLE>   CREATE TABLE STAFF
          (ID                 NUMBER(9),
           MWNAME             VARCHAR2(25),
           ADDRESS            VARCHAR2(60),
           CONSTRAINT         PK_STAFF_ID
                              PRIMARY KEY (ID)
                              USING INDEX
                              TABLESPACE  PK_ID_INDEX);

ORACLE>   CREATE TABLE PICTURE
          (ID                 NUMBER(9),
                              REFERENCES STAFF,
          PICTURE             LONG);
```

AVOIDING RUNTIME RECOMPILATIONS



- Avoid runtime recompilation of stored procedures, functions, and packages
- Do not redefine schema of objects while production applications are executing
- Place procedures and functions in packages whenever possible
- Use the Cost-Based Optimizer because it can select the local table or the remote table as the driving table
- Rule-based optimization always selects the remote table as the driving table, and performs a full table scan
- Cost-based queries can use indexes on remote tables and considers more execution plans
- There are usually several ways to accomplish any goal. Be aware of ORACLE's built-in functions and take advantage of them.

TUNING WITH DENORMALIZATION



- Try denormalization only after you have tried everything else
- Denormalize when you are constantly joining tables to retrieve just one column
- Denormalizing causes your queries to execute much faster, but you are exposed to the risks of:
  - Redundant data which has to be synchronized
  - Processing anomalies

For example:

```
ORACLE>   ALTER TABLE EMP
          ADD DNAME VARCHAR2(14);

ORACLE>   UPDATE EMP
          SET DNAME = ( SELECT DNAME
                        FROM DEPT WHERE
                        DEPT.DEPTNO = EMP.DEPTNO)

ORACLE>   SELECT ENAME, DNAME
          FROM EMP
          WHERE ENAME = 'FORD';
```

LAB 11

1. Write a select statement against the big table or the table the instructor identifies for you.

```
select * from big
where bigno < 5000;
```

2. Write a query against the V$SQLAREA that shows the number of executions, the number of buffer gets, the number of parse calls, and the first 20 characters of TEXT for all statements that execute more than 3000 times.

3. Determine approximately how many rows are being stored in each block of the EMP table & BIG table by writing an adhoc query using ROWID.
See tunex8.sql

4. Insert a duplicate row into the org table.

   insert into org values (10,'HEAD OFFICE',160,'CORPORATE','NEW YORK');

5. DELETE the row you just inserted using the ROWID.

6. Retrieve Lu from the staff table using ROWID.

7. Determine the approximate number of blocks in the BIG table.
(HINT: look at tunex8.sql)

8. Create statistics on the user SCOTT's tables so that they will run CBO.

8. Provide hints to the optimizer for executing SQL statements.
   1. Write a query to retrieve the id, employee name, years of service, and salary from the staff table for all employees in department 15. Use COST based hints. (SEE lab4_8.sql)

```
SELECT /*+ FIRST_ROWS */   * FROM STAFF WHERE DEPT = 15;
```

   2. Write another query on the ORD table which provides a hint for using an index called ord_index .

```
SELECT /*+ INDEX (ORD.ORD_INDEX) */ * FROM ORD
WHERE ORDID > 200;
```

PL/SQL, PRO*C AND PACKAGE PERFORMANCE TIPS

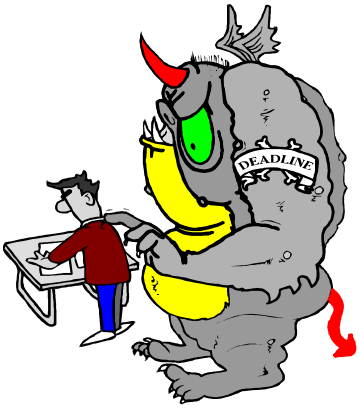
OVERLOADING PACKAGED SUBPROGRAMS

```
CREATE OR REPLACE PACKAGE ClassPackage AS
  /*  add a new student into a specified class
 PROCEDURE AddStudent(p_StudentID        IN      students.id%type,
                      p_Department       IN      students.department%type,
                           p_Course              IN      students.course%type);

/*  Add a new student by specifying first and last name rather than ID  */
PROCEDURE AddStudent(p_FirstName        IN      students.first_name%type,
                      p_LastName        IN      students.last_name%type,
                      p_Department      IN      students.department%type,
                      p_Course          IN      students.course%type);
END ClassPackage;

CREATE OR REPLACE PACKAGE BODY ClassPackage AS
PROCEDURE AddStudent(p_StudentID IN      students.id%type,
                      p_Department      IN      students.department%type,
                           p_Course              IN      students.course%type)
IS
BEGIN
INSERT INTO registered_students (studentid, department, course)
VALUES ( p_StudentID, p_Department, p_Course); Commit;
END AddStudent;
```

- Add the student by name

```
PROCEDURE AddStudent(p_FirstName        IN      students.first_name%type,
                      p_LastName        IN      students.last_name%type,
                      p_Department      IN      students.department%type,
                      p_Course          IN      students.course%type) IS
v_StudentID   students.ID%type;
/*  First we retrieve the student ID to ensure a INSERT for a NOT NULL column.
BEGIN
SELECT ID  INTO v_StudentID from students
WHERE first_name = p_FirstName and last_name = p_LastName;
INSERT INTO registered_students (studentid, department, course)
VALUES ( v_StudentID, p_Department, p_Course); Commit;
END AddStudent;
END ClassPackage;
/*  Now we can add a student to Music 410 with either:
BEGIN
   ClassPackage.AddStudent(10000,'MUS',410);  -- or
   ClassPackage.AddStudent('RICHARD','JERRY','MUS',410);
END;
```

Restrictions



LIMITATIONS

Only local or packaged subprograms can be overloaded. Standalone subprograms cannot be overloaded. Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode. For example, the following two procedures cannot be overloaded:

```
PROCEDURE send_letter (custno IN  INTEGER) IS…

PROCEDURE send_letter (custno OUT INTEGER) IS…
```

Furthermore, you cannot overload two subprograms if their formal parameters differ only in datatype and the different datatypes are in the same family. For example, you cannot overload the following procedures because the datatypes INTEGER and REAL are in the NUMBER family:

```
PROCEDURE calc_tax (amount INTEGER) IS…

PROCEDURE calc_tax (amount REAL) IS…
```

Finally, you cannot overload two functions that differ only in return type, even if the types are in different families. For example, you cannot overload the following functions:

```
FUNCTION acct_ok (acct_num INTEGER) RETURN BOOLEAN…

FUNCTION acct_ok (acct_num INTEGER) RETURN INTEGER
```

PRO*C **PL/SQl**

The Best of both worlds

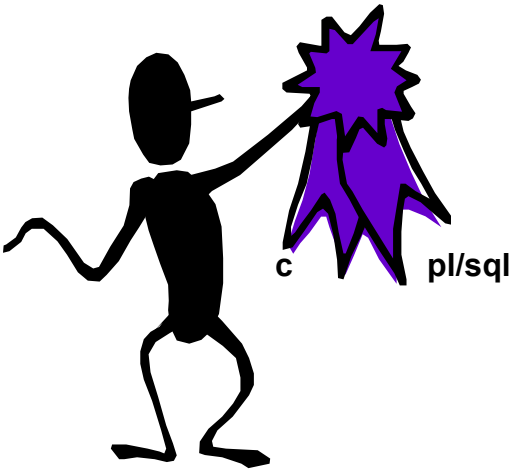
Using anonymous PL/SQL blocks in PRO*C is easy. The PL/SQL code must be enclosed within the EXEC SQL EXECUTE and END-EXEC keywords.

```
Int    age;
long int employee_code = 2431;
/* ----- PL/SQL Block ------------------------
EXEC SQL EXECUTE
BEGIN
          SELECT emp_age into :age
     where emp_code = : employee_code;
end;
END-EXEC;
/*-----------------PL/SQL BLOCKS --------------------
PRINTF("The age of the employee code 2431 is %d\n", age);
…..
…..
```

Host variables may be freely intermixed with PL/SQL code. Unfortunately, there are a few limitations. C structures cannot be used directly, and host arrays must be mapped to PL/SQL tables. Additionally because PL/SQL is best when applied with business logic on the server make sure you do not bring a large number of rows to the server using PL/SQL logic.

NOTE SQLCHECK must be used to parse PL/SQL i.e.
SQLCHECK=SEMANTICS, USERID=urid/passwd

Using Stored Procedures



Variables that are declared in the declare section are legal to use inside embedded PL/SQL blocks and embedded SQL statements. Inside of an embedded statement, the bind variables are delimited by a leading colon. For example, the following PRO*C program fragment calls the register STORED PROCEDURE.

```
EXEC SQL BEGIN DECLARE SECTION;
   /* Declare C variables */
VARCHAR v_Department[4]; /* The VARCHAR psuedo-type is
available only in PRO*C, and is converted into a record type
with two fields - .arr and .len  */
int v_Course;               /*   v_Course is an integer. */
int v_StudentID;            /*   So is v_StudentID. */

EXEC SQL END DECLARE SECTION;

/*  Initialize the host variables. */
strcpy(v_Department.arr, "ECN");
v_Department.len = 3;
v_Course = 101;
v_StudentID = 10006;
/* Begin the embedded PL/SQL block. Note the EXEC SQL EXECUTE
and END-EXEC; keywords, which delimit the block for the
precompiler */
EXEC SQL EXECUTE
BEGIN
     Register(:v_Department, :v_Course, :v_StudentID)
END;
END-EXEC;
```

PERFORMANCE MEASUREMENTS IN PL/SQL w DBMS_PROFILER

We found the statements creating the problem!!

Oracle11g provides a performeance measurement tool for Oracle applications which identifies the bottlenecks in PL/SQL applications. primarily it identifies the specific part of a PL/SQL code which takes the most time. This allows the developer and/or DBA to concentrate on improving the specific parts using the most resources.

The first item is to ensure that the DMBS-PROFILE package is created and the necessary profile tables created as well. The scripts are called PROFLOAD.sql and PROFTAB.sql. You must have the SYS user create the DBMS_PROFILE objects.

SQL> @c:\oracle\(studentx)\product\11.2.0\dbhome_1\rdbms\admin\profload.sql -- may be in another directory
Package created.
Grant succeeded.
Synonym created.
Library created.
Package body created.
Testing for correct installation
SYS.DBMS_PROFILER successfully loaded.
PL/SQL procedure successfully completed.

SQL> @c:\oracle\studentx\product\11.2.0\dbhome_1\rdbms\admin\proftab -- run as system or with developers username
Table created. -- Creates table PLSQL_PROFILER_DATA
Comment created.
Table created. -- Creates table PLSQL_PROFILER_RUNS
Comment created.
Table created. -- Creates table PLSQL_PROFILER_UNITS
Comment created.
Sequence created.

USING DBMS_PROFILER

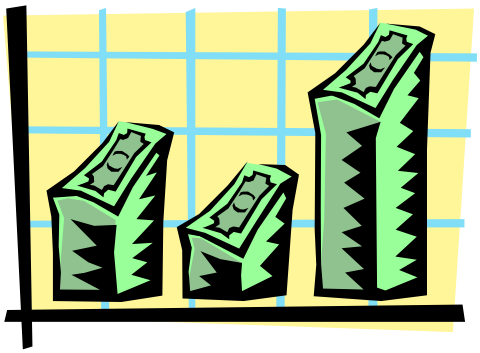
We create a tablespace called jerry and a table called big which will have 100,000 rows in it and thus provide us with some performance measurements.

```
SQL>  create tablespace jerry
      datafile c:\oracle\jerdat.dbf size 10m;

      CREATE TABLE       BIG
            (BIGNO      NUMBER(6),
            BNAME       VARCHAR2(20))
      tablespace jerry;  See cr8big.sql

      Create the procedure to load 100,000 rows.

      CREATE OR REPLACE PROCEDURE CALL_BIG
      AS
      BEGIN
            FOR I IN 1..100000 LOOP
            IF I BETWEEN 1 AND 1999 THEN
            INSERT INTO BIG VALUES(I,'ONE THOUSAND');
            ELSIF I BETWEEN 2000 AND 5000 THEN
             INSERT INTO BIG VALUES(I,'BET 2K AND 5K');
            ELSE
            INSERT INTO BIG VALUES(I,NULL);
            END IF;
            IF MOD(I,1000) = 0 THEN COMMIT;
            END IF;
            END LOOP;
      END CALL_BIG; see cr8big.sql
```

CREATING THE PROCEDURE

```
SQL> Set Serveroutput on
SQL>

SQL> DECLARE
        vrun     number;
BEGIN
   vrun := SYS.DBMS_PROFILER.start_profiler('FIRSTRUN');
    dbms_output.put_line ('START PROFILER STATUS ' || vrun);
    call_big;   -- call the procedure
    vrun := SYS.DBMS_PROFILER.stop_profiler;
    DBMS_OUTPUT.put_line ('STOP PROFILER STATUS ' || vrun);
    DBMS_OUTPUT.put_line ('0   successful ');
    DBMS_OUTPUT.put_line ('1   incorrect parameter ');
    DBMS_OUTPUT.put_line ('2   data flush operation failed ');
    DBMS_OUTPUT.put_line ('-1   version mismatch between package and tables ');
END;  see run_prof.sql
/
START PROFILER STATUS 0
STOP PROFILER STATUS 0
0   successful
1   incorrect parameter
2   data flush operation failed
-1   version mismatch between package and tables

PL/SQL procedure successfully completed.
```

IDENTIFYING THE PERFORMANCE PROBLEMS



The next step is to look at the information stored in the Profile tables about the running PL/SQL statements. This is done by executing the following query:

```
set linesize 132
COL unit format a20
COL tc format 9999999 HEADING “Exec”
COL stext format a40
SELECT u.unit_owner }} ‘.’ }} u.unit_name unit, l#,
            ROUND (d.total_time / 1000000000) time, d.total_occur tc,
            SUBSTR(s.text, 1, 40) stext
   FROM       plsql_profiler_runs r,
              plsql_profiler_units u,
              plsql_profiler_data, d,
              all_source s
  WHERE r.run_comment = ‘FIRSTRUN’
       AND  r.runid = u.runid
       AND  d.runid = u.runid
       AND  u.unit_number - d.unit_number
       AND  s.owner = u.unit_owner
       AND  s.TYPE = u.unit_type
       AND  s.NAME = u.unit_name
       AND  s.line = d.line#
ORDER BY line# -- see cprofdata.sql
Use the below time(s) value
UNIT                  L#   TIME Exec    STEXT
--------------         ---   ----    ---------   ------------------------------------------------
SYSTEM.CALL_BIG 4     41   100001 FOR I IN 1..100000 LOOP
SYSTEM.CALL_BIG 5     45   100000 IF I BETWEEN 1 AND 1999 THEN
SYSTEM.CALL_BIG 6    184  100000 INSERT INTO BIG VALUES(I,'ONE
SYSTEM.CALL_BIG 7     24    98001 ELSIF I BETWEEN 2000 AND 5000 TH
SYSTEM.CALL_BIG 8    241    98001 INSERT INTO BIG VALUES(I,'BET
SYSTEM.CALL_BIG 10 6173   95000 INSERT INTO BIG VALUES(I,NULL
SYSTEM.CALL_BIG 12   371 100000 IF MOD(I,1000) = 0 THEN COMMIT;
```

FINDING Oracle Invalid Objects:

Whenever an oracle object is marked as invalid because of a table, that has been changed, the Oracle professional can change the object to valid by using a SQL*Plus script called invalid in your ora11glabs directory.

This script will search the Oracle data dictionary for invalid objects, create a spool file with the names of the invalid objects, then invoke a script to re-compile all of these invalid objects. The following script should be used whenever a change is made to an Oracle table or index, or any other Oracle object, since the DBA must ensure that all objects are valid and executable.

We now have a supported script to perform this function of re-compiling invalid objects. The utility is called Recompile PL/SQL, or RP for short. The script is called utlrp.sql and is located in the $ORACLE_HOME/rdbms/admin/ directory. However, most experienced Oracle DBAs prefer a home-made script for this purpose.

Oracle highly recommends running a script called *invalid.sql* towards the end of any migration/upgrade/downgrade. Additionally the following show errors in packages, procedures and functions that may cause problems.

```
set heading off;
set feedback off;
set echo off;
Set lines 999;

Spool run_invalid.sql

select
'ALTER ' || OBJECT_TYPE || ' ' ||
OWNER || '.' || OBJECT_NAME || ' COMPILE;'
from
dba_objects
where
status = 'INVALID'
and
object_type in ('PACKAGE','FUNCTION','PROCEDURE')
;
spool off;
set heading on;
set feedback on;
set echo on;

@run_invalid.sql
```

Oracle invalid objects sometimes have dependencies, so it may be necessary to run the Oracle invalid objects recompile repeatedly.

ARRAY PROCESSING LAB - FAST LOADS

There are two actions that require array processing getting data to the database and get ting data from the database. Oracle 11g introduces two new array interfaces for updates and deletes.

- BULK COLLECT
- FORALL

The BULK COLLECT syntax allows you to get all the rows for a select statement from the database into local variables in a single call. In order for our example to work please create the child table and load as follows: Create the child table by running “cr8child.sql”. using the script “child.sql” load the child table, the running the following code which is found in file “childload.sql”. Then:

```
SQL> create or replace type jpl_sel_row as object (n1 number, n2 number);
SQL> create or replace type jpl_sel_tab as table of jpl_sel_row;
declare
  m_selection jpl_sel_tab;
begin
  select jpl_sel_row(n1, n2)
   BULK COLLECT into m_selection
   from child
   where n2 in (1,2,3);
for I in 1..m_selection.count loop
     dbms_output.put_line(m_selection(i).n1 || ‘ ‘ || m_selection(i).n2 );
end loop;
end;
```

To run childload from SQL*PLUS or ISQL*PLUS: SQL> @childload

BULK COPY CONTINUED

The critical point to notice here is the BULK COLLECT that appears just before the INTO clause. This directs Oracle to fetch all rows that meet the test and return them to the relevant PL/SQL variables. In the example on the previous page I used the jpl_sel_row() constructor on the base table columns. I could just as well have used PL/SQL tables.
The drawback to BULK COLLECT is that if a user accidentally hits the execute query key before they have filled in any query conditions on a FORM for example, the database will start building a huge result set based on a PL/SQL table which keeps running until all the memory is used up.
Using a VARRAY puts a limit to how much data is placed into the object.

PL/SQL FORALL

Let's get moving

The FORALL statement allows Oracle to batch all the values in a contiguous section of a list and send them to the database with the SQL code as a single request. To run the code below execute cr8test_table. For instance:

```
declare
        type num_type is table of number index by
binary_integer;
        type vc_type is table of varchar2(32) index by
binary_integer;
        num_tab  num_type;
        vc_tab        vc_type;
begin
        num_tab(1) := 2;
        vc_tab(1) := 'asdf';
        num_tab(2)  := 3;
        vc_tab(2)  := 'wret';
        forall j in 1..num_tab.count
        insert into test_table values (num_tab(j), vc_tab(j));
end;
```

Note: the insert statement is packaged and passed to the database only once, not once per set of values.

This page left blank intentionally