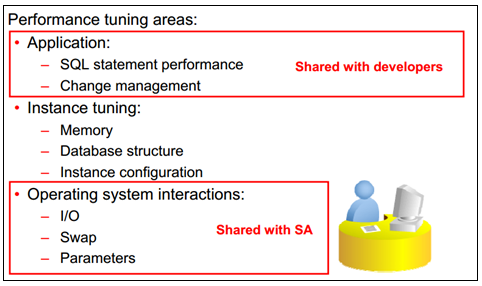
Oracle Performance Tuning 11Gr2

# Introduction

## Performace Tuning Concept:

* Monitoring and Diagnostics
  + Monitoring using available tools
  + Identifying the problem
  + Using AWR-based tools
* SQL Tuning
  + Identifying and tuning SQL statements by influencing the optimizer
  + Managing change
    - SQL Performance Management
    - Real Application Testing
* Instance Tuning
  + Tuning memory components
  + Tuning space usage and I/O
* Tuning Backup performance
* Tuning for Oracle Data Guard
* Tuning for Oracle Real Application Clusters.



## Tuning Methodology

Tuning steps:

* Identify the scope of the problem (OS, database, and so on).
* Tune the following from the top down:
  + The design before tuning the application code
  + The code before tuning the instance
* Tune the area with the greatest potential benefit:
  + Identify the performance problem (AWR, Statspack).
  + Analyze the problem, looking for skewed and tunable components.
  + Use appropriate tools to tune the components implicated.
* Stop tuning when the goal is met.

## General Tuning Session

Tuning sessions have the same procedure:

* (1) Define the problem and state the goal.
* (2) Collect current performance statistics.
* (3) Consider some common performance errors.
* (4) Build a trial solution.
* (5) Implement and measure the change.
* (6) Decide: “Did the solution meet the goal?”
  + No? Then go to step (3) and repeat.
  + Yes? Then create a new baseline.

# Database Tuning

## Optimizing Instance Memory

### Optimizing Memory

* **Automatic Memory Management**: tự động tuning để cấu hình size SGA và PGA dựa trên tổng dung lương Ram được cung cấp
* **Automatic Shared Memory Management:** Fix cứng dung lượng của SGA và PGA, từ đó chúng tự tuning cấu

**Tuning AMM**: Để xác định tổng dung lương memory cho Database:

SQL> select \* from v$memory\_target\_advice order by memory\_size;

MEMORY\_SIZE MEMORY\_SIZE\_FACTOR ESTD\_DB\_TIME ESTD\_DB\_TIME\_FACTOR VERSION

----------- ------------------ ------------ ------------------- ---------

180 .5 458 1.344 0

270 .75 367 1.0761 0

360 1 341 1 0

450 1.25 335 .9817 0

540 1.5 335 .9817 0

Bao giờ **ESTD\_DB\_TIME** và **ESTD\_DB\_TIME\_FACTOR** ổn định thì chọn **MEMORY\_SIZE** tương ứng

**Tuning ASMM**

* **Tuning SGA**

SQL> select \* from v$sga\_target\_advice order by sga\_size;

SGA\_SIZE SGA\_SIZE\_FACTOR ESTD\_DB\_TIME ESTD\_DB\_TIME\_FACTOR ESTD\_PHYSICAL\_READS

---------- --------------- ------------ ------------------- -------------------

290 .5 448176 1.6578 1636103

435 .75 339336 1.2552 1636103

580 1 270344 1 1201780

725 1.25 239038 .8842 907584

870 1.5 211517 .7824 513881

1015 1.75 201866 .7467 513881

Bao giờ **ESTD\_DB\_TIME** và **ESTD\_DB\_TIME\_FACTOR** ổn định thì chọn **SGA\_SIZE** tương ứng

* **Tuning PGA**

SQL> select \* from v$pga\_target\_advice order by sga\_size;

PGA\_SIZE SGA\_SIZE\_FACTOR ESTD\_DB\_TIME ESTD\_DB\_TIME\_FACTOR ESTD\_PHYSICAL\_READS

---------- --------------- ------------ ------------------- -------------------

290 .5 448176 1.6578 1636103

435 .75 339336 1.2552 1636103

580 1 270344 1 1201780

725 1.25 239038 .8842 907584

870 1.5 211517 .7824 513881

1015 1.75 201866 .7467 513881

1160 2 200703 .7424 513881

Bao giờ **ESTD\_DB\_TIME** và **ESTD\_DB\_TIME\_FACTOR** ổn định thì chọn **SGA\_SIZE** tương ứng

### Managing Multiple Buffer Pools

You can use multiple buffer pools instead of Oracle’s single default buffer pool, to ensure that frequently used segments stay cached in the buffer pool without being recycled out of the buffer pool.

Here’s how you create the two types of buffer pools. In the SPFILE or the init.ora file, specify the two parameters and the sizes you want to assign to each of the pools:

db\_keep\_cache\_size=1000m

db\_recycle\_cache\_size=100m

SQL> alter table employees  
storage (buffer\_pool=keep);  
Table altered.  
SQL>

### Configuring the Server Query Cache

You’d like to set up the server query cache that’s part of Oracle’s memory allocation. You can control the behavior of the server query cache by setting three initialization parameters:

* RESULT\_CACHE\_MAX\_SIZE: This sets the memory allocated to the server result cache.
* RESULT\_CACHE\_MAX\_RESULT: This is the maximum amount of memory a single result in the cache can use, in percentage terms. The default is 5% of the server result cache.

For example, you can use the following set of values for the three server result cache-related initialization parameters:

RESULT\_CACHE\_MAX\_SIZE=500M /\* Megabytes  
RESULT\_CACHE\_MAX\_RESULT=20 /\* Percentage  
RESULT\_CACHE\_REMOTE\_EXPIRATION=3600 /\* Minutes

Force result\_cache

SQL> alter session set result\_cache\_mode=force;

SQL> alter table stores result\_cache (mode force);

For no result\_cache

SQL>select /\*+ no\_result\_cache \*/ \*  
from stores  
order by time\_id desc;

You can remove cached results from the server result cache by using the FLUSH procedure from the DBMS\_RESULT\_CACHE package, as shown here:

SQL> execute dbms\_result\_cache.flush

### Configuring the Oracle Database Smart Flash Cache

Depending on your operating system, you can use the new Oracle Database Smart Flash Cache feature, in cases where the database indicates that it needs a much larger amount of memory for the buffer cache. Right now, the Flash Cache feature is limited to Solaris and Oracle Linux operating systems, up to 16 OS devices

Set the following parameters to turn the Flash Cache feature on:

DB\_FLASH\_CACHE\_FILE= "/dev/sdc"

DB\_FLASH\_CACHE\_FILE = "/export/home/oracle/file\_raw" /\* raw file

DB\_FLASH\_CACHE\_FILE = "+dg1/file\_asm" /\* using ASM storage

SQL> alter system set db\_flash\_cache\_size = 0; /\* disables flash cache

SQL> alter system set db\_flash\_cache\_size = 8G; /\* reenables flash cache

### Tuning the Redo Log Buffer

SQL> alter system set log\_buffer=4096000 scope=spfile;

It’s fairly easy to tune the size of the LOG\_BUFFER parameter. Just execute the following statement to get the current “redo log space request ratio”:

SQL> select round(t.value/s.value,5) "Redo Log Space Request Ratio"

from v$sysstat s, v$sysstat t

where s.name = 'redo log space requests'

and t.name = 'redo entries'

and s.value !=0;

The redo log space request ratio is simply the ratio of total redo log space requests to redo entries. You can also query the V$SYSSTAT view to find the value of the statistic redo buffer allocation retries. This statistic shows the number of times processes waited for space in the redo log buffer:

SQL> select name,value from V$SYSSTAT

where name= 'redo buffer allocation retries';

Execute this SQL query multiple times over a period of time. If the value of the “redo buffer allocation retries” statistic is rising steadily over this period, it indicates that the redo log buffer is under space pressure and as a result, processes are waiting to write their redo log entries to the redo log buffer. You must increase the size of the redo log buffer if you continue to see this.

## Table Performance

### Creating Tablespaces

* Smallfile Tablespace:

create tablespace tools  
datafile '+DATA'  
size 100m -- Fixed datafile size  
extent management local -- Locally managed  
uniform size 128k -- Uniform extent size  
segment space management auto -- ASSM  
/

* Bigfile Tablespace

create bigfile tablespace tools\_bf  
datafile '/ora01/dbfile/O11R2/tools\_bf01.dbf'  
size 100m  
extent management local  
uniform size 128k  
segment space management auto  
/

### Choosing Table Features for Performance

Table Features That Impact Performance

* If a column always contains numeric data, make it a number data type
* If you have a business rule that defines the length and precision of a number field, then enforce it—for example, **NUMBER(7,2)**. If you don’t have a business rule, make it **NUMBER(38)**.
* For character data that is of variable length, use **VARCHAR2** (and not **VARCHAR**).
* Use **DATE** and **TIMESTAMP** data types appropriately.
* Consider setting the physical attribute **PCTFREE** to a value higher than the default of 10% if the table initially has rows inserted with null values that are later updated with large values.
* Most tables should be created with a primary key.
* Create a numeric surrogate key to be the primary key for each table. Populate the surrogate key from a sequence.
* Create a unique key for the logical business key-a recognizable combination of columns that makes a row unique
* Define foreign keys where appropriate
* Consider special features such as virtual columns, read-only, parallel, compression, no logging, and so on.

Table Features That Impact Scalability and Maintainability

* Use standards when naming tables, columns, constraints, triggers, indexes…
* If you have a business rule that specifies the maximum length of a column, then use that length, as opposed to making all columns VARCHAR2(4000).
* Specify a separate tablespace for the table and indexes.
* Let tables and indexes inherit storage attributes from the tablespaces
* Create primary-key constraints out of line
* Create comments for the tables and columns.
* Avoid large object (LOB) data types if possible. If you use LOBs in Oracle Database 11g or higher, use the new SecureFiles architecture.
* If a column should always have a value, then enforce it with a NOT NULL constraint.
* Create audit-type columns, such as
* CREATE\_DTT and UPDATE\_DTT, that are automatically populated with default values and/or triggers.
* Use check constraints where appropriate.

### Maximizing Data Loading Speeds

Use a combination of the following two features to maximize the speed of insert statements:

* Set the table’s logging attribute to NOLOGGING; this minimizes the generation redo for direct path operations (this feature has no effect on regular DML operations).
* Use a direct path loading feature, such as the following:
  + INSERT /\*+ APPEND \*/ on queries that use a subquery for determining which records are inserted
  + INSERT /\*+ APPEND\_VALUES \*/ on queries that use a VALUES clause
  + CREATE TABLE…AS SELECT

The prior output verifies that the table was created with **LOGGING** enabled (the default). To enable **NOLOGGING**, use the **ALTER TABLE** statement as follows:

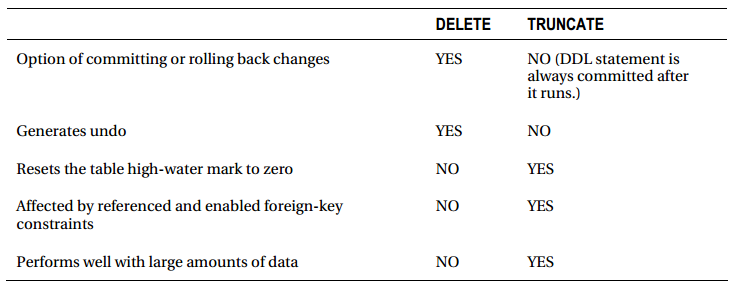
SQL> alter table f\_regs nologging;

Now that **NOLOGGING** has been enabled, there should be a minimal amount of redo generated for direct path operations. The following example uses a direct path **INSERT** statement to load data into the table:

insert /\*+APPEND \*/ into f\_regs  
select \* from reg\_master;

The prior statement is an efficient method for loading data because direct path operations such as **INSERT /\*+APPEND \*/** combined with **NOLOGGING** generate a minimal amount of redo.

### Efficiently Removing Table Data



### Rebuilding Rows Spanning Multiple Blocks

* Remove row changing

SELECT  
'Task Name : ' || f.task\_name || chr(10) ||  
'Segment Name : ' || o.attr2 || chr(10) ||  
'Segment Type : ' || o.type || chr(10) ||  
'Partition Name : ' || o.attr3 || chr(10) ||  
'Message : ' || f.message || chr(10) ||  
'More Info : ' || f.more\_info TASK\_ADVICE  
FROM dba\_advisor\_findings f  
,dba\_advisor\_objects o  
WHERE o.task\_id = f.task\_id  
AND o.object\_id = f.object\_id  
ORDER BY f.task\_name;

TASK\_ADVICE  
-------------------------------------------------------------------------  
Task Name : EMP Advice  
Segment Name : EMP  
Segment Type : TABLE  
Partition Name :  
Message : The object has chained rows that can be removed by re-org.  
More Info : 47 percent chained rows can be removed by re-org.

* Solution:

SQL> alter table emp move;

* Find index and rebuild

Select owner ,index\_name ,status  
from dba\_indexes  
where table\_name='EMP';

OWNER INDEX\_NAME STATUS  
------------------------------ ------------------------------ --------  
MV\_MAINT EMP\_PK UNUSABLE

* Rebuilding the index will make it usable again:

SQL> alter index emp\_pk rebuild;

### Freeing Unused Table Space

* When you shrink a table, this requires that rows (if any) be moved. This means you must enable row movement. This example enables row movement for the INV table:

SQL> alter table inv enable row movement;

* Next the table shrink operation is executed via an ALTER TABLE statement:

SQL> alter table inv shrink space;

* You can also shrink the space associated with any index segments via the CASCADE clause:

SQL> alter table inv shrink space cascade;

### Compressing Data for Direct Path Loading

Use Oracle’s basic compression feature to compress direct path–loaded data into a heap-organized table. Basic compression is enabled as follows:

* Use the COMPRESS clause to enable compression either when creating, altering, or moving an existing table.
* Load data via a direct path mechanism such as CREATE TABLE…AS SELECT or INSERT /\*+ APPEND \*/.

Example:

SQL> select table\_name, compression, compress\_for  
from user\_tables  
where table\_name='REGS\_DSS';

TABLE\_NAME COMPRESS COMPRESS\_FOR

------------------------------ -------- ------------

REGS\_DSS ENABLED BASIC

SQL> alter table regs\_dss compress;

When you alter a table to enable basic compression, this does not affect any data currently existing in the table; rather it only compresses subsequent direct path data load operations. If you want to enable basic compression for data in an existing table, use the MOVE COMPRESS clause:

SQL> alter table regs\_dss move compress;

Keep in mind that when you move a table, all of the associated indexes are invalidated. You’ll have to rebuild any indexes associated with the moved table.

If you have enabled basic compression for a table, you can disable it via the NOCOMPRESS clause—for example:

SQL> alter table regs\_dss nocompress;

SQL> alter table regs\_dss move nocompress;

### Compressing Data for All DML

Use the COMPRESS FOR OLTP clause when creating a table to enable data compression when using regular DML statements to manipulate data. This example creates an OLTP compression–enabled table:

SQL> create table regs

(reg\_id number

,reg\_name varchar2(2000))

compress for oltp;

SQL> alter table regs compress for oltp;

SQL> select table\_name, compression, compress\_for  
from user\_tables  
where table\_name='REGS';  
  
TABLE\_NAME COMPRESS COMPRESS\_FOR  
------------------------------ -------- ------------  
REGS ENABLED OLTP

SQL> alter table regs move compress for oltp;

SQL> alter table regs nocompress;

SQL> CREATE TABLESPACE comp\_data  
DATAFILE '/ora01/dbfile/O11R2/comp\_data01.dbf'  
SIZE 10M  
EXTENT MANAGEMENT LOCAL  
UNIFORM SIZE 1M  
SEGMENT SPACE MANAGEMENT AUTO  
DEFAULT COMPRESS FOR OLTP;

SQL> alter tablespace comp\_data default compress for oltp;

SQL> select tablespace\_name, def\_tab\_compression, compress\_for  
from dba\_tablespaces where tablespace\_name = 'COMP\_DATA';

TABLESPACE\_NAME DEF\_TAB\_ COMPRESS\_FOR  
------------------------------ -------- ------------  
COMP\_DATA ENABLED OLTP

### Compressing Data at the Column Level

To enable hybrid columnar compression, when creating a table, use either the COMPRESS FOR QUERY or the COMPRESS FOR ARCHIVE clause—for example:

* COMPRESS FOR QUERY LOW
* COMPRESS FOR QUERY HIGH
* COMPRESS FOR ARCHIVE LOW
* COMPRESS FOR ARCHIVE HIGH

SQL> create table f\_regs(

reg\_id number

,reg\_desc varchar2(4000))

compress for query;

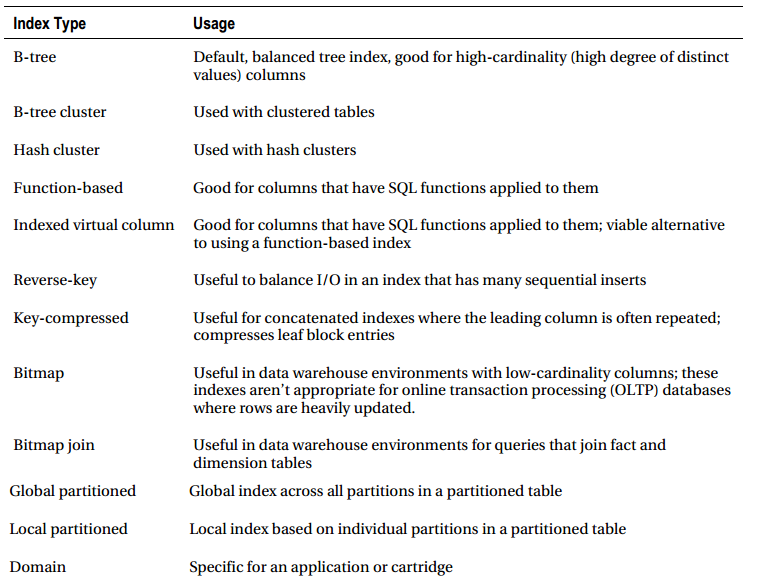
SQL> create table f\_regs(  
reg\_id number  
,reg\_desc varchar2(4000))  
compress for query high;

select table\_name, compression, compress\_for  
from user\_tables where table\_name='F\_REGS';

TABLE\_NAME COMPRESS COMPRESS\_FOR  
------------------------------ -------- ------------  
F\_REGS ENABLED QUERY HIGH

## Choosing and Optimizing Indexes

### Index Type



### Rule to create index

* Use the default **B-tree index** unless you have a solid reason to use a different index type.
* Create a **separate tablespace** for the indexes. This allows you to more easily manage indexes separately from tables for tasks such as backup and recovery.
* Let the index inherit its storage properties from the tablespace. This allows you to specify the storage properties when you create the tablespace and not have to manage storage properties for individual indexes.
* If you have a variety of storage requirements for indexes, then consider creating separate tablespaces for each type of index—for example, INDEX\_LARGE, INDEX\_MEDIUM, and INDEX\_SMALL tablespaces, each defined with storage characteristics appropriate for the size of the index.
* Add indexes judiciously. **Test first** to determine quantifiable performance gains.
* Use the **correct type** of index, consistent naming standards.
* **Monitor your indexes**, and drop indexes that aren’t used.
* Don’t rebuild indexes unless you have a solid reason to do so.
* Before dropping an index, consider marking it as unusable or invisible.
* Consider creating concatenated indexes that result in only the index structure being required to return the result set.
* Consider creating indexes on columns used in the **WHERE, JOIN, ORDER BY**, **GROUP BY, UNION**, or **DISTINCT** clauses.
* If columns are often used together in the WHERE clause, consider creating a concatenated index.
* If a column is also used (in other queries) by itself in the WHERE clause, place that column at the leading edge of the index (first column defined).

### Creating a Primary Key Index

When you define a primary key constraint for a table, Oracle will automatically create an associated index for you.

alter table cust add constraint cust\_pk primary key (cust\_id)  
using index tablespace users;

Or

create table cust(cust\_id number constraint cust\_pk primary key  
using index tablespace users);

### Creating a Unique Index

You have a column (or combination of columns) that contains values that should always be unique. You want to create an index on this column (or combination of columns) that enforces the uniqueness and also provides efficient access to the table when using the unique column in the WHERE clause of a query.

SQL> alter table cust add constraint cust\_ux1 unique (last\_name, first\_name) using index tablespace users;

create table cust(  
cust\_id number  
,last\_name varchar2(30)  
,first\_name varchar2(30)  
,constraint cust\_ux1 unique(last\_name, first\_name)  
using index tablespace users);

SQL> create unique index cust\_uidx1 on cust(last\_name, first\_name) tablespace users;  
SQL> alter table cust add constraint cust\_uidx1 unique (last\_name, first\_name);

### Reverse-key indexes

Reverse-key indexes can perform better in scenarios where you need a way to evenly distribute index data that would otherwise have similar values clustered together. Thus, when using a reverse-key index, you avoid having I/O concentrated in one physical disk location within the index **during large inserts** of sequential values. The downside to this type of index is that it can’t be used for index range scans, which therefore limits its usefulness.

You can rebuild an existing index to be reverse-key by using the REBUILD REVERSE clause—for example:

SQL> alter index f\_regs\_idx1 rebuild reverse;

Similarly, if you want to make an index that is reverse-key into a normally ordered index, then use the REBUILD NOREVERSE clause:

SQL> alter index f\_regs\_idx1 rebuild noreverse;

### Function-Based Index

Fuction column

* Create a function-based index in column have SQL fuction applied

SQL> create index cust\_fidx1 on cust(UPPER(first\_name));

Indexing a Virtual Column

* Use a virtual column in combination with an index. ( make a virtual column like a SQL fuction and make a index in it)
* First a virtual column is added to the table that encapsulates the SQL function:

SQL> alter table cust add(up\_name generated always as (UPPER(first\_name)) virtual);

* Next an index is created on the virtual column:

SQL> create index cust\_vidx1 on cust(up\_name);

### Adding an Index Without Impacting Existing Applications

You can create the index as invisible and then explicitly instruct a query to use the index via a hint—for example

SQL> create index inv\_idx1 on inv(inv\_id) invisible;

Next, ensure that the OPTIMIZER\_USE\_INVISIBLE\_INDEXES initialization parameter is set to TRUE (the default is FALSE). This instructs the optimizer to consider invisible indexes:

SQL> alter system set optimizer\_use\_invisible\_indexes=true;

Now, use a hint to tell the optimizer that the index exists:

SQL> select /\*+ index (inv INV\_IDX1) \*/ inv\_id from inv where inv\_id=1;

You can verify that the index is being used by setting AUTOTRACE TRACE EXPLAIN and running the SELECT statement:

SQL> set autotrace trace explain;

SQL> select /\*+ index (inv INV\_IDX1) \*/ inv\_id from inv where inv\_id=1;

Make it visible

SQL> alter index inv\_idx1 visible;

### Creating a Bitmap Index

You have a data warehouse that contains a star schema. The star schema consists of a large fact table and several dimension (lookup) tables. The primary key columns of the dimension tables map to foreign key columns in the fact table. You would like to create bitmap indexes on all of the foreign key columns in the fact table.

SQL> create bitmap index f\_sales\_cust\_fk1 on f\_sales(cust\_id);

The type of index is verified with the following query:

SQL> select index\_name, index\_type

from user\_indexes

where index\_name='F\_SALES\_CUST\_FK1';

INDEX\_NAME INDEX\_TYPE

------------------------------ ---------------------------

F\_SALES\_CUST\_FK1 BITMAP

### Creating a Bitmap Join Index

You’re working in a data warehouse environment. You have a fairly large dimension table that is often joined to an extremely large fact table. You wonder if there’s a way to create a bitmap index in such a way that it can eliminate the need for the optimizer to access the dimension table blocks to satisfy the results of a query

create bitmap index <index\_name>  
on <fact\_table> (<dimension\_table.dimension\_column>)  
from <fact\_table>, <dimension\_table>  
where <fact\_table>.<foreign\_key\_column> = <dimension\_table>.<primary\_key\_column>;

### Creating an Index-Organized Table

You want to create a table that is the intersection of a many-to-many relationship between two tables. The intersection table will consist of two columns. Each column is a foreign key that maps back to a corresponding primary key in a parent table.

create table cust\_assoc  
(cust\_id number  
,user\_group\_id number  
,create\_dtt timestamp(5)  
,update\_dtt timestamp(5)  
,constraint cust\_assoc\_pk primary key(cust\_id, user\_group\_id))  
organization index  
including create\_dtt  
pctthreshold 30  
tablespace nsestar\_index  
overflow  
tablespace dim\_index;

### Monitoring Index Usage

set pagesize 0 head off linesize 132  
spool enable\_mon.sql  
select  
'alter index ' || index\_name || ' monitoring usage;'  
from user\_indexes;  
spool off;

select io.name, t.name,  
decode(bitand(i.flags, 65536), 0, 'NO', 'YES'),  
decode(bitand(ou.flags, 1), 0, 'NO', 'YES'),  
ou.start\_monitoring,  
ou.end\_monitoring  
from sys.obj$ io  
,sys.obj$ t  
,sys.ind$ i  
,sys.object\_usage ou  
where i.obj# = ou.obj#  
and io.obj# = ou.obj#  
and t.obj# = i.bo#;

### Maximizing Index Creation Speed

#### Turning Off Redo Generation

create index inv\_idx1 on inv(inv\_id, inv\_id2)  
nologging  
tablespace inv\_mgmt\_index;

#### Increasing the Degree of Parallelism

create index inv\_idx1 on inv(inv\_id)  
parallel 2  
tablespace inv\_mgmt\_data;

### Reclaiming Unused Index Space

SELECT  
'Task Name : ' || f.task\_name || CHR(10) ||  
'Start Run Time : ' || TO\_CHAR(execution\_start, 'dd-mon-yy hh24:mi') || chr (10) ||  
'Segment Name : ' || o.attr2 || CHR(10) ||  
'Segment Type : ' || o.type || CHR(10) ||  
'Partition Name : ' || o.attr3 || CHR(10) ||  
'Message : ' || f.message || CHR(10) ||  
'More Info : ' || f.more\_info || CHR(10) ||

'------------------------------------------------------' Advice  
FROM dba\_advisor\_findings f  
,dba\_advisor\_objects o  
,dba\_advisor\_executions e  
WHERE o.task\_id = f.task\_id  
AND o.object\_id = f.object\_id  
AND f.task\_id = e.task\_id  
AND e. execution\_start > sysdate - 1  
AND e.advisor\_name = 'Segment Advisor'  
ORDER BY f.task\_name;

SQL> alter index f\_regs\_idx1 rebuild;

SQL> alter index f\_regs\_idx1 shrink space;

# SQL Tuning

## Creating Efficient SQL

### Avoiding Accidental Full Table Scans

When constructing a SQL statement, a fundamental rule to try to always observe, if possible, is to avoid using functions on the left side of the comparison operator. A function essentially turns a column into a literal value, and therefore the Oracle optimizer does not recognize that converted value as a column any longer, but as a value instead. scan, even though the HIRE\_DATE column is indexed:

SELECT employee\_id, salary, hire\_date

FROM employees

**WHERE TO\_CHAR(hire\_date,'yyyy-mm-dd') >= '2000-01-01';**

------------------------------------  
| Id | Operation | Name |  
------------------------------------  
| 0 | SELECT STATEMENT | |  
| 1 | TABLE ACCESS FULL| EMPLOYEES |  
------------------------------------

SELECT employee\_id, salary, hire\_date  
FROM employees  
**WHERE hire\_date >= TO\_DATE('2000-01-01','yyyy-mm-dd');**

-------------------------------------------------  
| Id | Operation | Name |  
-------------------------------------------------  
| 0 | SELECT STATEMENT | |  
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES |  
| 2 | INDEX RANGE SCAN | EMP\_I5 |  
-------------------------------------------------

### Avoiding the NOT Clause

* Avoid not clause
* Take it into subqueries

SELECT last\_name, first\_name, salary, email  
FROM employees\_big  
**WHERE department\_id NOT IN(20,30)**  
AND commission\_pct > 0;

-------------------------------------------------------------------------  
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |  
-------------------------------------------------------------------------  
| 0 | SELECT STATEMENT | | 697K| 21M| **4480** (1)| 00:00:54 |  
|\* 1 | TABLE ACCESS FULL| EMPLOYEES\_BIG | 697K| 21M| 4480 (1)| 00:00:54 |  
-------------------------------------------------------------------------

SELECT last\_name, first\_name, salary, email  
FROM employees\_big  
**WHERE department\_id IN  
(SELECT department\_id FROM departments  
WHERE department\_id NOT IN (20,30))**  
AND commission\_pct > 0;

-------------------------------------------------------------------------  
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |  
-------------------------------------------------------------------------  
| 0 | SELECT STATEMENT | | 33 | 1188 | **3** (0)| 00:00:01 |  
| 1 | NESTED LOOPS | | 33 | 1188 | 3 (0)| 00:00:01 |  
|\* 2 | TABLE ACCESS FULL| EMPLOYEES | 34 | 1088 | 3 (0)| 00:00:01 |  
|\* 3 | INDEX UNIQUE SCAN| DEPT\_ID\_PK | 1 | 4 | 0 (0)| 00:00:01 |  
-------------------------------------------------------------------------

### **Seeing Execution Statistics for Currently** Running SQL

SELECT sid, sql\_text FROM v$sql\_monitor  
WHERE status = 'EXECUTING';

SELECT sid, buffer\_gets, disk\_reads, round(cpu\_time/1000000,1) cpu\_seconds  
FROM v$sql\_monitor  
WHERE SID=100  
AND status = 'EXECUTING';

SID BUFFER\_GETS DISK\_READS CPU\_SECONDS  
---------- ----------- ---------- -----------  
100 149372 4732 39.1

SELECT \* FROM (  
SELECT sid, buffer\_gets, disk\_reads, round(cpu\_time/1000000,1) cpu\_seconds  
FROM v$sql\_monitor  
ORDER BY cpu\_time desc)  
WHERE rownum <= 5;

SID BUFFER\_GETS DISK\_READS CPU\_SECONDS  
---------- ----------- ---------- -----------  
20 1332665 30580 350.5  
105 795330 13651 269.7  
20 259324 5449 71.6  
20 259330 5485 71.3  
100 259236 8188 67.9

### Implementing Query Hints

Using hints to get another way to execute SQL statements

Get detail in book

### Executing SQL in Parallel

Using parael to increase the performace of SQL statement, only for large amount of data

Get detail in book

# Backup Tuning

# Tuning for Oracle Data Guard

# Tuning for Oracle Real Application Clusters.