Oracle Asia Pacific

Oracle Insert Statements for DBAs and Developers

Daniel A. Morgan
dmorgan@forsythe.com
+1 206-669-2949

Monday: August 14, 2017

Unsafe Harbor

- This room is an unsafe harbor
- You can rely on the information in this presentation to help you protect your data, your databases, your organization, and your career
- No one from Oracle has previewed this presentation
- No one from Oracle knows what I'm going to say
- No one from Oracle has supplied any of my materials
- Everything I will present is existing, proven, functionality

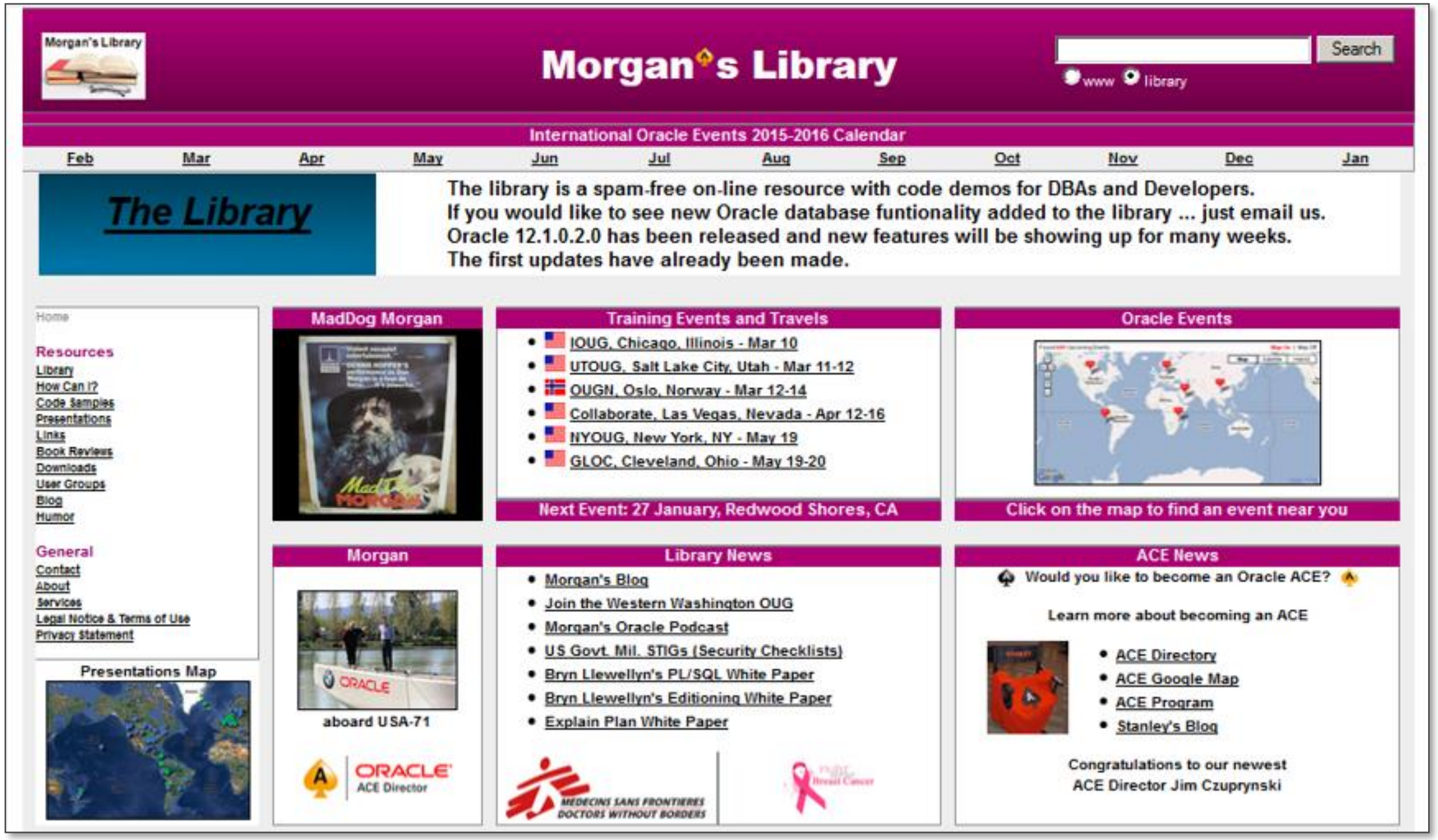
Introduction

Daniel Morgan

- Oracle ACE Director Alumni
- Oracle Educator
  - Curriculum author and primary program instructor at University of Washington
  - Consultant: Harvard University
  - University Guest Lecturers
    - APAC: University of Canterbury (NZ)
    - EMEA: University of Oslo (Norway)
    - Latin America: Universidad Cenfotec, Universidad Latina de Panama, Technologico de Costa Rica
- IT Professional
  - First computer: IBM 360/40 in 1969: Fortran IV
  - Oracle Database since 1988-9 and Oracle Beta tester
  - The Morgan behind www.morganslibrary.org
  - Member Oracle Data Integration Solutions Partner Advisory Council
  - Vice President Twin Cities Oracle Users Group (Minneapolis-St. Paul)
  - Co-Founder International GoldenGate Oracle Users Group
- Principal Adviser: Forsythe **Meta7**

System/370-145 system console

My Websites: Morgan's Library



Morgan's Library

Search

www library

International Oracle Events 2015-2016 Calendar

Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec Jan

The Library

The library is a spam-free on-line resource with code demos for DBAs and Developers.
If you would like to see new Oracle database funtionality added to the library ... just email us.
Oracle 12.1.0.2.0 has been released and new features will be showing up for many weeks.
The first updates have already been made.

Home

Resources
Library
How Can I?
Code Samples
Presentations
Links
Book Reviews
Downloads
User Groups
Blog
Humor

General
Contact
About
Services
Legal Notice & Terms of Use
Privacy Statement

Presentations Map

MadDog Morgan

Training Events and Travels
- IOUG, Chicago, Illinois - Mar 10
- UTOUG, Salt Lake City, Utah - Mar 11-12
- OUGN, Oslo, Norway - Mar 12-14
- Collaborate, Las Vegas, Nevada - Apr 12-16
- NYOUG, New York, NY - May 19
- GLOC, Cleveland, Ohio - May 19-20

Next Event: 27 January, Redwood Shores, CA

Oracle Events

Click on the map to find an event near you

Morgan

aboard USA-71

ORACLE ACE Director

Library News
- Morgan's Blog
- Join the Western Washington OUG
- Morgan's Oracle Podcast
- US Govt. Mil. STIGs (Security Checklists)
- Bryn Llewellyn's PL/SQL White Paper
- Bryn Llewellyn's Editioning White Paper
- Explain Plan White Paper

MEDECINS SANS FRONTIERES DOCTORS WITHOUT BORDERS

ACE News

Would you like to become an Oracle ACE?

Learn more about becoming an ACE

- ACE Directory
- ACE Google Map
- ACE Program
- Stanley's Blog

Congratulations to our newest
ACE Director Jim Czuprynski

www.morganslibrary.org

META7™ Solutions for the Red Stack

Stability: IT Fire Fighting

META7™ Solutions for the Red Stack

Oracle Stack Security

Scalability: VLDBs and Partitioning

META7™ Solutions for the Red Stack

Database Performance

Zero Downtime Migration

Just In Time IT Procurement

with the Oracle Bare Metal Cloud and Infrastructure as Code

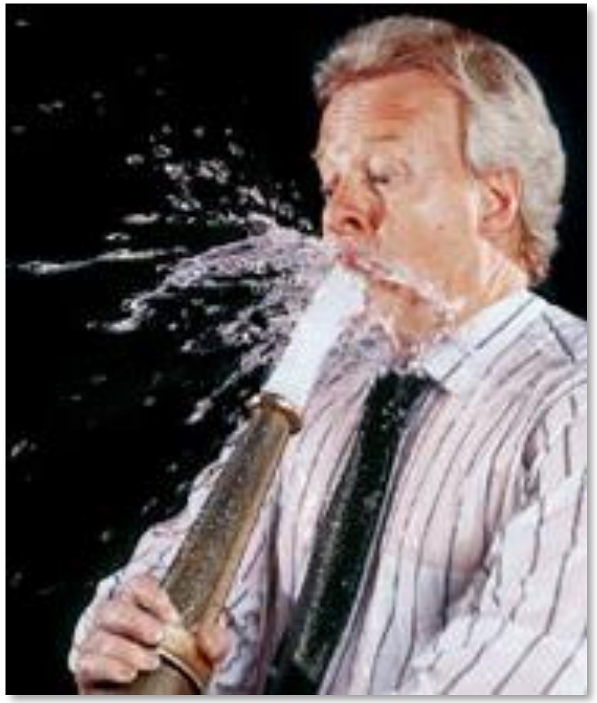
META7™ Solutions for the Red Stack

Travel Log: 2010 - Lima Peru

Travel Log: 2014 - Galapagos Islands Ecuador

Travel Log: 2015 - Turkey

Content Density Warning



Take Notes ... Email Questions

Why Am I Focusing On INSERT Statements?

- Because no one else is
- Because Oracle University doesn't teach this material
- Because there are 17 pages in the 12c docs on INSERT statements
- Because almost no one knows the full syntax for basic DML statements
- Because we have now spent more than 30 years talking about performance tuning and yet the number one conference and training topic remains tuning which proves that we need to stop focusing on edge cases and focus, instead, on the basics
- Because explain plans, AWR Reports, and trace files will never fix a problem if you don't know the full range of syntaxes available
- Because the best way to achieve high performance is to choose techniques that reduce resource utilization

Insert Statements

SQL DML

- DML stands for **D**ata **M**anipulation **L**anguage
- DML is a direct reference to the following SQL statements
  - INSERT
  - UPDATE
  - DELETE
  - MERGE

SQL INSERT Statement Topics

- Basic Insert
- INSERT WHEN
- INSERT ALL
- INSERT ALL WHEN
- INSERT FIRST WHEN
- INSERT INTO A SELECT STATEMENT
- INSERT WITH CHECK OPTION
- View Inserts
- Editioning View Inserts
- Partitioned Table Inserts
- Cluster Table Inserts
- Tables with Virtual Columns Insert
- Tables with Hidden Columns Insert
- Create Table As Inserts
- Nested Table Inserts
- VARRAY Table Inserts
- MERGE Statement Insert

PL/SQL INSERT Statement Topics

- Record inserts
- FORALL INSERTs
- FORALL MERGE Inserts
- LOB Inserts
- DBMS_SQL Dynamic Inserts
- Native Dynamic SQL Inserts
- RETURNING Clause with a Sequence
- RETURNING Clause with an Identity Column

Performance Tuning INSERT Statement Topics

- Too Many Columns
- Column Ordering
- Aliasing and Fully Qualified Names
- Implicit Casts
- APPEND hint
- APPEND_VALUES hint
- DBMS_ERRLOG built-in package
  - `CHANGE_DUPKEY_ERROR_INDEX` hint
  - `IGNORE_ON_DUPKEY_INDEX` hint
- DBMS_STATS
- Insert Statement Most Common Error

Part 1: SQL Insert Statements

Basic INSERT Statement (1:2)

- Use this syntax to perform inserts into a single column in a heap, global temporary, IOT, and most partitioned tables

```
INSERT INTO <table_name>
(<column_name>)
VALUES
(<value>);
```

```
CREATE TABLE state (
state_abbrev VARCHAR2(2));

INSERT INTO state
(state_abbrev)
VALUES
('NY');

COMMIT;

SELECT * FROM state;
```

Basic INSERT Statement (2:2)

- Use this syntax to perform inserts into a single column in a heap, global temporary, IOT, and most partitioned tables

```
INSERT INTO <table_name>
(<column_name>, <column_name> [,...])
VALUES
(<value>, <value> [,<value>]);
```

```
CREATE TABLE state (
state_abbrev VARCHAR2(2),
state_name   VARCHAR2(30));

INSERT INTO state
(state_abbrev, state_name)
VALUES
('NY', 'New York');

COMMIT;

SELECT * FROM state;
```

INSERT WHEN and INSERT ALL WHEN

- Use this syntax to conditionally insert rows into multiple tables

```
INSERT
WHEN (<condition>) THEN
  INTO <table_name> (<column_list>)
  VALUES (<values_list>)
WHEN (<condition>) THEN
  INTO <table_name> (<column_list>)
  VALUES (<values_list>)
ELSE
  INTO <table_name> (<column_list>)
  VALUES (<values_list>)
SELECT <column_list> FROM <table_name>;
```

```
INSERT ALL
WHEN (<condition>) THEN
  INTO <table_name> (<column_list>)
  VALUES (<values_list>)
WHEN (<condition>) THEN
  INTO <table_name> (<column_list>)
  VALUES (<values_list>)
ELSE
  INTO <table_name> (<column_list>)
  VALUES (<values_list>)
SELECT <column_list> FROM <table_name>;
```

```
INSERT
WHEN (deptno=10) THEN
  INTO emp_10 (empno,ename,job,mgr,sal,deptno)
  VALUES (empno,ename,job,mgr,sal,deptno)
WHEN (deptno=20) THEN
  INTO emp_20 (empno,ename,job,mgr,sal,deptno)
  VALUES (empno,ename,job,mgr,sal,deptno)
WHEN (deptno=30) THEN
  INTO emp_30 (empno,ename,job,mgr,sal,deptno)
  VALUES (empno,ename,job,mgr,sal,deptno)
ELSE
  INTO leftover (empno,ename,job,mgr,sal,deptno)
  VALUES (empno,ename,job,mgr,sal,deptno)
SELECT * FROM emp;
```

```
INSERT ALL
WHEN (location < 6) THEN
  INTO hq_employee (empno,ename,job,mgr,sal,deptno)
  VALUES (empno,ename,job,mgr,sal,deptno)
WHEN (term_date IS NOT NULL) THEN
  INTO current_emp (empno,ename,job,mgr,sal,deptno)
  VALUES (empno,ename,job,mgr,sal,deptno)
WHEN (rehire = 1) THEN
  INTO rehires (empno,ename,job,mgr,sal,deptno)
  VALUES (empno,ename,job,mgr,sal,deptno)
ELSE
  INTO other_emps (empno,ename,job,mgr,sal,deptno)
  VALUES (empno,ename,job,mgr,sal,deptno)
SELECT * FROM emp;
```

INSERT ALL

- Use this syntax to unconditionally insert data into multiple tables
- Note that columns can go into one target table, multiple target tables, or all target tables

```
INSERT ALL
INTO <table_name> VALUES <column_name_list)
INTO <table_name> VALUES <column_name_list)
...
<SELECT Statement>;
```

```
INSERT ALL
  INTO ap_cust VALUES (customer_id, program_id, delivered_date)
  INTO ap_orders VALUES (order_date, program_id)
SELECT program_id, delivered_date, customer_id, order_date
FROM airplanes;
```

INSERT FIRST WHEN

- With "FIRST" the database evaluates each WHEN clause in the order in which it appears in the statement and only performs an insert for the first match

```
INSERT FIRST
WHEN <condition> THEN
INTO <table_name> VALUES <column_name_list)
INTO <table_name> VALUES <column_name_list)
...
<SELECT Statement>;
```

```
INSERT FIRST
WHEN customer_id < 'I' THEN
  INTO cust_ah
  VALUES (customer_id, program_id, delivered_date)
WHEN customer_id < 'Q' THEN
  INTO cust_ip
  VALUES (customer_id, program_id, delivered_date)
WHEN customer_id > 'PZZZ' THEN
  INTO cust_qz
  VALUES (customer_id, program_id, delivered_date)
SELECT program_id, delivered_date, customer_id, order_date
FROM airplanes;
```

INSERT into a SELECT Statement

- Use this syntax to INSERT rows into a table a part of a SELECT statement from itself or one or more different tables

```
INSERT INTO (
<SELECT Statement>)
VALUES (value_list);
```

```
CREATE TABLE dept (
dept_no   NUMBER(3) NOT NULL,
dept_name VARCHAR2(2) NOT NULL,
dept_loc  VARCHAR2(30));

INSERT INTO (
 SELECT dept_no, dept_name, dept_loc
 FROM dept)
VALUES (99, 'TRAVEL', 'SEATTLE');
```

INSERT with Check Option

- Use this syntax to limit inserted rows to only those that pass CHECK OPTION validation

```
INSERT INTO (
<SELECT_statement> WITH CHECK OPTION)
VALUES (value_list);
```

```
CREATE TABLE dept (
dept_no   NUMBER(3) NOT NULL,
dept_name VARCHAR2(2) NOT NULL,
dept_loc  VARCHAR2(30));

INSERT INTO (
  SELECT dept_no, dept_name, dept_loc
  FROM dept
  WHERE deptno < 30 WITH CHECK OPTION)
VALUES (99, 'TRAVEL', 'SEATTLE');
```

INSERTing into a View

- Evaluate whether a view column is insertable
- Views with aggregations, CONNECT BY, and other syntaxes may not be insertable

```
desc cdb_updatable_columns

SELECT cuc.con_id, cuc.owner, cuc.insertable, COUNT(*)
FROM cdb_updatable_columns cuc
WHERE (cuc.con_id, cuc.owner, cuc.table_name) IN
  (SELECT cv.con_id, cv.owner, cv.view_name
   FROM cdb_views cv)
GROUP BY cuc.con_id, cuc.owner, cuc.insertable
ORDER BY 1,2,3;

 CON_ID    OWNER                     INS   COUNT(*)
---------- ------------------------- --- ----------
         2 ORDSYS                    NO           4
         2 ORDSYS                    YES          4
         2 SYS                       NO       45190
         2 SYS                       YES      22415
         2 SYSTEM                    NO         172
         2 SYSTEM                    YES         14
         2 WMSYS                     NO         736
         2 WMSYS                     YES        160
```

INSERTing into an Editioning View

- All editioning views are insertable ... but be sure you are in the correct edition

```
SQL> CREATE EDITION demo_ed;

SQL> CREATE OR REPLACE EDITIONING VIEW test AS
  2  SELECT program_id, line_number
  3  FROM airplanes;

View created.

SQL> ALTER SESSION SET EDITION=demo_ed;

Session altered.

SQL> CREATE OR REPLACE EDITIONING VIEW test AS
  2  SELECT line_number, program_id
  3  FROM airplanes;

View created.

SQL> SELECT * FROM user_editioning_views_ae;

VIEW_NAME    TABLE_NAME               EDITION_NAME
------------ ------------------------ -------------
TEST         AIRPLANES                ORA$BASE
TEST         AIRPLANES                DEMO_ED
```

INSERTing into a Partitioned Table

- With HASH, LIST, and RANGE partitioning any INSERT statement will work
- With Partition by SYSTEM you must name the partition

```
CREATE TABLE syst_part (
tx_id   NUMBER(5),
begdate DATE)
PARTITION BY SYSTEM (
PARTITION p1,
PARTITION p2,
PARTITION p3);

INSERT INTO syst_part VALUES (1, SYSDATE-10);
             *
ERROR at line 1:
ORA-14701: partition-extended name or bind variable must be used
for DMLs on tables partitioned by the System method

INSERT INTO syst_part PARTITION (p1) VALUES (1, SYSDATE-10);
INSERT INTO syst_part PARTITION (p2) VALUES (2, SYSDATE);
INSERT INTO syst_part PARTITION (p3) VALUES (3, SYSDATE+10);

SELECT * FROM syst_part PARTITION(p2);
```

INSERTing into a Cluster Table (1:3)

- In an Oracle Database a **CLUSTER** is an object similar in concept to a SecureFile in that it redefines storage in a portion of a tablespace
- When part of a tablespace is defined as a cluster each block in the cluster is able to hold more than a single segment minimizing I/O through co-location
  - A table and an index
    - Single Table Hash Cluster
    - Sorted Hash Cluster
  - Multiple tables and an index
    - Multi-Table Hash Cluster
    - Index Cluster
- Oracle's Data Dictionary is almost entirely defined by clusters which optimize access

```
SQL> SELECT cluster_name, cluster_type
  2  FROM dba_clusters
  3  WHERE owner = 'SYS'
  4  ORDER BY 1;

CLUSTER_NAME                   CLUST
------------------------------ -----
C_COBJ#                        INDEX
C_FILE#_BLOCK#                 INDEX
C_MLOG#                        INDEX
C_OBJ#                         INDEX
C_OBJ#_INTCOL#                 INDEX
C_RG#                          INDEX
C_TOID_VERSION#                INDEX
C_TS#                          INDEX
C_USER#                        INDEX
SMON_SCN_TO_TIME_AUX           INDEX
```

INSERTing into a Cluster Table (2:3)

- It makes a lot of sense to be able to read a single 8K block and get all of this information and you likely have similar situation in your applications

```
SQL> SELECT table_name
  2  FROM dba_tables
  3  WHERE cluster_name = 'C_OBJ#'
  4* ORDER BY 1;

TABLE_NAME
------------
ASSEMBLY$
ATTRCOL$       -- column attributes
CLU$           -- clusters
COL$           -- columns
COLTYPE$       -- column types
ICOL$          -- index columns
ICOLDEP$       -- index column dependencies
IND$           -- indexes
LIBRARY$
LOB$           -- CLOBs and BLOBs
NTAB$          -- nested tables
OPQTYPE$       -- opaque data types
REFCON$        -- reference constraints
SUBCOLTYPE$    -- object column attributes
TAB$           -- tables
TYPE_MISC$     -- type miscellaneous information
VIEWTRCOL$     -- view column attributes
```

INSERTing into a Cluster Table (3:3)

- So let's look at one type of cluster ... the Sorted Hash Cluster
- A very simple way to make the overhead of an ORDER BY clause go away

```
SQL> CREATE CLUSTER sorted_hc (
  2  program_id  NUMBER(3),
  3  line_id     NUMBER(10) SORT,
  4  delivery_dt DATE       SORT)
  5  TABLESPACE uwdata
  6  HASHKEYS 9
  7  SIZE 750
  8  HASH IS program_id;

Cluster created.

SQL> CREATE TABLE shc_airplane (
  2  program_id  NUMBER(3),
  3  line_id     NUMBER(10) SORT,
  4  delivery_dt DATE       SORT,
  5  customer_id VARCHAR2(3),
  6  order_dt    DATE)
  7  CLUSTER sorted_hc (program_id, line_id, delivery_dt);
```

INSERTing into a Table With Virtual Columns

- Virtual columns will appear in a DESCRIBE statement but you cannot insert values into them

```
CREATE TABLE vcol (
salary     NUMBER(8),
bonus      NUMBER(3),
total_comp NUMBER(10) AS (salary+bonus));

desc vcol

SELECT column_id, column_name, virtual_column
FROM user_tab_cols
WHERE table_name = 'VCOL'

INSERT INTO vcol
(salary, bonus, total_comp)
VALUES
(1,2,3);

INSERT INTO vcol
(salary, bonus)
VALUES
(1,2);

SELECT * FROM vcol;
```

INSERTing into a Table with Invisible Columns

- Invisible columns will not appear in a DESCRIBE statement but you can insert into them directly

```
CREATE TABLE vis (
rid     NUMBER,
testcol VARCHAR2(20));

CREATE TABLE invis (
rid     NUMBER,
testcol VARCHAR2(20) INVISIBLE);

desc vis

desc invis

SELECT table_name, column_name, hidden_column
FROM user_tab_cols                -- not found in user_tab_columns
WHERE table_name like '%VIS';

INSERT INTO invis
(rid, testcol)
VALUES
(1, 'TEST');

SELECT * FROM invis;

SELECT rid, testcol FROM invis;
```

CREATE TABLE as an INSERT Statement

- Use this syntax to create a new table as the result of a SELECT statement from one or more source tables

```
CREATE TABLE <table_name> AS
<SELECT Statement>;
```

```
CREATE TABLE column_subset AS
SELECT col1, col3, col5
FROM servers;

desc column_subset

SELECT COUNT(*)
FROM column_subset;
```

Nested Table Insert

- Cast column values using the object column's data type

```
CREATE OR REPLACE NONEDITIONABLE TYPE CourseList AS TABLE OF VARCHAR2(64);
/

CREATE TABLE department (
name     VARCHAR2(20),
director VARCHAR2(20),
office   VARCHAR2(20),
courses  CourseList)
NESTED TABLE courses STORE AS courses_tab;

INSERT INTO department
(name, director, office, courses)
VALUES
('English', 'Tara Havemeyer', 'Breakstone Hall 205', CourseList(
 'Expository Writing',
 'Film and Literature',
 'Modern Science Fiction',
 'Discursive Writing',
 'Modern English Grammar',
 'Introduction to Shakespeare',
 'Modern Drama',
 'The Short Story',
 'The American Novel'));
```

VARRAY Table Insert

- Cast column values using the VARRAY column's data type

```
CREATE OR REPLACE TYPE ProjectList AS VARRAY(50) OF Project;
/

CREATE TABLE department (
dept_id  NUMBER(2),
dname    VARCHAR2(15),
budget   NUMBER(11,2),
projects ProjectList);

INSERT INTO department
(dept_id, dname, budget, projects)
VALUES
(30, 'Accounting', 1205700,
ProjectList (Project(1, 'Design New Expense Report', 3250),
Project(2, 'Outsource Payroll', 12350),
Project(3, 'Evaluate Merger Proposal', 2750),
Project(4, 'Audit Accounts Payable', 1425)));
```

MERGE Statement Insert

- Use MERGE statements where an insert or other DML action is conditioned on the results of a SELECT statement result match

```
MERGE INTO bonuses b
USING (
  SELECT employee_id, salary, dept_no
  FROM employee
  WHERE dept_no =20) e
ON (b.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET b.bonus = e.salary * 0.1
  DELETE WHERE (e.salary < 40000)
WHEN NOT MATCHED THEN
  INSERT (b.employee_id, b.bonus)
  VALUES (e.employee_id, e.salary * 0.05)
  WHERE (e.salary > 40000);
```

Part 2: PL/SQL Insert Statements

Cursor Loops: One Row At A Time

- If you want to make insert statements as slow as possible ... do them one row at a time. Make each insert statement find a block into which it can be inserted and then check everything sequentially

```
CREATE TABLE parent (
part_num  NUMBER,
part_name VARCHAR2(15));

CREATE TABLE child AS
SELECT *
FROM parent;

CREATE OR REPLACE PROCEDURE slow_way AUTHID CURRENT_USER IS
BEGIN
  FOR r IN (SELECT * FROM parent) LOOP
    -- modify record values
    r.part_num := r.part_num * 10;
    -- store results
    INSERT INTO child
    VALUES
    (r.part_num, r.part_name);
  END LOOP;
  COMMIT;
END slow_way;
/
```

Record Inserts

- Use this syntax to insert based on an array that matches the target table rather than named individual columns
  - Adding a new column to the table will not break the statement

```
CREATE TABLE t AS
SELECT table_name, tablespace_name
FROM all_tables;

SELECT COUNT(*)
FROM t;

DECLARE
 trec  t%ROWTYPE;
BEGIN
  trec.table_name := 'NEW';
  trec.tablespace_name := 'NEW_TBSP';

  INSERT INTO t
  VALUES trec;

  COMMIT;
END;
/

SELECT COUNT(*) FROM t;
```

FORALL INSERTs (1:3)

- Use this syntax to greatly enhance performance but be sure you understand the concept of DIRECT LOAD INSERTs
- With this syntax I can insert 500,000 rows per second on my laptop
- Learn
  - Limits Clause
  - Save Exceptions
  - Partial Collections
  - Sparse Collections
  - In Indices Of Clause

```
CREATE OR REPLACE PROCEDURE fast_way AUTHID CURRENT_USER IS
 TYPE myarray IS TABLE OF parent%ROWTYPE;
 l_data myarray;

 CURSOR r IS
 SELECT part_num, part_name
 FROM parent;

 BatchSize CONSTANT POSITIVE := 1000;
BEGIN
  OPEN r;
  LOOP
    FETCH r BULK COLLECT INTO l_data LIMIT BatchSize;

    FOR j IN 1 .. l_data.COUNT LOOP
      l_data(j).part_num := l_data(j).part_num * 10;
    END LOOP;

    FORALL i IN 1..l_data.COUNT
    INSERT INTO child VALUES l_data(i);

    EXIT WHEN l_data.COUNT < BatchSize;
  END LOOP;
  COMMIT;
  CLOSE r;
END fast_way;
/
```

FORALL INSERTs (2:3)

- Use this syntax to greatly enhance performance but be sure you understand the concept of DIRECT LOAD INSERTs
- With this syntax I can insert 500,000 rows per second on my laptop
- Learn
  - Limits Clause
  - Save Exceptions
  - Partial Collections
  - Sparse Collections
  - In Indices Of Clause

```
CREATE OR REPLACE PROCEDURE fast_way AUTHID CURRENT_USER IS
 TYPE PartNum IS TABLE OF parent.part_num%TYPE
 INDEX BY BINARY_INTEGER;

 pnum_t PartNum;

 TYPE PartName IS TABLE OF parent.part_name%TYPE
 INDEX BY BINARY_INTEGER;

 pnam_t PartName;
BEGIN
  SELECT part_num, part_name
  BULK COLLECT INTO pnum_t, pnam_t
  FROM parent;

  FOR i IN pnum_t.FIRST .. pnum_t.LAST LOOP
    pnum_t(i) := pnum_t(i) * 10;
  END LOOP;

  FORALL i IN pnum_t.FIRST .. pnum_t.LAST
  INSERT INTO child
  (part_num, part_name)
  VALUES
  (pnum_t(i), pnam_t(i));
  COMMIT;
END fast_way;
/
```

FORALL INSERTs (3:3)

- Use this syntax to greatly enhance performance but be sure you understand the concept of DIRECT LOAD INSERTs
- With this syntax I can insert 500,000 rows per second on my laptop
- Learn
  - Limits Clause
  - Save Exceptions
  - Partial Collections
  - Sparse Collections
  - In Indices Of Clause

```
CREATE OR REPLACE PROCEDURE fast_way AUTHID CURRENT_USER IS
 TYPE parent_rec IS RECORD (
 part_num   dbms_sql.number_table,
 part_name  dbms_sql.varchar2_table);

 p_rec parent_rec;

 CURSOR c IS
 SELECT part_num, part_name FROM parent;

 l_done BOOLEAN;
BEGIN
  OPEN c;
  LOOP
    FETCH c BULK COLLECT INTO p_rec.part_num, p_rec.part_name
    LIMIT 500;
    l_done := c%NOTFOUND;

    FOR i IN 1 .. p_rec.part_num.COUNT LOOP
      p_rec.part_num(i) := p_rec.part_num(i) * 10;
    END LOOP;

    FORALL i IN 1 .. p_rec.part_num.COUNT
    INSERT INTO child
    (part_num, part_name)
    VALUES
    (p_rec.part_num(i), p_rec.part_name(i));

    EXIT WHEN (l_done);
  END LOOP;
  COMMIT;
  CLOSE c;
END fast_way;
/
```

FORALL MERGE Inserts

- Use this syntax to execute a MERGE statement using data in an array data (most likely selected using BULK COLLECT)

```
CREATE OR REPLACE PROCEDURE forall_merge AUTHID CURRENT_USER IS
 TYPE ridVal IS TABLE OF forall_tgt.rid%TYPE
 INDEX BY BINARY_INTEGER;
 l_data ridVal;
BEGIN
  SELECT rid BULK COLLECT INTO l_data
  FROM forall_src;

  FORALL i IN l_data.FIRST .. l_data.LAST
  MERGE INTO forall_tgt ft
  USING (
    SELECT rid
    FROM forall_src fs
    WHERE fs.rid = l_data(i)) al
  ON (al.rid = ft.rid)
  WHEN MATCHED THEN
    UPDATE SET upd = 'U'
  WHEN NOT MATCHED THEN
    INSERT (rid, ins, upd)
    VALUES (l_data(i), 'I', NULL);

  COMMIT;
END forall_merge;
/
```

LOB Insert

- When creating LOB objects be sure to use SecureFiles and be sure that you understand PCTVERSION, CHUNK, and other storage parameters
- Failure to understand how LOBs process undo can result in massive waste of space

```
DECLARE
 src_file BFILE;
 dst_file BLOB;
 lgh_file BINARY_INTEGER;
 retval   VARCHAR2(30);
BEGIN
  src_file := bfilename('CTEMP', 'sphere.mpg');

  INSERT INTO sct
  (rid, bcol)
  VALUES
  (1, EMPTY_BLOB())
  RETURNING bcol INTO dst_file;

  SELECT bcol
  INTO dst_file
  FROM sct
  WHERE rid = 1
  FOR UPDATE;

  dbms_lob.fileopen(src_file, dbms_lob.file_readonly);
  lgh_file := dbms_lob.getlength(src_file);
  dbms_lob.loadFromFile(dst_file, src_file, lgh_file);

  UPDATE sct
  SET bcol = dst_file
  WHERE rid = 1;

  dbms_lob.setContentType(dst_file, 'MPG Movie');
  retval := dbms_lob.getContentType(dst_file);
  dbms_output.put_line(retval);

  dbms_lob.fileclose(src_file);
END load_file;
/
```

DBMS_SQL Dynamic Inserts

- DBMS_SQL is the legacy implementation of dynamic SQL in the Oracle database introduced in version 7

```
CREATE OR REPLACE PROCEDURE single_row_insert(c1 NUMBER, c2 NUMBER, r OUT NUMBER) IS
 c NUMBER;
 n NUMBER;
BEGIN
  c := dbms_sql.open_cursor;

  dbms_sql.parse(c, 'INSERT INTO tab VALUES (:bnd1, :bnd2) ' || 'RETURNING c1*c2 into :bnd3', 2);

  dbms_sql.bind_variable(c, 'bnd1', c1);
  dbms_sql.bind_variable(c, 'bnd2', c2);
  dbms_sql.bind_variable(c, 'bnd3', r);

  n := dbms_sql.execute(c);

  dbms_sql.variable_value(c, 'bnd3', r); -- get value of outbind
  dbms_sql.close_cursor(c);
END single_row_insert;
/
```

Native Dynamic SQL Inserts

- Native Dynamic SQL has largely replaced DBMS_SQL as it is robust and more easily coded

```
BEGIN
  FOR i IN 1 .. 10000
  LOOP
    EXECUTE IMMEDIATE 'INSERT INTO t VALUES (:x)'
    USING i;
  END LOOP;
END;
/
```

RETURNING Clause with a Sequence

- Use this syntax to return values from an insert statement unknown to the program inserting the row

```
INSERT INTO <table_name>
 (column_list)
 VALUES
 (values_list)
 RETURNING <value_name>
 INTO <variable_name>;
```

```
DECLARE
 x emp.empno%TYPE;
 r rowid;
BEGIN
  INSERT INTO emp
  (empno, ename)
  VALUES
  (seq_emp.NEXTVAL, 'Morgan')
  RETURNING rowid, empno
  INTO r, x;

  dbms_output.put_line(r);
  dbms_output.put_line(x);
END;
/
```

RETURNING Clause with an Identify Column

- Use this syntax to return values from an insert statement unknown to the program inserting the row

```
CREATE TABLE idcoltab (
rec_id NUMBER(38) GENERATED ALWAYS AS IDENTITY,
coltxt VARCHAR2(30));

DECLARE
 rid idcoltab.rec_id%TYPE;
BEGIN
  INSERT INTO idcoltab
  (coltxt)
  VALUES
  ('Morgan')
  RETURNING rec_id
  INTO rid;

  dbms_output.put_line(rid);
END;
/
```

RETURNING Clause with Native Dynamic SQL

- Use this syntax to return values from an insert statement created using Native Dynamic SQL

```
DECLARE
 sql_stmt VARCHAR2(128);
 dno      dept_ret.deptno%TYPE;
BEGIN
  sql_stmt := 'INSERT INTO dept_ret (deptno, dname, location) ' ||
              'VALUES (seq.NEXTVAL, ''PERSONNEL'', ''SEATTLE'') ' ||
              'RETURNING deptno INTO :retval';
  EXECUTE IMMEDIATE sql_stmt RETURNING INTO dno;
  dbms_output.put_line(TO_CHAR(dno));
END;
/
```

Performance Tuning Insert Statements

Considerations

- Table structure
- Indexes
- Triggers
- It is always more efficient if you code it right once rather than making the database fix it thousands or millions of times

Too Many Columns

- Oracle claims that a table can contain up to 1,000 columns: It is not true. No database can do 1,000 columns no matter what their marketing claims may be
- The maximum number of real table columns is 255
- Break the 255 barrier and optimizations such as advanced and hybrid columnar compression no longer work
- A 1,000 column table is actually four segments joined together behind the scenes just as a partitioned table appears to be a single segment but isn't
- Be suspicious of any table with more than 50 columns. At 100 columns it is time to take a break and re-read the Codd-Date rules on normalization
- Think vertically not horizontally
- Be very suspicious of any table with column names in the form "SPARE1", "SPARE2", "..."
- The more columns a table has the more cpu is required when accessing columns to the right (as the table is displayed in a SELECT * query ... or at the bottom if the table is displayed by a DESCribe)

Column Ordering

- Computers are not humans and tables are not paper forms
- CBO's column retrieval cost
  - Oracle stores columns in variable length format
  - Each row is parsed in order to retrieve one or more columns
  - Each subsequently parsed column introduces a cost of 20 cpu cycles regardless of whether it is of value or not
- These tables will be accessed by person_id or state: No one will ever put the address2 column into the WHERE clause as a filter ... they won't filter on middle initial either

Common Design

```
CREATE TABLE customers (
person_id   NUMBER,
first_name  VARCHAR2(30) NOT NULL,
middle_init VARCHAR2(2),
last_name   VARCHAR2(30) NOT NULL,
address1    VARCHAR2(30),
address2    VARCHAR2(30),
city        VARCHAR2(30),
state       VARCHAR2(2));
```

CPU

```
 20
 40
 60
 80
100
120
140
160
```

Optimized Design

```
CREATE TABLE customers (
person_id   NUMBER,
last_name   VARCHAR2(30) NOT NULL,
state       VARCHAR2(2)  NOT NULL,
city        VARCHAR2(30) NOT NULL,
first_name  VARCHAR2(30) NOT NULL,
address1    VARCHAR2(30),
address2    VARCHAR2(30),
middle_init VARCHAR2(2));
```

Proof Column Ordering Matters in 12.1.0.2

```
CREATE TABLE read_test AS
SELECT *
FROM apex_040200.wwv_flow_page_plugs
WHERE rownum = 1;

SQL> explain plan for
  2  select * from read_test;

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |            |     1 |   214K|     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| READ_TEST |     1 |   214K|     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------

-- fetch value from column 1
Final cost for query block SEL$1 (#0) - All Rows Plan:
  Best join order: 1
  Cost: 2.0002  Degree: 1  Card: 1.0000  Bytes: 13
  Resc: 2.0002  Resc_io: 2.0000  Resc_cpu: 7271
  Resp: 2.0002  Resp_io: 2.0000  Resc_cpu: 7271

-- fetch value from column 193
Final cost for query block SEL$1 (#0) - All Rows Plan:
  Best join order: 1
  Cost: 2.0003  Degree: 1  Card: 1.0000  Bytes: 2002
  Resc: 2.0003  Resc_io: 2.0000  Resc_cpu: 11111
  Resp: 2.0003  Resp_io: 2.0000  Resc_cpu: 11111
```

It Still Matters in 12.2.0.1

- Run the query while performing a 10053 Level 1 Trace

```
ALTER SESSION SET tracefile_identifier = 'test_plan2';

ALTER SESSION SET EVENTS '10053 trace name context forever, level 1';

SELECT * FROM  audsys.aud$unified WHERE inst_id = 2 AND ROWNUM = 1;

SELECT * FROM  audsys.aud$unified WHERE con_id = 9 AND ROWNUM = 1;

ALTER SESSION SET EVENTS '10053 trace name context OFF';
```

Column 1

```
SINGLE TABLE ACCESS PATH
  Single Table Cardinality Estimation for AUD$UNIFIED[AUD$UNIFIED]
  SPD: Return code in qosdDSDirSetup: NOCTX, estType = TABLE

 kkecdn: Single Table Predicate:"AUD$UNIFIED"."INST_ID"=2
  Column (#1): INST_ID(NUMBER)
    AvgLen: 22 NDV: 0 Nulls: 81 Density: 0.000000
  Estimated selectivity: 0.010000 , col: #1
  Table: AUD$UNIFIED  Alias: AUD$UNIFIED
    Card: Original: 81.000000  Rounded: 1  Computed: 0.000000  Non Adjusted: 0.000000
  Scan IO  Cost (Disk) =   4.000000
  Scan CPU Cost (Disk) =   85364.400000
  Cost of predicates:
    io = NOCOST, cpu = 50.000000, sel = 0.010000 flag = 2048  ("AUD$UNIFIED"."INST_ID"=2)
  Total Scan IO  Cost  =   4.000000 (scan (Disk))
                       + 0.000000 (io filter eval) (= 0.000000 (per row) * 81.000000 (#rows))
                     =   4.000000
   Total Scan CPU  Cost =   85364.400000 (scan (Disk))
                       + 4050.000000 (cpu filter eval) (= 50.000000 (per row) * 81.000000 (#rows))
                           =   89414.400000
  Access Path: TableScan
    Cost:  4.002375  Resp: 4.002375  Degree: 0
      Cost_io: 4.000000  Cost_cpu: 89414
      Resp_io: 4.000000  Resp_cpu: 89414
  Best:: AccessPath: TableScan
         Cost: 4.002375  Degree: 1  Resp: 4.002375  Card: 0.000000  Bytes: 0.000000
```

Column 101

```
SINGLE TABLE ACCESS PATH
  Single Table Cardinality Estimation for AUD$UNIFIED[AUD$UNIFIED]
  SPD: Return code in qosdDSDirSetup: NOCTX, estType = TABLE

 kkecdn: Single Table Predicate:"AUD$UNIFIED"."CON_ID"=9
  Column (#101): CON_ID(NUMBER)
    AvgLen: 22 NDV: 0 Nulls: 81 Density: 0.000000
  Estimated selectivity: 0.010000 , col: #101
  Table: AUD$UNIFIED  Alias: AUD$UNIFIED
    Card: Original: 81.000000  Rounded: 1  Computed: 0.000000  Non Adjusted: 0.000000
  Scan IO  Cost (Disk) =   4.000000
  Scan CPU Cost (Disk) =   245364.400000
  Cost of predicates:
    io = NOCOST, cpu = 50.000000, sel = 0.010000 flag = 2048  ("AUD$UNIFIED"."CON_ID"=9)
  Total Scan IO  Cost  =   4.000000 (scan (Disk))
                       + 0.000000 (io filter eval) (= 0.000000 (per row) * 81.000000 (#rows))
                     =   4.000000
   Total Scan CPU  Cost =   245364.400000 (scan (Disk))
                       + 4050.000000 (cpu filter eval) (= 50.000000 (per row) * 81.000000 (#rows))
                           =   249414.400000
  Access Path: TableScan
    Cost:  4.006624  Resp: 4.006624  Degree: 0
      Cost_io: 4.000000  Cost_cpu: 249414
      Resp_io: 4.000000  Resp_cpu: 249414
  Best:: AccessPath: TableScan
         Cost: 4.006624  Degree: 1  Resp: 4.006624  Card: 0.000000  Bytes: 0.000000
```

How much are you willing to pay for accessing column 101?

Aliasing and Fully Qualified Names

- When you do not use fully qualified names Oracle must do the work for you
- You write code once ... the database executes it many times

```
SELECT DISTINCT s.srvr_id
FROM servers s, serv_inst i
WHERE s.srvr_id = i.srvr_id;

SELECT DISTINCT s.srvr_id
FROM uwclass.servers s, uwclass.serv_inst i
WHERE s.srvr_id = i.srvr_id;
```

Implicit Casts

- Code that does not correctly define data types will either fail to run or run very inefficiently

  The following example shows both the correct way and the incorrect way to work with dates. The correct way is to perform an explicit cast

```
SQL> create table t (
  2  datecol date);

Table created.

SQL> insert into t values ('01-JAN-2017');

1 row created.

SQL> insert into t values (TO_DATE('01-JAN-2017'));

1 row created.
```

Jonathan Lewis' Rules for Hints

1. Don't

2. If you must use hints, then assume you've used them incorrectly

3. On every patch or upgrade to Oracle, assume every piece of hinted SQL is going to do the wrong thing

   Because of (2) above; you've been lucky so far, but the patch/upgrade lets you discover your mistake

4. Every time you apply some DDL to an object that appears in a piece of hinted SQL assume that the hinted SQL is going to do the wrong thing

   Because of (2) above; you've been lucky so far, but the structural change lets you discover your mistake

APPEND Hint

- The APPEND hint enables direct-path INSERT if the database is running in serial mode. The database is in serial mode if you are not using Enterprise Edition. Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode
- In direct-path INSERT data is appended above the high-water mark potentially improving performance

```
INSERT /*+ APPEND */ INTO t
SELECT * FROM servers;
```

APPEND_VALUES Hint

- Use this new 12c hint instructs the optimizer to use direct-path INSERT with the VALUES clause
- If you do not specify this hint, then conventional INSERT is used
- This hint is only supported with the VALUES clause of the INSERT statement
- If you specify it with an insert that uses the subquery syntax it is ignored

```
SQL> EXPLAIN PLAN FOR
  2  INSERT INTO t
  3  VALUES
  4  ('XYZ');

SQL> SELECT * FROM TABLE(dbms_xplan.display);

---------------------------------------------------------------------------------
| Id  | Operation                | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | INSERT STATEMENT         |      |     1 |   100 |     1   (0)| 00:00:01 |
|   1 |  LOAD TABLE CONVENTIONAL | T    |       |       |            |          |
---------------------------------------------------------------------------------

SQL> EXPLAIN PLAN FOR
  2  INSERT /*+ APPEND_VALUES */ INTO t
  3  VALUES
  4  ('XYZ');

SQL> SELECT * FROM TABLE(dbms_xplan.display);

-------------------------------------------------------------------------
| Id  | Operation        | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------
|   0 | INSERT STATEMENT |      |     1 |   100 |     1   (0)| 00:00:01 |
|   1 |  LOAD AS SELECT  | T    |       |       |            |          |
|   2 |   BULK BINDS GET |      |       |       |            |          |
-------------------------------------------------------------------------
```

CHANGE_DUPKEY_ERROR_INDEX Hint

- Use this hint to unambiguously identify a unique key violation for a specified set of columns or for a specified index
- When a unique key violation occurs for the specified index, an ORA-38911 error is reported instead of an ORA-00001

```
INSERT /*+ CHANGE_DUPKEY_ERROR_INDEX(T,TESTCOL) */ INTO t
(testcol)
VALUES
('A');
```

IGNORE_ON_DUPKEY_INDEX Hint

- This hint applies only to single-table INSERT operations
- It causes the statement to ignore a unique key violation for a specified set of columns or for a specified index
- When a unique key violation is encountered, a row-level rollback occurs and execution resumes with the next input row
- If you specify this hint when inserting data with DML error logging enabled, then the unique key violation is not logged and does not cause statement termination

```
INSERT /*+ IGNORE_ROW_ON_DUPKEY_INDEX(T,UC_T_TESTCOL)) */ INTO t
(testcol)
VALUES
(1);
```

DBMS_ERRLOG (1:2)

- Provides a procedure that enables creating an error logging table so that DML operations can continue after encountering errors rather than performing an abort and rollback
- Tables with LONG, CLOB, BLOB, BFILE, and ADT data types are not supported
- LOG ERRORS effectively it turns array processing into single row processing, so it adds an expense at the moment of inserting, even though it saves you the overhead of an array rollback if a duplicate gets into the data (Jonathan Lewis)

```
CREATE TABLE t AS
SELECT *
FROM all_tables
WHERE 1=2;

ALTER TABLE t
ADD CONSTRAINT pk_t
PRIMARY KEY (owner, table_name)
USING INDEX;

ALTER TABLE t
ADD CONSTRAINT cc_t
CHECK (blocks < 11);

INSERT /*+ APPEND */ INTO t
SELECT *
FROM all_tables;
```

DBMS_ERRLOG (2:2)

```
exec
dbms_errlog.create_error_log('T');

desc err$_t

INSERT /*+ APPEND */ INTO t
SELECT *
FROM all_tables
LOG ERRORS
REJECT LIMIT UNLIMITED;

SELECT COUNT(*) FROM t;

COMMIT;

SELECT COUNT(*) FROM t;

SELECT COUNT(*) FROM err$_t;

set linesize 121
col table_name format a30
col blocks format a7
col ora_err_mesg$ format a60

SELECT ora_err_mesg$, table_name,
blocks
FROM err$_t;
```

DBMS_STATS: Statistics

- System Stats
- Fixed Object Stats
- Dictionary Stats
- Set stats for new partitions so that when inserts take place the optimizer knows what you are inserting

```
SQL> exec dbms_stats.gather_system_stats('INTERVAL', 15);

SQL> SELECT * FROM sys.aux_stats$;

SNAME           PNAME                PVAL1 PVAL2
--------------- --------------- ---------- -----------------
SYSSTATS_INFO   STATUS                     COMPLETED
SYSSTATS_INFO   DSTART                     05-27-2015 09:45
SYSSTATS_INFO   DSTOP                      05-27-2015 09:51
SYSSTATS_INFO   FLAGS                    0
SYSSTATS_MAIN   CPUSPEEDNW            3010
SYSSTATS_MAIN   IOSEEKTIM               10
SYSSTATS_MAIN   IOTFRSPEED            4096
SYSSTATS_MAIN   SREADTIM             3.862
SYSSTATS_MAIN   MREADTIM             1.362
SYSSTATS_MAIN   CPUSPEED              2854
SYSSTATS_MAIN   MBRC                    17
SYSSTATS_MAIN   MAXTHR
SYSSTATS_MAIN   SLAVETHR
```

DBMS_STATS: Processing Rate (1:2)

- Processing Rate collection is new as of version 12cR1
- Besides the amount of work the optimizer also needs to know the HW characteristics of the system to understand how much time is needed to complete that amount of work
- Consequently, the HW characteristics describe how much work a single process can perform on that system, these are expressed as bytes per second and rows per second and are called processing rates
- As they indicate a system's capability it means you will need fewer processes (which means less DOP) for the same amount of work as these rates go higher; the more powerful a system is, the less resources you need to process the same statement in the same amount of time
- Processing rates are collected manually

```
SQL> exec dbms_stats.gather_processing_rate('START', 20);

SQL> SELECT operation_name, manual_value, calibration_value, default_value
  2  FROM v$optimizer_processing_rate
  3  ORDER BY 1;
```

DBMS_STATS: Processing Rate (2:2)

```
OPERATION_NAME            MANUAL_VAL CALIBRATIO DEFAULT_VA
------------------------- ---------- ---------- ----------
AGGR                                            1000.00000
ALL                                              200.00000
CPU                                              200.00000
CPU_ACCESS                                       200.00000
CPU_AGGR                                         200.00000
CPU_BYTES_PER_SEC                               1000.00000
CPU_FILTER                                       200.00000
CPU_GBY                                          200.00000
CPU_HASH_JOIN                                    200.00000
CPU_IMC_BYTES_PER_SEC                           2000.00000
CPU_IMC_ROWS_PER_SEC                            2000000.00
CPU_JOIN                                         200.00000
CPU_NL_JOIN                                      200.00000
CPU_RANDOM_ACCESS                                200.00000
CPU_ROWS_PER_SEC                             1000000.00000
CPU_SEQUENTIAL_ACCESS                            200.00000
CPU_SM_JOIN                                      200.00000
CPU_SORT                                         200.00000
HASH                                             200.00000
IO                                               200.00000
IO_ACCESS                                        200.00000
IO_BYTES_PER_SEC                                 200.00000
IO_IMC_ACCESS                                   1000.00000
IO_RANDOM_ACCESS                                 200.00000
IO_ROWS_PER_SEC                              1000000.00000
IO_SEQUENTIAL_ACCESS                             200.00000
MEMCMP                                           500.00000
MEMCPY                                          1000.00000

SQL> exec dbms_stats.set_processing_rate('IO', 100);
```

INSERT Statement Most Common Error

- If you do not name columns DDL can break your statement and not doing so will use a less efficient code path

```
INSERT INTO <table_name>
~~(<comma_separated_column_name_list>)~~
VALUES
(<comma_separated_value_list>);
```

```
CREATE TABLE state (
state_abbrev VARCHAR2(2),
state_name   VARCHAR2(30),
city_name    VARCHAR2(30));

INSERT INTO state
(state_abbrev, state_name)
VALUES
('NY', 'New York');

INSERT INTO state
VALUES
('NY', 'New York');
```

Wrap Up

Conclusion

- How comfortable are you with your knowledge of UPDATE and DELETE statements?
- The most important principle in INSERT statements, and everything else in Oracle is "do the least work"
  - Minimize CPU utilization
  - Minimize I/O
    - Take the load off the storage array
    - Off the HBA cards
    - Off the SAN switch
    - Off the Fibre
  - Minimize network utilization
    - Bandwidth
    - Round Trips
  - Minimize your memory footprint

```
*
ERROR at line 1:
ORA-00028: your session has been killed
```

Thank You

Daniel A. Morgan
email: dmorgan@forsythe.com
mobile: +1 206-669-2949