Oracle Insert Statements
for DBAs and Developers
Oracle ACE Director
Consultant to Harvard University
University of Washington Oracle Instructor, ret.

The Morgan of Morgan’s Library on the web
www.morganslibrary.org/library.html

- Executive Board Member: Vancouver OUG

- Upcoming Presentations & Events
  - September: Oracle OpenWorld
  - October: Croatia Oracle Users Group
  - October: Slovenian Oracle Users Group
  - November APAC Tour: Thailand & New Zealand

- 10g, 11g, & 12c Beta Tester
The Legend of Mad Dog Morgan?

- Oracle ACE 2007
- Mad Dog Morgan

"Violent escapist entertainment." — NY Times

"DENNIS HOPPER’S performance as Dan Morgan is a tour de force... it’s powerful." — Variety

Mad Dog Morgan
The library is a spam-free on-line resource with code demos for DBAs and Developers. If you would like to see Oracle database functionality added to the library ... just email us. Oracle 12.1.0.1.0 has been released and content will start showing up every day for weeks.
cd $MORGAN_HOME
cd $MORGAN_BASE/San_Francisco
My Sled Meets Larry's
Travel Log: Peru 2010
Travel Log: Galapagos 2014
IGGOUG: The New Users Group On The Block

www.iggoug.org
If you are coming to OpenWorld and are interested in GoldenGate there is a special session on Tuesday evening at 6:30pm ... location to be announced on the IGGOUG website within a week.

This will be the place to meet the GG product management and help form the new group.
To become a member of the International GoldenGate OUG, you must meet one or more of the following qualifications.

1. You have an interest in Oracle’s GoldenGate software
2. You have an interest in database replication
3. You have an interest in database platform migration
4. You have an interest in database version migration
5. You have been asked to be a presenter an an IGGOUG event
6. You are willing to buy the first round for everyone else that does meet one or more of the above qualifications

Want to be fully hardware, software, complete? Join us at our 2015 conference

we will fly from San Francisco, over the bridge, on our way to:

Bora Bora, Tahiti: 28 October - 1 November, 2015

the conference will include time for a "personal software" refresh
Disclaimer

- This room is an unsafe harbour
- 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

- This presentation is about capabilities built into the Oracle database that Oracle has never promoted but that can have a substantial impact on database performance

This disclaimer has not been approved by Oracle Legal
Why Is An ACE Director 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
- 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
What Is SQL DML?

- DML stands for **Data Manipulation Language**
- DML is a direct reference to the following SQL statements
  - INSERT
  - UPDATE
  - DELETE
  - MERGE
SQL INSERT Statement Topics (1:2)

- 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 Insert
SQL INSERT Statement Topics (2:2)

- 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
SQL Insert Statements
Basic INSERT Statement (1:2)

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

```sql
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;
```
Use this syntax to perform inserts into multiple columns in a heap, global temporary, IOT, or most partitioned tables.

```sql
INSERT INTO state
(state_abbrev, state_name)
VALUES
('NY', 'New York');
COMMIT;
SELECT * FROM state;
```
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
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;
```
• Use this syntax to unconditionally insert data into multiple tables

• Note that some columns go into one table ... others into both

```
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;
```
With "ALL", the default value, the database evaluates each WHEN sequentially

```sql
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>;
```

```sql
INSERT ALL
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;
```
With FIRST the database evaluates each \texttt{WHEN} clause in the order in which it appears in the statement

\begin{verbatim}
INSERT FIRST
WHEN \texttt{<condition>} THEN
INTO \texttt{<table_name>} VALUES <column_name_list>)
INTO \texttt{<table_name>} VALUES <column_name_list>)
...
<SELECT Statement>;
\end{verbatim}

\begin{verbatim}
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;
\end{verbatim}
Use this syntax to INSERT rows into one table as part of a SELECT statement from itself or a different table or tables

```
INSERT INTO <table_name> <SELECT Statement>;
```

```
CREATE TABLE state (  
  zip_code VARCHAR2(5) NOT NULL,  
  state_abbrev VARCHAR2(2) NOT NULL,  
  city_name VARCHAR2(30));
```

```
(  
  INSERT INTO (  
    SELECT deptno, dname, loc  
    FROM dept)  
  VALUES (99, 'TRAVEL', 'SEATTLE');
```
Use this syntax to limit inserted rows to those that pass CHECK OPTION validation

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

```
INSERT INTO ( 
SELECT deptno, dname, 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

```sql
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;
```

<table>
<thead>
<tr>
<th>CON_ID</th>
<th>OWNER</th>
<th>INS</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ORDSYS</td>
<td>NO</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>ORDSYS</td>
<td>YES</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>SYS</td>
<td>NO</td>
<td>45190</td>
</tr>
<tr>
<td>2</td>
<td>SYS</td>
<td>YES</td>
<td>22415</td>
</tr>
<tr>
<td>2</td>
<td>SYSTEM</td>
<td>NO</td>
<td>172</td>
</tr>
<tr>
<td>2</td>
<td>SYSTEM</td>
<td>YES</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>WMSYS</td>
<td>NO</td>
<td>736</td>
</tr>
<tr>
<td>2</td>
<td>WMSYS</td>
<td>YES</td>
<td>160</td>
</tr>
</tbody>
</table>
INSERTing Into An Editioning View

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

```sql
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;

<table>
<thead>
<tr>
<th>VIEW_NAME</th>
<th>TABLE_NAME</th>
<th>EDITION_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>AIRPLANES</td>
<td>ORA$BASE</td>
</tr>
<tr>
<td>TEST</td>
<td>AIRPLANES</td>
<td>DEMO_ED</td>
</tr>
</tbody>
</table>
```
INSERTing Into A Partitioned Table

- With HASH, LIST, and RANGE partitioning any INSERT statement will work
- With Partition by SYSTEM you must think more clearly

```sql
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);
```
Virtual columns will appear in a DESCRIBE statement but you cannot insert into them directly.

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

```sql
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
WHERE table_name like '%VIS';

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

SELECT * FROM invis;

SELECT rid, testcol FROM invis;
```
CREATE TABLE AS INSERTS

- Use this syntax to create a new table as the result of a SELECT statement

```
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

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

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

INSERT INTO department  
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 is conditioned on the results of a SELECT statement

```
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);
```
PL/SQL Insert Statements
**Record Inserts**

- Use this syntax to insert based on an array that matches the target table rather than named individual columns

```
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

```sql
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 LOOP; /
```
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

```sql
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;
/```

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;
/

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
Use this syntax to perform MERGE statements using array data

```sql
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;
/
```
When creating LOB objects be sure to use SecureFiles and be sure that you understand PCTVERSION, CHUNK, and other storage parameters.
DBMS_SQL Dynamic Inserts

- **DBMS_SQL** is the legacy implementation of dynamic SQL in the Oracle database introduced in version 7.3.4.

```sql
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 has largely replaced DBMS_SQL as it is robust and more easily coded.

```sql
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

```sql
CREATE TABLE idcoltab (  
  rec_id NUMBER 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; /
```
Performance Tuning Insert Statements
Oracle claims that a table can contain up to 1,000 columns: It is not true.
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 tables joined together seamlessly 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 reread 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)
Column Ordering (1:3)

- 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
Column Ordering (2:3)

- 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

```
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));
```

```
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 order matters

```sql
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
<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>214K</td>
<td>2 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS FULL</td>
<td>READ_TEST</td>
<td>1</td>
<td>214K</td>
<td>2 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

-- 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  Resp: 2.0002  Resc_cpu: 7271
    Resc_io: 2.0000  Resp_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  Resp: 2.0003  Resc_cpu: 11111
    Resc_io: 2.0000  Resp_cpu: 11111
```
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

```sql
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-2012');

1 row created.

SQL> insert into t values (TO_DATE('01-JAN-2012'));

1 row created.
```
**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 (1:2)

- 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.
### APPEND_VALUES Hint (2:2)

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

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

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INSERT STATEMENT</td>
<td></td>
<td>1</td>
<td>100</td>
<td>1   (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>LOAD TABLE CONVENTIONAL</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BULK BINDS GET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INSERT STATEMENT</td>
<td></td>
<td>1</td>
<td>100</td>
<td>1   (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>LOAD AS SELECT</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BULK BINDS GET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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).

```sql
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;
```
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;
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.

```sql
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

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

- 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.set_table_stats(USER, 'EMP', numrows=>1000000, numblks=>10000, avgrlen=>74);
exec dbms_stats.set_index_stats(USER, 'ix_emp_deptno', numrows=>1000000, numblks=>1000, numdist=>10000, clstfct=>1);
exec dbms_stats.set_column_stats(USER, 'emp', 'deptno', distcnt=>10000);
exec dbms_stats.set_table_stats(USER, 'dept', numrows=>100, numblks=>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

```
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');
```

```
INSERT INTO <table_name>  
(<comma_separated_column_name_list>)  
VALUES  
(<comma_separated_value_list>);
```
Insert Statements for DBAs and Developers

Presented: NYOUG, 12 September, 2014
Conclusion

- How comfortable are you with your knowledge of UPDATE and DELETE statements?
- The most important principle in INSERT statements, and anything else in Oracle is "do the least work"
  - Minimize CPU utilization
  - Minimize I/O
  - Minimize network utilization
    - Bandwidth
    - Round Trips
  - Minimize your memory footprint
The following was written by Jonathan Lewis: I've never heard better advice

Rules for Hinting

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.
Questions

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

Feel free to ask questions now or contact me at PTC
daniel.morgan@perftuning.com / +1 206-669-2949