**Some Topics in Advanced SQL**

1. CASE Statement
2. DECODE Statement (page 2)
3. CONCATENATE (page 3)
4. Sequence and CURRVAL, NEXTVAL (page 4)
5. DEFINE statement (page 5)
6. ACCEPT, PROMPT, & variable; (page 7)
7. ROWNUM (Top N) (Pseudocolumns) (page 8)
8. WITH Clause (page 9)
9. ROWID (Pseudocolumns) (page 11)
10. Hierarchical Queries (LEVEL) (Pseudocolumns) (page 12)
11. MERGE Statement (page 17)
12. SQL Creating SQL (page 18)
13. Creating Datafiles for Other Programs (page 21)
14. **CASE Statement**

This part will be discussed in class #2 (or #3) with Control statement.

case\_expression ::=

**CASE** {simple\_case\_expression | searched\_case\_expression}

[else else\_expr] **END**

simple\_case\_expression ::=

expr WHEN comparison\_expr THEN return\_expr

[WHEN comparison\_expr THEN return\_expr]...

searched\_case\_expression ::=

WHEN condition THEN return\_expr

[WHEN condition THEN return\_expr]...

In a simple CASE expression, Oracle searches for the first WHEN ... THEN pair for which *expr* is equal to *comparison\_expr* and returns *return\_expr*. If none of the WHEN ... THEN pairs meet this condition, and an ELSE clause exists, then Oracle returns *else\_expr*. Otherwise, Oracle returns null. You cannot specify the literal NULL for all the *return\_expr*s and the *else\_expr*.

In a searched CASE expression, Oracle searches from left to right until it finds an occurrence of *condition* that is true, and then returns *return\_expr*. If no *condition* is found to be true, and an ELSE clause exists, Oracle returns *else\_expr*. Otherwise, Oracle returns null.

For the same query request as in “DECODE” section, the “CASE” statement could be as bellows:

SELECT ename,

CASE **job** WHEN 'ANALYST' THEN 'Tech'

WHEN 'CLERK' THEN 'Admin'

WHEN 'MANAGER' THEN 'Admin'

WHEN 'PRESIDENT' THEN 'Admin'

ELSE 'OTHER' END JOB,

sal

FROM EMP;

or

SELECT ename, (CASE job WHEN 'ANALYST' THEN 'Tech'

WHEN 'CLERK' THEN 'Admin'

WHEN 'MANAGER' THEN 'Admin'

WHEN 'PRESIDENT' THEN 'Admin'

ELSE 'OTHER' END) JOB, SAL

FROM EMP;

**or**

SELECT ename,

CASE WHEN job ='ANALYST' THEN 'Tech'

WHEN JOB in ('CLERK', 'MANAGER', 'PRESIDENT') THEN 'Admin'

ELSE 'OTHER' END JOB,

SAL

FROM EMP;

1. **DECODE Statement**

The **decode** statement is how you implement if-then logic in SQL\*Plus. Decode statement is a powerful function and can be used to implement logic within SQL\*Plus. The decode statement has the else clause, which can handle exceptions.

**DECODE** (expr, search, result [, search, result]... [, default])

DECODE compares *expr* to each *search* value one by one. If *expr* is equal to a *search*, then Oracle returns the corresponding *result*. If no match is found, then Oracle returns *default*. If *default* is omitted, then Oracle returns null.

Example:

SET PAGESIZE 50;

SELECT ENAME, DECODE (job, 'ANALYST', 'Tech',

'CLERK', 'Admin',

'MANAGER', 'Admin',

'PRESIDENT', 'Admin',

'OTHER') JOB, sal

FROM EMP;

ENAME JOB SAL

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

KING Admin 5000

SMITH Admin 800

ALLEN OTHER 1600

WARD OTHER 1250

JONES Admin 2975

MARTIN OTHER 1250

BLAKE Admin 2850

CLARK Admin 2450

SCOTT Tech 3000

TURNER OTHER 1500

ADAMS Admin 1100

AMES Admin 950

FORD Tech 3000

MILLER Admin 1300

The logic expressed by this listing is as follows:

if job = 'ANALYST' then display 'Tech',

else if job = 'CLERK' then display 'Admin',

else if job = 'MANAGER' then display 'Admin',

else if job = 'PRESIDENT' then display 'Admin',

else display 'OTHER'.

1. **CONCATENATION** operator ||

Concatenates character strings data

The concatenation operator || appends one string operand to another.

SELECT last\_name || ', ' || first\_name AS Full\_Name

FROM employees

WHERE employee\_id = 152;

Result:

FULL\_NAME

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

Hall, Peter

The concatenation operator ignores null operands. Statement below gives the same result as last statement.

SELECT last\_name || ', ' ***|| null*** || first\_name AS Full\_Name

FROM employees

WHERE employee\_id = 152;

**CONCAT** (char1, char2)

It returns char1 concatenated with char2. Both char1 and char2 can be any of the datatypes char, chvarchar2,

This function is equivalent to the concatenation operator (||).

Below statement will give the same result as the above statement.

SELECT Concat(concat (last\_name, ', ' ), first\_name) AS Full\_Name

FROM employees

WHERE employee\_id = 152;

This example uses nesting to concatenate three character strings:

SELECT CONCAT(CONCAT(last\_name, '''s job category is '),

job\_id) "Job"

FROM employees

WHERE employee\_id = 152;

Job

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

Hall's job category is SA\_REP

The statement below gives the same output as above.

SELECT (last\_name || '''s job category is ' || job\_id) as Job

FROM employees

WHERE employee\_id = 152

Note:

'''s job category is '

the outer side two 's are a pair.

The second “'” is the escape for the third “'”.

1. **Sequence (autonumber), CURRVAL, NEXTVAL**

A sequence is an object in Oracle that is used to generate a number sequence. This can be useful when you need to create a unique number to act as a primary key. (SQL manual 15.1)

CREATE SEQUENCE sequence\_name

START WITH value -- the first sequence number

INCREMENT BY value -- the interval between sequence numbers. This integer

-- value can be any positive or negative integer, but it cannot be 0.

CACHE value;

DROP SEQUENCE sequence\_name;

**Example**

CREATE SEQUENCE employees\_seq

START WITH 100

INCREMENT BY 1

NOCACHE

NOCYCLE; /\* indicate that the sequence cannot generate more values after

reaching its maximum or minimum value. This is the default \*/

After a sequence is created, you can access its values in SQL statements with the

**CURRVAL** pseudocolumn, which returns the current value of the sequence, or the

**NEXTVAL** pseudocolumn, which increments the sequence and returns the new value.

To reference these pseudocolumns, use dot notation. For example,

*sequence\_name*.CURRVAL.

Each time you reference *sequence\_name*.NEXTVAL, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

**Example**

Assuming that you have created the SEQUENCE employees\_seq as in the example above.

DROP TABLE employees\_temp;

CREATE TABLE employees\_temp AS

SELECT employee\_id, first\_name, last\_name

FROM employees

WHERE employee\_id = 999;

SELECT \* from employees\_temp;

- result: no rows selected.

INSERT INTO employees\_temp (employee\_id, first\_name, last\_name)

VALUES (employees\_seq.NEXTVAL, 'Lynette', 'Smith');

SELECT \* from employees\_temp;

RESULT:

EMPLOYEE\_ID FIRST\_NAME LAST\_NAME

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

100 Lynette Smith

SELECT employees\_seq.CURRVAL from dual; -- result: 100

SELECT employees\_seq.NEXTVAL from dual; -- result: 101

UPDATE employees\_temp

SET employee\_id = employees\_seq.NEXTVAL

WHERE first\_name = 'Lynette' and last\_name = 'Smith';

SELECT \* from employees\_temp;

EMPLOYEE\_ID FIRST\_NAME LAST\_NAME

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

**102** Lynette Smith

INSERT INTO employees\_temp (employee\_id, first\_name, last\_name)

VALUES (employees\_seq.NEXTVAL, 'Mike', 'Ford');

SELECT \* from employees\_temp;

result:

EMPLOYEE\_ID FIRST\_NAME LAST\_NAME

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

102 Lynette Smith

103 Mike Ford

1. **DEFINE statement**

Syntax:

DEF[INE] [*variable*] | [*variable* = *text*]

In SQL\*Plus, it is possible to define variables that can then be used further on in the same program. Remember that a command file can have one or more SQL queries within it. By using the SQL\*Plus define statement, we are able to define a variable that can be referenced in all SQL statements in the command file. The following code is an example:

DEFINE c\_l = 'MANAGER' -- same if just use MANAGER without quote,

SELECT empno, deptno FROM EMP where job = '&c\_l';

old:SELECT empno, deptno FROM EMP where job = '&c\_l'

new:SELECT empno, deptno FROM EMP where job = 'MANAGER'

We have assigned the variable named "c\_l" the value MANAGER. Thus, when issuing a query, we can prefix the variable name with an ampersand (&) and enclose it in single quotes.

Example:

DEFINE mytable = emp -- same as "emp", 'emp'

SELECT \* FROM &mytable;

System will replace the &mytable with the value you defined (emp), then execute the query.

To **view** all variables that have been defined, type:

DEFINE.

To see the value of one variable, enter define followed by the name of the variable.

DEFINE mytable

DEFINE MYTABLE = "emp" (CHAR)

To clear the defined variables:

UNDEFINE variable\_name

UNDEFINE mytable

Note:

* All variables, regardless of the data type assigned, are character data.
* The **define** and **undefine** words may be abbreviated to **def** and **undef**.
* Some variables are predefined when SQL\*Plus starts. Enter DEFINE to see their definitions.

Example

DEFINE f = 7782

Even though you enter the number 7782, SQL\*Plus assigns a CHAR value to f consisting of four characters, 7, 7, 8 and 2.

But when comparing, the system will convert the datatype to the right one for the definition of the column.

SELECT empno, ename, deptno FROM emp WHERE empno = &f;

old:select \* from emp where empno = &f

new:select \* from emp where empno = 7782

1. **ACCEPT, PROMPT**

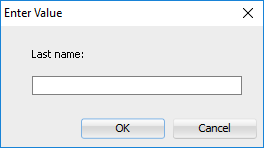
It reads a line of input and stores it in a given substitution variable.

ACCEPT variable [NUMBER|CHAR|DATE] [FORMAT format] [PROMPT 'message']

We may prefix the variable with an ampersand (&)

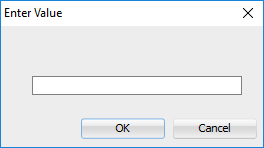
**Example**

ACCEPT l\_ename CHAR FORMAT A10 PROMPT "Last name:"



-- hint for input

ACCEPT l\_empno NUMBER FORMAT '9999' -- no hint in the popup window



ACCEPT l\_hired DATE FORMAT 'mm/dd/yyyy' prompt 'Hire date mm/dd/yyyy) '

-- &JOB\_CLERK asks for input for the “job”

INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)

VALUES (&l\_empno, '&l\_ename', '&JOB\_CLERK', 7839,

TO\_DATE('&l\_hired','mm/dd/yyyy'), 1222, null, 20);

/

The order of screen prompt windows: l\_ename, “enter value (for empno)”,

l\_hired, then &JOB\_CLERK.

-- SELECT \* FROM emp;

-- ROLLBACK;

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

----- ------- --------- ----- --------- -------- --------- ------- 1111 test CLERK 7839 01-JAN-10 1222 20

1. **ROWNUM Statement (Top N)**

For each row returned by a query, the ROWNUM *pseudocolumn* returns a number indicating the order in which Oracle selects the row from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second has 2, and so on. You can use ROWNUM to limit the number of rows returned by a query. (other pseudocolumns: nextval, rowid, level)

**Example**

List three employee names with their salary that get the top payments in the company

(this sample will be repeated and changed in the page 9, section #8 WITH clause).

SELECT ROWNUM as RANK, ename AS name, sal AS salary

FROM ( SELECT ename, sal

FROM emp

ORDER BY sal DESC)

WHERE ROWNUM <= 3;

Result:

RANK NAME SALARY

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

1 KING 5000

2 SCOTT 3000

3 FORD 3000

Note: the code below does not get same result as above code. It lists the first three employees according to the order of that table installed, then order these three records:

SELECT ename, sal

FROM emp

WHERE ROWNUM <= 3

ORDER BY sal DESC;

Result:

ENAME SAL

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

KING 5000

ALLEN 1600

SMITH 800

Select ename, sal from emp;

ENAME SAL

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

*KING 5000*

*SMITH 800*

*ALLEN 1600*

WARD 1250

JONES 2975

MARTIN 1250

BLAKE 2850

CLARK 2450

. . .

System picks up the first three rows, then sorts them by sal.

1. **WITH Clause**

Syntax:

WITH query\_name as subquery [, query\_name2 as subquery2]

The WITH clause, or subquery factoring clause, is part of the SQL-99 standard and was added into the Oracle SQL syntax in Oracle 9.2. The query\_name assigned to the sub-query is treated as though it was an inline view or table. You can specify this clause in any top-level SELECT statement and in most types of subqueries. The query\_name is visible to the main query and to all subsequent subqueries.

**Example**: In the 1st sample in ROWNUM on p8, the FROM clause includes a subquery:

FROM (SELECT ename, sal FROM emp ORDER BY sal DESC)

Here, we use WITH clause to declare a query\_name salrank as an inline table/view, thus the SELECT statement will be much easier to read.

With salrank as (SELECT ename, sal

FROM emp

ORDER BY sal DESC)

select ename, sal from salrank where rownum <= 3;

**Example,** declare two query\_names,

WITH

sum\_sal AS

( select sum(sal) total\_salaries from emp ),

number\_employees AS

( select count(\*) nbr\_employees from emp )

SELECT total\_salaries/nbr\_employees as Ave\_sal

From sum\_sal, number\_employees;

Result

AVE\_SAL

----------

2073.21429

Following coding will get same result as above:

Select sum(sal)/count(\*) as AVE\_SAL from emp;

**Example,**

WITH BIGDEPTS (Dept\_number) as

-- Dept\_number is not same as Deptno in sub\_query

( SELECT Deptno

FROM EMP

GROUP BY Deptno

HAVING COUNT(\*) > 4 )

SELECT Dept\_number

-- but this select column must match with column in query\_name

FROM BIGDEPTS;

RESULT:

DEPT\_NUMBER

-----------

30

20

WITH BIGDEPTS (Deptno) as

( SELECT Deptno

FROM EMP

GROUP BY Deptno

HAVING COUNT(\*) > 4 )

SELECT dept.Deptno, dname

FROM dept, BIGDEPTS

Where dept.deptno = BIGDEPTS.deptno

RESULT

DEPTNO DNAME

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

30 SALES

20 RESEARCH

WITH BIGDEPTS (Deptno, No\_Employees) as

( SELECT Deptno, count (\*) as Number\_Employees

FROM EMP

GROUP BY Deptno

HAVING COUNT(\*) > 4 )

SELECT dept.Deptno, dname, No\_Employees

FROM dept, BIGDEPTS

Where dept.deptno = BIGDEPTS.deptno ;

RESULT

- - - -

DEPTNO DNAME NUMBER\_EMPLOYEES

30 SALES 6

20 RESEARCH 5

**Example**

SELECT e.ename AS employee\_name,

dc.dept\_count AS emp\_dept\_count

FROM emp e

JOIN (SELECT deptno, COUNT(\*) AS dept\_count

FROM emp

GROUP BY deptno) dc

ON e.deptno = dc.deptno;

Using a WITH clause, this would look like the following.

WITH dept\_count AS (

SELECT deptno, COUNT(\*) AS dept\_count

FROM emp

GROUP BY deptno )

SELECT e.ename AS employee\_name, dc.dept\_count AS emp\_dept\_count

FROM emp e JOIN dept\_count dc ON e.deptno = dc.deptno;

1. **ROWID** (Pseudocolumn)

Oracle uses a ROWID datatype to store the address of every row in the database.

Oracle uses the ROWID values in the pseudocolumn ROWID internally for the construction of indexes

**Physical rowids** store the addresses of rows in ordinary tables (excluding index-organized tables), clustered tables, table partitions and subpartitions, indexes, and index partitions and subpartitions.

**Logical rowids** store the addresses of rows in index-organized tables.

Each table in an Oracle database internally has a **pseudocolumn** named ROWID.

It will not show in SELECT \* FROM ...;

But it will show when you explicitly name the pseudocolumn:

SELECT rowid, ename FROM emp;

**[Physical Rowids]**

* Provide the fastest possible access to a row, down to the specific block.
* As long as the row exists, its rowid does not change.

(unless the row is exported and imported using the Import and Export utilities)

* Every row in a nonclustered table is assigned a unique rowid

**[Format]** **extended rowid** (Oracle8i or higher)

Extended rowids use **a base 64** encoding of the physical address for each row selected. The encoding characters are A-Z, a-z, 0-9, +, and /.

An extended rowid has a four-piece format, OOOOOO FFF BBBBBB RRR:

OOOOOO: The **data object number** that identifies the database segment. Schema objects in the same segment, such as a cluster of tables, have the same data object number.

FFF: The tablespace-relative **datafile number** of the datafile that contains the row.

BBBBBB: The **data block** that contains the row. Block numbers are relative to their datafile, **not** tablespace.

RRR: The **row** in the block.

**[Using rowid to find the file structure]**

Rowids can be useful for revealing information about the physical storage of a table's data.

SELECT ROWID,

SUBSTR(ROWID,1,6) "OBJECT",

SUBSTR(ROWID,7,3) "FIL",

SUBSTR(ROWID,10,6) "BLOCK",

SUBSTR(ROWID,16,3) "ROW"

FROM scott.emp;

ROWID OBJECT FIL BLOCK ROW

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

AAAHW7AABAAAMUiAAA AAAHW7 AAB AAAMUi AAA

AAAHW7AABAAAMUiAAB AAAHW7 AAB AAAMUi AAB

.. .

AAAHW7AABAAAMUiAAN AAAHW7 AAB AAAMUi AAN

The following query of an extended rowid tells how many blocks contain rows of a given table:

SELECT COUNT(DISTINCT(SUBSTR(ROWID,10,6))) "BLOCKS" FROM data\_bids;

BLOCKS

----------

668

SELECT SUBSTR(ROWID,1,15) OFB,

COUNT(DISTINCT(SUBSTR(ROWID,16,3))) "# of ROWs in a block"

FROM data\_bids

Group by SUBSTR(ROWID,1,15);

OFB # of ROWs in a block

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

AAAHmJAABAAAMsD 251

. . .

Data block size : $ORACLE\_HOME\admin\Chtest\pfile\init.ora.6242003123641

Note: Chtest is the name of instance.

db\_block\_size=8192

db\_file\_multiblock\_read\_count=16

SQL> desc bids

Name Null? Type

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

USERID NOT NULL VARCHAR2(30)

ITEMID NOT NULL VARCHAR2(30)

PRICE NUMBER(9,2)

TIMESTAMP DATE

column "MAX|AVG|Min" format 99,999

SELECT MAX (COUNT(DISTINCT(SUBSTR(ROWID,16,3)))) "MAX|AVG|MIN"

FROM data\_bids

Group by SUBSTR(ROWID,1,15)

UNION

SELECT AVG (COUNT(DISTINCT(SUBSTR(ROWID,16,3))))

FROM data\_bids

Group by SUBSTR(ROWID,1,15)

UNION

SELECT Min (COUNT(DISTINCT(SUBSTR(ROWID,16,3))))

FROM data\_bids

Group by SUBSTR(ROWID,1,15);

1. **Hierarchical Queries**

Syntax:

Select ... from ... Where ...

[**START WITH** condition] **CONNECT BY** condition

* **START WITH** specifies the root row(s) of the hierarchy. If you omit this clause, then Oracle uses all rows in the table as root rows. The START WITH condition can contain a subquery, but it cannot contain a scalar subquery expression.
* **CONNECT BY** specifies the relationship between parent rows and child rows of the hierarchy. It must use the **PRIOR** operator to refer to the parent row. For example,

... PRIOR expr = expr

or

... expr = PRIOR expr

SELECT statements that contain hierarchical queries can contain the LEVEL pseudocolumn in the select list. LEVEL returns the value 1 for a root node, 2 for a child node of a root node, 3 for a grandchild, and so on.

Example

SELECT empno, ename name, mgr Manager#

FROM emp

Order by 3;

EMPNO NAME MANAGER#

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

7788 SCOTT 7566

7902 FORD 7566

7499 ALLEN 7698

7521 WARD 7698

7900 JAMES 7698

7844 TURNER 7698

7654 MARTIN 7698

7934 MILLER 7782

7876 ADAMS 7788

7566 JONES 7839

7782 CLARK 7839

7698 BLAKE 7839

7369 SMITH 7902

7839 KING

14 rows selected.

The tree structure is as below:

**King (7839)**

Jones (7566) Blake (7698) Clark (7782)

Scott Ford Allen Ward Martin Turner James Miller

7788 7902 7499 7521 7654 7844 7900 7934

Adams Smith

7876 7369

The following hierarchical query uses the CONNECT BY clause to define the relationship between employees and managers:

SELECT empno, ename name, mgr Manager#, LEVEL

FROM emp

start with mgr is null

CONNECT BY PRIOR empno = mgr;

The output:

EMPNO NAME MANAGER# LEVEL

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

**7839 KING 1**

7566 JONES 7839 2

7788 SCOTT 7566 3

7876 ADAMS 7788 4

7902 FORD 7566 3

7369 SMITH 7902 4

7698 BLAKE 7839 2

7499 ALLEN 7698 3

7521 WARD 7698 3

7654 MARTIN 7698 3

7844 TURNER 7698 3

7900 JAMES 7698 3

7782 CLARK 7839 2

7934 MILLER 7782 3

14 rows selected.

Example

SELECT empno, ename name, mgr Manager#, LEVEL

FROM emp

**Where ename != 'FORD'**

start with mgr is null

CONNECT BY PRIOR empno = mgr;

EMPNO NAME MANAGER# LEVEL

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

7839 KING 1

7566 JONES 7839 2

7788 SCOTT 7566 3

7876 ADAMS 7788 4

7369 SMITH 7902 4

7698 BLAKE 7839 2

7499 ALLEN 7698 3

7521 WARD 7698 3

7654 MARTIN 7698 3

7844 TURNER 7698 3

7900 JAMES 7698 3

7782 CLARK 7839 2

7934 MILLER 7782 3

13 rows selected.

REM Eliminate a row ‘FORD’ itself alone.

5th row

***“7902 FORD 7566 3”***

is deleted

Example

select empno, ename name, mgr manager#, level

from emp

start with mgr is null

connect by prior empno = mgr **and ename != 'ford'** ;

EMPNO NAME MANAGER# LEVEL

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

7839 KING 1

7566 JONES 7839 2

7788 SCOTT 7566 3

7876 ADAMS 7788 4

7698 BLAKE 7839 2

7499 ALLEN 7698 3

7521 WARD 7698 3

7654 MARTIN 7698 3

7844 TURNER 7698 3

7900 JAMES 7698 3

7782 CLARK 7839 2

7934 MILLER 7782 3

12 rows selected.

REM Eliminate a branch of ‘FORD’.

***7902 FORD 7566 3***

***7369 SMITH 7902 4***

Two rows are deleted.

Query to print out the hierarchy of the emp. Using LPAD ( char1, n [, char2] ).

(LPAD returns *char1*, left-padded to length *n* with the sequence of characters in *char2*)

SELECT lpad (' ', (level-1)\*4, ' ') || ename name

FROM emp

start with mgr is null

CONNECT BY PRIOR empno = mgr;

NAME

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

KING

JONES

SCOTT

ADAMS

FORD

SMITH

BLAKE

ALLEN

WARD

MARTIN

TURNER

JAMES

CLARK

MILLER

14 rows selected.

Column name format A30

Column "Tree PATH" format A40

SELECT lpad (' ', (level-1)\*4, ' ') || ename name,

Sys\_connect\_by\_path (ename, '--') "TREE PATH"

FROM emp

start with mgr is null

CONNECT BY PRIOR empno = mgr;

**Function:**

Sys\_connect\_by\_path (column, separator\_char)

NAME TREE PATH

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

KING --KING

JONES --KING--JONES

SCOTT --KING--JONES--SCOTT

ADAMS --KING--JONES--SCOTT--ADAMS

FORD --KING--JONES--FORD

SMITH --KING--JONES--FORD--SMITH

BLAKE --KING--BLAKE

ALLEN --KING--BLAKE--ALLEN

WARD --KING--BLAKE--WARD

MARTIN --KING--BLAKE--MARTIN

TURNER --KING--BLAKE--TURNER

JAMES --KING--BLAKE--JAMES

CLARK --KING--CLARK

MILLER --KING--CLARK--MILLER

14 rows selected.

1. **MERGE Statement**

Use the MERGE statement to select rows from one table for update or insertion into another table. The decision whether to *update* or *insert* into the target table is based on a condition in the ON clause.

Syntax:

MERGE INTO target\_table

USING source\_table -- table/view/result of subquery

ON search\_condition

WHEN MATCHED THEN

UPDATE SET col1 = value1, col2 = value2, ...

WHERE <update\_condition>

[DELETE WHERE <delete\_condition>]

WHEN NOT MATCHED THEN

INSERT (col1, col2,)

values (value1, value2,)

WHERE <insert\_condition>;

Basic logic:

Target table A, source table (view, or sub query) B

If condition is True, then update the row in A;

If condition is False, then insert a row into A;

Example

The following example creates a bonus table with a default bonus of 200. It then inserts into the bonus table those employees who do not receive commission. Finally, the Human Resources manager decides that all employees should receive a bonus. Those who got commission get a bonus of 15% of their salary. Those who do not get commission get an increase in their bonus equal to 1% of their salary. The MERGE statement implements these changes in one step:

CREATE TABLE emp\_bonus (empno NUMBER, bonus NUMBER (7,2) DEFAULT 200);

insert into emp\_bonus (empno) select empno from emp where comm is null;

column bonus format $999,999.99

select \* from emp\_bonus;

EMPNO BONUS

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

7369 $200.00

7566 $200.00

7698 $200.00

7782 $200.00

7788 $200.00

7839 $200.00

7844 $200.00

7876 $200.00

7900 $200.00

7902 $200.00

7934 $200.00

11 rows selected.

**MERGE INTO** emp\_bonus b

**USING**

(SELECT empno, sal, comm

FROM emp ) e

**ON** (b.empno = e.empno)

WHEN **MATCHED** THEN

UPDATE SET bonus = bonus + sal\*0.01

WHEN **NOT MATCHED** THEN

INSERT (empno, bonus) VALUES (e.empno, sal\*0.15);

* Merge into - define the target table you are updating or inserting into.
* Using - define the source of data to be updated or inserted. the source can be a table, view, or the result of a subquery.
* ON - define the condition upon which the MERGE operation either updates or inserts. For each row in the target table for which the search condition is true, Oracle updates the row based with corresponding data from the source table. If false, then Oracle inserts into the target table based on the source table row.
* When MATCHED then UPDATE - if the condition of the ON clause is true, then modify (UPDATE) the existing rows (on the target table).
* When NOT MATCHED then INSERT values - if the condition of the ON clause is false, then add (INSERT) new row(s) into the target table.

select \* from emp\_bonus;

EMPNO BONUS

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

7369 $208.00

7566 $229.75

7698 $228.50

7782 $224.50

7788 $230.00

7839 $250.00

7844 $215.00

7876 $211.00

7900 $209.50

7902 $230.00

7934 $213.00

7499 $240.00

7521 $187.50

7654 $187.50

14 rows selected.

Note. An UPSERT is a word combines Update and Insert, a statement that updates existing rows in a table or inserts new rows depending on a match condition.

1. **SQL Creating SQL**

There is no reason why you cannot get SQL to create SQL programs. In fact, this is a technique most DBAs find very useful. The following SQL codes will generate SQL files called xxx.sql that will contain SQL statements.

Example: write a script file to Grant "select" to public for all your tables.

A crt\_grt.sql file is as below:

set heading off

set pagesize 0

set feedback off

set echo off

set termout off

spool d:\grt.sql -- in Windows, set the file directory

select 'set pagesize 55' from dual;

select 'grant select on '||table\_name||' to public;'

from user\_tables where table\_name like '%EMP%';

-- for demo, added limited condition here

spool off

set termout on

set heading on

= = = =

The grt.sql will be as following

set pagesize 55

grant select on EMP to public;

grant select on EMP2 to public;

grant select on EMPLOYEE2 to public;

grant select on EMPLOYEES to public;

grant select on EMPLOYEE\_TEMP to public;

grant select on EMP\_CHANGING\_LOG to public;

grant select on EMP\_DEL\_LOG to public;

grant select on EMP\_LOG to public;

grant select on EMP\_SAL\_LOG to public;

grant select on EMP\_STMT to public;

Then you can run this file by typing *path*\grt.sql

(in Unix, using / )

As you can see, we have a program that creates another SQL program. The first select references the dual table. This is a table owned by Oracle user SYS that contains only one row. We use it in this case to load a setup command for the output file called grt.sql; we then follow up with the grant statements. You have all the elements you need to generate numerous SQL programs from SQL programs.

By the end of the quarter, I found there are too many “tables” (functions, procedures) in my schema. These lines of code will create a script file for dropping all my existing tables (of course, you can edit it before running)

set heading off

set pagesize 0

set feedback off

set echo off

set termout off

spool d:\drop\_tables.txt

select 'set pagesize 55' from dual;

select 'DROP TABLE ' || table\_name || ' ;'

from user\_tables;

spool off

set termout on

set heading on

Similar to drop the functions:

column object\_name format A50

select object\_name from user\_objects where object\_type in 'FUNCTION';

column object\_name format A50

select object\_name from user\_objects where object\_type in 'PROCEDURE';

set heading off

set pagesize 0

set feedback off

set echo off

set termout off

spool d:\drop\_functions.txt

select 'set pagesize 55' from dual;

select 'DROP FUNCTION ' || OBJECT\_NAME || ' ;'

from user\_objects where object\_TYPE = 'FUNCTION';

spool off

set termout on

set heading on

Check the output: open the file of drop\_functions.txt

set pagesize 55

DROP FUNCTION BONUS ;

DROP FUNCTION BONUSF ;

DROP FUNCTION BONUS\_CALCULATOR ;

DROP FUNCTION CELSIUS\_TO\_FAHRENHEIT ;

DROP FUNCTION COMPUTE\_BONUS ;

DROP FUNCTION CONVERT\_C\_F ;

DROP FUNCTION CTOF ;

DROP FUNCTION C\_TO\_F ;

DROP FUNCTION EMPCURTYP ;

DROP FUNCTION F ;

DROP FUNCTION FACTORIAL ;

DROP FUNCTION FF ;

DROP FUNCTION GET\_NUM\_OF\_EMPLOYEES ;

DROP FUNCTION HWF2 ;

DROP FUNCTION HW\_2 ;

DROP FUNCTION IS\_NUMBER ;

DROP FUNCTION IS\_NUMERIC ;

DROP FUNCTION RAISE ;

DROP FUNCTION TEST ;

DROP FUNCTION TEST\_ERR2 ;

When feeling safe, then I run:

@d:\drop\_functions.txt

set termout on

set heading on

set feedback on

set echo on

Similar to Procedures.

1. **Creating Datafiles for Other Programs**

One of the most common programs we see people write is one that will feed data from an Oracle database into a spreadsheet. Let's use SQL\*Plus to do this. Most spreadsheets require data in which each item is separated by a comma. Character-type data needs to be enclosed in single or double quotes. If we need single quotes, the following code will do this for us:

Rem \* Make spreadsheet data program

set heading off

set pagesize 0

set feedback off

set echo off

spool d:\out.txt

REM Notice you need to select four single quotes to put a single

REM quote in the output data file

select ''''||ename|| ''''||', '|| ''''||job|| ''''||', '||sal

from emp;

spool off

**Explain why we need Four single quotes**: the outside pair of single quotes is routine - anything inside that pair of single quotes will be treated as literal; to make a single quote as literal we prefix another single quote to this single quote - escape.

The content of out.dat file:

'SMITH', 'CLERK', 800

'ALLEN', 'SALESMAN', 1600

'WARD', 'SALESMAN', 1250

'JONES', 'MANAGER', 2975

'MARTIN', 'SALESMAN', 1250

'BLAKE', 'MANAGER', 2850

'CLARK', 'MANAGER', 2450

'SCOTT', 'ANALYST', 3000

'KING', 'PRESIDENT', 5000

'TURNER', 'SALESMAN', 1500

'ADAMS', 'CLERK', 1100

'JAMES', 'CLERK', 950

'FORD', 'ANALYST', 3000

'MILLER', 'CLERK', 1300

Since the single quote means something special to Oracle (i.e., it starts and ends a literal), you must **select** four of them together if you want the single quote character in your query results. *The second single quote plays a role of ESCAPE for the third single quote.*

If we need double quotes, the following will work:

set heading off

set pagesize 0

set feedback off

set echo off

spool out.dat

/\* Notice you place a double quote between two single quotes \*/

/\* to place a double quote in the output data file \*/

select '"'||ename||'","'||job||'",'||sal

from emp;

spool off

The output would be

"SMITH","CLERK",800

"ALLEN","SALESMAN",1600

"WARD","SALESMAN",1250

...

"FORD","ANALYST",3000

"MILLER","CLERK",1300

The following table discusses the lines in the program to make the spreadsheet data.

**Component Meaning**

set heading off Since you are creating a datafile, you do not want headings

set pagesize 0 You do not want page breaks, so you set this to zero for datafile output

set linesize 80 You set this to the size of the longest line in your output datafile.

set feedback off Suppresses SQL\*Plus from telling you how many rows are retrieved to satisfy the query

set echo off Tells SQL\*Plus not to echo the SQL statement as it runs.

spool out.dat Tells Oracle to send the results of the query to a file named out.dat.

spool off Tells Oracle to close the output datafile.