



# More SQL



# Ch.8

11e

## Database Systems Design, Implementation, and Management



# Chapter 8

## Advanced SQL

©2015 Cengage Learning. All Rights Reserved. May not be scanned, copied or duplicated, or posted to a publicly accessible website, in whole or in part.



# Objectives

## Learning Objectives

- In this chapter, the student will learn:
  - How to use the advanced SQL JOIN operator syntax
  - About the different types of subqueries and correlated queries
  - How to use SQL functions to manipulate dates, strings, and other data
  - About the relational set operators UNION, UNION ALL, INTERSECT, and MINUS



# Objectives

## Learning Objectives

- In this chapter, the student will learn:
  - How to create and use views and updatable views
  - How to create and use triggers and stored procedures
  - How to create embedded SQL



# The 'JOIN' operation

Recall that we looked at examples of joining - entries from two tables, and entries from a single table.

These joins were based on 'join conditions'.

It is also possible to join tables using the 'JOIN' keyword..



# JOIN conditions

Note that JOINS can be based on != (aka <>), >, <, >= and <= as well, in addition to equality. Eg. to list all students who will be getting a 'A' (uses two inequality comparisons indirectly):

```
// http://www.comp.nus.edu.sg/~ooibc/courses/sql/dml_query_join.htm
SELECT a.name, a.score
FROM student_scores a, grade_class b
WHERE b.grade = 'A' AND a.score BETWEEN
b.low_end AND b.high_end;
```

# SQL Join Operators

- Relational join operation merges rows from two tables and returns rows with one of the following
  - Natural join - Have common values in common columns
  - Equality or inequality - Meet a given join condition
  - **Outer join:** Have common values in common columns or have no matching values
- **Inner join:** Only rows that meet a given criterion are selected



# Ways to specify JOIN conditions

Table 8.1 - SQL Join Expression Styles

JOIN CLASSIFICATION	JOIN TYPE	SQL SYNTAX EXAMPLE	DESCRIPTION
CROSS	CROSS JOIN	SELECT * FROM T1, T2	Returns the Cartesian product of T1 and T2 (old style)
		SELECT * FROM T1 CROSS JOIN T2	Returns the Cartesian product of T1 and T2
INNER	Old-style JOIN	SELECT * FROM T1, T2 WHERE T1.C1=T2.C1	Returns only the rows that meet the join condition in the WHERE clause (old style); only rows with matching values are selected
	NATURAL JOIN	SELECT * FROM T1 NATURAL JOIN T2	Returns only the rows with matching values in the matching columns; the matching columns must have the same names and similar data types
	JOIN USING	SELECT * FROM T1 JOIN T2 USING (C1)	Returns only the rows with matching values in the columns indicated in the USING clause
	JOIN ON	SELECT * FROM T1 JOIN T2 ON T1.C1=T2.C1	Returns only the rows that meet the join condition indicated in the ON clause

Cengage Learning © 2015

©2015 Cengage Learning. All Rights Reserved. May not be scanned, copied or duplicated, or posted to a publicly accessible website, in whole or in part.



# Outer vs inner vs full ('both') JOINS

Table 8.1 - SQL Join Expression Styles

JOIN CLASSIFICATION	JOIN TYPE	SQL SYNTAX EXAMPLE	DESCRIPTION
OUTER	LEFT JOIN	<code>SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1=T2.C1</code>	Returns rows with matching values and includes all rows from the left table (T1) with unmatched values
	RIGHT JOIN	<code>SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T1.C1=T2.C1</code>	Returns rows with matching values and includes all rows from the right table (T2) with unmatched values
	FULL JOIN	<code>SELECT * FROM T1 FULL OUTER JOIN T2 ON T1.C1=T2.C1</code>	Returns rows with matching values and includes all rows from both tables (T1 and T2) with unmatched values

Cengage Learning © 2015



# Full (left+right outer) JOIN example

For example, the following query lists the product code, vendor code, and vendor name for all products and includes all product rows (products without matching vendors) as well as all vendor rows (vendors without matching products):

```
SELECT      P_CODE, VENDOR.V_CODE, V_NAME  
FROM        VENDOR FULL JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;
```

The SQL code and its results are shown in Figure 8.12.

FIGURE  
8.12

FULL JOIN results

The screenshot shows the Oracle SQL\*Plus interface with the title bar "Oracle SQL\*Plus". The command line displays:

```
SQL> SELECT P.CODE, VENDOR.V_CODE, V_NAME  
  2  FROM VENDOR FULL JOIN PRODUCT ON VENDOR.V_CODE = PRODUCT.V_CODE;
```

The output window shows the results of the query:

P_CODE	V_CODE	V_NAME
11QER/31	25595	Rubicon Systems
13-Q2/P2	21344	Gomez Bros.
14-Q1/L9	21944	Gomez Bros.
1546-QQ2	23119	Randssets Ltd.
1558-QW1	23119	Randssets Ltd.
2232/QTY	24288	ORDUA, Inc.
2232/QWE	24288	ORDUA, Inc.
2238/QPD	25595	Rubicon Systems
29180-HB	21226	Bryson, Inc.
54778-2T	21344	Gomez Bros.
89-WRE-Q	24288	ORDUA, Inc.
SM-18277	21225	Bryson, Inc.
SW-23116	21231	D&E Supply
VR3/TI3	25595	Rubicon Systems
	22567	Dome Supply
	21226	SuperLoo, Inc.
	24884	Brickman Bros.
	25501	Danai Supplies
	25443	B&H, Inc.
29114-AA		
PUC230RT		

At the bottom of the output window, it says "21 rows selected."



# 'SELECT' subqueries

SELECT SUBQUERY EXAMPLES	EXPLANATION
INSERT INTO PRODUCT SELECT * FROM P;	Inserts all rows from Table P into the PRODUCT table. Both tables must have the same attributes. The subquery returns all rows from Table P.
UPDATE PRODUCT SET P_PRICE = (SELECT AVG(P_PRICE) FROM PRODUCT) WHERE V_CODE IN (SELECT V_CODE FROM VENDOR WHERE V_AREACODE = '615')	Updates the product price to the average product price, but only for products provided by vendors who have an area code equal to 615. The first subquery returns the average price; the second subquery returns the list of vendors with an area code equal to 615.
DELETE FROM PRODUCT WHERE V_CODE IN (SELECT V_CODE FROM VENDOR WHERE V_AREACODE = '615')	Deletes the PRODUCT table rows provided by vendors with an area code equal to 615. The subquery returns the list of vendor codes with an area code equal to 615.

Cengage Learning © 2015

## Subqueries and Correlated Queries

- Subquery is a query inside another query
- Subquery can return:
  - One single value - One column and one row
  - A list of values - One column and multiple rows
  - A virtual table - Multicolumn, multirow set of values
  - No value - Output of the outer query might result in an error or a null empty set



# 'WHERE' subqueries

## WHERE Subqueries

- Uses inner SELECT subquery on the right side of a WHERE comparison expression
- Value generated by the subquery must be of a comparable data type
- If the query returns more than a single value, the DBMS will generate an error
- Can be used in combination with joins



# WHERE subquery example

FIGURE  
8.13

WHERE subquery example

```
Oracle SQLPlus
File Edit Search Options Help
SQL> SELECT P_CODE, P_PRICE FROM PRODUCT
  2 WHERE P_PRICE >= (SELECT AVG(P_PRICE) FROM PRODUCT);

P_CODE          P_PRICE
-----        -----
11QER/31      109.99
2292/QTV       109.92
2232/QWE       99.87
89-WRE-Q       256.99
VR3/TT3       119.95

SQL> SELECT DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
  2 FROM CUSTOMER JOIN INVOICE USING (CUS_CODE)
  3           JOIN LINE USING (INV_NUMBER)
  4           JOIN PRODUCT USING (P_CODE)
  5 WHERE P_CODE IN (SELECT P_CODE FROM PRODUCT WHERE P_DESCRIFT = 'Clav hammer');

CUS_CODE CUS_LNAME          CUS_FNAME
-----  -----
10011  Dunne                 Leona
10014  Orlando               Myron
```



# IN, HAVING subqueries

## IN and HAVING Subqueries

- IN subqueries
  - Used to compare a single attribute to a list of values
- HAVING subqueries
  - HAVING clause restricts the output of a GROUP BY query by applying conditional criteria to the grouped rows

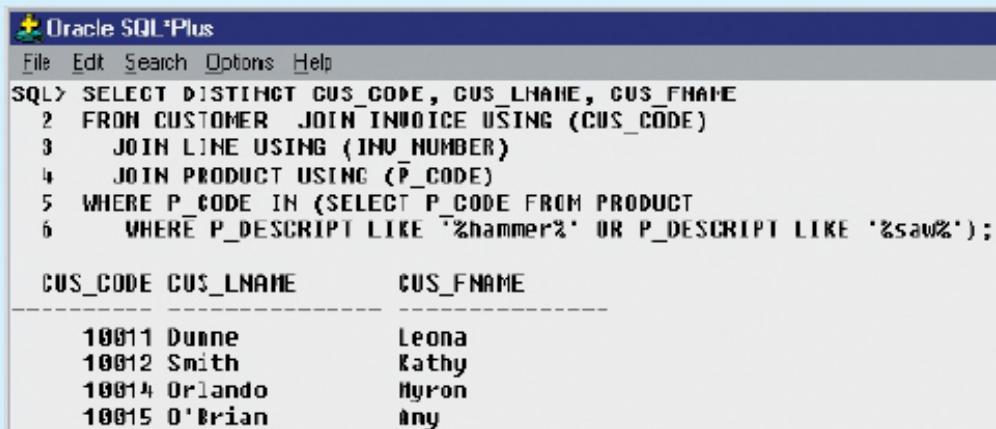


# 'IN' subqueries

Compare against a LIST of values..

FIGURE  
8.14

IN subquery example



The screenshot shows a Windows-style window titled "Oracle SQL\*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main area displays an SQL query and its results.

```
SQL> SELECT DISTINCT CUS_CODE, CUS_LNAME, CUS_FNAME
  2  FROM CUSTOMER JOIN INVOICE USING (CUS_CODE)
  3  JOIN LINE USING (INV_NUMBER)
  4  JOIN PRODUCT USING (P_CODE)
  5  WHERE P_CODE IN (SELECT P_CODE FROM PRODUCT
  6    WHERE P_DESCRIPT LIKE '%hammer%' OR P_DESCRIPT LIKE '%saw%');

  CUS_CODE CUS_LNAME      CUS_FNAME
  -----  -----
  10011  Dunne          Leona
  10012  Smith          Kathy
  10014  Orlando         Myron
  10015  O'Brian        Amy

SQL>
```

The query selects distinct customer codes, last names, and first names. It joins four tables: CUSTOMER, INVOICE, LINE, and PRODUCT. The WHERE clause uses an IN subquery to filter products based on their descriptions containing either "%hammer%" or "%saw%". The resulting output shows four customers: Dunne, Smith, Orlando, and O'Brian, each with their respective first and last names.



# 'HAVING' subqueries

As we saw earlier, this restricts the results of a GROUP BY clause. Eg. here's how to list all products sold, whose totals are greater than the average quantity sold:

**FIGURE  
8.15** HAVING subquery example

```
Oracle SQL*Plus
File Edit Search Options Help
SQL> SELECT P_CODE, SUM(LINE_UNITS)
  2  FROM LINE
  3  GROUP BY P_CODE
  4  HAVING SUM(LINE_UNITS) > (SELECT AVG(LINE_UNITS) FROM LINE);

P_CODE      SUM(LINE_UNITS)
-----  -----
13-Q2/P2          8
23109-HB         5
54778-2T         6
PUC23DRT        17
SM-18277         3
MR9/TT9           9

6 rows selected.

SQL>
```



# ALL, ANY (inequality comparisons)

Recall that 'IN' is an equality comparison against a list. To do inequality **comparison of a value against a list of values** (eg. need to be greater than ALL, need to be less than ANY..), use ALL, ANY.

## Multirow Subquery Operators: ANY and ALL

- ALL operator
  - Allows comparison of a single value with a list of values returned by the first subquery
    - Uses a comparison operator other than equals
- ANY operator
  - Allows comparison of a single value to a list of values and selects only the rows for which the value is greater than or less than any value in the list

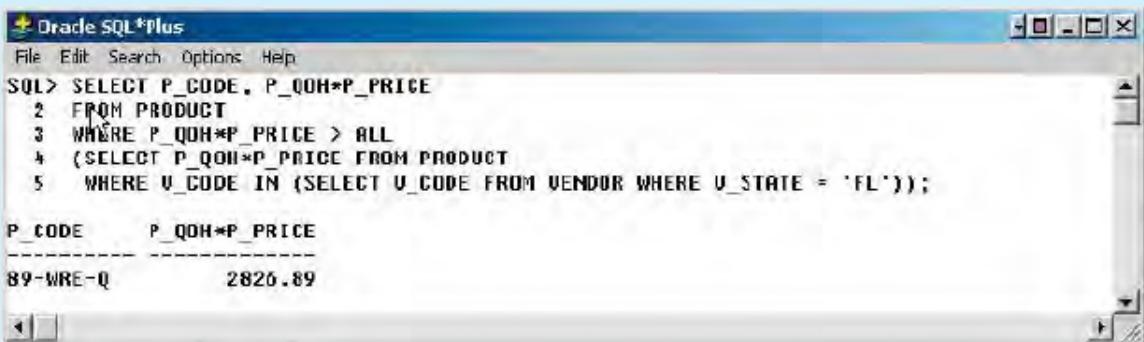


# ALL, ANY

Eg. "which products do we own [in our store], whose value is more than ALL other products's values supplied by vendors in Florida?"

FIGURE  
8.16

Multirow subquery operator example



The screenshot shows an Oracle SQL\*Plus window. The title bar says "Oracle SQL\*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The SQL command entered is:

```
SQL> SELECT P_CODE, P_QOH*p_PRICE
  2  FROM PRODUCT
  3  WHERE P_QOH*p_PRICE > ALL
  4  (SELECT P_QOH*p_PRICE FROM PRODUCT
  5  WHERE V_CODE IN (SELECT V_CODE FROM VENDOR WHERE V_STATE = 'FL'));
```

The output shows one row:

P_CODE	P_QOH*p_PRICE
89-WRE-Q	2826.89

Note that 'greater than ALL' is eqvt to 'greater than the largest of'. 'ALL' is used to select rows [plural in general] that comparison-succeed against all values in a list.

Another powerful operator is the ANY multirow operator (the near cousin of the ALL multirow operator). The ANY operator allows you to compare a single value to a list of values, selecting only the rows for which the inventory cost is greater than any value of the list or less than any value of the list. You could use the equal to ANY operator, which would be the equivalent of the IN operator.

'ANY' is used to select rows [plural in general] that comparison-succeed with any value in a list.

Note that '= ANY(list of values)' is equivalent to the 'IN' operator (which is itself equivalent to multiple == conditions joined by ORs). So the following are all equivalent, for a given value of 'M':

**(M==6) OR (M==8) OR (M==10)**

**M IN (6,8,10)**

**M = ANY (6,8,10)**

So loosely speaking, ALL is equivalent to AND, and ANY is equivalent to OR.



# 'FROM' subqueries

A SELECT query that appears in FROM, creates a \*\*virtual table\*\* against which the main query can run.

## FROM Subqueries

- FROM clause:
  - Specifies the tables from which the data will be drawn
  - Can use SELECT subquery

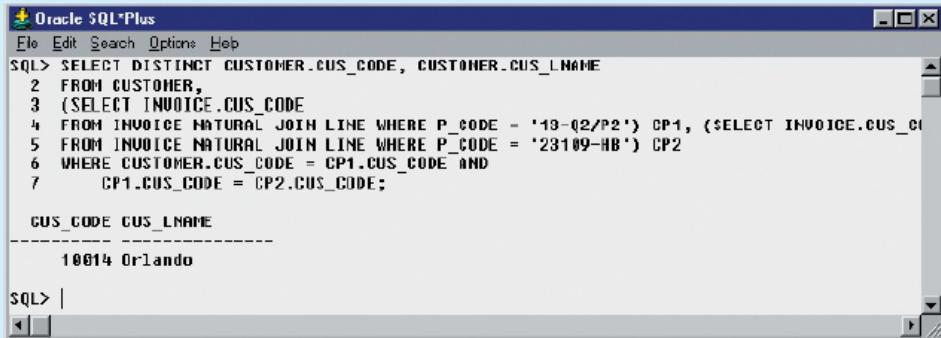


# FROM subquery example

All customers who bought both specified products

FIGURE  
8.17

FROM subquery example



The screenshot shows a window titled "Oracle SQL\*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main area displays the following SQL query and its results:

```
SQL> SELECT DISTINCT CUSTOMER.CUS_CODE, CUSTOMER.CUS_LNAME
  2  FROM CUSTOMER,
  3  (SELECT INVOICE.CUS_CODE
  4  FROM INVOICE NATURAL JOIN LINE WHERE P_CODE = '10-Q2/P2') CP1,
  5  (SELECT INVOICE.CUS_CODE
  6  FROM INVOICE NATURAL JOIN LINE WHERE P_CODE = '23109-HB') CP2
  7  WHERE CUSTOMER.CUS_CODE = CP1.CUS_CODE AND
        CP1.CUS_CODE = CP2.CUS_CODE;

 CUS_CODE CUS_LNAME
-----
 10014 Orlando
```

The command prompt "SQL>" is visible at the bottom left, and there is a small input field below it.



# Attribute list subqueries

These subqueries determine what columns get output by the main query - they can be actual (existing) columns or computed columns or results of aggregate functions.

These are also known as 'column subqueries' or 'inline subqueries'.

## Attribute List Subqueries

- SELECT statement uses attribute list to indicate what columns to project in the resulting set
- Inline subquery
  - Subquery expression included in the attribute list that must return one value
- Column alias cannot be used in attribute list computation if alias is defined in the same attribute list



# Attribute subquery example

FIGURE  
8.18

Inline subquery example

The screenshot shows the Oracle SQL\*Plus interface. The title bar reads "Oracle SQL\*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays the following SQL query and its results:

```
SQL> SELECT P_CODE, P_PRICE, (SELECT AVG(P_PRICE) FROM PRODUCT) AS AVGPRICE,
  2       P_PRICE-(SELECT AVG(P_PRICE) FROM PRODUCT) AS DIFF
  3   FROM PRODUCT;
```

P_CODE	P_PRICE	AVGPRICE	DIFF
11QER/81	189.99	56.42125	133.56875
13-Q2/P2	18.99	56.42125	-41.43125
14-Q1/L3	17.49	56.42125	-38.93125
1546-QQ2	39.95	56.42125	-16.47125
1558-QV1	43.99	56.42125	-12.43125
2232/QTY	189.92	56.42125	133.49875
2232/QWE	99.87	56.42125	43.44675
2238/OPD	38.95	56.42125	-17.47125
29109-HB	9.95	56.42125	-46.47125
23114-AA	14.4	56.42125	-42.02125
54778-2T	4.99	56.42125	-51.43125
89-WRE-Q	256.99	56.42125	200.56875
PVC230RT	5.87	56.42125	-50.55125
SM-18277	6.99	56.42125	-49.43125
SV-23116	8.45	56.42125	-47.97125
VR3/TT3	119.95	56.42125	63.52875

16 rows selected.

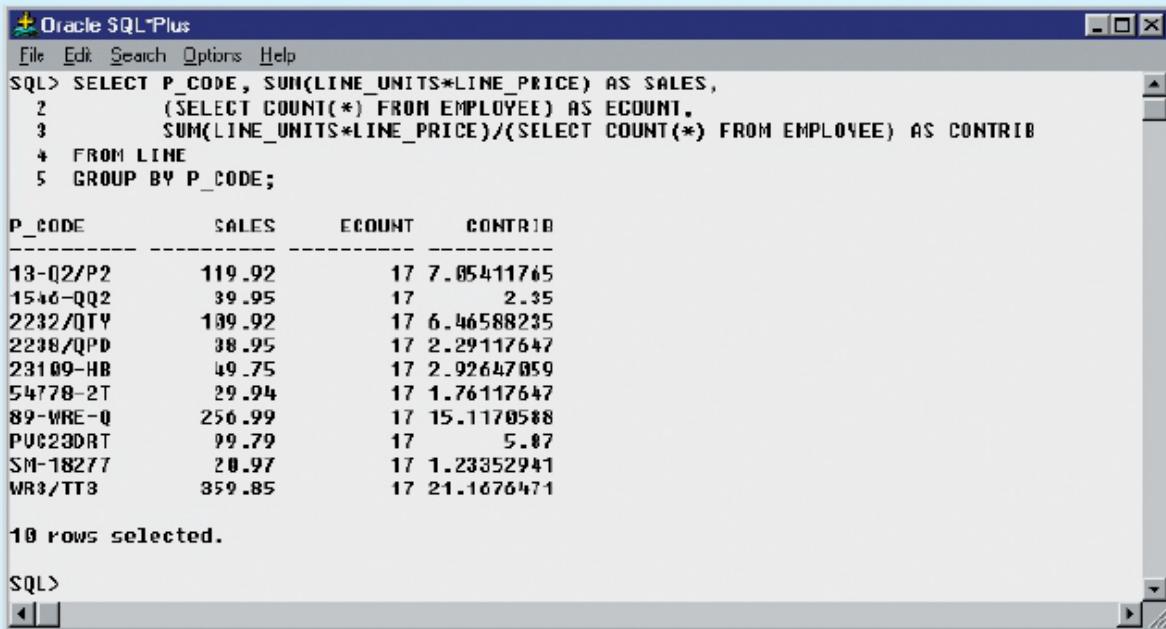
SQL>



# Another attribute subquery example

FIGURE  
8.19

Another example of an inline subquery



The screenshot shows the Oracle SQL\*Plus interface with a blue header bar. The title bar reads "Oracle SQL\*Plus". The menu bar includes "File", "Edit", "Search", "Options", and "Help". The main window displays an SQL query and its results.

```
SQL> SELECT P_CODE, SUM(LINE_UNITS*LINE_PRICE) AS SALES,
  2      (SELECT COUNT(*) FROM EMPLOYEE) AS ECOUNT,
  3      SUM(LINE_UNITS*LINE_PRICE)/(SELECT COUNT(*) FROM EMPLOYEE) AS CONTRIB
  4  FROM LINE
  5 GROUP BY P_CODE;
```

P_CODE	SALES	ECOUNT	CONTRIB
13-Q2/P2	119.92	17	7.05411765
1546-QQ2	39.95	17	2.35
2232/QTY	109.92	17	6.46588235
2238/QPD	38.95	17	2.29117647
23109-HB	49.75	17	2.92647059
54778-2T	29.94	17	1.76117647
89-WRE-Q	256.99	17	15.1170588
PUC23DRT	99.79	17	5.87
SM-18277	20.97	17	1.23352941
WR3/TT3	359.85	17	21.1676471

10 rows selected.

SQL>



# Correlated subqueries

## Correlated Subquery

- Executes once for each row in the outer query
- Inner query references a column of the outer subquery
- Can be used with the EXISTS special operator

In a correlated subquery, the inner (sub) query is repeatedly run, for each row of the outer query! The inner is said to be (co-)related with the outer query when it references a column in the outer query's table. This is in effect, like a double (nested) 'for' loop..

```
for each row in OUTER table
  run subquery on EACH row in INNER table
, gather
  results, use in outer table's query
```

Here is the Wikipedia entry on correlated subqueries. This is the example shown there [select employees who make more than the average salary for their department]:

```
SELECT employee_number, name
  FROM employees AS Bob
 WHERE salary > (
   SELECT AVG(salary)
     FROM employees
    WHERE department = Bob.department
 );
```

In the above, the outer query "passes in", for each employee (each row), the employee's dept. [which the inner query refers to as Bob.department]. The inner query selects all salaries for that dept., computes the

average, compares it with the passed-in employee's salary; if the test passes, the outer query selects the employee's # and name.



# Correlated subqueries [cont'd]

Until now, all subqueries you have learned execute independently. That is, each subquery in a command sequence executes in a serial fashion, one after another. The inner subquery executes first; its output is used by the outer query, which then executes until the last outer query executes (the first SQL statement in the code).

In contrast, a **correlated subquery** is a subquery that executes once for each row in the outer query. That process is similar to the typical nested loop in a programming language. For example:

```
FOR X = 1 TO 2
    FOR Y = 1 TO 3
        PRINT "X = "X, "Y = "Y
    END
END
```

1. It initiates the outer query.
2. For each row of the outer query result set, it executes the inner query by passing the outer row to the inner query.

That process is the opposite of that of the subqueries as you have already seen. The query is called a **correlated subquery** because the inner query is *related* to the outer query by the fact that the inner query references a column of the outer subquery.



# Correlated subquery examples

To see the correlated subquery in action, suppose that you want to know all product sales in which the units sold value is greater than the average units sold value *for that product* (as opposed to the average for *all* products). In that case, the following procedure must be completed:

1. Compute the average units sold for a product.
2. Compare the average computed in Step 1 to the units sold in each sale row, and then select only the rows in which the number of units sold is greater.

The following correlated query completes the preceding two-step process:

```
SELECT    INV_NUMBER, P_CODE, LINE_UNITS
FROM      LINE LS
WHERE     LS.LINE_UNITS > (SELECT AVG(LINE_UNITS)
                           FROM LINE LA
                           WHERE LA.P_CODE = LS.P_CODE);
```

```
SQL> SELECT INV_NUMBER, P_CODE, LINE_UNITS
  2  FROM LINE LS
  3 WHERE LS.LINE_UNITS > (SELECT AVG(LINE_UNITS)
  4                           FROM LINE LA
  5                           WHERE LA.P_CODE = LS.P_CODE);

INV_NUMBER P_CODE      LINE_UNITS
----- -----
  1003 13-Q2/P2          5
  1004 54778-2T          3
  1004 23109-HB          2
  1005 PVC23DRT         12

SQL> SELECT INV_NUMBER, P_CODE, LINE_UNITS,
  2      (SELECT AVG(LINE_UNITS) FROM LINE LX WHERE LX.P_CODE = LS.P_CODE) AS AUG
  3  FROM LINE LS
  4 WHERE LS.LINE_UNITS > (SELECT AVG(LINE_UNITS)
  5                           FROM LINE LA
  6                           WHERE LA.P_CODE = LS.P_CODE);

INV_NUMBER P_CODE      LINE_UNITS      AVG
----- -----
  1003 13-Q2/P2          5  2.666666667
  1004 54778-2T          3   2
  1004 23109-HB          2   1.25
  1005 PVC23DRT         12   8.5

SQL>
```

In the top query and its result in Figure 8.14, note that the LINE table is used more than once, so you must use table aliases. In this case, the inner query computes the average units sold of the product that matches the P\_CODE of the outer query P\_CODE. That is, the inner query runs once, using the first product code found in the outer LINE table, and returns the average sale for that product. When the number of units sold in the outer LINE row is greater than the average computed, the row is added to the output. Then the inner query runs again, this time using the second product code found in the outer LINE table. The process repeats until the inner query has run for all rows in the outer LINE table. In this case, the inner query will be repeated as many times as there are rows in the outer query.

To verify the results and to provide an example of how you can combine subqueries, you can add a correlated inline subquery to the previous query. (See the second query and its results in Figure 8.14.) As you can see, the new query contains a correlated inline subquery that computes the average units sold for each product. You not only get an answer, you can also verify that the answer is correct.

**In the second query above, we have TWO correlated subqueries (that are identical), both of which need to run for every row of the main query.**



# UNION, INTERSECTION, DIFFERENCE

## Relational Set Operators

- SQL data manipulation commands are set-oriented
  - **Set-oriented:** Operate over entire sets of rows and columns at once
- UNION, INTERSECT, and Except (MINUS) work properly when relations are union-compatible
  - **Union-compatible:** Number of attributes are the same and their corresponding data types are alike
- UNION
  - Combines rows from two or more queries without including duplicate rows



# UNION, INTERSECTION, DIFFERENCE [cont'd]

## Relational Set Operators

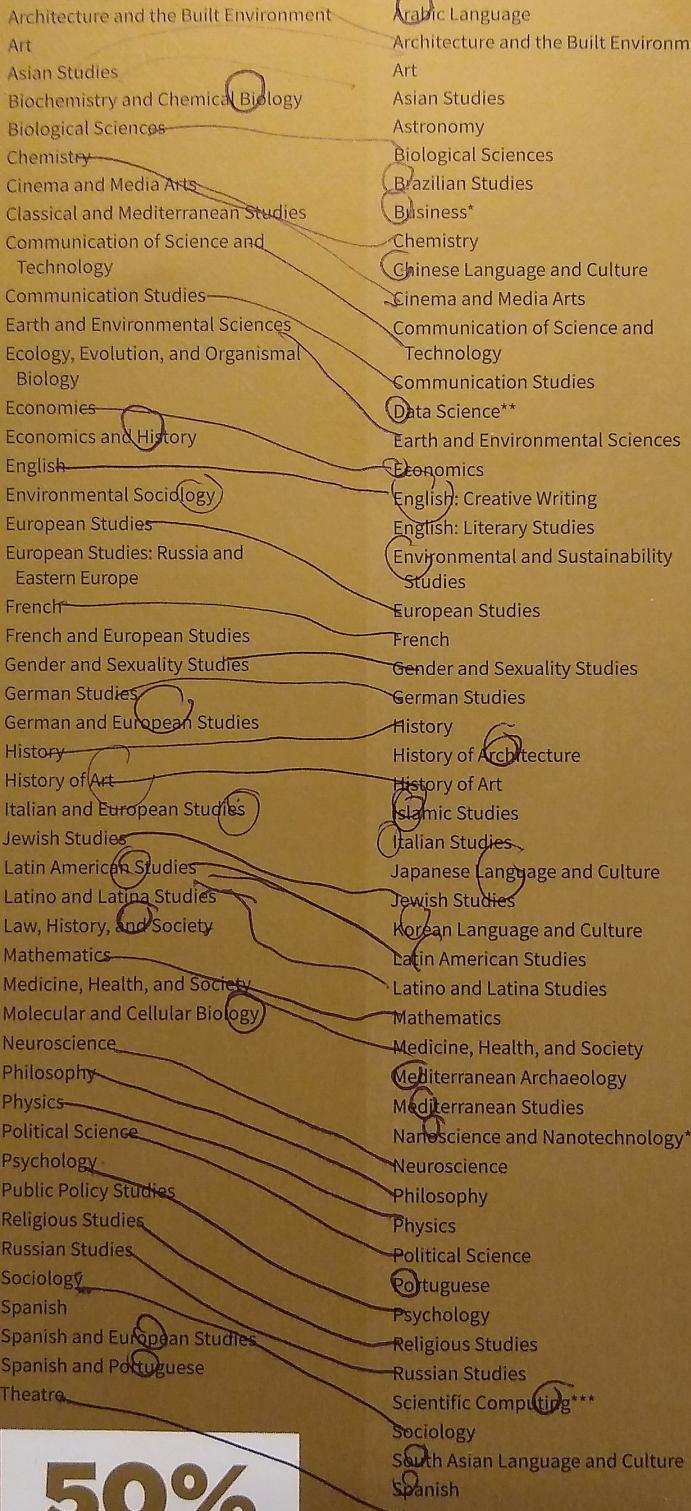
- Syntax - query UNION query
- UNION ALL
  - Produces a relation that retains duplicate rows
  - Can be used to unite more than two queries
- INTERSECT
  - Combines rows from two queries, returning only the rows that appear in both sets
  - Syntax - query INTERSECT query

©2015 Cengage Learning. All Rights Reserved. May not be scanned, copied or duplicated, or posted to a publicly accessible website, in whole or in part.

27

If the columns below are in two different tables, the intersection of them would list items in both (eg. History, Physics...):

MAJORS	MINORS
African American and Diaspora Studies	African American and Diaspora Studies
American Studies	American Studies
Anthropology	Anthropology



**50%**  
**ADD A**  
**MINOR**  
TO THEIR PRIMARY  
**DEGREE**

\* Jointly administered by the four undergraduate schools and Owen Graduate School of Management

\*\* Jointly administered by the four undergraduate schools

\*\*\* Jointly administered by the School of Engineering

---



# UNION, INTERSECTION, DIFFERENCE [cont'd]

## Relational Set Operators

- EXCEPT (MINUS)
  - Combines rows from two queries and returns only the rows that appear in the first set
  - Syntax
    - query EXCEPT query
    - query MINUS query
- Syntax alternatives
  - IN and NOT IN subqueries can be used in place of INTERSECT



# VIEWS

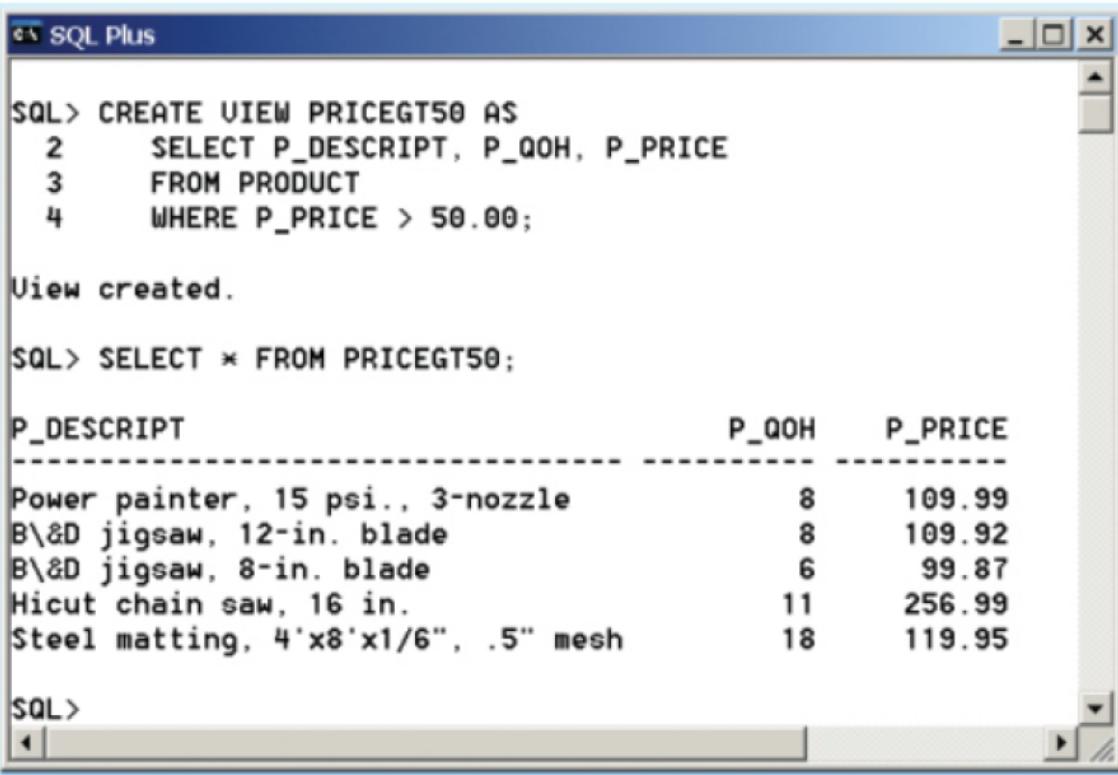
## Virtual Tables: Creating a View

- **View:** Virtual table based on a SELECT query
- **Base tables:** Tables on which the view is based
- **CREATE VIEW** statement: Data definition command that stores the subquery specification in the data dictionary
  - CREATE VIEW command
    - CREATE VIEW *viewname* AS SELECT query



# VIEW example

## Creating a Virtual Table with the CREATE VIEW Command



The screenshot shows a Windows application window titled "SQL Plus". Inside, SQL commands are being run and their results displayed.

```
SQL> CREATE VIEW PRICEGT50 AS
  2      SELECT P_DESCRPT, P_QOH, P_PRICE
  3      FROM PRODUCT
  4     WHERE P_PRICE > 50.00;

View created.

SQL> SELECT * FROM PRICEGT50;

P_DESCRPT                  P_QOH    P_PRICE
-----  -----  -----
Power painter, 15 psi., 3-nozzle      8    109.99
B\&D jigsaw, 12-in. blade            8    109.92
B\&D jigsaw, 8-in. blade             6    99.87
Hicut chain saw, 16 in.           11   256.99
Steel matting, 4'x8'x1/6", .5" mesh 18   119.95

SQL>
```

Cengage Learning © 2015

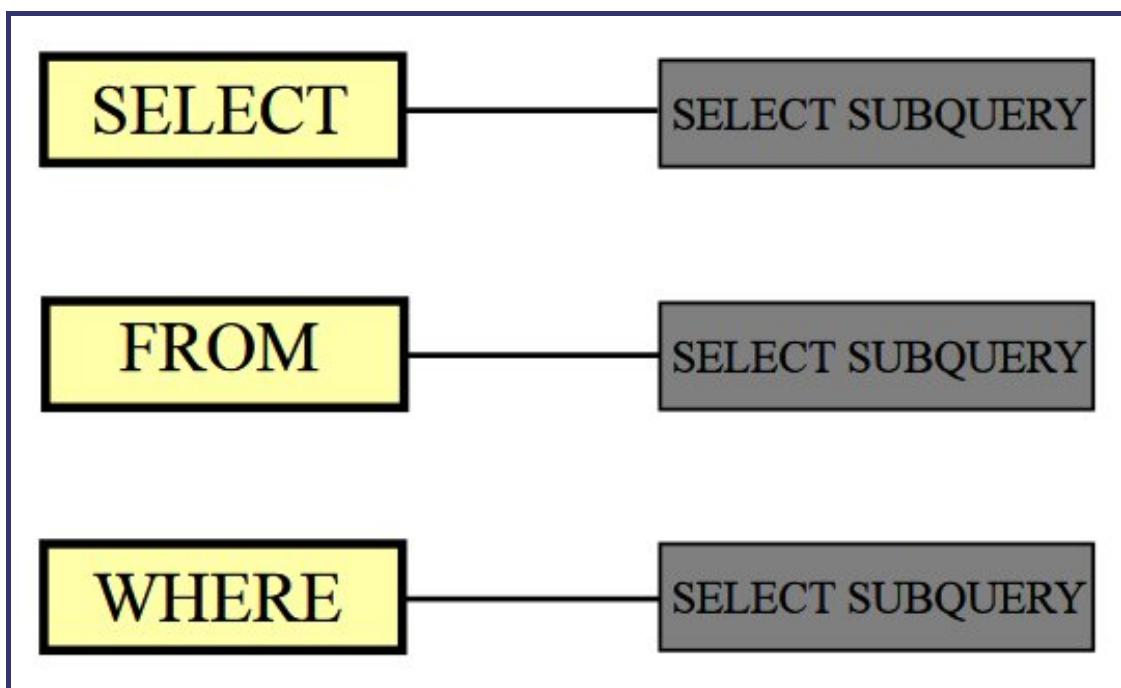
©2015 Cengage Learning. All Rights Reserved. May not be scanned, copied or duplicated, in whole or in part.



# Queries: summary

We looked at several variations of queries and subqueries (SELECT, WHERE, HAVING, IN..).

Most interestingly, a SELECT subquery can appear at the top (SELECT), middle (FROM) or bottom (WHERE) of a parent query, which provides a flexible way to express **complex logic** (since such subqueries can be recursively nested):





# SQL functions (built-ins)

Functions **return values**...

## SQL Functions

- Functions always use a numerical, date, or string value
- Value may be part of a command or may be an attribute located in a table
- Function may appear anywhere in an SQL statement where a value or an attribute can be used



# SQL functions [cont'd]

## SQL Functions

Date and time functions

Numeric functions

String functions

Conversion functions



# Sequences

## Oracle Sequences

- Independent object in the database
- Have a name and can be used anywhere a value expected
- Not tied to a table or column
- Generate a numeric value that can be assigned to any column in any table
- Table attribute with an assigned value can be edited and modified
- Can be created and deleted any time



# Sequence creation example

Figure 8.27 - Oracle Sequence

The screenshot shows a Windows application window titled "SQL Plus". Inside, SQL commands are being run:

```
SQL> CREATE SEQUENCE CUS_CODE_SEQ START WITH 20010 NOCACHE;
Sequence created.

SQL> CREATE SEQUENCE INV_NUMBER_SEQ START WITH 4010 NOCACHE;
Sequence created.

SQL> SELECT * FROM USER_SEQUENCES;
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	C	O	CACHE_SIZE	LAST_NUMBER
CUS_CODE_SEQ	1	1.0000E+27	1	N	N	0	20010
INV_NUMBER_SEQ	1	1.0000E+27	1	N	N	0	4010

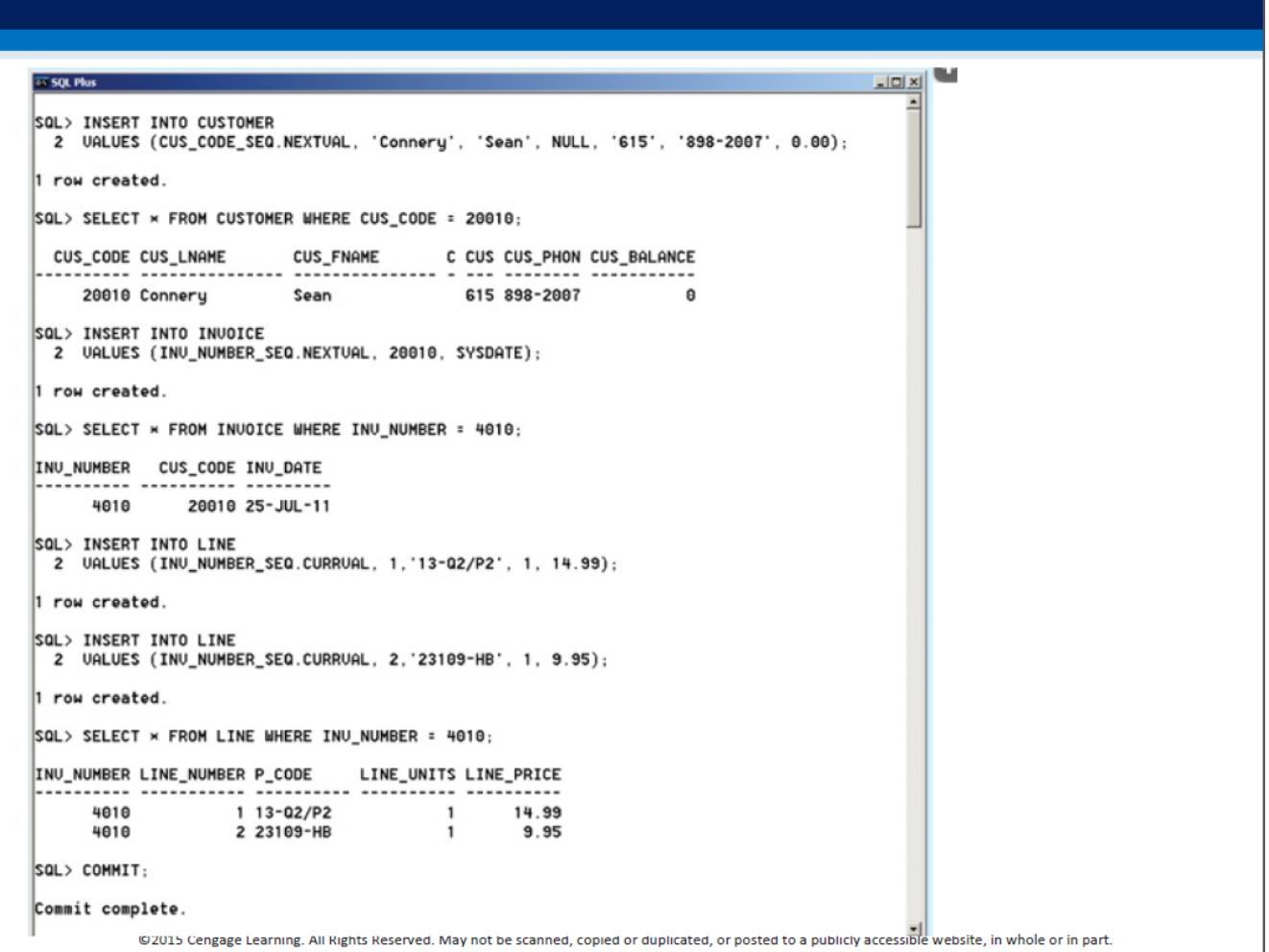
```
SQL>
```

```
INSERT INTO CUSTOMER
VALUES (CUS_CODE_SEQ.NEXTVAL, 'Connery', 'Sean', NULL, '615', '898-2007', 0.00);
```



# Sequence: NEXTVAL, CURRVAL

NEXTVAL returns the current value, **then** does ++ (ie. it does 'post increment', ie. C++ as opposed to ++C); CURRVAL on the other hand, just fetches the current value (does not ++ it).



The screenshot shows a window titled "SQL\*Plus" with the following SQL session:

```
SQL> INSERT INTO CUSTOMER
  2  VALUES (CUS_CODE_SEQ.NEXTVAL, 'Connery', 'Sean', NULL, '615', '898-2007', 0.00);
1 row created.

SQL> SELECT * FROM CUSTOMER WHERE CUS_CODE = 20010;
   CUS_CODE CUS_LNAME      CUS_FNAME      C CUS_CUS_PHON CUS_BALANCE
-----  -----
  20010 Connery          Sean           615 898-2007          0

SQL> INSERT INTO INVOICE
  2  VALUES (INU_NUMBER_SEQ.NEXTVAL, 20010, SYSDATE);
1 row created.

SQL> SELECT * FROM INVOICE WHERE INU_NUMBER = 4010;
  INU_NUMBER  CUS_CODE INU_DATE
-----  -----
  4010        20010 25-JUL-11

SQL> INSERT INTO LINE
  2  VALUES (INU_NUMBER_SEQ.CURRVAL, 1,'13-Q2/P2', 1, 14.99);
1 row created.

SQL> INSERT INTO LINE
  2  VALUES (INU_NUMBER_SEQ.CURRVAL, 2,'23109-HB', 1, 9.95);
1 row created.

SQL> SELECT * FROM LINE WHERE INU_NUMBER = 4010;
  INU_NUMBER LINE_NUMBER P_CODE      LINE_UNITS LINE_PRICE
-----  -----
  4010        1 13-Q2/P2            1       14.99
  4010        2 23109-HB           1       9.95

SQL> COMMIT;
Commit complete.
```

©2015 Cengage Learning. All Rights Reserved. May not be scanned, copied or duplicated, or posted to a publicly accessible website, in whole or in part.



# Procedural Language SQL (PL/SQL)

PL/SQL involves extra (augmented) syntax that lets us do looping, branching, variable declaration and function declaration - these are of course not possible using 'plain' SQL.

PL/SQL can be used to create:

- **blocks of code** for one-time execution
- **triggers** - callbacks to invoke
- **stored procedures** - named procedures (no return values) for repeated calling
- **stored functions** - named functions (with return values) for repeated calling

# Procedural SQL

- Performs a conditional or looping operation by isolating critical code and making all application programs call the shared code
  - Yields better maintenance and logic control
- **Persistent stored module (PSM):** Block of code containing:
  - Standard SQL statements
  - Procedural extensions that is stored and executed at the DBMS server



# PL/SQL [cont'd]

## Procedural SQL

- **Procedural Language SQL (PL/SQL)**
  - Use and storage of procedural code and SQL statements within the database
  - Merging of SQL and traditional programming constructs
- Procedural code is executed as a unit by DBMS when invoked by end user
- End users can use PL/SQL to create:
  - Anonymous PL/SQL blocks and triggers
  - Stored procedures and PL/SQL functions



# [Unnamed] block creation example

The screenshot shows a Windows application window titled "SQL\*Plus". Inside, a PL/SQL block is being run. The code creates a new vendor record and prints a success message. It then performs a SELECT query on the VENDOR table, displaying 13 rows of vendor information.

```
SQL> BEGIN
 2  INSERT INTO UENDOR
 3  VALUES (25678, 'Microsoft Corp.', 'Bill Gates', '765', '546-8484', 'WA', 'N');
 4 END;
 5 /
PL/SQL procedure successfully completed.

SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
 2  INSERT INTO UENDOR
 3  VALUES (25772, 'Clue Store', 'Issac Hayes', '456', '323-2009', 'VA', 'N');
 4  DBMS_OUTPUT.PUT_LINE('New Vendor Added!');
 5 END;
 6 /
New Vendor Added!

PL/SQL procedure successfully completed.

SQL> SELECT * FROM UENDOR;

U_CODE U_NAME          U_CONTACT      U_A U_PHONE   U_ U
----- -----
25678 Microsoft Corp.    Bill Gates    765 546-8484 WA N
25772 Clue Store        Issac Hayes   456 323-2009 VA N
21225 Bryson, Inc.       Smithson     615 223-3234 TN Y
21226 SuperLoo, Inc.    Flushing     904 215-8995 FL N
21231 D&E Supply       Singh        615 228-3245 TN Y
21344 Gomez Bros.       Ortega       615 889-2546 KY N
22567 Dome Supply       Smith        901 678-1419 GA N
23119 Randsets Ltd.     Anderson    901 678-3998 GA Y
24004 Brackman Bros.   Browning    615 228-1410 TN N
24288 ORDVUA, Inc.      Hakford     615 898-1234 TN Y
25443 B&K, Inc.         Smith        904 227-0093 FL N
25501 Damal Supplies    Smythe      615 890-3529 TN N
25595 Rubicon Systems   Orton       904 456-0092 FL Y

13 rows selected.

SQL>
```

e website, in whole or in part.



# Triggers

## Triggers

- Procedural SQL code automatically invoked by RDBMS when given data manipulation event occurs
- Parts of a trigger definition
  - Triggering timing - Indicates when trigger's PL/SQL code executes
  - Triggering event - Statement that causes the trigger to execute
  - Triggering level - **Statement-** and **row-level**
  - Triggering action - PL/SQL code enclosed between the BEGIN and END keywords



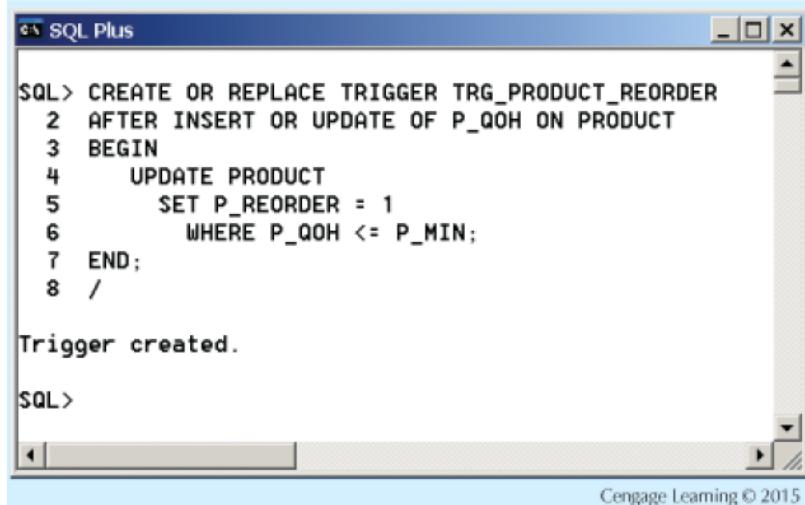
# Triggers [cont'd]

- *The triggering timing:* BEFORE or AFTER. This timing indicates when the trigger's PL/SQL code executes—in this case, before or after the triggering statement is completed.
- *The triggering event:* The statement that causes the trigger to execute (INSERT, UPDATE, or DELETE).
- *The triggering level:* The two types of triggers are statement-level triggers and row-level triggers.
  - A **statement-level trigger** is assumed if you omit the FOR EACH ROW keywords. This type of trigger is executed once, before or after the triggering statement is completed. This is the default case.
  - A **row-level trigger** requires use of the FOR EACH ROW keywords. This type of trigger is executed once for each row affected by the triggering statement. (In other words, if you update 10 rows, the trigger executes 10 times.)
- *The triggering action:* The PL/SQL code enclosed between the BEGIN and END keywords. Each statement inside the PL/SQL code must end with a semicolon ( ; ).



# Trigger example

```
CREATE OR REPLACE TRIGGER trigger_name
[BEFORE / AFTER] [DELETE / INSERT / UPDATE OF column_name] ON table_name
[FOR EACH ROW]
[DECLARE]
[variable_name data type[:=initial_value]]
BEGIN
PL/SQL instructions;
.....
END;
```



The screenshot shows a Windows-style window titled "SQL Plus". Inside, SQL code is being entered and executed. The code creates a trigger named "TRG\_PRODUCT\_REORDER" that fires after an insert or update of the "P\_QOH" column in the "PRODUCT" table. The trigger updates the "P\_REORDER" column to 1 where "P\_QOH" is less than or equal to "P\_MIN". The execution is successful, returning the message "Trigger created.".

```
SQL> CREATE OR REPLACE TRIGGER TRG_PRODUCT_REORDER
  2  AFTER INSERT OR UPDATE OF P_QOH ON PRODUCT
  3  BEGIN
  4    UPDATE PRODUCT
  5      SET P_REORDER = 1
  6      WHERE P_QOH <= P_MIN;
  7  END;
  8  /

Trigger created.

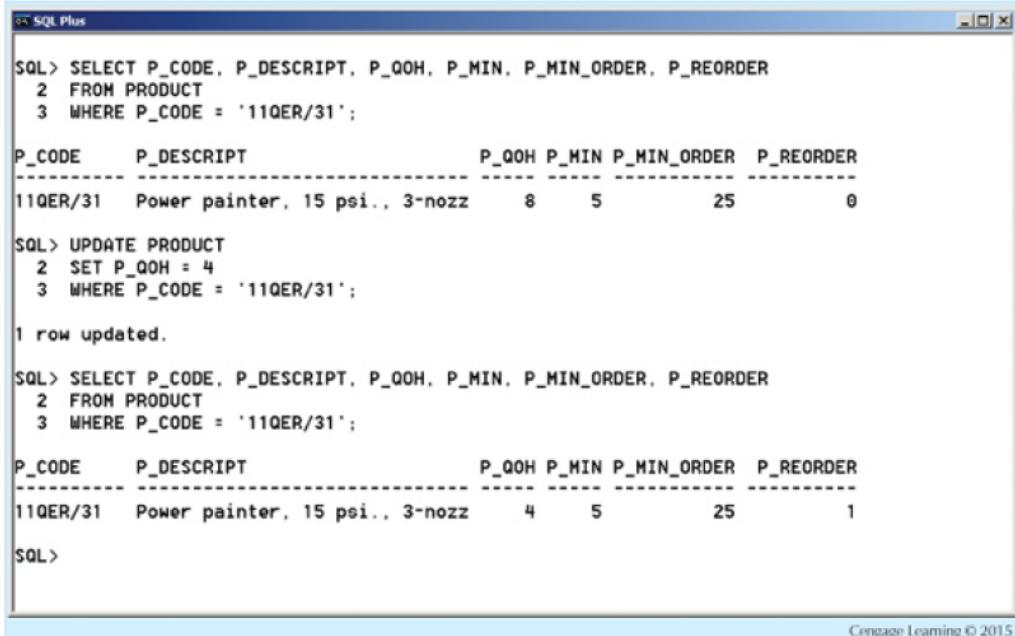
SQL>
```

Cengage Learning © 2015

©2015 Cengage Learning. All Rights Reserved. May not be scanned, copied or duplicated, or posted to a publicly accessible website, in whole or in part.



# Trigger example



The screenshot shows an SQL Plus window with the following session history:

```
SQL> SELECT P_CODE, P_DESCRIP, P_QOH, P_MIN, P_MIN_ORDER, P_REORDER
  2  FROM PRODUCT
  3  WHERE P_CODE = '11QER/31';

P_CODE      P_DESCRIP          P_QOH  P_MIN P_MIN_ORDER  P_REORDER
-----      -----          -----   ---   -----       -----
11QER/31    Power painter, 15 psi., 3-nozz     8      5        25        0

SQL> UPDATE PRODUCT
  2  SET P_QOH = 4
  3  WHERE P_CODE = '11QER/31';

1 row updated.

SQL> SELECT P_CODE, P_DESCRIP, P_QOH, P_MIN, P_MIN_ORDER, P_REORDER
  2  FROM PRODUCT
  3  WHERE P_CODE = '11QER/31';

P_CODE      P_DESCRIP          P_QOH  P_MIN P_MIN_ORDER  P_REORDER
-----      -----          -----   ---   -----       -----
11QER/31    Power painter, 15 psi., 3-nozz     4      5        25        1

SQL>
```

Cengage Learning © 2015

Our trigger worked! [look at P\_REORDER]



# Triggers [cont'd]

## Triggers

- **DROP TRIGGER trigger\_name command**
  - Deletes a trigger without deleting the table
- Trigger action based on DML predicates
  - Actions depend on the type of DML statement that fires the trigger



# Stored procedures

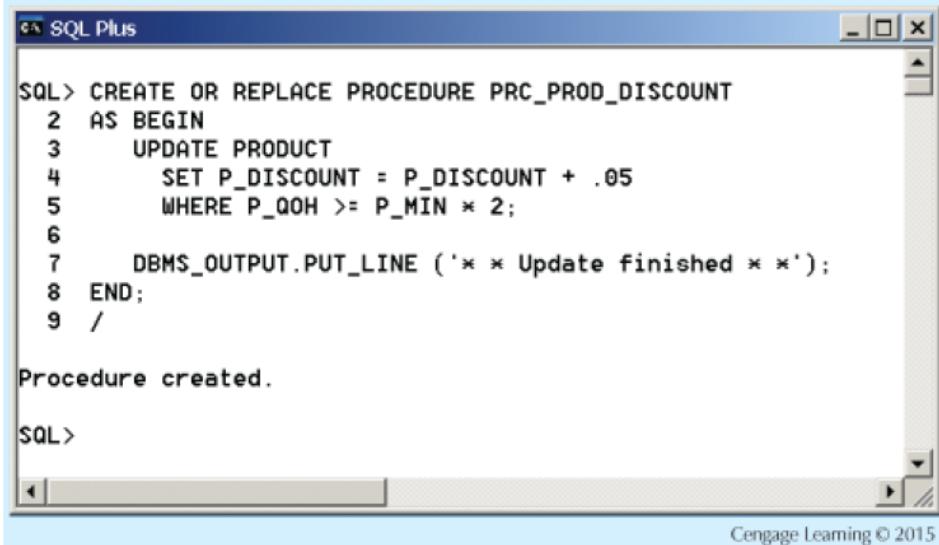
## Stored Procedures

- Named collection of procedural and SQL statements
- Advantages
  - Reduce network traffic and increase performance
  - Reduce code duplication by means of code isolation and code sharing



# Stored procedure example

```
CREATE OR REPLACE PROCEDURE procedure_name [(argument [IN/OUT] data-type, ...)]  
    [IS/AS]  
    [variable_name data-type[:=initial_value] ]  
BEGIN  
    PL/SQL or SQL statements;  
    ...  
END;
```



The screenshot shows an 'SQL Plus' window with the following content:

```
SQL> CREATE OR REPLACE PROCEDURE PRC_PROD_DISCOUNT  
2 AS BEGIN  
3     UPDATE PRODUCT  
4         SET P_DISCOUNT = P_DISCOUNT + .05  
5         WHERE P_QOH >= P_MIN * 2;  
6  
7     DBMS_OUTPUT.PUT_LINE ('* * Update finished * *');  
8 END;  
9 /  
  
Procedure created.  
  
SQL>
```

The window has a standard Windows-style title bar with minimize, maximize, and close buttons. The main area contains the SQL code and its execution results. The status bar at the bottom right displays "Cengage Learning © 2015".

©2015 Cengage Learning. All Rights Reserved. May not be scanned, copied or duplicated, or posted to a publicly accessible website, in whole or in part.



# Stored procedure example

The screenshot shows a Windows application window titled "SQL Plus". Inside, there are two separate SQL session panes. The top pane shows the initial state of the product table:

```
SQL> SELECT P_CODE, P_DESCRIFT, P_QOH, P_MIN, P_DISCOUNT FROM PRODUCT;
```

P_CODE	P_DESCRIFT	P_QOH	P_MIN	P_DISCOUNT
11QER-31	Power painter, 15 psi., 3-nozz	29	5	0.00
13-QZ-P2	7.25-in. pur. saw blade	32	15	0.05
14-QJ-L3	9.00-in. pur. saw blade	18	12	0.00
1546-QQ2	Hrd. cloth, 1/4-in., 2x50	15	8	0.00
1550-QW1	Hrd. cloth, 1/2-in., 3x50	23	5	0.00
2232-QTY	BBD jigsaw, 12-in. blade	8	5	0.05
2232-QWE	BBD jigsaw, 8-in. blade	6	7	0.05
2238-QPD	BBD cordless drill, 1/2-in.	12	5	0.05
23109-HB	Claw hammer	23	10	0.10
23114-AA	Sledge hammer, 12 lb.	8	10	0.05
54778-2T	Rat-tail file, 1/8-in. fine	43	20	0.00
89-WRE-Q	Hicut chain saw, 16 in.	11	5	0.05
PUC230RT	PVC pipe, 3.5-in., 8-ft	188	75	0.00
SM-18277	1.25-in. metal screw, 25	172	75	0.00
SM-23116	2.5-in. wd. screw, 50	237	100	0.00
WR3-TT3	Steel matting, 4'x8'x1/6", .5"	18	5	0.10

16 rows selected.

```
SQL> EXEC PRC_FROD_DISCOUNT;
```

\* \* Update finished \* \*

```
PL/SQL procedure successfully completed.
```

```
SQL> SELECT P_CODE, P_DESCRIFT, P_QOH, P_MIN, P_DISCOUNT FROM PRODUCT;
```

P_CODE	P_DESCRIFT	P_QOH	P_MIN	P_DISCOUNT
11QER-31	Power painter, 15 psi., 3-nozz	29	5	0.05
13-QZ-P2	7.25-in. pur. saw blade	32	15	0.10
14-QJ-L3	9.00-in. pur. saw blade	18	12	0.00
1546-QQ2	Hrd. cloth, 1/4-in., 2x50	15	8	0.00
1550-QW1	Hrd. cloth, 1/2-in., 3x50	23	5	0.05
2232-QTY	BBD jigsaw, 12-in. blade	8	5	0.05
2232-QWE	BBD jigsaw, 8-in. blade	6	7	0.05
2238-QPD	BBD cordless drill, 1/2-in.	12	5	0.10
23109-HB	Claw hammer	23	10	0.15
23114-AA	Sledge hammer, 12 lb.	8	10	0.05
54778-2T	Rat-tail file, 1/8-in. fine	43	20	0.05
89-WRE-Q	Hicut chain saw, 16 in.	11	5	0.10
PUC230RT	PVC pipe, 3.5-in., 8-ft	188	75	0.05
SM-18277	1.25-in. metal screw, 25	172	75	0.05
SM-23116	2.5-in. wd. screw, 50	237	100	0.05
WR3-TT3	Steel matting, 4'x8'x1/6", .5"	18	5	0.15

16 rows selected.

```
SQL>
```

Cengage Learning © 2015

©2015 Cengage Learning. All Rights Reserved. May not be scanned, copied or duplicated, or posted to a publicly accessible website, in whole or in part.



# Stored functions

Reminder - these can RETURN a value.

## PL/SQL Stored Functions

- **Stored function:** Named group of procedural and SQL statements that returns a value
  - As indicated by a RETURN statement in its program code
- Can be invoked only from within stored procedures or triggers



# Stored functions - syntax

```
CREATE FUNCTION function_name (argument IN data-type, ...) RETURN data-type [IS]
BEGIN
    PL/SQL statements;
    ...
    RETURN (value or expression);
END;
```

Once such a function is defined, it can be CALLED inside triggers or in stored procedures..



# Stored functions - example

The following is an example from  
<http://www.tutorialspoint.com/plsql>.

Creating/defining a function:

```
FUNCTION findMax(x IN number, y IN number
)
RETURN number
IS
    z number;
BEGIN
    IF x > y THEN
        z:= x;
    ELSE
        z:= y;
    END IF;

    RETURN z;
END;
```

Calling/executing/running the function:

```
DECLARE
    a number;
    b number;
    c number;
BEGIN
    a:= 23;
    b:= 45;

    c := findMax(a, b);
    dbms_output.put_line(' Maximum of (23,
45): ' || c);
END;
/
```

Result:

```
Maximum of (23,45): 45
```

