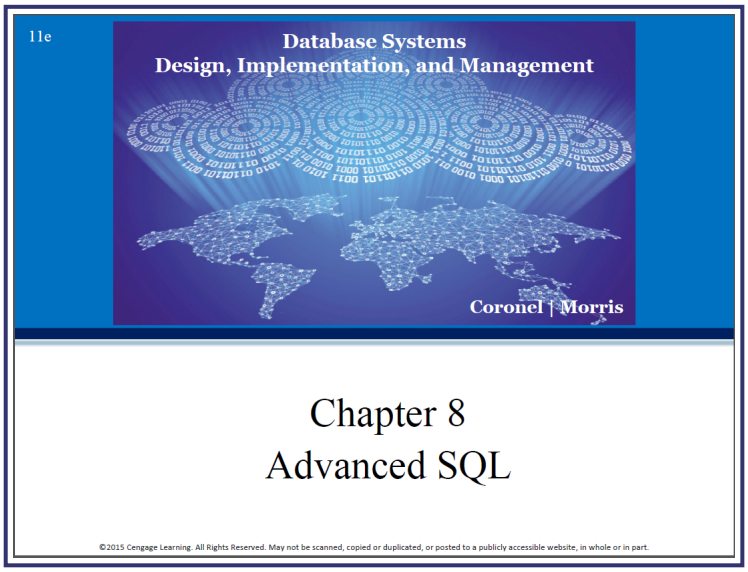
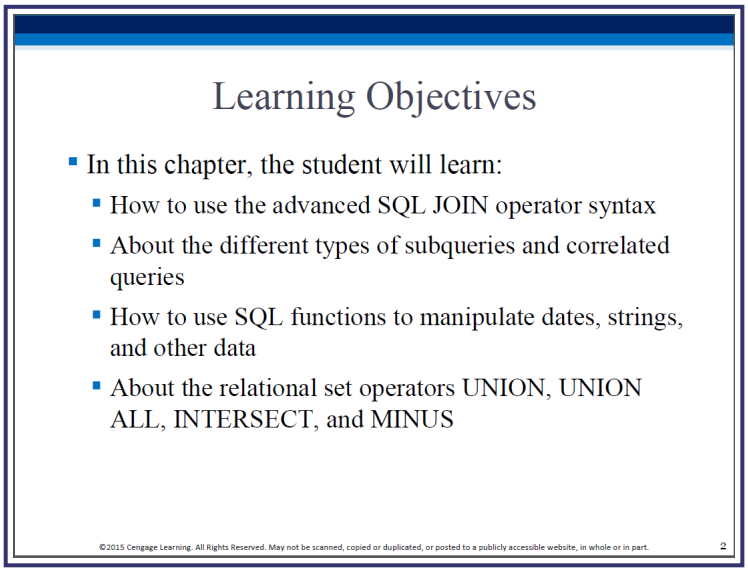
1/50 7:26:23 ***

More SQL

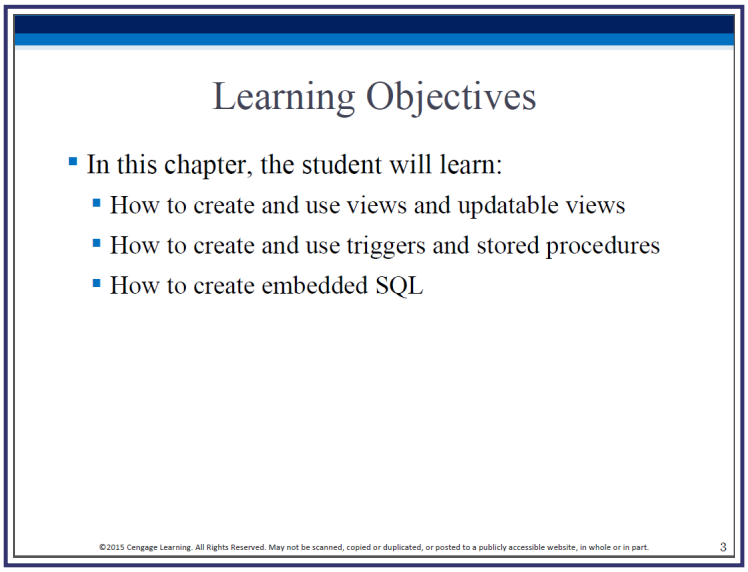
Ch.8


11e
Database Systems
Design, Implementation, and Management
Coronel | Morris
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
©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.
2

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
©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.
3

The 'JOIN' operation

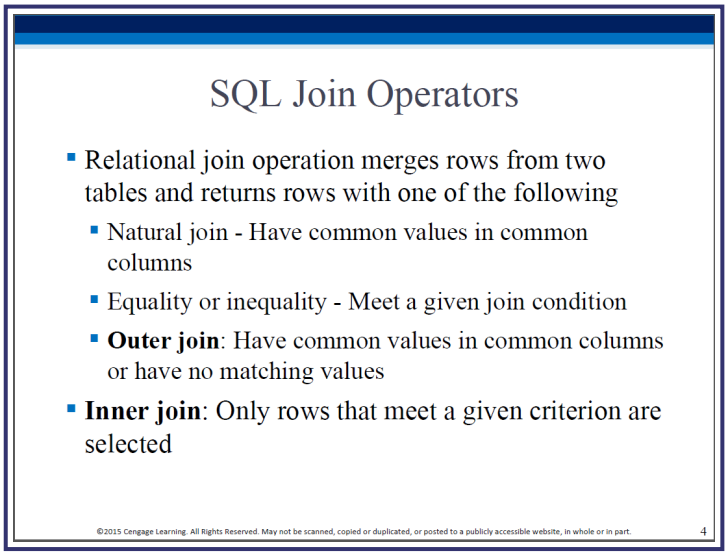
Recall that we looked at examples of joining - entries from two tables, and entries from a single table. These joins were based on 'join conditions'.

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;
```



Ways to specify JOIN conditions

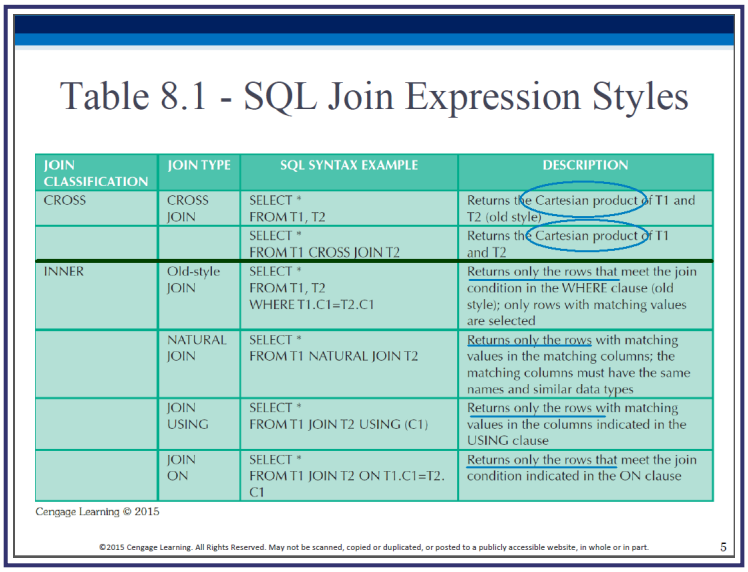


Table 8.1 - SQL Join Expression Styles

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

5

Outer vs inner vs full ('both') JOINs

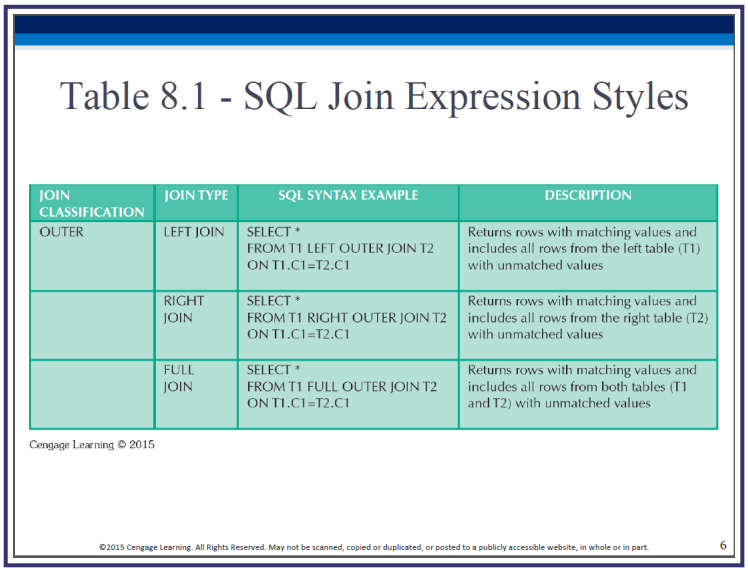


Table 8.1 - SQL Join Expression Styles

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

Cengage Learning © 2015

6

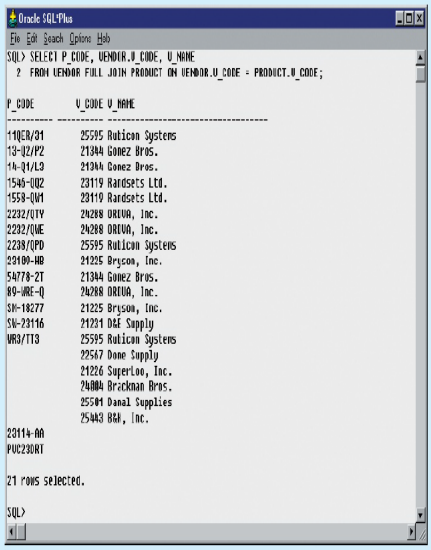
Full (left+right outer) JOIN example

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



'SELECT' subqueries

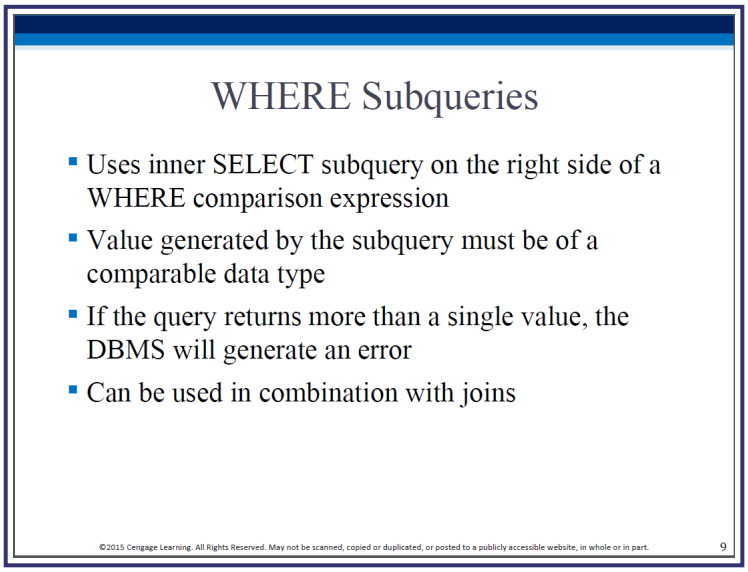
SELECT SUBQUERY EXAMPLES	EXPLANATION
INSERT INTO PRODUCT SELECT * FROM P;	Inserts all rows from Table P into the PRODUCT table. Both tables must have the same attributes. The subquery returns all rows from Table P.
UPDATE PRODUCT SET P_PRICE = (SELECT AVG(P_PRICE) FROM PRODUCT) WHERE V_CODE IN (SELECT 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.

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

8

'WHERE' subqueries



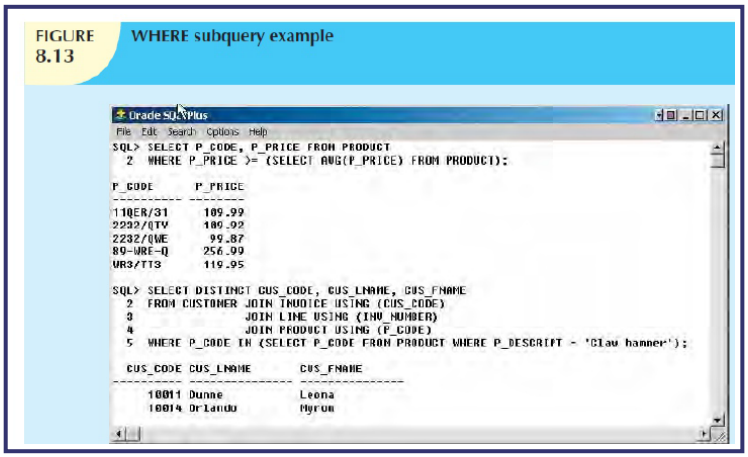
WHERE Subqueries

- 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

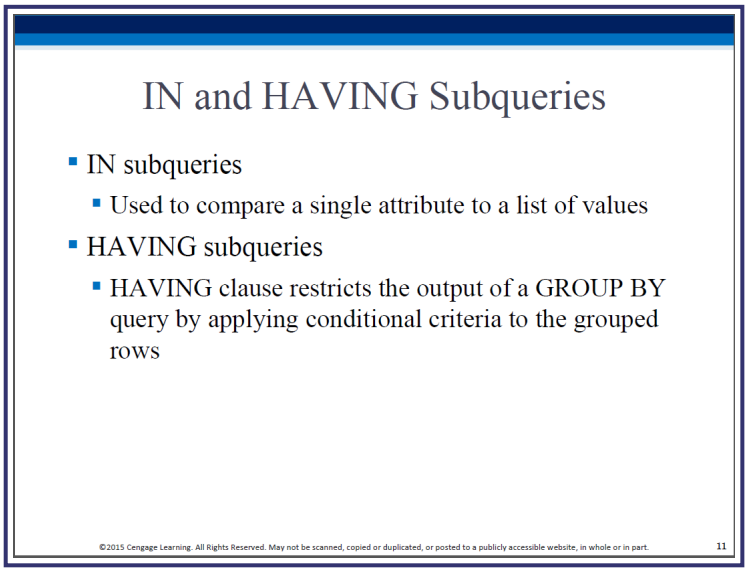
9

WHERE subquery example

FIGURE 8.13 WHERE subquery example



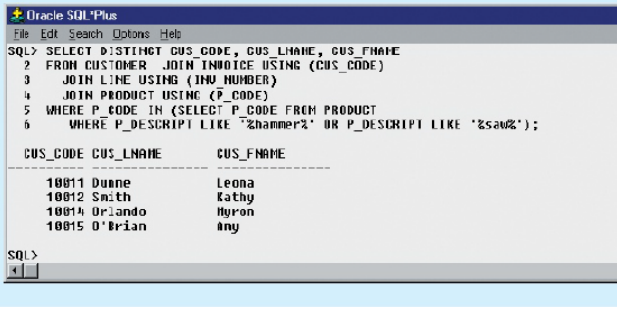
IN, HAVING subqueries



'IN' subqueries

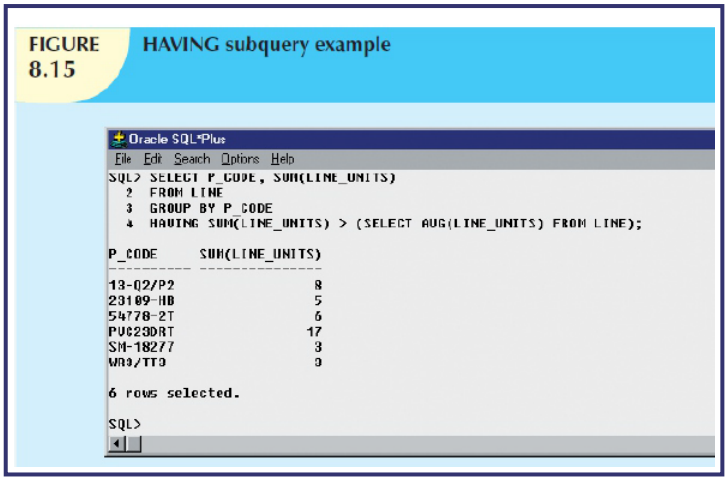
Compare against a LIST of values..

FIGURE 8.14 IN subquery example



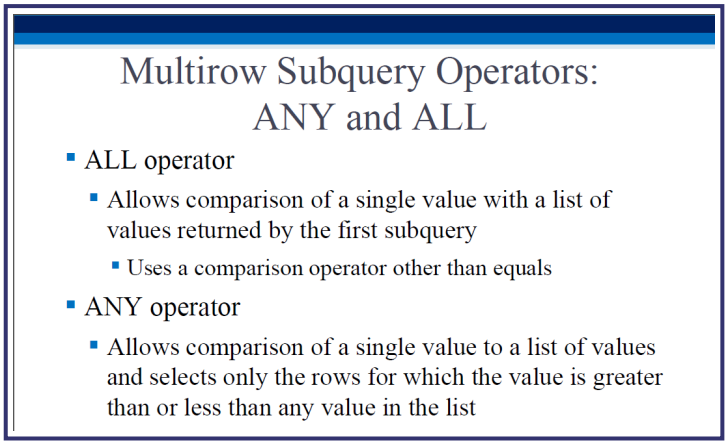
'HAVING' subqueries

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:



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.



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?"

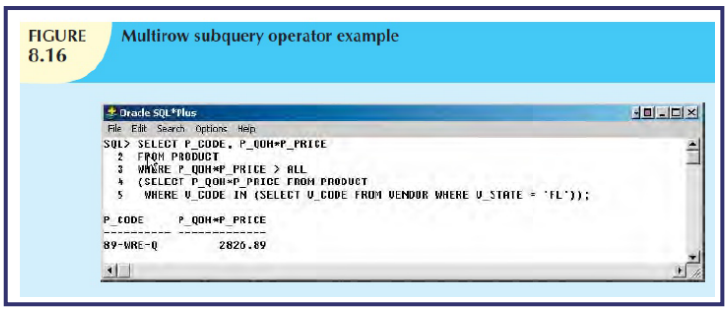


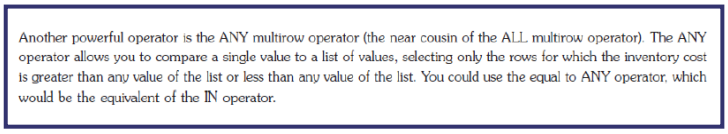
FIGURE 8.16 Multirow subquery operator example

Oracle SQL*Plus

File Edit Search Options Help

```
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'));

P_CODE     P_QOH*P_PRICE
---------- -------------
89-WRE-Q         2825.89
```

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':

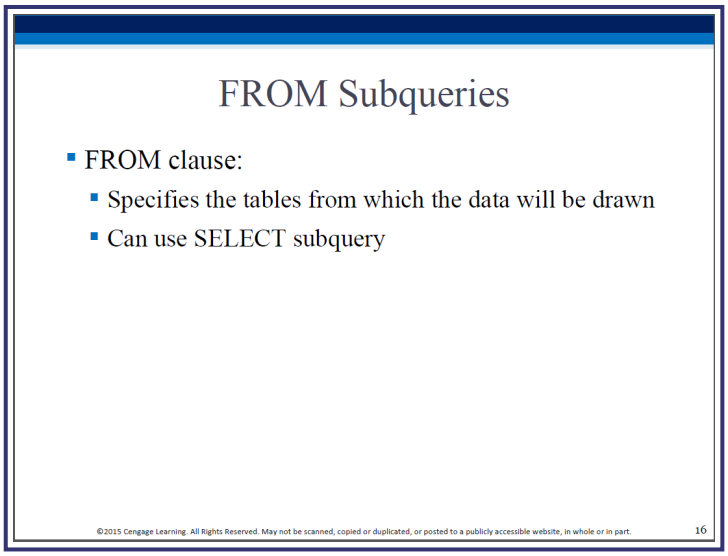
```
(M==6) OR (M==8) OR (M==10)

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

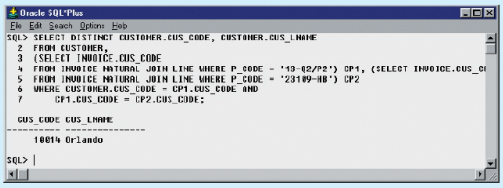
A SELECT query that appears in FROM, creates a **virtual table** against which the main query can run.



FROM subquery example

All customers who bought both specified products

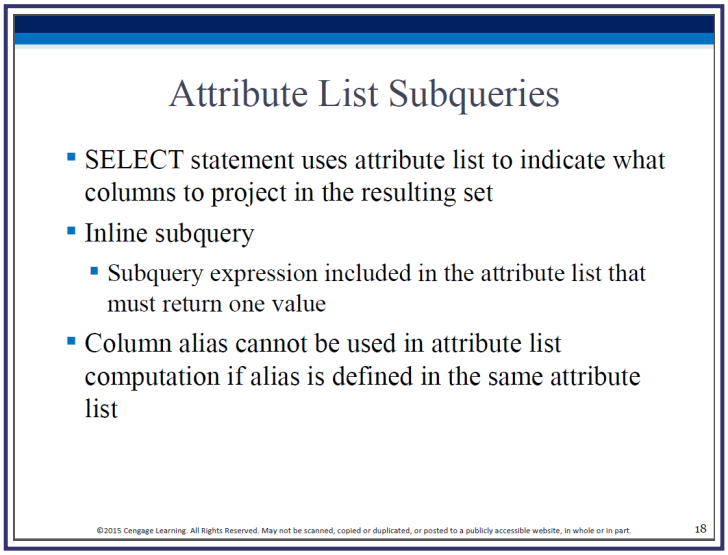
FIGURE 8.17 FROM subquery example



Attribute list subqueries

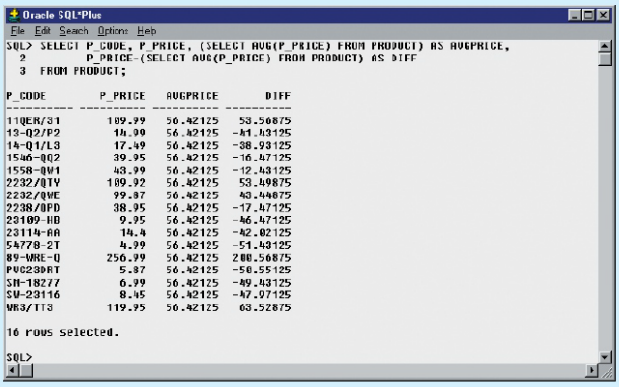
These subqueries determine what columns get output by the main query - they can be actual (existing) columns or computed columns or results of aggregate functions.

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



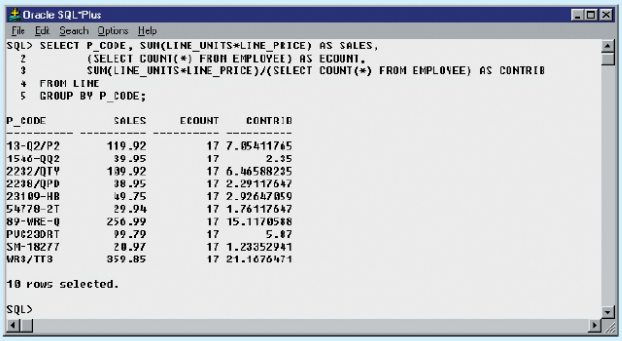
Attribute subquery example

FIGURE 8.18 Inline subquery example



Another attribute subquery example

FIGURE 8.19 Another example of an inline subquery

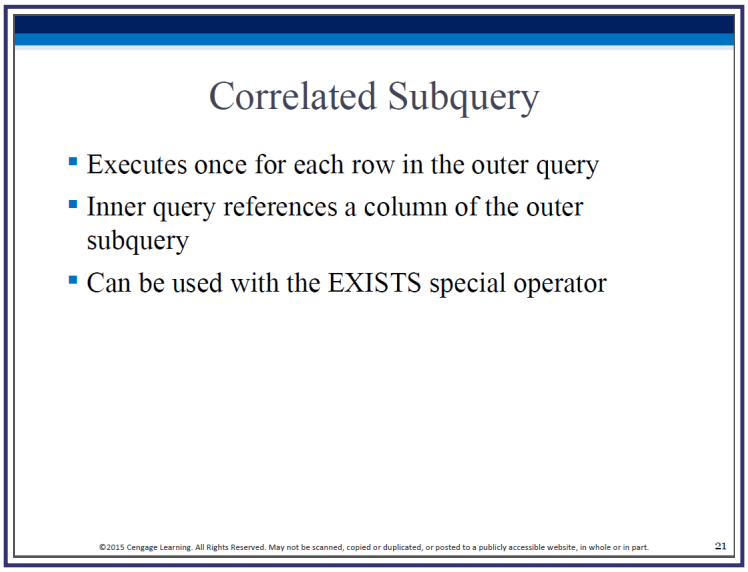


Oracle SQL*Plus

File Edit Search Options Help

```
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        139.92         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
PVC23DRT         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



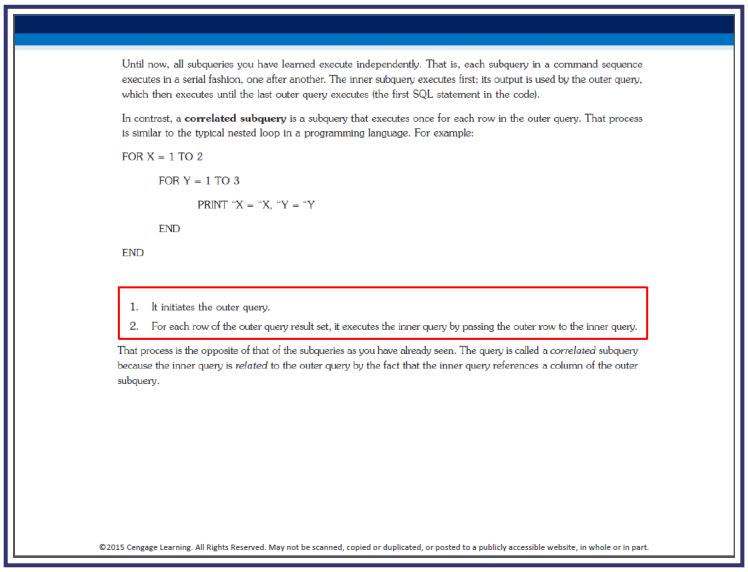
In a correlated subquery, the inner (sub) query is repeatedly run, for each row of the outer query! The inner is said to be (co-)related with the outer query when it references a column in the outer query's table. This is in effect, like a double (nested) 'for' loop..

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.

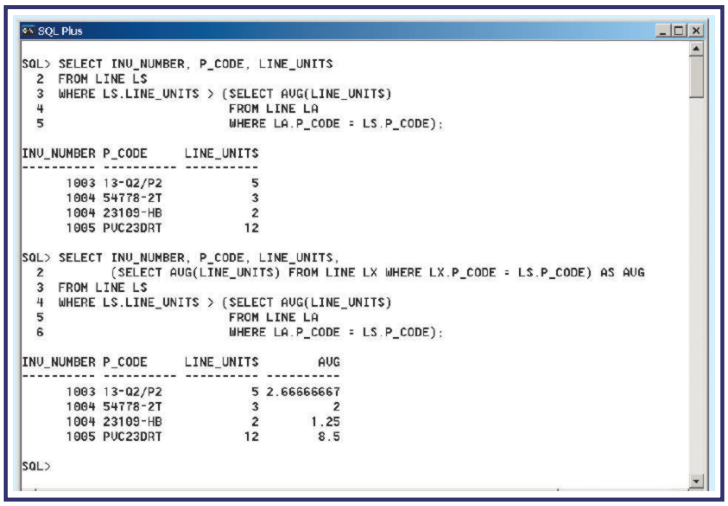
Correlated subquery examples

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 AVG
  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 2.66666667
      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.

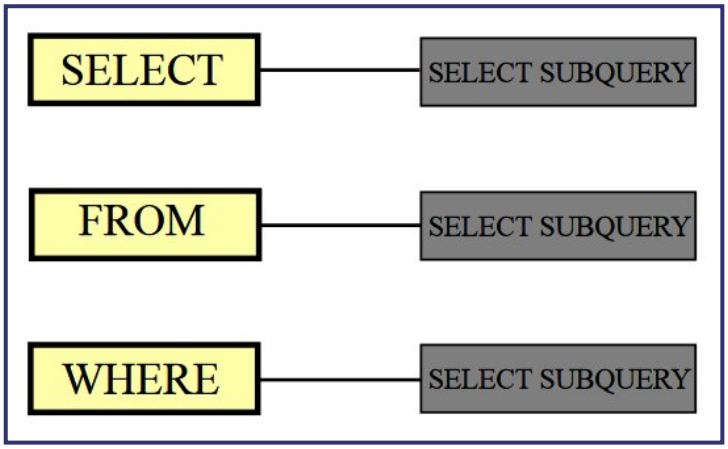
To verify the results and to provide an example of how you can combine subqueries, you can add a correlated inline subquery to the previous query. (See the second query and its results in Figure 8.14.) As you can see, the new query contains a correlated inline subquery that computes the average units 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.

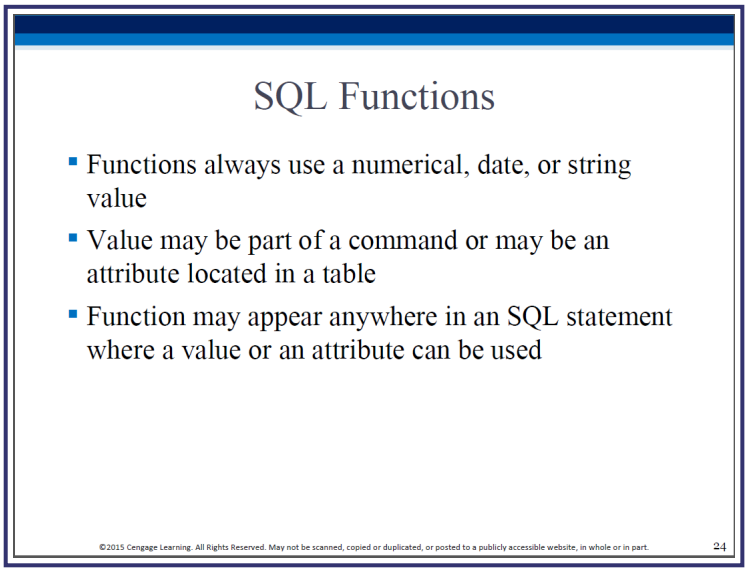
Queries: summary

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

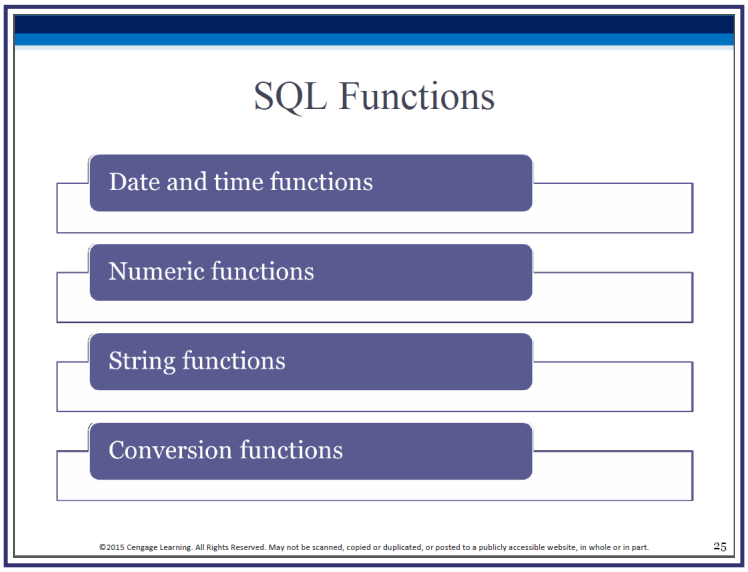
Most interestingly, a SELECT subquery can appear at the top (SELECT), middle (FROM) or bottom (WHERE) of a parent query, which provides a flexible way to express complex logic (since such subqueries can be recursively nested):



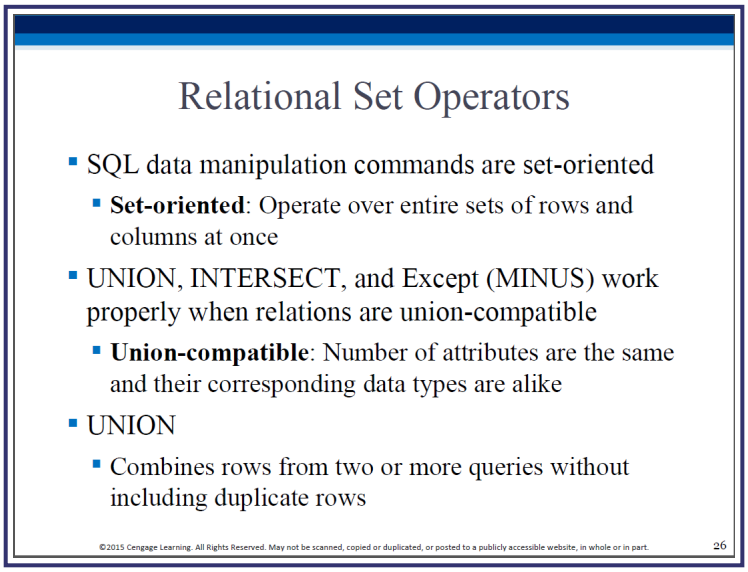
SQL functions



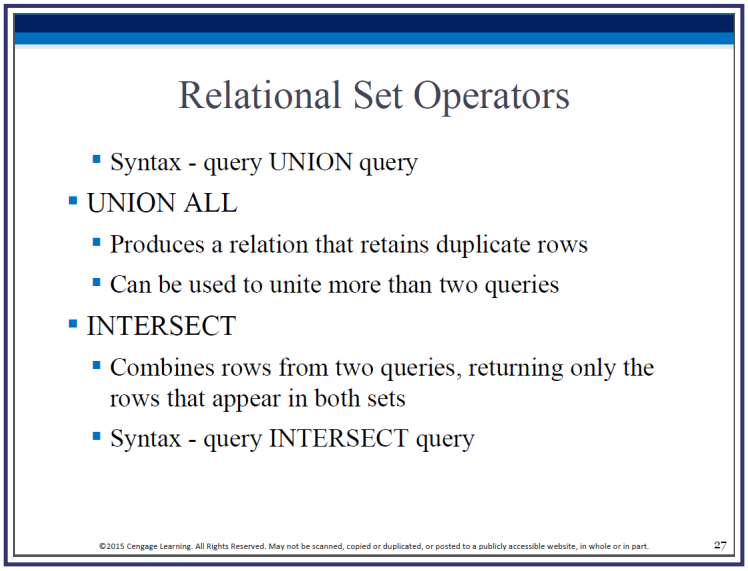
SQL functions [cont'd]



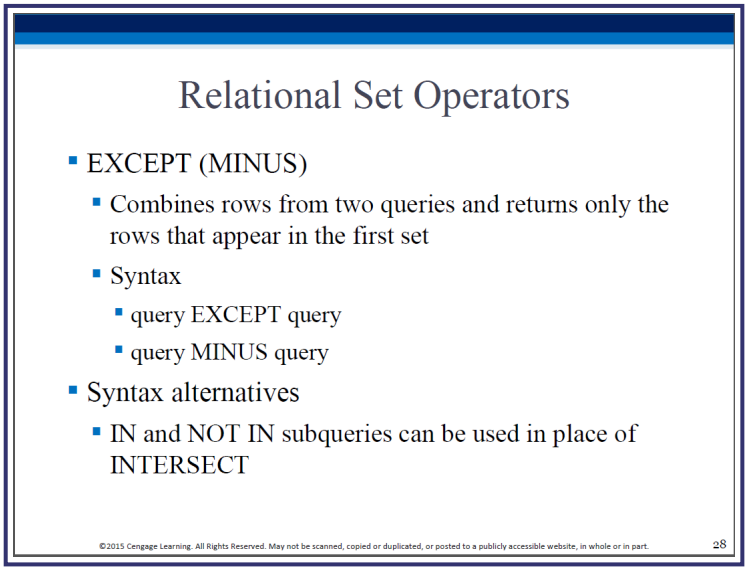
UNION, INTERSECTION, DIFFERENCE



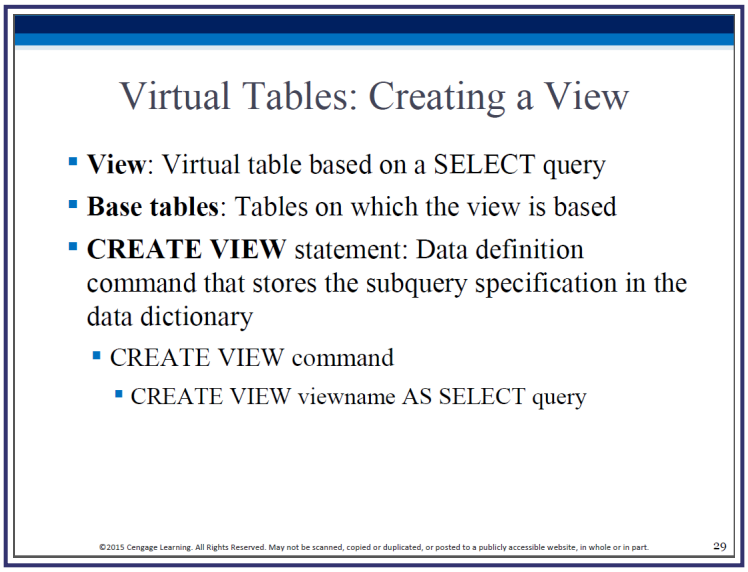
UNION, INTERSECTION, DIFFERENCE [cont'd]



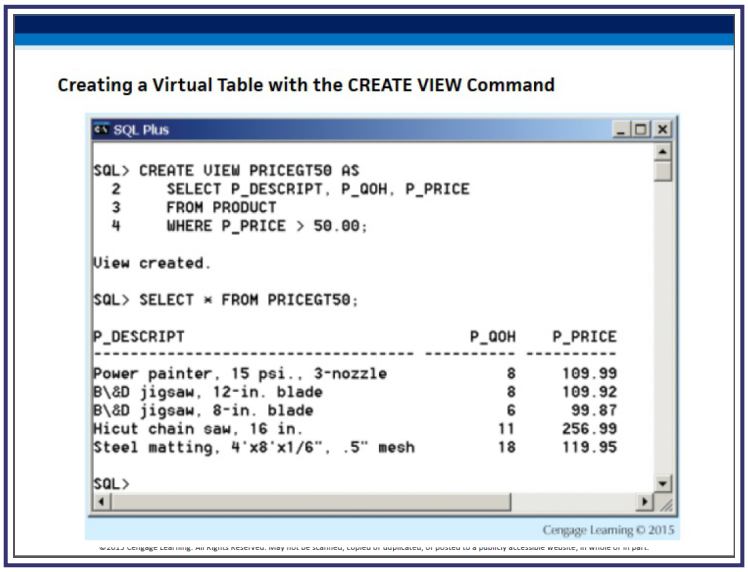
UNION, INTERSECTION, DIFFERENCE [cont'd]



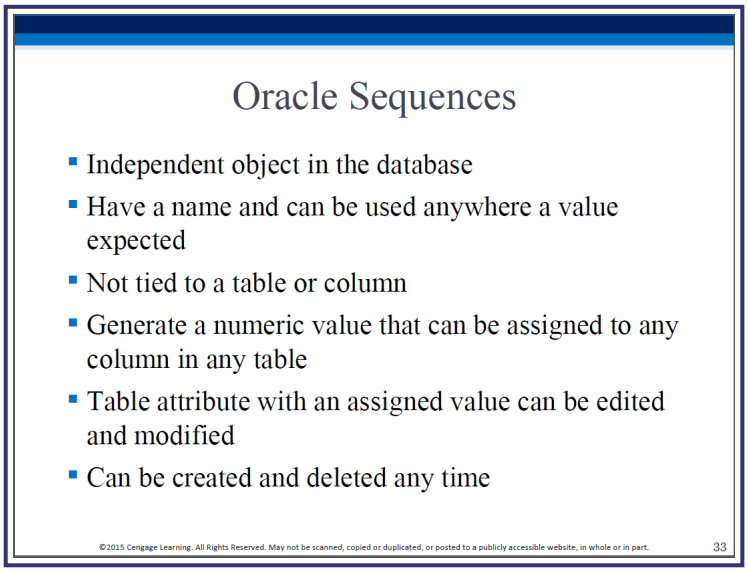
VIEWs


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
©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.
29

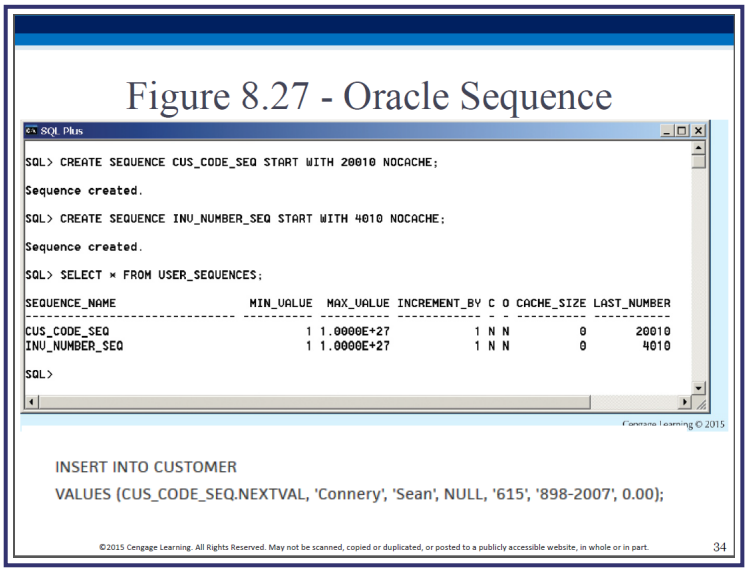
VIEW example



Sequences

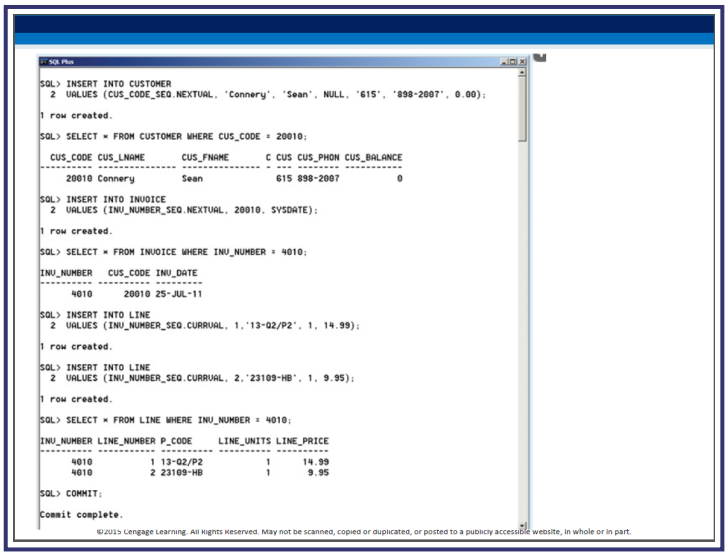

Oracle Sequences
Independent object in the database
Have a name and can be used anywhere a value expected
Not tied to a table or column
Generate a numeric value that can be assigned to any column in any table
Table attribute with an assigned value can be edited and modified
Can be created and deleted any time
©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.
33

Sequence creation example



Sequence: NEXTVAL, CURRVAL

NEXTVAL returns the current value, then does ++; CURRVAL just fetches the current value (does not ++ it).



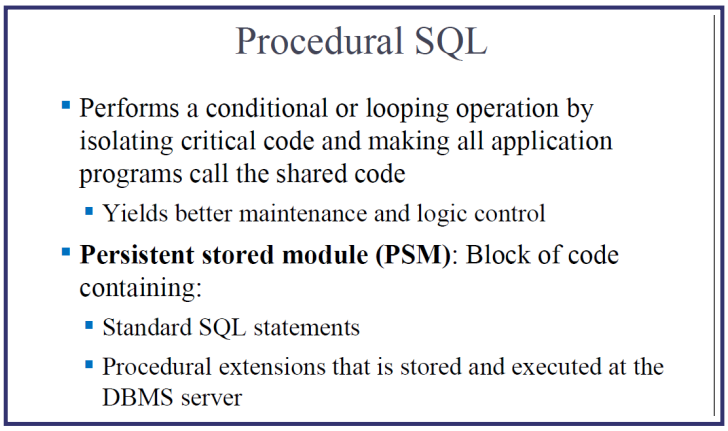
```
SQL> INSERT INTO CUSTOMER
  2  VALUES (CUS_CODE_SEQ.NEXTVAL, 'Connery', 'Sean', NULL, '615', '898-2007', 0.00);

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 (INV_NUMBER_SEQ.NEXTVAL, 20010, SYSDATE);

1 row created.

SQL> SELECT * FROM INVOICE WHERE INV_NUMBER = 4010;

INV_NUMBER   CUS_CODE INV_DATE
---------- ---------- ---------
      4010      20010 25-JUL-11

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

1 row created.

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

1 row created.

SQL> SELECT * FROM LINE WHERE INV_NUMBER = 4010;

INV_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.
```

Procedural Language SQL (PL/SQL)

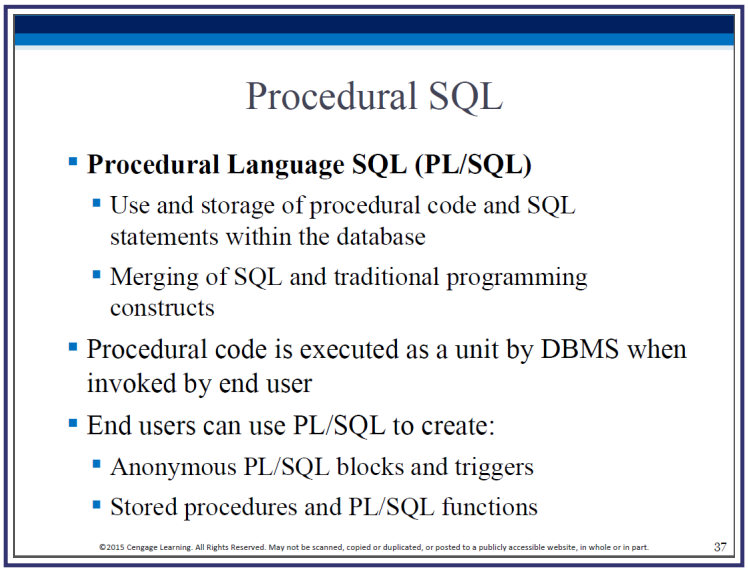
PL/SQL involves extra (augmented) syntax that lets us do looping, branching, variable declaration and function declaration - these are of course not possible using 'plain' SQL.

PL/SQL can be used to create:

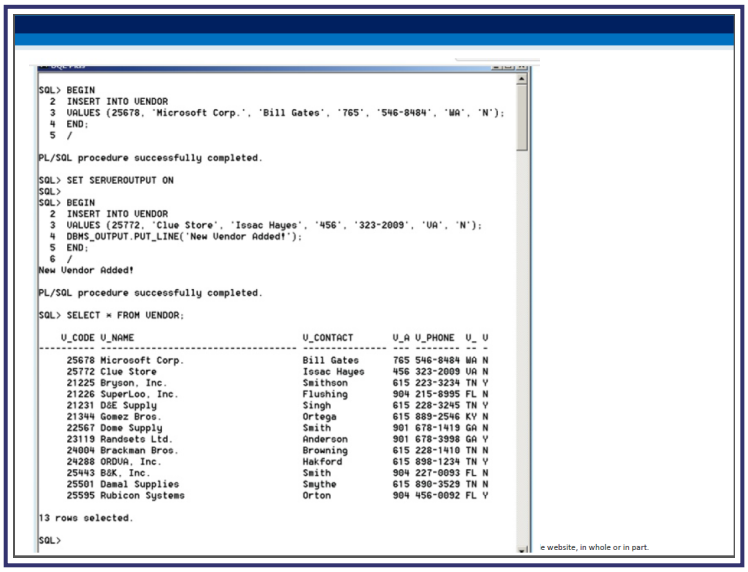
- **blocks of code** for one-time execution
- **triggers** - callbacks to invoke
- **stored procedures** - named procedures (no return values) for repeated calling
- **stored functions** - named functions (with return values) for repeated calling



PL/SQL [cont'd]

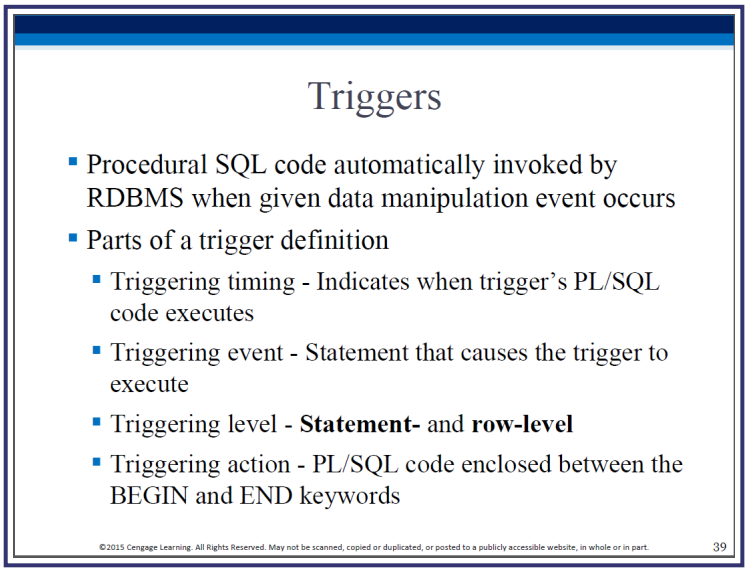

Procedural SQL
Procedural Language SQL (PL/SQL)
Use and storage of procedural code and SQL statements within the database
Merging of SQL and traditional programming constructs
Procedural code is executed as a unit by DBMS when invoked by end user
End users can use PL/SQL to create:
Anonymous PL/SQL blocks and triggers
Stored procedures and PL/SQL functions
©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.
37

[Unnamed] block creation example

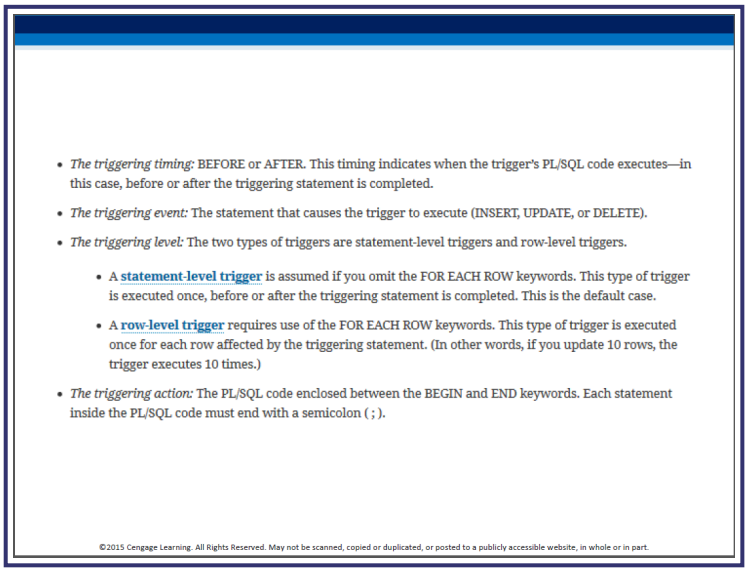


```
SQL> BEGIN
  2  INSERT INTO VENDOR
  3  VALUES (25678, 'Microsoft Corp.', 'Bill Gates', '765', '546-8484', 'WA', 'N');
  4  END;
  5  /

PL/SQL procedure successfully completed.

SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
  2  INSERT INTO VENDOR
  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 VENDOR;

    V_CODE V_NAME                             V_CONTACT       V_A V_PHONE  V_ V
---------- ---------------------------------- --------------- --- -------- -- -
     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 ORDVA, 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>
```

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
©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.
39

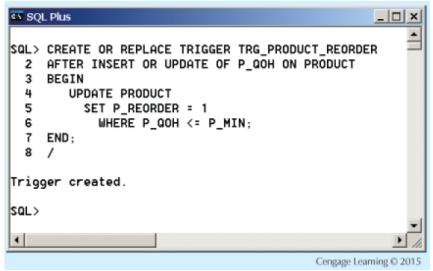
Triggers [cont'd]



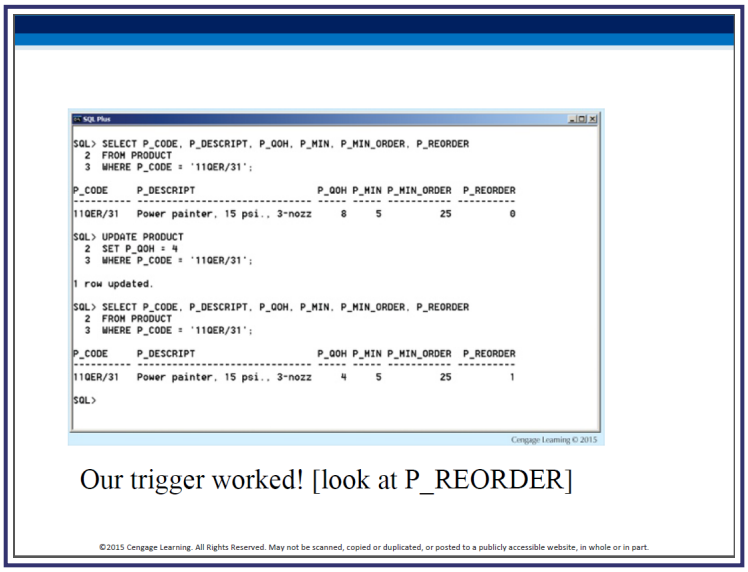
- *The triggering timing:* BEFORE or AFTER. This timing indicates when the trigger's PL/SQL code executes—in this case, before or after the triggering statement is completed.
- *The triggering event:* The statement that causes the trigger to execute (INSERT, UPDATE, or DELETE).
- *The triggering level:* 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_namedata type*[:=*initial_value*]]
BEGIN
PL/SQL instructions;
.....
END;

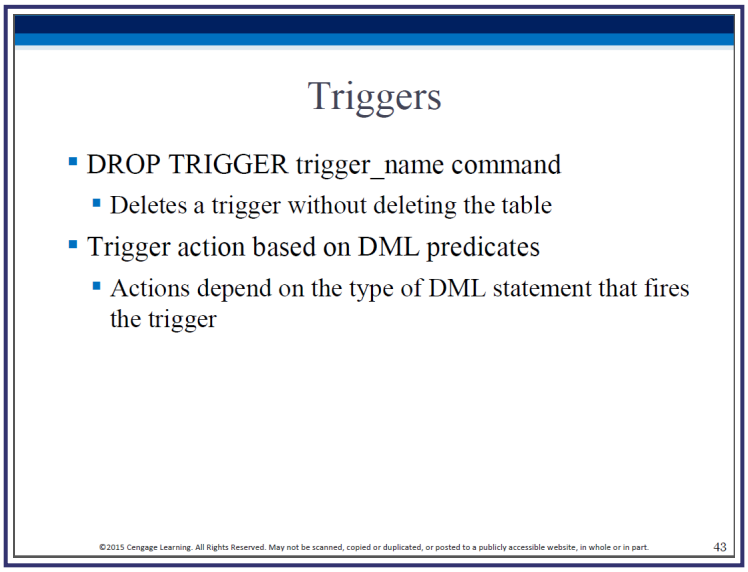


Trigger example

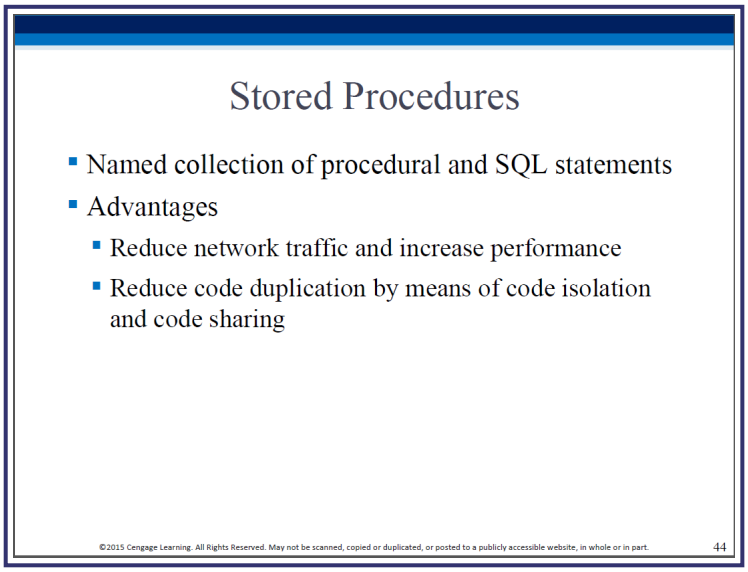


Our trigger worked! [look at P_REORDER]

Triggers [cont'd]



Stored procedures



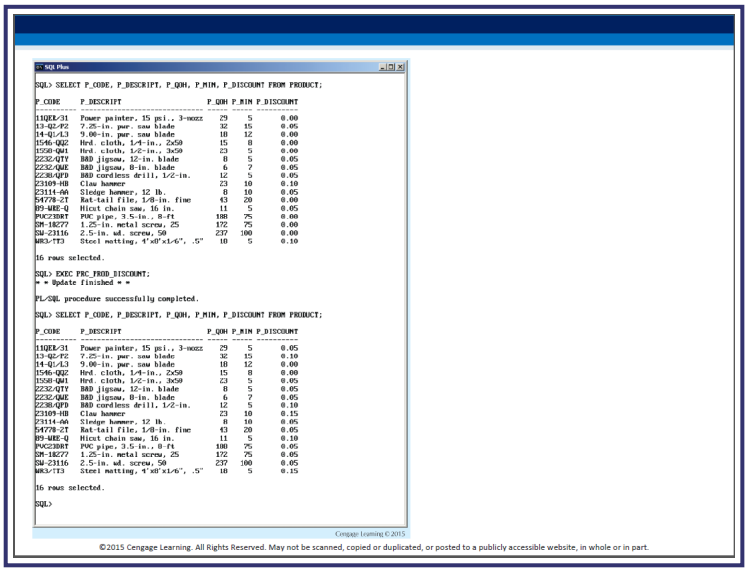
Stored procedure example

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

SQL Plus

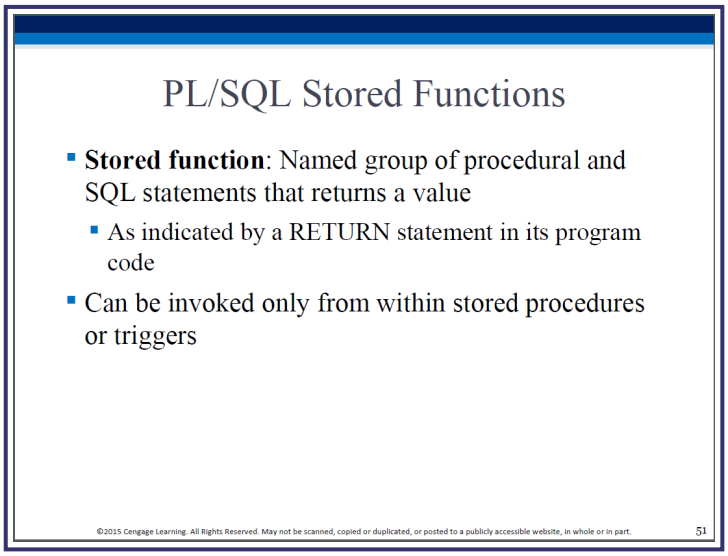
```
SQL> CREATE OR REPLACE PROCEDURE PRC_PROD_DISCOUNT
  2  AS BEGIN
  3     UPDATE PRODUCT
  4       SET P_DISCOUNT = P_DISCOUNT + .05
  5       WHERE P_QOH >= P_MIN * 2;
  6
  7     DBMS_OUTPUT.PUT_LINE ('* * Update finished * *');
  8  END;
  9  /

Procedure created.

SQL>
```

Stored procedure example



Stored functions

Reminder - these can RETURN a value.

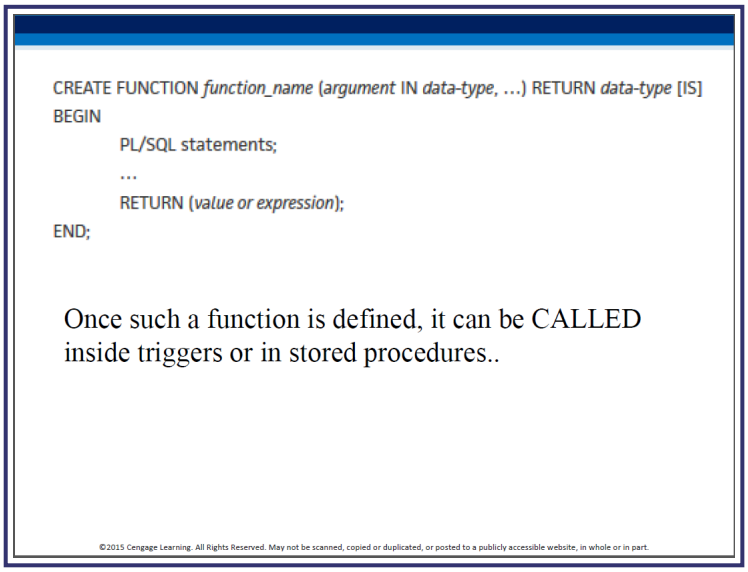


PL/SQL Stored Functions

- **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

51

Stored functions - syntax



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
```