Database Management Systems
Chapter 7: Advanced SQL

UNIVERSITY OF MARYLAND

ROBERT H. SMITH SCHOOL OF BUSINESS

DR. ADAM LEE

Objectives

- Define terms.
- Write single and multiple table SQL queries.
- Define and use three types of joins.
- Write noncorrelated and correlated subqueries.
- Understand and use SQL in procedural languages.
- Understand triggers and stored procedures.
- Discuss SQL:2008 standard and its enhancements and extensions.

Processing Multiple Tables

- **Join** – a relational operation that causes two or more tables with a common domain to be combined into a single table or view.
- **Equi-join** – a join in which the joining condition is based on equality between values in the common columns; common columns appear redundantly in the result table.
- **Natural join** – an equi-join in which one of the duplicate columns is eliminated in the result table.
- The common columns in joined tables are usually the primary key of the dominant table and the foreign key of the dependent table in 1:M relationships.

Processing Multiple Tables

- **Outer join** – a join in which rows that do not have matching values in common columns are nonetheless included in the result table (as opposed to inner join, in which rows must have matching values in order to appear in the result table).
- **Union join** – includes all columns from each table in the join, and an instance for each row of each table.

Figure 7-2: Visualization of Different Join Types with Results Returned in Shaded area

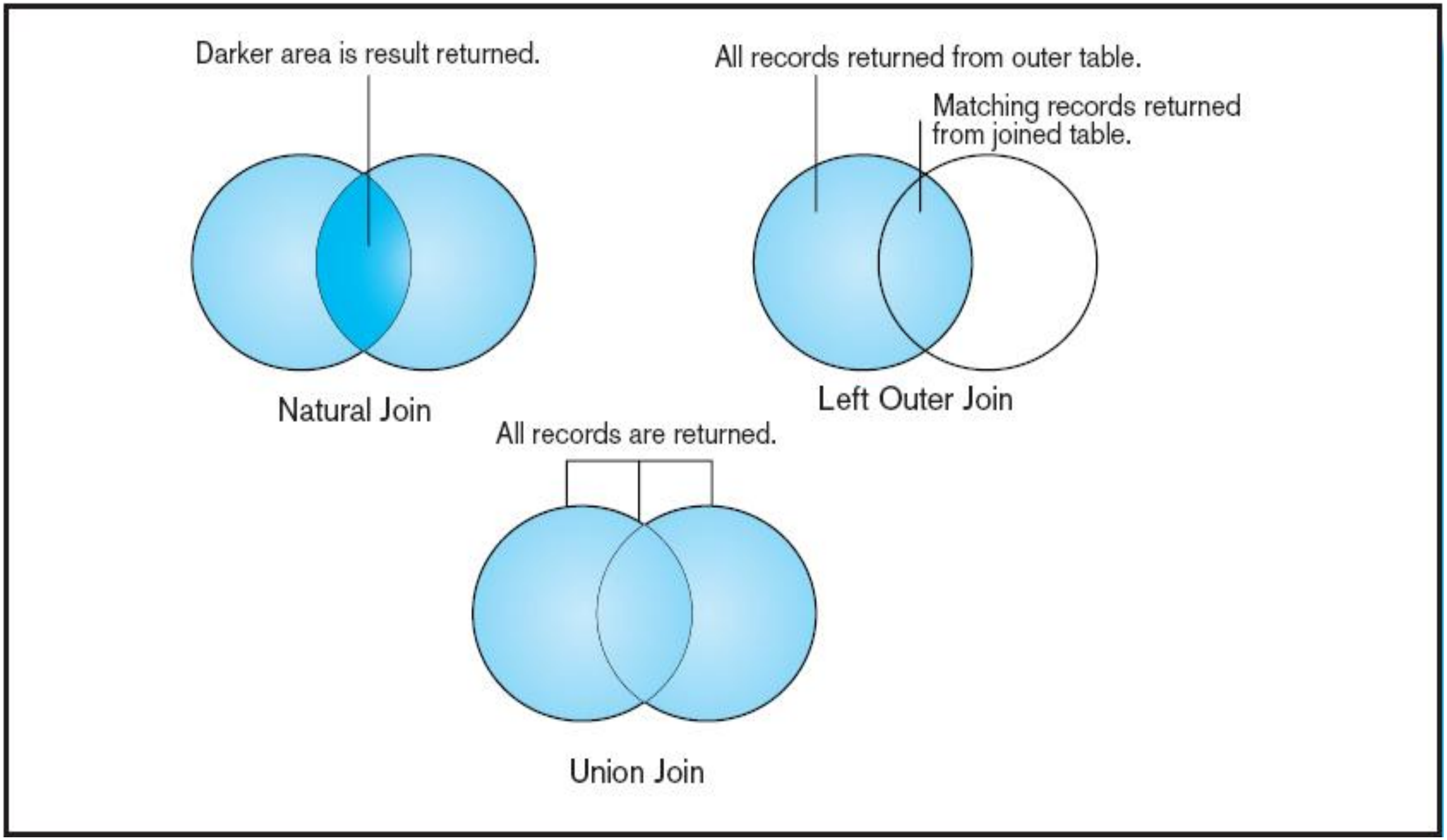

Darker area is result returned.
Natural Join
All records returned from outer table.
Matching records returned from joined table.
Left Outer Join
All records are returned.
Union Join

Figure 1-3: Project Level Data Models

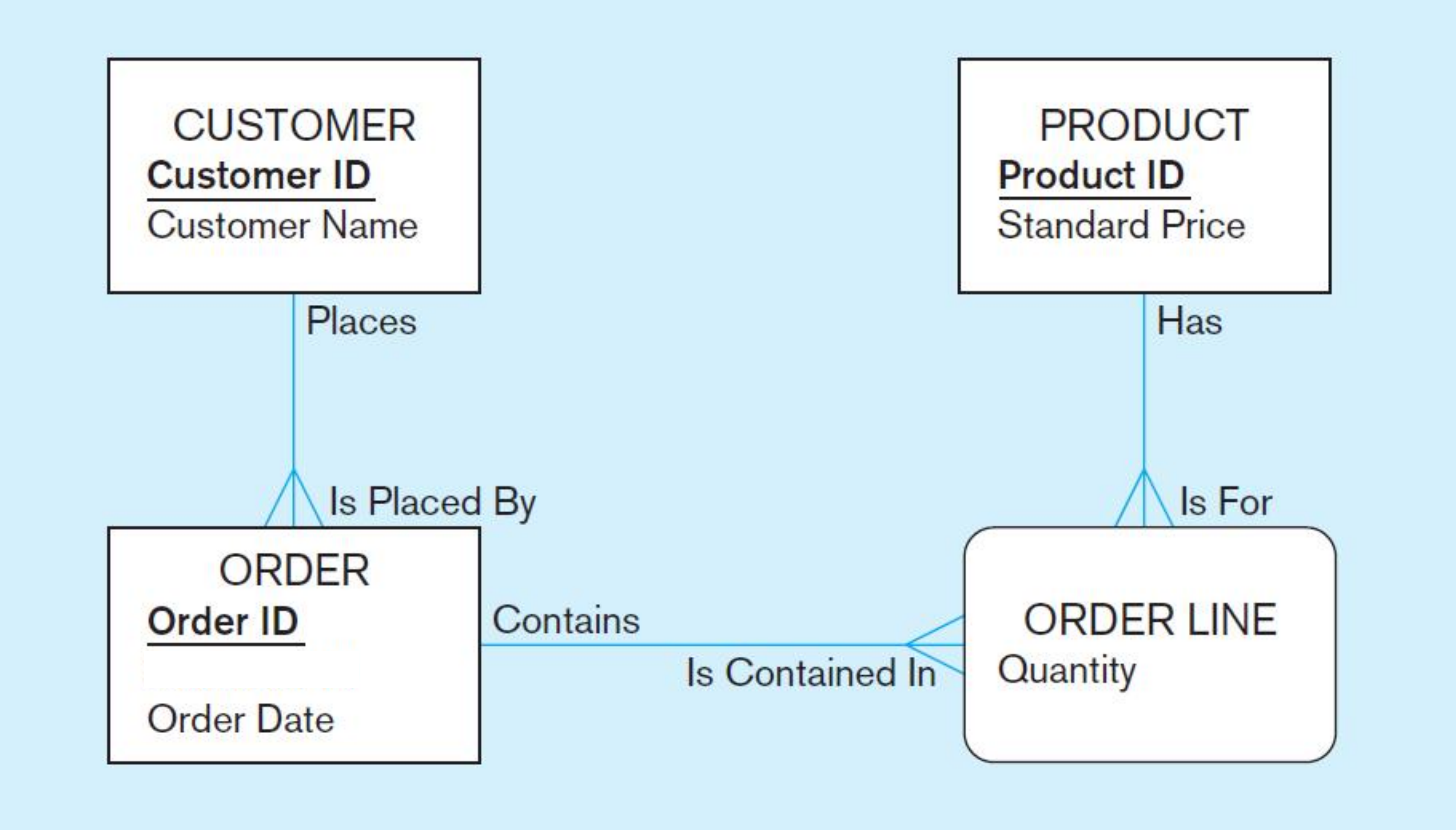
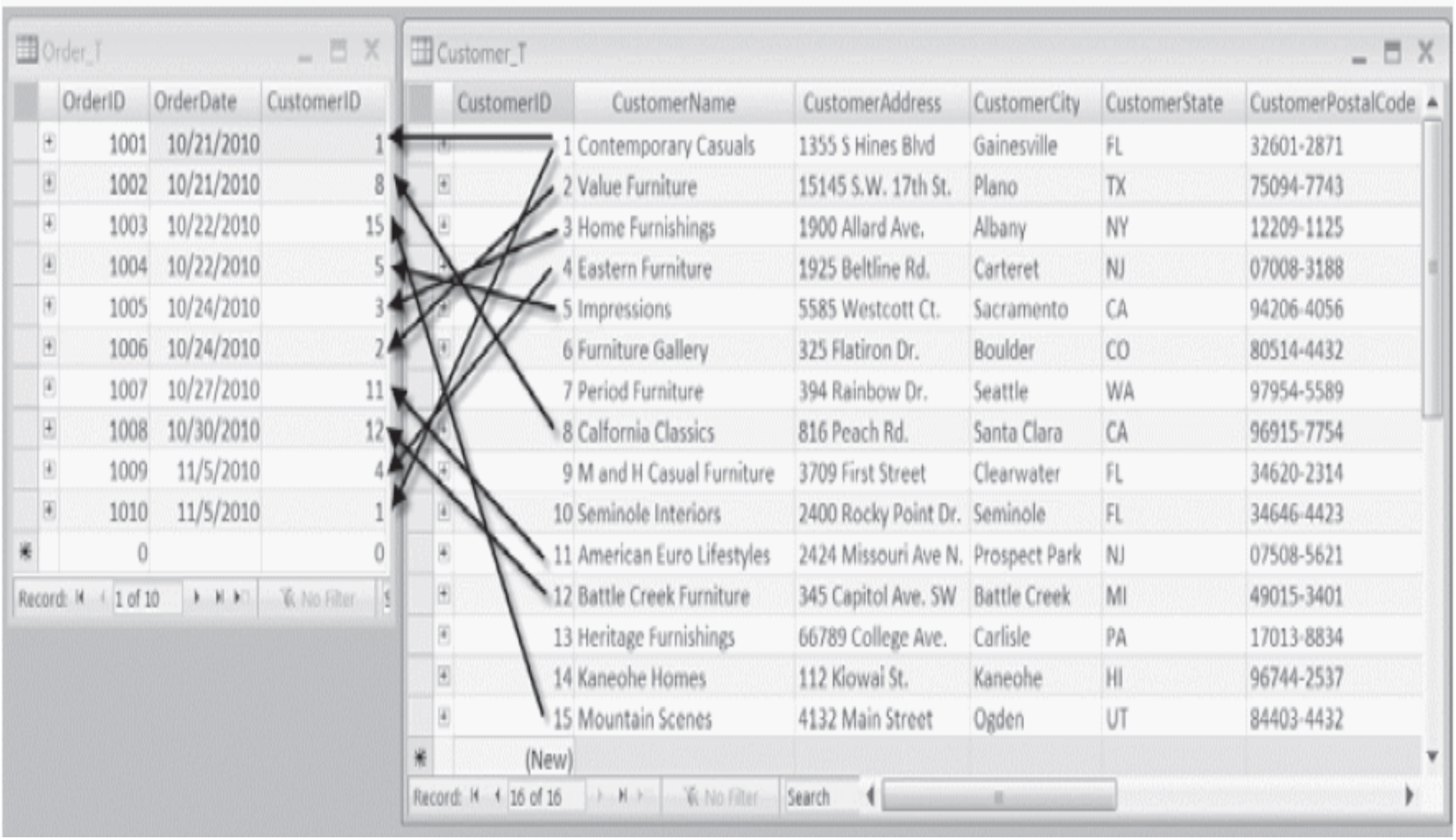


Figure 7-1: Customer_T and Order_T Tables with Pointers from Customers to Their Orders

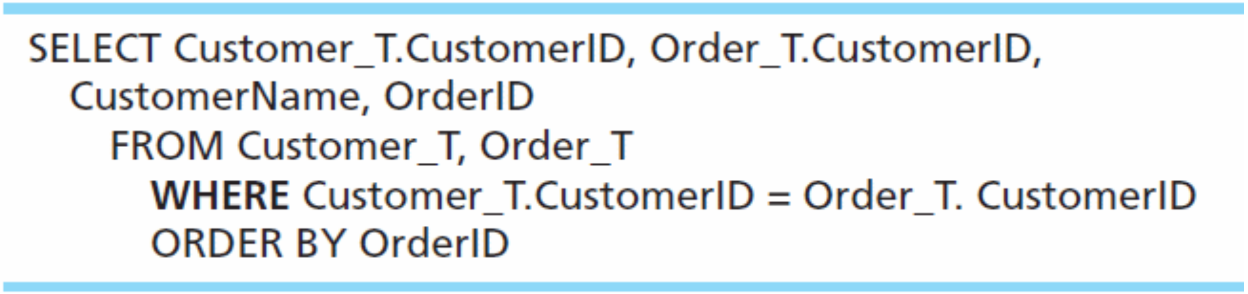


OrderID	OrderDate	CustomerID
1001	10/21/2010	1
1002	10/21/2010	8
1003	10/22/2010	15
1004	10/22/2010	5
1005	10/24/2010	3
1006	10/24/2010	2
1007	10/27/2010	11
1008	10/30/2010	12
1009	11/5/2010	4
1010	11/5/2010	1
0		0

CustomerID	CustomerName	CustomerAddress	CustomerCity	CustomerState	CustomerPostalCode
1	Contemporary Casuals	1355 S Hines Blvd	Gainesville	FL	32601-2871
2	Value Furniture	15145 S.W. 17th St.	Plano	TX	75094-7743
3	Home Furnishings	1900 Allard Ave.	Albany	NY	12209-1125
4	Eastern Furniture	1925 Beltline Rd.	Carteret	NJ	07008-3188
5	Impressions	5585 Westcott Ct.	Sacramento	CA	94206-4056
6	Furniture Gallery	325 Flatiron Dr.	Boulder	CO	80514-4432
7	Period Furniture	394 Rainbow Dr.	Seattle	WA	97954-5589
8	California Classics	816 Peach Rd.	Santa Clara	CA	96915-7754
9	M and H Casual Furniture	3709 First Street	Clearwater	FL	34620-2314
10	Seminole Interiors	2400 Rocky Point Dr.	Seminole	FL	34646-4423
11	American Euro Lifestyles	2424 Missouri Ave N.	Prospect Park	NJ	07508-5621
12	Battle Creek Furniture	345 Capitol Ave. SW	Battle Creek	MI	49015-3401
13	Heritage Furnishings	66789 College Ave.	Carlisle	PA	17013-8834
14	Kaneohe Homes	112 Kiowai St.	Kaneohe	HI	96744-2537
15	Mountain Scenes	4132 Main Street	Ogden	UT	84403-4432
(New)					

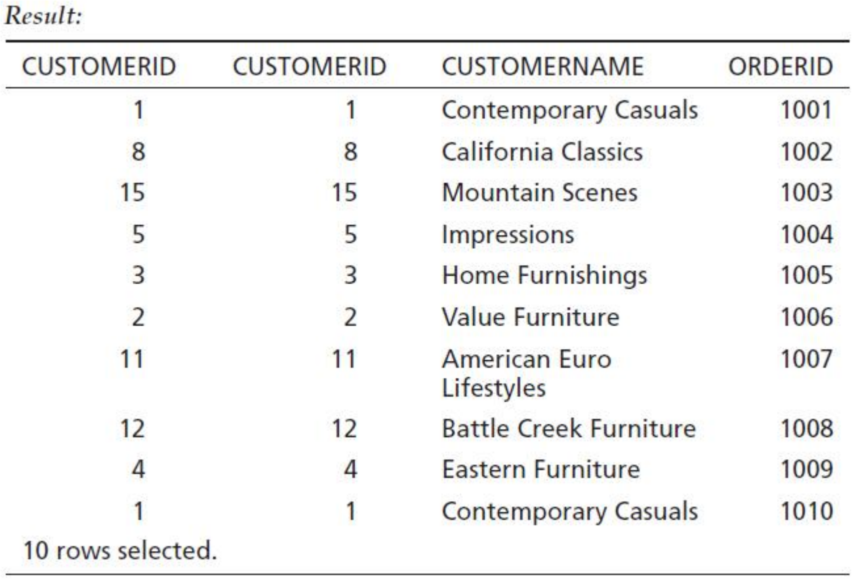
Equi-Join Example

- For each customer who placed an order, what is the customer's name and order number?



```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
   CustomerName, OrderID
      FROM Customer_T, Order_T
         WHERE Customer_T.CustomerID = Order_T. CustomerID
         ORDER BY OrderID
```

Result:

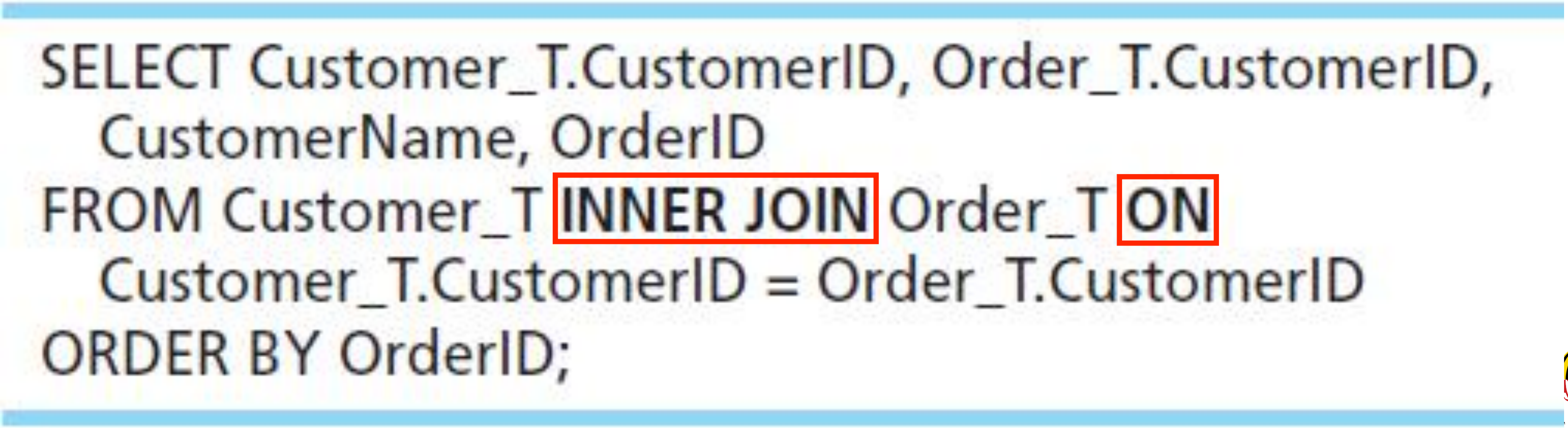


```
CUSTOMERID   CUSTOMERID   CUSTOMERNAME              ORDERID
         1            1   Contemporary Casuals         1001
         8            8   California Classics          1002
        15           15   Mountain Scenes              1003
         5            5   Impressions                  1004
         3            3   Home Furnishings             1005
         2            2   Value Furniture              1006
        11           11   American Euro                1007
                          Lifestyles
        12           12   Battle Creek Furniture       1008
         4            4   Eastern Furniture            1009
         1            1   Contemporary Casuals         1010
10 rows selected.
```

CustomerID appears twice in the result

Equi-Join Example – Alternative Syntax

- **INNER JOIN** clause is an alternative to **WHERE** clause, and is used to match primary and foreign keys.
- An **INNER JOIN** will only return rows from each table that have matching rows in the other.
- This query produces same results as previous equi-join example.

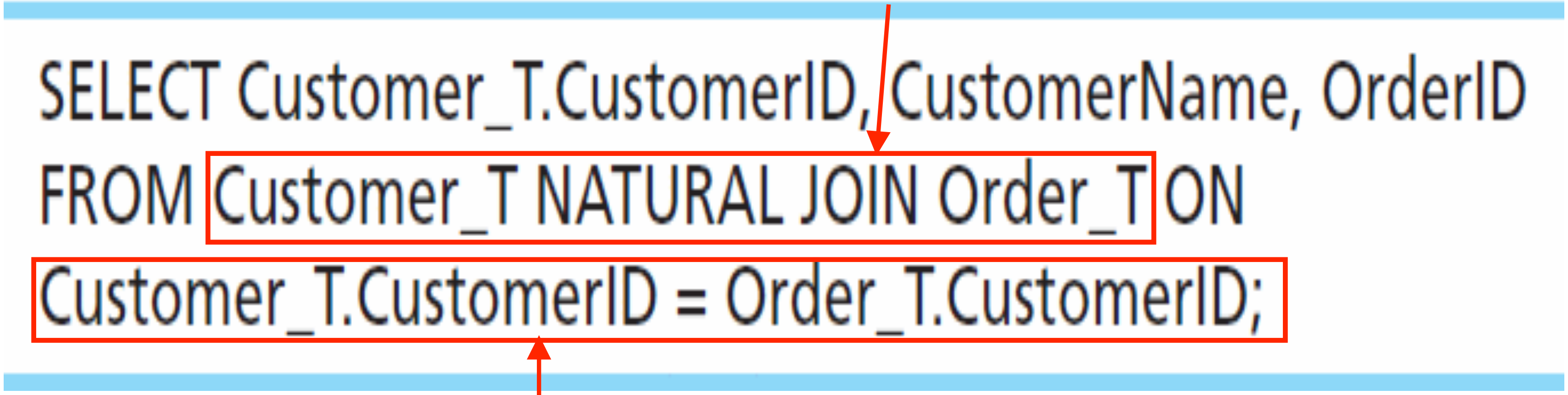


```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
   CustomerName, OrderID
FROM Customer_T INNER JOIN Order_T ON
   Customer_T.CustomerID = Order_T.CustomerID
ORDER BY OrderID;
```

Natural Join Example

- For each customer who placed an order, what is the customer's name and order number?

Join involves multiple tables in FROM clause



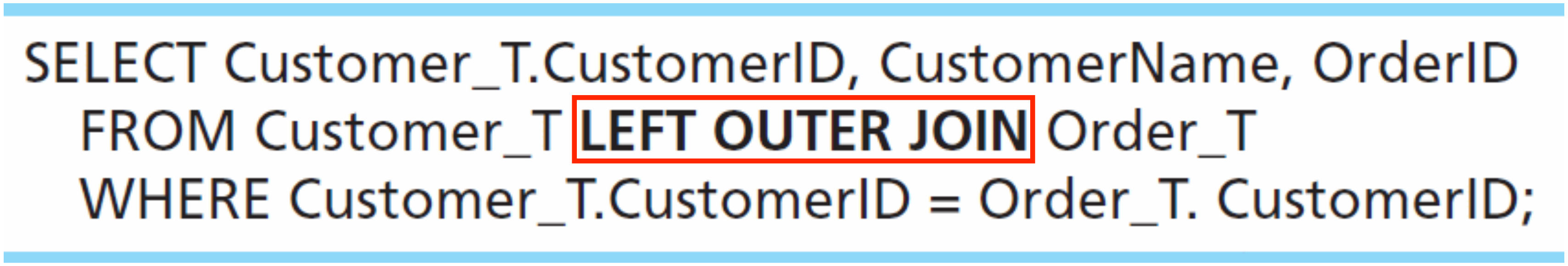
```
SELECT Customer_T.CustomerID, CustomerName, OrderID
FROM Customer_T NATURAL JOIN Order_T ON
Customer_T.CustomerID = Order_T.CustomerID;
```

ON clause performs the equality check for common columns of the two tables

Note: From Fig. 7-1, you see that only 10 Customers have links with orders.
➔ Only 10 rows will be returned from this INNER JOIN

Outer Join Example

- List the customer name, ID number, and order number for all customers. Include customer information even for customers that do have an order.

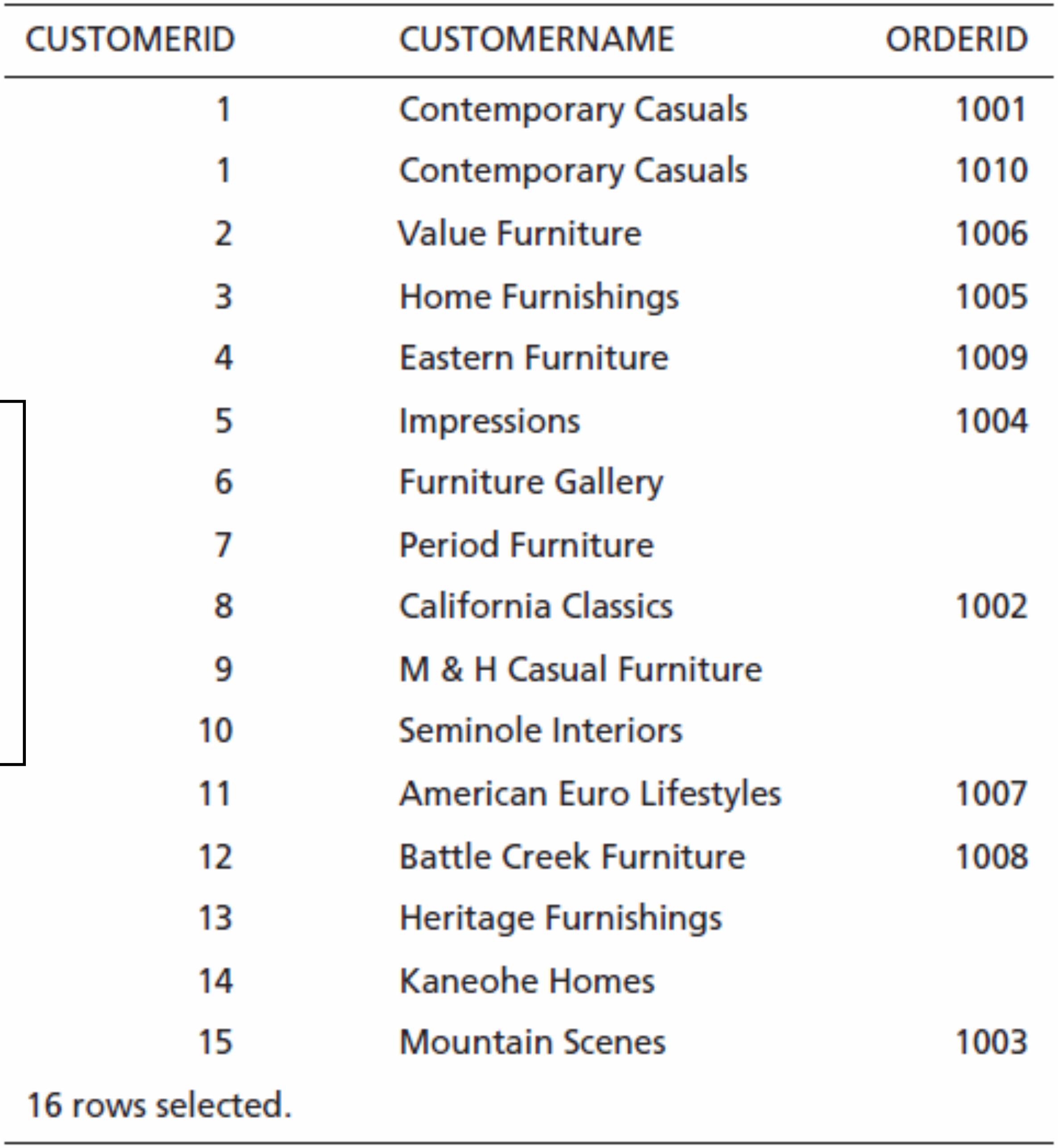


```
SELECT Customer_T.CustomerID, CustomerName, OrderID
   FROM Customer_T LEFT OUTER JOIN Order_T
   WHERE Customer_T.CustomerID = Order_T. CustomerID;
```

LEFT OUTER JOIN clause causes customer data to appear even if there is no corresponding order data

Unlike INNER join, this will include customer rows with no matching order rows

Outer Join Result



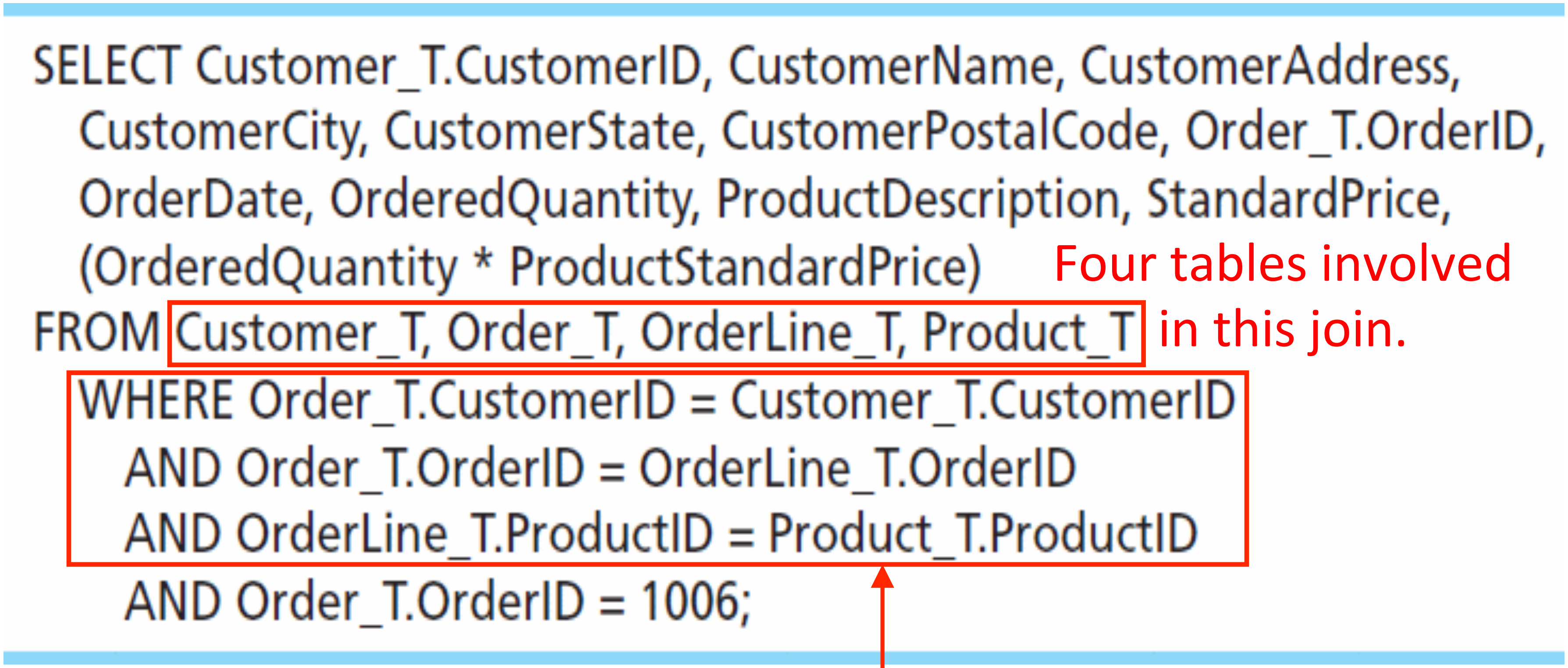
CUSTOMERID	CUSTOMERNAME	ORDERID
1	Contemporary Casuals	1001
1	Contemporary Casuals	1010
2	Value Furniture	1006
3	Home Furnishings	1005
4	Eastern Furniture	1009
5	Impressions	1004
6	Furniture Gallery	
7	Period Furniture	
8	California Classics	1002
9	M & H Casual Furniture	
10	Seminole Interiors	
11	American Euro Lifestyles	1007
12	Battle Creek Furniture	1008
13	Heritage Furnishings	
14	Kaneohe Homes	
15	Mountain Scenes	1003

16 rows selected.

Unlike INNER JOIN, this will include customer rows with no matching order rows

Multiple Table Join Example

- Assemble all information necessary to create an invoice for order number 1006.



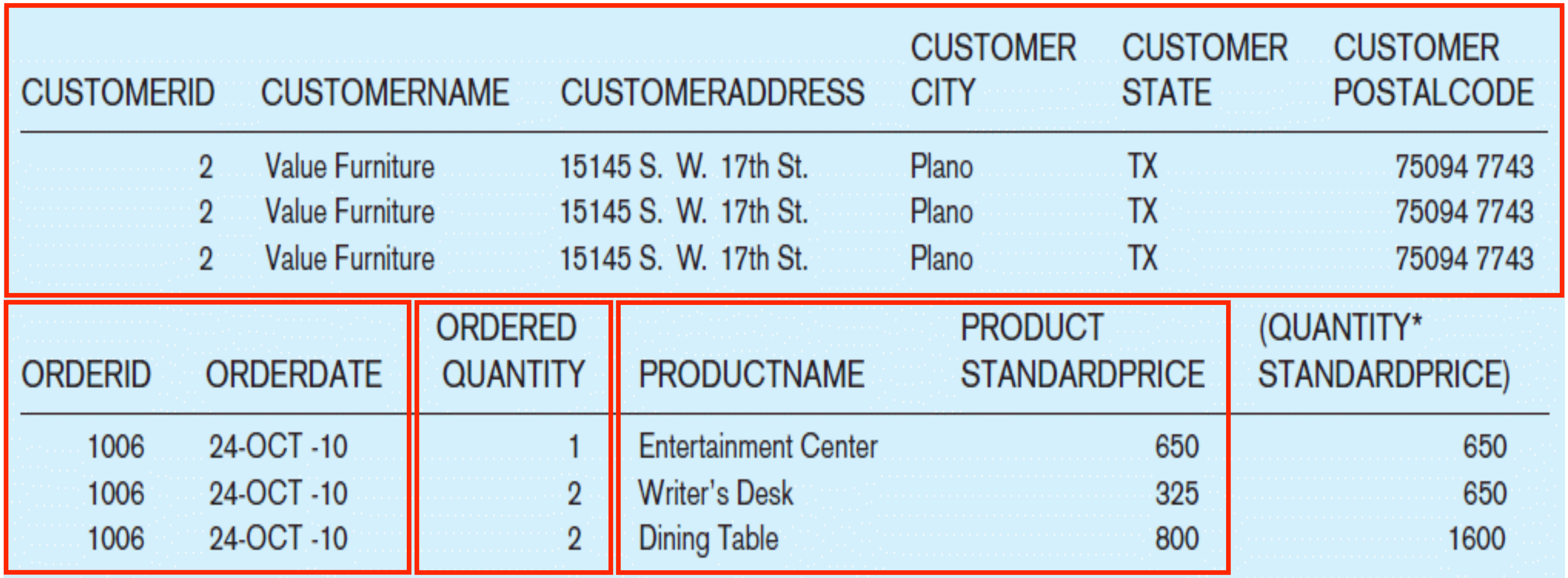
```
SELECT Customer_T.CustomerID, CustomerName, CustomerAddress,
   CustomerCity, CustomerState, CustomerPostalCode, Order_T.OrderID,
   OrderDate, OrderedQuantity, ProductDescription, StandardPrice,
   (OrderedQuantity * ProductStandardPrice)
FROM Customer_T, Order_T, OrderLine_T, Product_T
   WHERE Order_T.CustomerID = Customer_T.CustomerID
      AND Order_T.OrderID = OrderLine_T.OrderID
      AND OrderLine_T.ProductID = Product_T.ProductID
      AND Order_T.OrderID = 1006;
```

Four tables involved in this join.

Each pair of tables requires an equality-check condition in the WHERE clause, matching primary keys against foreign keys.

Figure 7-4: Results from a Four-Table Join (Edited for Readability)

From Customer_T table



CUSTOMERID	CUSTOMERNAME	CUSTOMERADDRESS	CUSTOMER CITY	CUSTOMER STATE	CUSTOMER POSTALCODE
2	Value Furniture	15145 S. W. 17th St.	Plano	TX	75094 7743
2	Value Furniture	15145 S. W. 17th St.	Plano	TX	75094 7743
2	Value Furniture	15145 S. W. 17th St.	Plano	TX	75094 7743

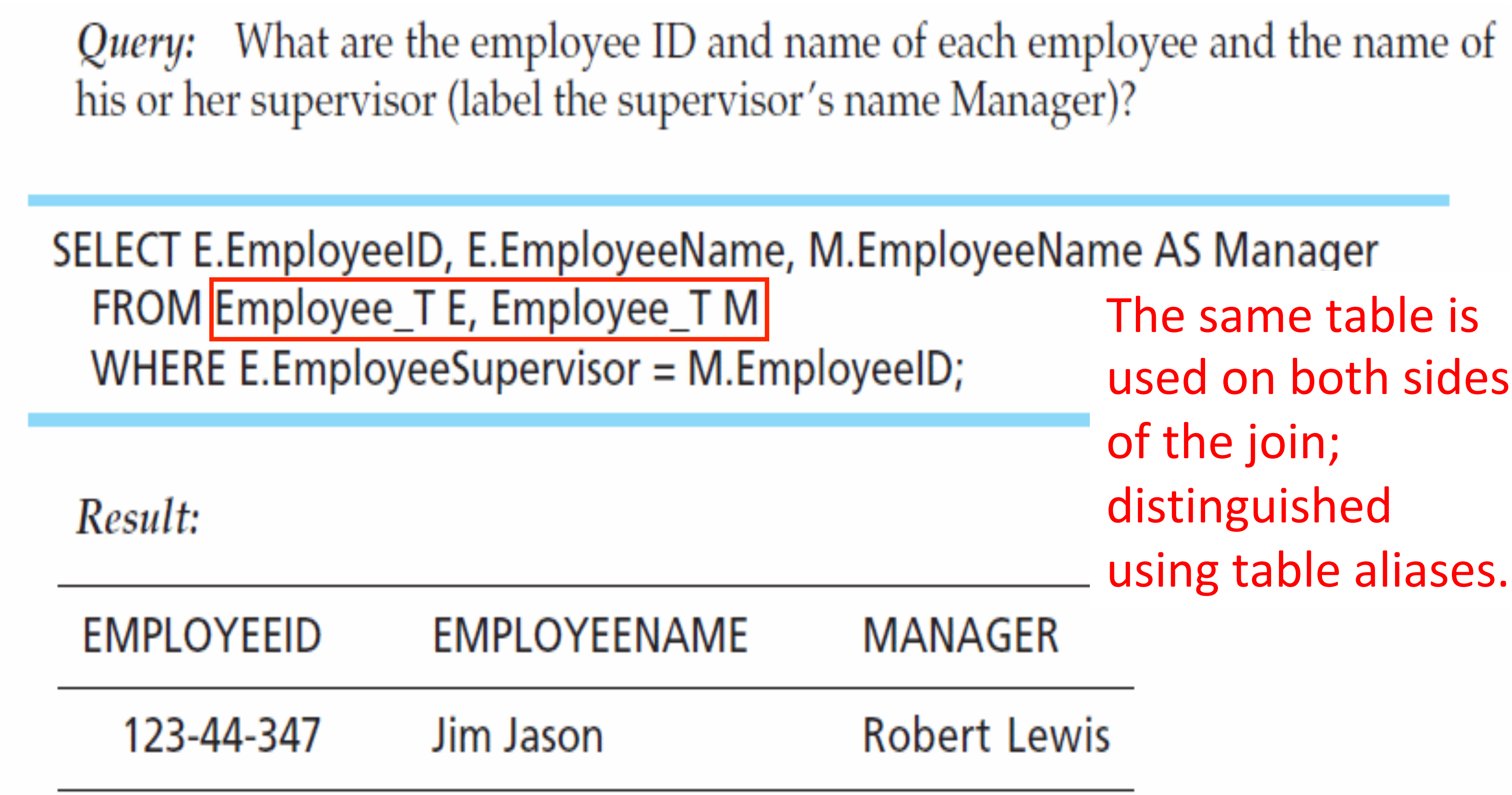
ORDERID	ORDERDATE	ORDERED QUANTITY	PRODUCTNAME	PRODUCT STANDARDPRICE	(QUANTITY* STANDARDPRICE)
1006	24-OCT -10	1	Entertainment Center	650	650
1006	24-OCT -10	2	Writer's Desk	325	650
1006	24-OCT -10	2	Dining Table	800	1600

From Order_T table

From Product_T table

From OrderLine_T table

Self-Join Example



Query: What are the employee ID and name of each employee and the name of his or her supervisor (label the supervisor's name Manager)?

```
SELECT E.EmployeeID, E.EmployeeName, M.EmployeeName AS Manager
  FROM Employee_T E, Employee_T M
  WHERE E.EmployeeSupervisor = M.EmployeeID;
```

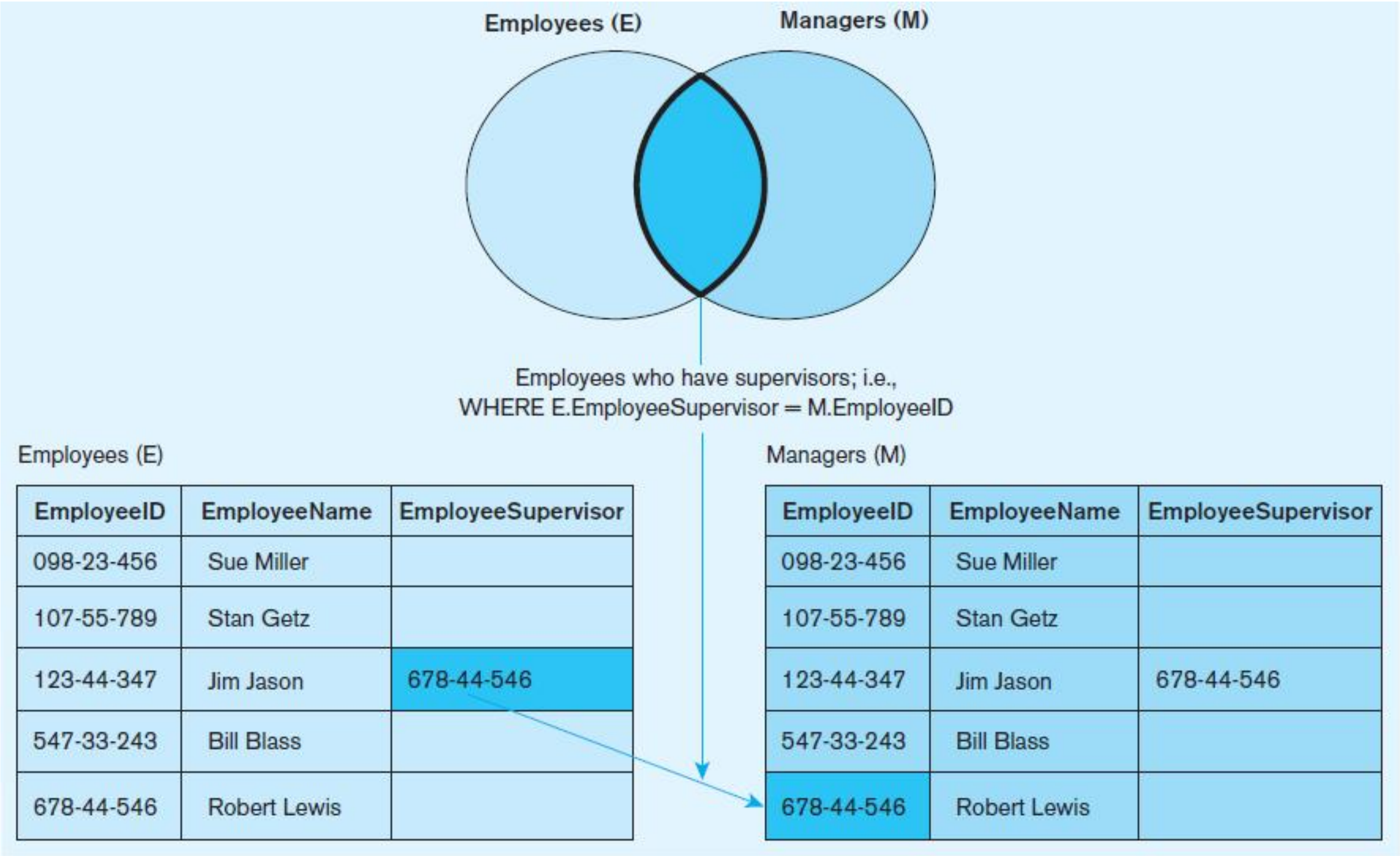
Result:

EMPLOYEEID	EMPLOYEENAME	MANAGER
123-44-347	Jim Jason	Robert Lewis

The same table is used on both sides of the join; distinguished using table aliases.

Self-joins are usually used on tables with unary relationships.

Figure 7-5: Example of a Self-Join



Employees who have supervisors; i.e.,
WHERE E.EmployeeSupervisor = M.EmployeeID

Employees (E)

EmployeeID	EmployeeName	EmployeeSupervisor
098-23-456	Sue Miller	
107-55-789	Stan Getz	
123-44-347	Jim Jason	678-44-546
547-33-243	Bill Blass	
678-44-546	Robert Lewis	

Managers (M)

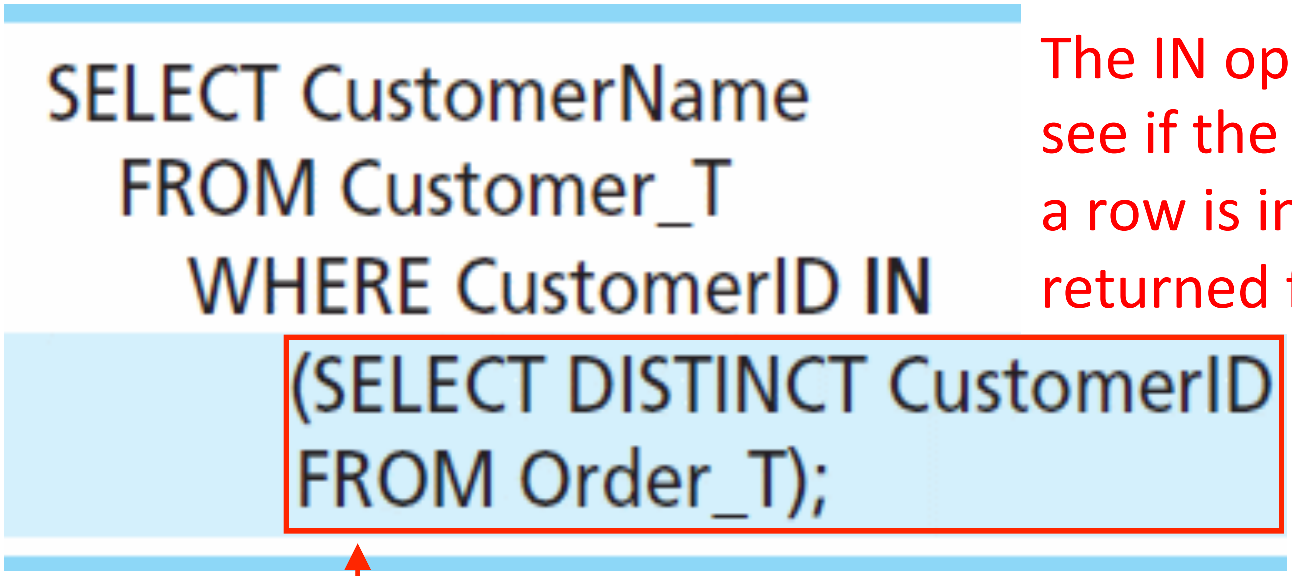
EmployeeID	EmployeeName	EmployeeSupervisor
098-23-456	Sue Miller	
107-55-789	Stan Getz	
123-44-347	Jim Jason	678-44-546
547-33-243	Bill Blass	
678-44-546	Robert Lewis	

Processing Multiple Tables Using Subqueries

- **Subquery** – placing an inner query (**SELECT** statement) inside an outer query.
- Options:
  - In a condition of the **WHERE** clause.
  - As a "table" of the **FROM** clause.
  - Within the **HAVING** clause.
- Subqueries can be:
  - **Noncorrelated** – executed once for the entire outer query.
  - **Correlated** – executed once for each row returned by the outer query.

Subquery Example

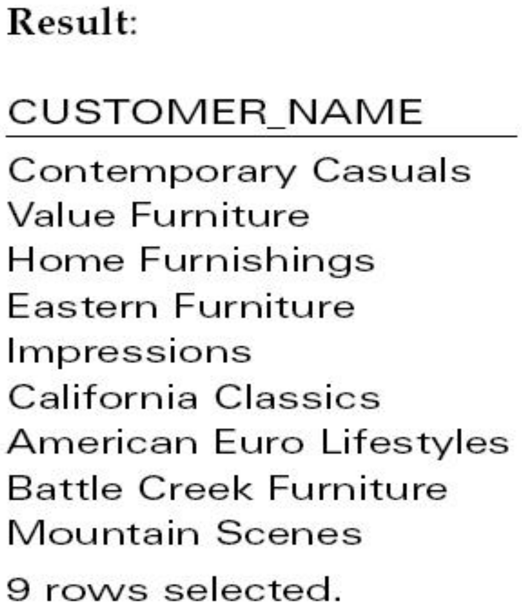
- Show all customers who have placed an order.



```
SELECT CustomerName
   FROM Customer_T
      WHERE CustomerID IN
         (SELECT DISTINCT CustomerID
         FROM Order_T);
```

The IN operator will test to see if the CustomerID value of a row is included in the list returned from the subquery.

Subquery is embedded in parentheses. In this case it returns a list that will be used in the WHERE clause of the outer query.



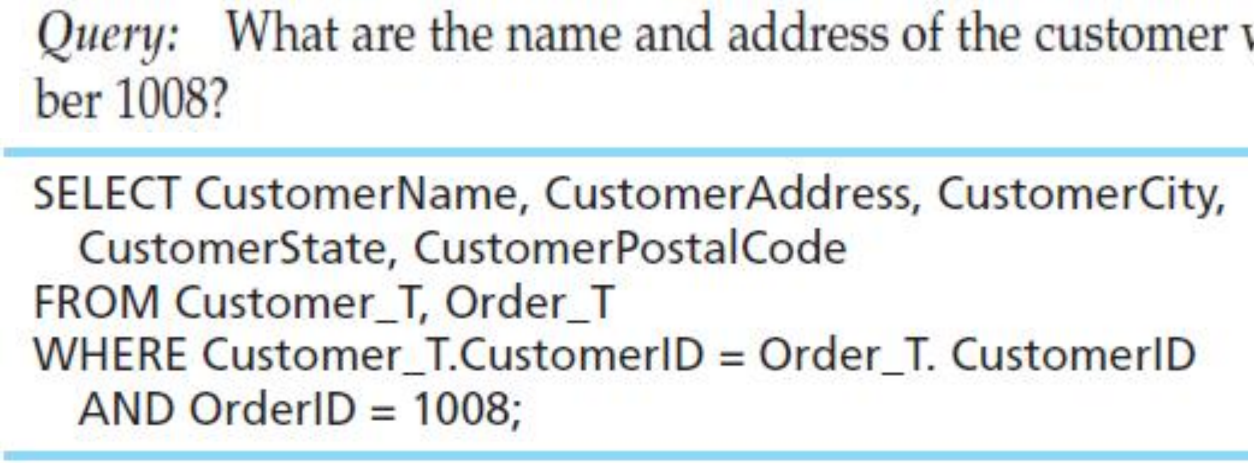
```
Result:

CUSTOMER_NAME
_______________________
Contemporary Casuals
Value Furniture
Home Furnishings
Eastern Furniture
Impressions
California Classics
American Euro Lifestyles
Battle Creek Furniture
Mountain Scenes

9 rows selected.
```

Join Vs. Subquery

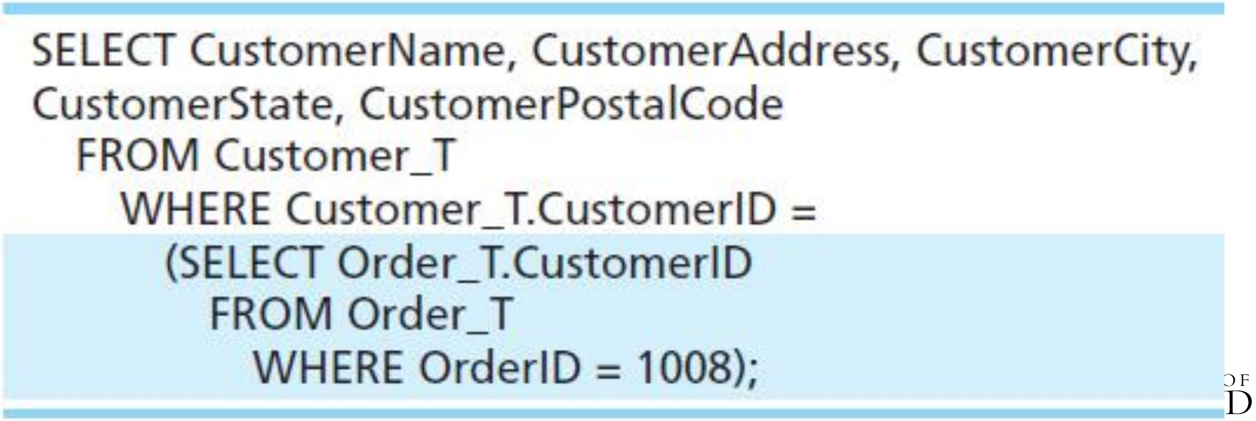
- Some queries could be accomplished by either a join or a subquery.



Query: What are the name and address of the customer who placed order number 1008?

```
SELECT CustomerName, CustomerAddress, CustomerCity,
   CustomerState, CustomerPostalCode
FROM Customer_T, Order_T
WHERE Customer_T.CustomerID = Order_T. CustomerID
   AND OrderID = 1008;
```

Join version

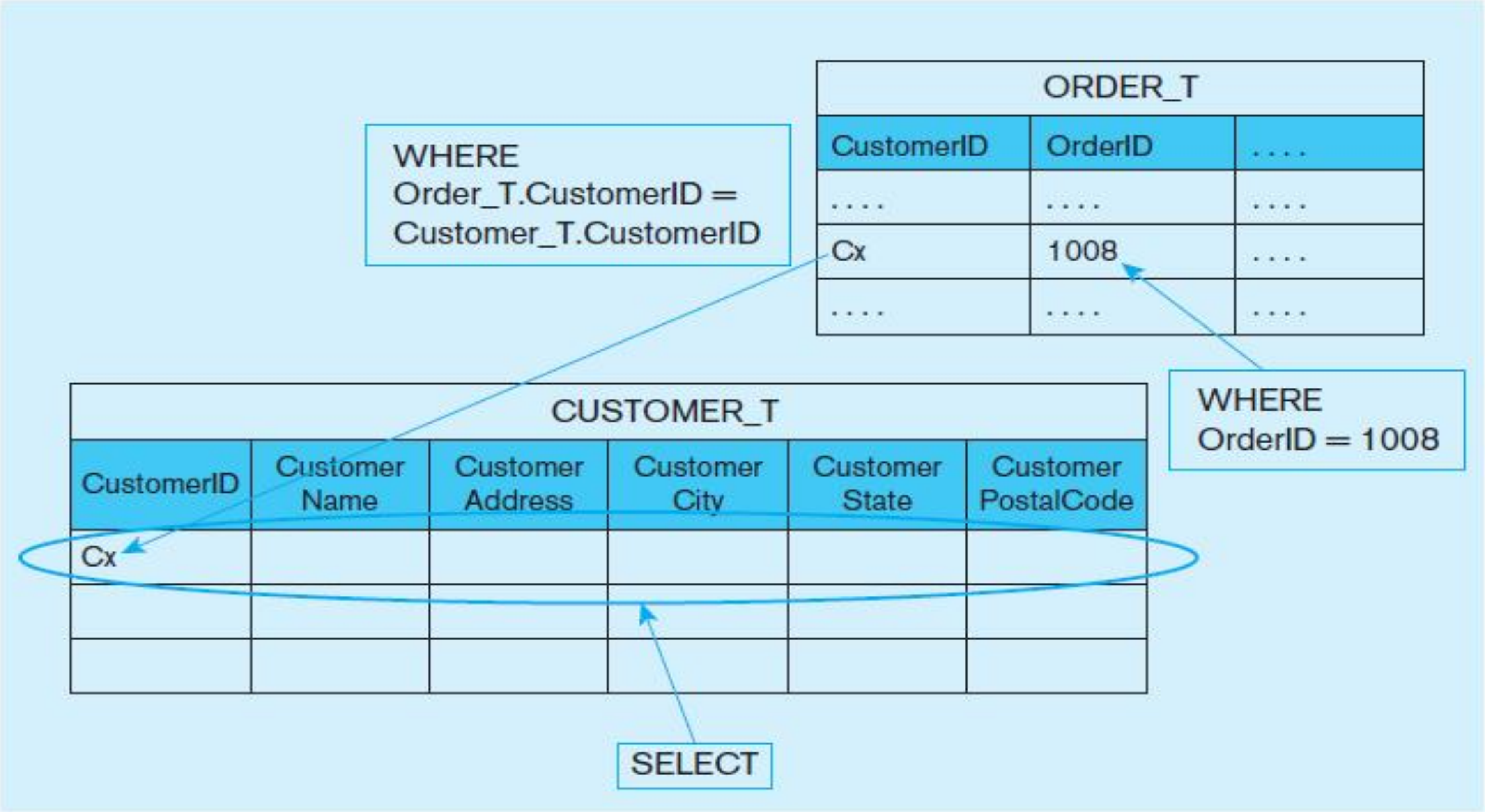


```
SELECT CustomerName, CustomerAddress, CustomerCity,
CustomerState, CustomerPostalCode
   FROM Customer_T
      WHERE Customer_T.CustomerID =
         (SELECT Order_T.CustomerID
            FROM Order_T
               WHERE OrderID = 1008);
```

Subquery version

Figure 7-6: Graphical Depiction of Two Ways to Answer a Query with Different Types of Joins

(a) Join query approach



WHERE
Order_T.CustomerID =
Customer_T.CustomerID

ORDER_T

CustomerID	OrderID	....
....	....	....
Cx	1008	....
....	....	....

WHERE
OrderID = 1008

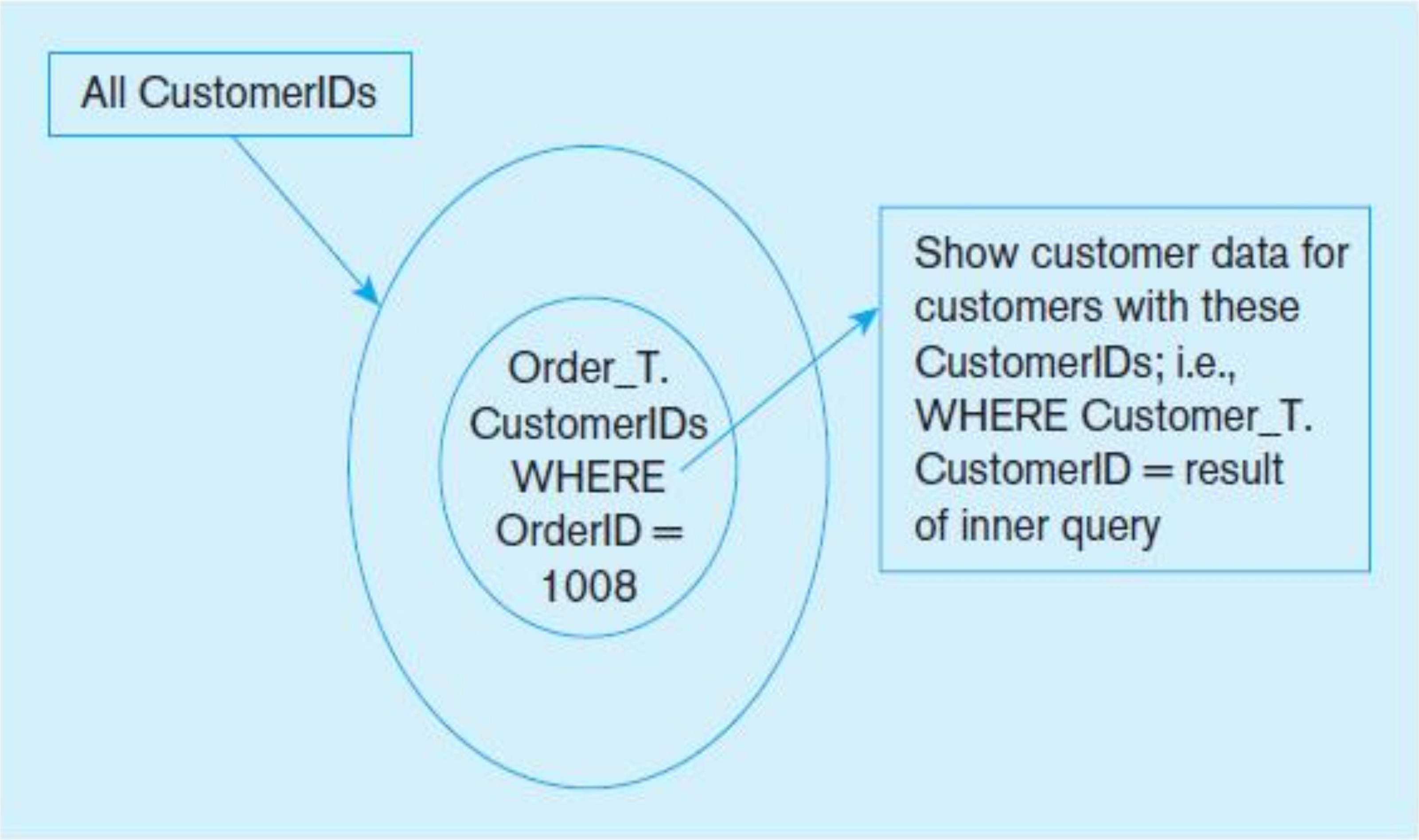
CUSTOMER_T

CustomerID	Customer Name	Customer Address	Customer City	Customer State	Customer PostalCode
Cx					

SELECT

Figure 7-6: Graphical Depiction of Two Ways to Answer a Query with Different Types of Joins

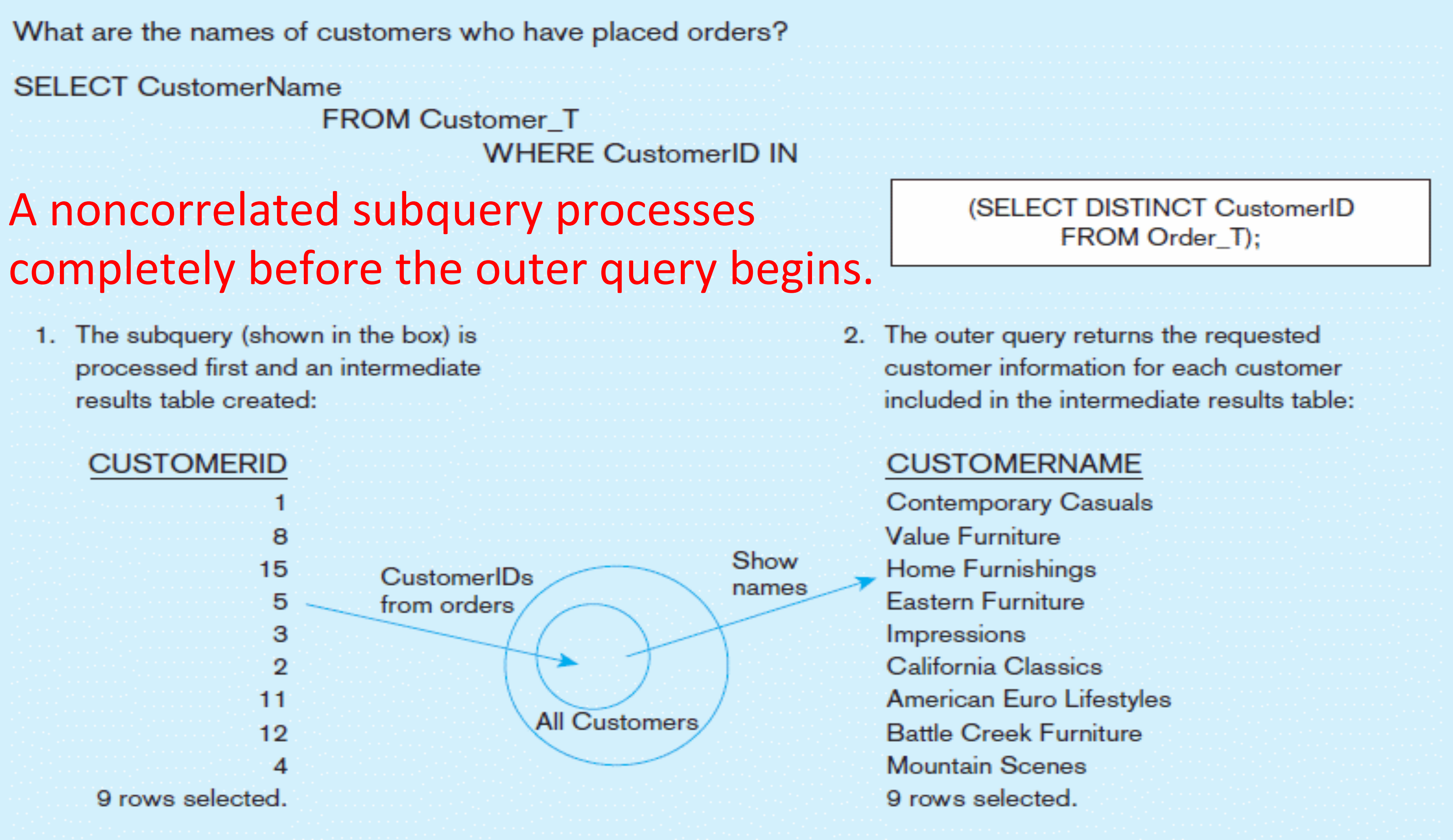
(b) Subquery approach


All CustomerIDs
Order_T. CustomerIDs WHERE OrderID = 1008
Show customer data for customers with these CustomerIDs; i.e., WHERE Customer_T. CustomerID = result of inner query

Correlated Versus Noncorrelated Subqueries

- Noncorrelated subqueries:
  - Do not depend on data from the outer query.
  - Execute once for the entire outer query.
- Correlated subqueries:
  - Make use of data from the outer query.
  - Execute once for each row of the outer query.
  - Can use the **EXISTS** operator.

Figure 7-8: Processing a Noncorrelated Subquery



What are the names of customers who have placed orders?

```
SELECT CustomerName
                    FROM Customer_T
                              WHERE CustomerID IN
                                        (SELECT DISTINCT CustomerID
                                         FROM Order_T);
```

A noncorrelated subquery processes completely before the outer query begins.

1. The subquery (shown in the box) is processed first and an intermediate results table created:

```
CUSTOMERID
         1
         8
        15
         5
         3
         2
        11
        12
         4
9 rows selected.
```

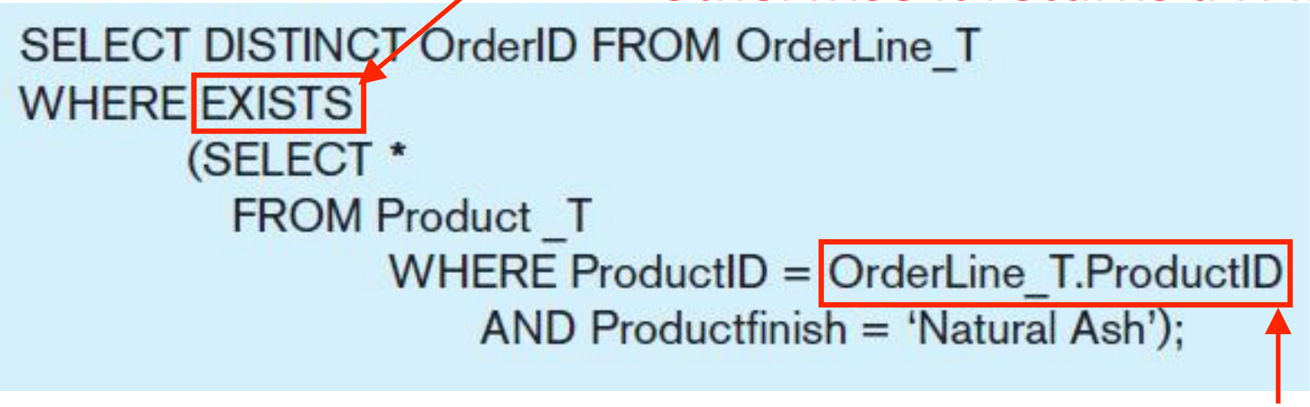
2. The outer query returns the requested customer information for each customer included in the intermediate results table:

```
CUSTOMERNAME
Contemporary Casuals
Value Furniture
Home Furnishings
Eastern Furniture
Impressions
California Classics
American Euro Lifestyles
Battle Creek Furniture
Mountain Scenes
9 rows selected.
```

Correlated Subquery Example

- Show all orders that include furniture finished in natural ash.

The **EXISTS** operator will return a TRUE value if the subquery resulted in a non-empty set, otherwise it returns a FALSE.



```
SELECT DISTINCT OrderID FROM OrderLine_T
WHERE EXISTS
        (SELECT *
          FROM Product _T
                  WHERE ProductID = OrderLine_T.ProductID
                       AND Productfinish = ‘Natural Ash’);
```

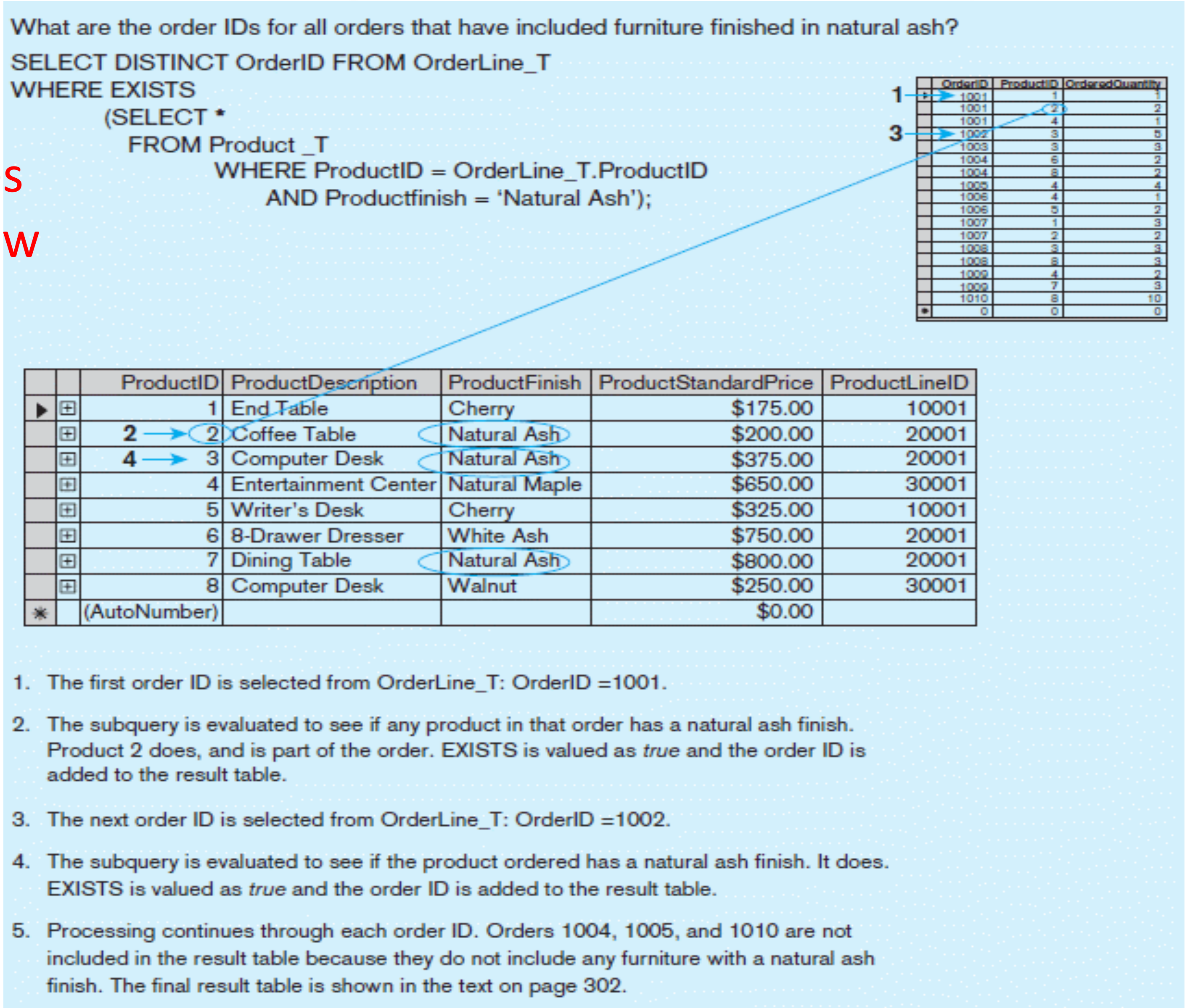
The subquery is testing for a value that comes from the outer query.

➔ A correlated subquery always refers to an attribute from a table referenced in the outer query.

Figure 7-8: Processing a Correlated Subquery

Subquery refers to outer-query data, so executes once for each row of outer query.

Note: Only the orders that involve products with Natural Ash will be included in the final results.



What are the order IDs for all orders that have included furniture finished in natural ash?

```
SELECT DISTINCT OrderID FROM OrderLine_T
WHERE EXISTS
          (SELECT *
            FROM Product _T
                  WHERE ProductID = OrderLine_T.ProductID
                       AND Productfinish = 'Natural Ash');
```

	OrderID	ProductID	OrderedQuantity
1 ▸	1001	1	1
	1001	2	2
	1001	4	1
3	1002	3	5
	1003	3	3
	1004	6	2
	1004	8	2
	1005	4	4
	1006	4	1
	1006	5	2
	1007	1	3
	1007	2	2
	1008	3	3
	1008	8	3
	1009	4	2
	1009	7	3
	1010	8	10
*	0	0	0

		ProductID	ProductDescription	ProductFinish	ProductStandardPrice	ProductLineID
▶	⊞	1	End Table	Cherry	$175.00	10001
	⊞	2 2	Coffee Table	Natural Ash	$200.00	20001
	⊞	4 3	Computer Desk	Natural Ash	$375.00	20001
	⊞	4	Entertainment Center	Natural Maple	$650.00	30001
	⊞	5	Writer's Desk	Cherry	$325.00	10001
	⊞	6	8-Drawer Dresser	White Ash	$750.00	20001
	⊞	7	Dining Table	Natural Ash	$800.00	20001
	⊞	8	Computer Desk	Walnut	$250.00	30001
*		(AutoNumber)			$0.00	

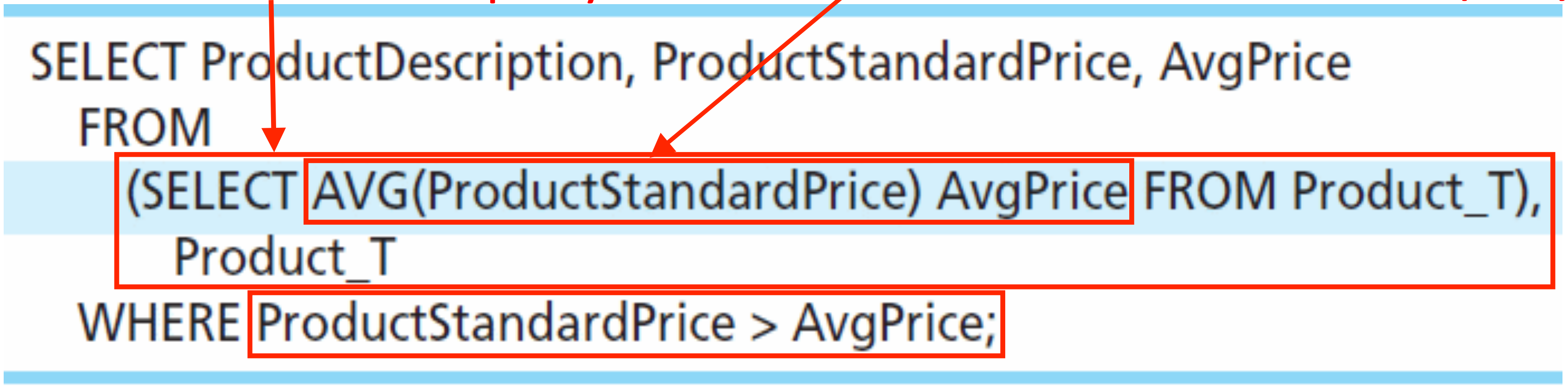
1. The first order ID is selected from OrderLine_T: OrderID =1001.
2. The subquery is evaluated to see if any product in that order has a natural ash finish. Product 2 does, and is part of the order. EXISTS is valued as *true* and the order ID is added to the result table.
3. The next order ID is selected from OrderLine_T: OrderID =1002.
4. The subquery is evaluated to see if the product ordered has a natural ash finish. It does. EXISTS is valued as *true* and the order ID is added to the result table.
5. Processing continues through each order ID. Orders 1004, 1005, and 1010 are not included in the result table because they do not include any furniture with a natural ash finish. The final result table is shown in the text on page 302.

Another Subquery Example

- Show all products whose standard price is higher than the average price.

Subquery forms the derived table used in the FROM clause of the outer query

One column of the subquery is an aggregate function that has an alias name. That alias can then be referred to in the outer query.

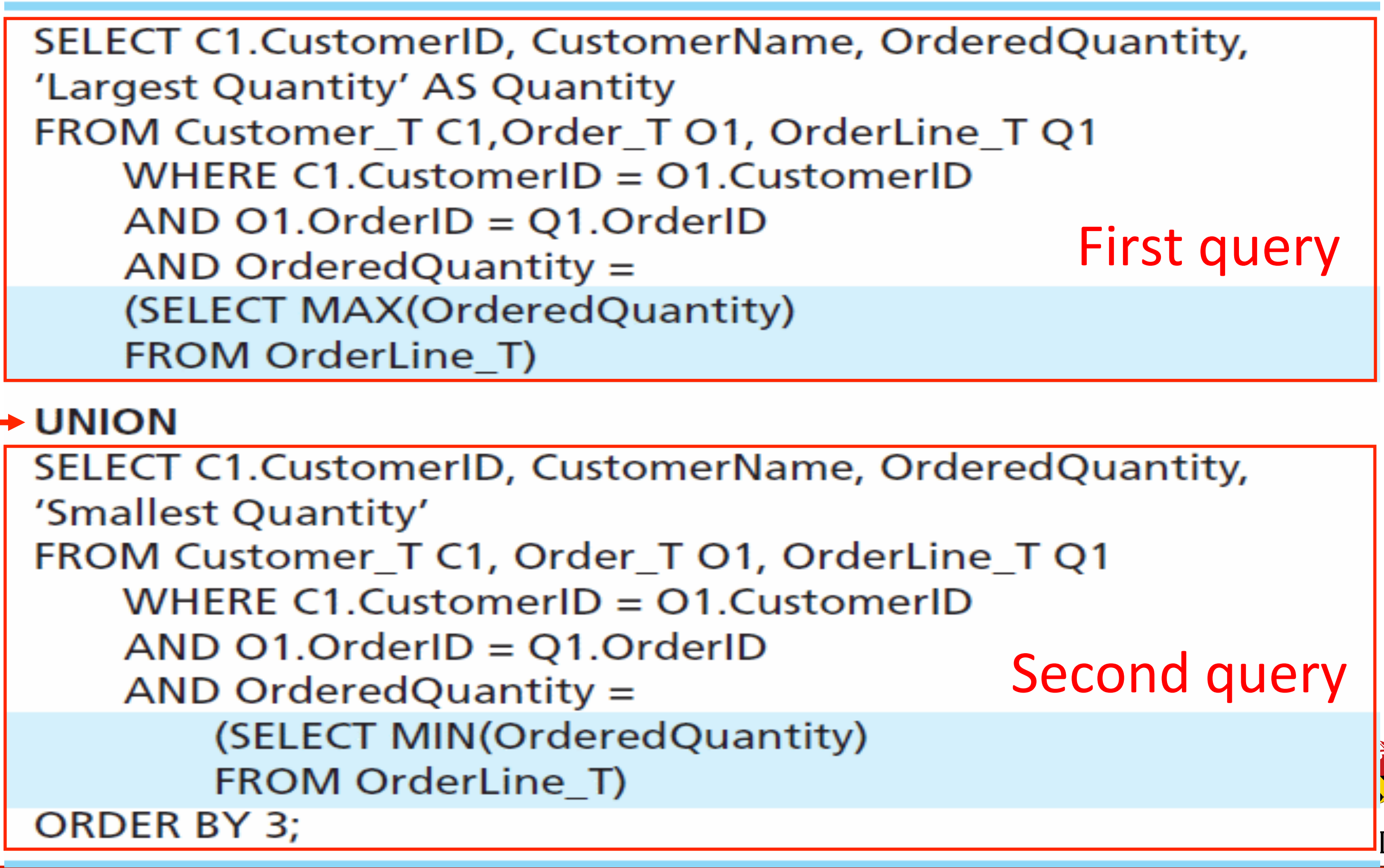


```
SELECT ProductDescription, ProductStandardPrice, AvgPrice
  FROM
    (SELECT AVG(ProductStandardPrice) AvgPrice FROM Product_T),
      Product_T
  WHERE ProductStandardPrice > AvgPrice;
```

The WHERE clause normally cannot include aggregate functions, but because the aggregate is performed in the subquery its result can be used in the outer query' s WHERE clause.

Union Queries

- Combine the output (union of multiple queries) together into a single result table.



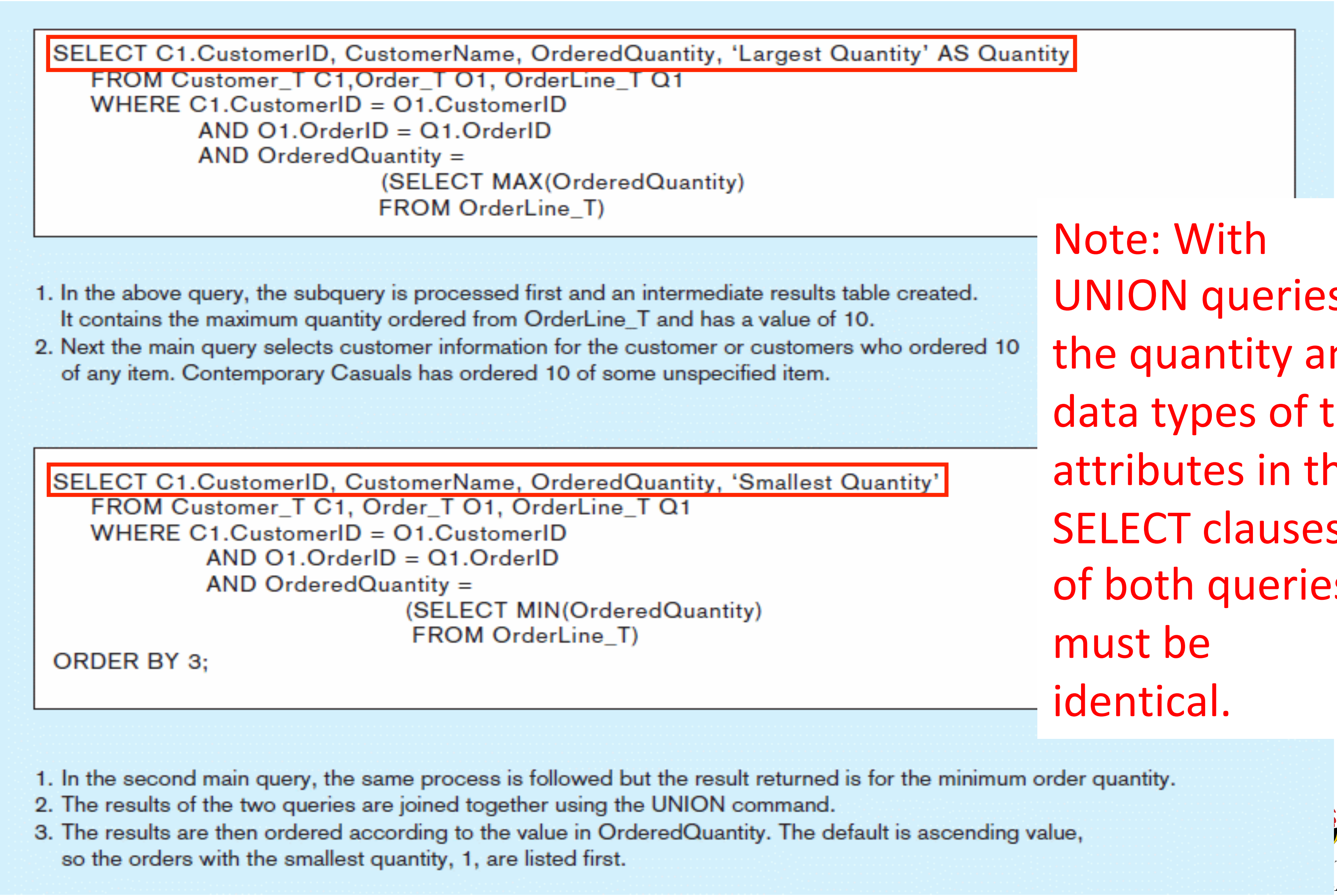
```
SELECT C1.CustomerID, CustomerName, OrderedQuantity,
‘Largest Quantity’ AS Quantity
FROM Customer_T C1,Order_T O1, OrderLine_T Q1
      WHERE C1.CustomerID = O1.CustomerID
      AND O1.OrderID = Q1.OrderID
      AND OrderedQuantity =
      (SELECT MAX(OrderedQuantity)
      FROM OrderLine_T)
UNION
SELECT C1.CustomerID, CustomerName, OrderedQuantity,
‘Smallest Quantity’
FROM Customer_T C1, Order_T O1, OrderLine_T Q1
      WHERE C1.CustomerID = O1.CustomerID
      AND O1.OrderID = Q1.OrderID
      AND OrderedQuantity =
            (SELECT MIN(OrderedQuantity)
            FROM OrderLine_T)
ORDER BY 3;
```

First query

Combine

Second query

Figure 7-9: Combining Queries Using UNION



```
SELECT C1.CustomerID, CustomerName, OrderedQuantity, 'Largest Quantity' AS Quantity
    FROM Customer_T C1,Order_T O1, OrderLine_T Q1
    WHERE C1.CustomerID = O1.CustomerID
                AND O1.OrderID = Q1.OrderID
                AND OrderedQuantity =
                                    (SELECT MAX(OrderedQuantity)
                                    FROM OrderLine_T)
```

1. In the above query, the subquery is processed first and an intermediate results table created. It contains the maximum quantity ordered from OrderLine_T and has a value of 10.
2. Next the main query selects customer information for the customer or customers who ordered 10 of any item. Contemporary Casuals has ordered 10 of some unspecified item.

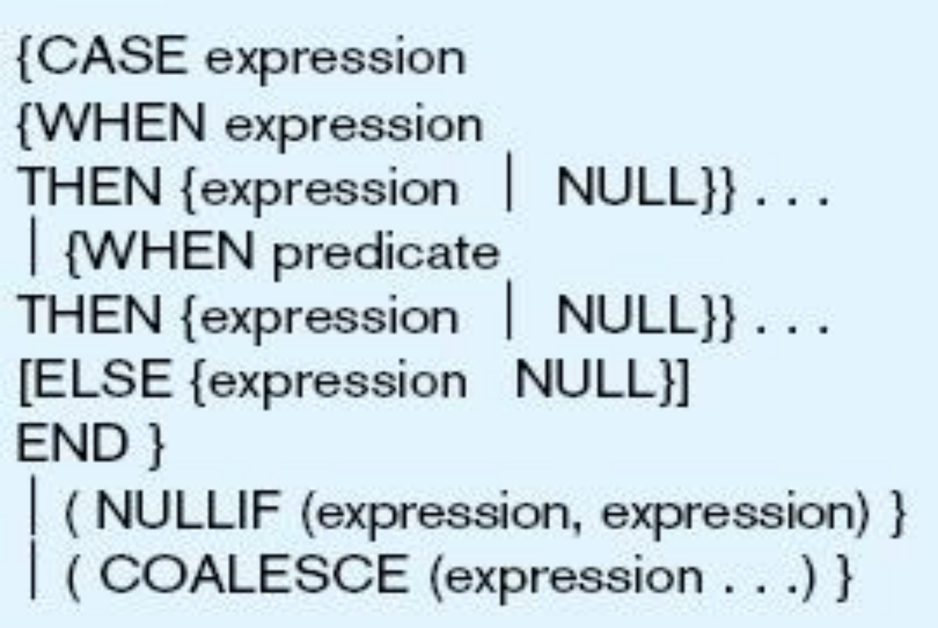
```
SELECT C1.CustomerID, CustomerName, OrderedQuantity, 'Smallest Quantity'
    FROM Customer_T C1, Order_T O1, OrderLine_T Q1
    WHERE C1.CustomerID = O1.CustomerID
                AND O1.OrderID = Q1.OrderID
                AND OrderedQuantity =
                                    (SELECT MIN(OrderedQuantity)
                                    FROM OrderLine_T)
ORDER BY 3;
```

1. In the second main query, the same process is followed but the result returned is for the minimum order quantity.
2. The results of the two queries are joined together using the UNION command.
3. The results are then ordered according to the value in OrderedQuantity. The default is ascending value, so the orders with the smallest quantity, 1, are listed first.

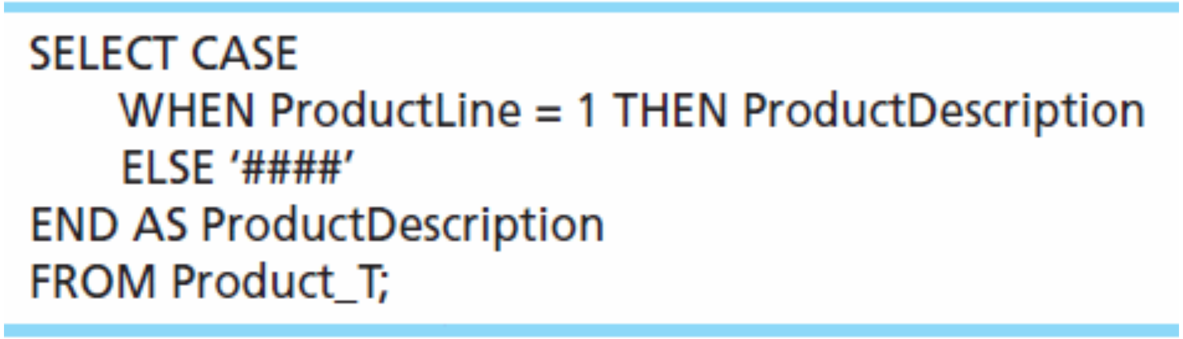
Note: With UNION queries, the quantity and data types of the attributes in the SELECT clauses of both queries must be identical.

Figure 7-10: Conditional Expressions Using CASE Syntax

- This is available with newer versions of SQL, previously not part of the standard.



```
{CASE expression
{WHEN expression
THEN {expression  |  NULL}} . . .
 | {WHEN predicate
THEN {expression  |  NULL}} . . .
[ELSE {expression   NULL}]
END }
 | ( NULLIF (expression, expression) }
 | ( COALESCE (expression . . .) }
```



```
SELECT CASE
      WHEN ProductLine = 1 THEN ProductDescription
      ELSE '####'
END AS ProductDescription
FROM Product_T;
```

Tips for Developing Queries

- Be familiar with the data model (entities and relationships).
- Understand the desired results.
- Know the attributes desired in results.
- Identify the entities that contain desired attributes.
- Review ERD.
- Construct a **WHERE** equality for each link.
- Fine tune with **GROUP BY** and **HAVING** clauses if needed.
- Consider the effect on unusual data.

Query Efficiency Considerations

- Instead of **SELECT ***, identify the specific attributes in the **SELECT** clause; this helps reduce network traffic of result set.
- Limit the number of subqueries; try to make everything done in a single query if possible.
- If data is to be used many times, make a separate query and store it as a view.

Guidelines for Better Query Design

- Understand how indexes are used in query processing.
- Keep optimizer statistics up-to-date.
- Use compatible data types for fields and literals.
- Write simple queries.
- Break complex queries into multiple simple parts.
- Don’t nest one query inside another query.
- Don’t combine a query with itself (if possible avoid self-joins).

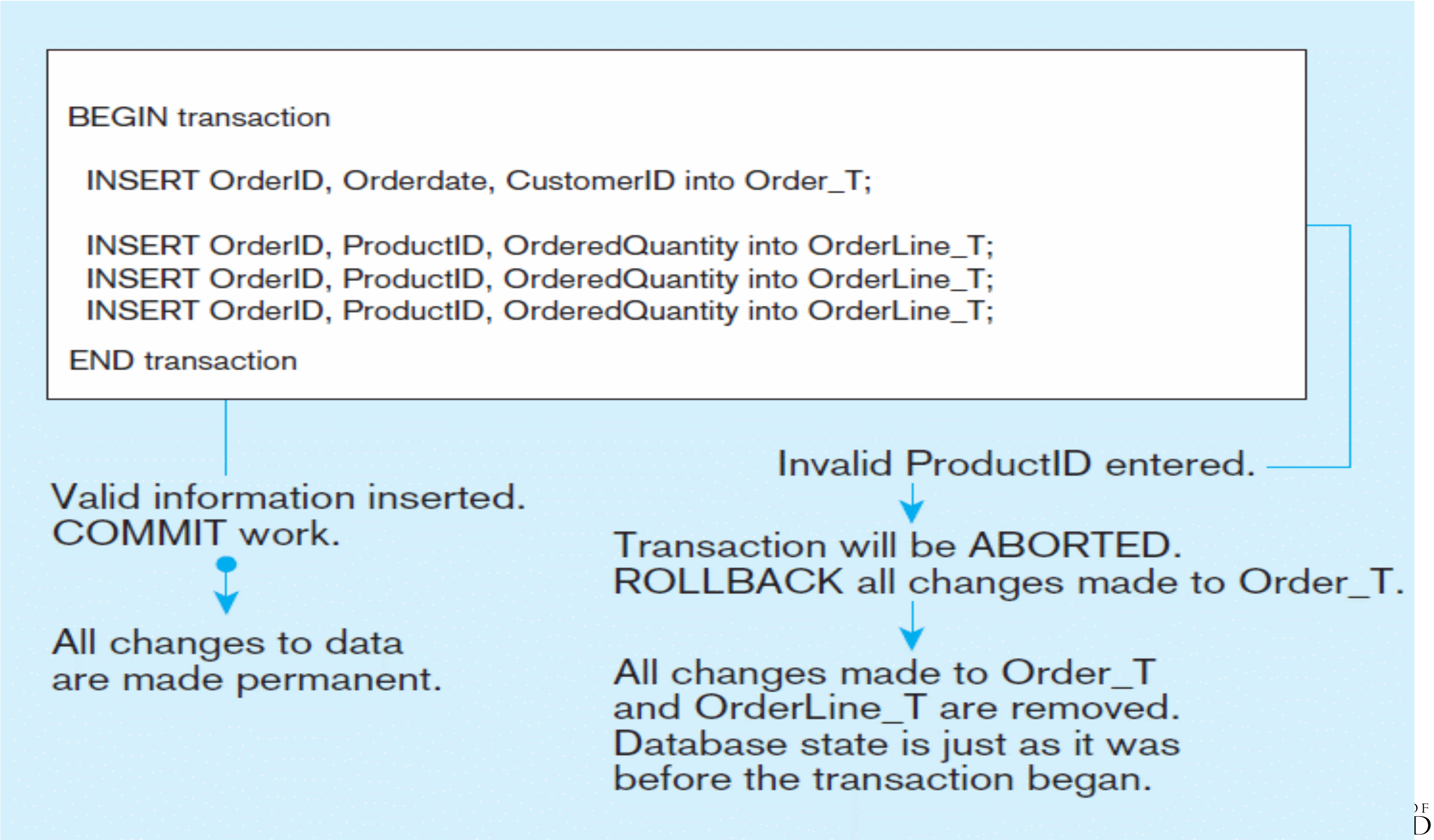
Guidelines for Better Query Design

- Create temporary tables for groups of queries.
- Combine update operations.
- Retrieve only the data you need.
- Don’t have the DBMS sort without an index.
- Learn!
- Consider the total query processing time for ad hoc queries.

Ensuring Transaction Integrity

- **Transaction** – A discrete unit of work that must be completely processed or not processed at all.
  - May involve multiple updates.
  - If any update fails, then all other updates must be cancelled.
- SQL commands for transactions:
  - **BEGIN TRANSACTION** | **END TRANSACTION** – Marks boundaries of a transaction.
  - **COMMIT** – Makes all updates permanent.
  - **ROLLBACK** – Cancels updates since the last **COMMIT**.

Figure 7-12: An SQL Transaction Sequence (in Pseudocode)



```
BEGIN transaction

  INSERT OrderID, Orderdate, CustomerID into Order_T;

  INSERT OrderID, ProductID, OrderedQuantity into OrderLine_T;
  INSERT OrderID, ProductID, OrderedQuantity into OrderLine_T;
  INSERT OrderID, ProductID, OrderedQuantity into OrderLine_T;

END transaction
```

Valid information inserted.
COMMIT work.

All changes to data
are made permanent.

Invalid ProductID entered.

Transaction will be ABORTED.
ROLLBACK all changes made to Order_T.

All changes made to Order_T
and OrderLine_T are removed.
Database state is just as it was
before the transaction began.

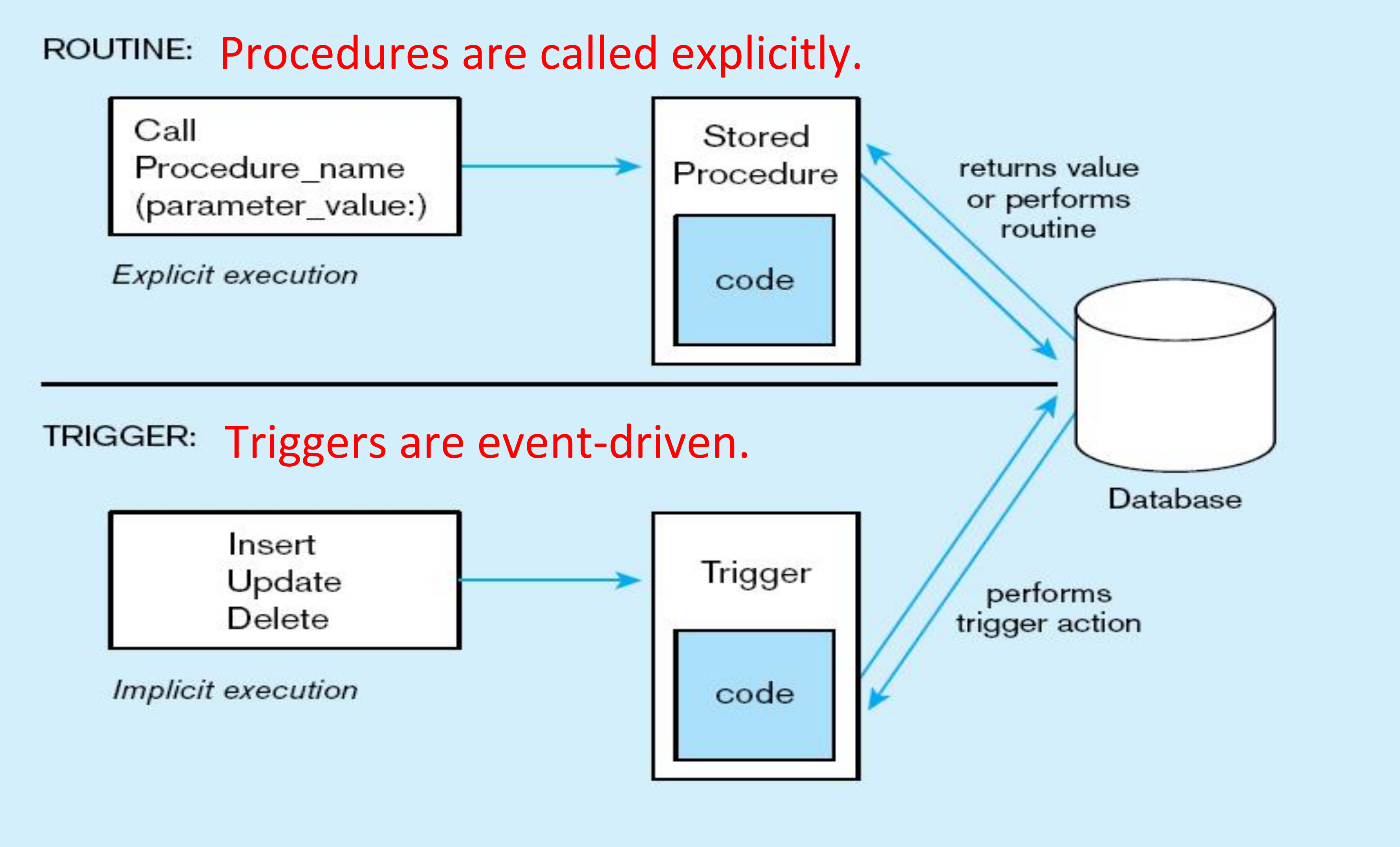
Data Dictionary Facilities

- System tables that store metadata.
- Users usually can view some of these tables.
- Users are restricted from updating them.
- Some examples in Oracle 11g:
  - **DBA_TABLES** – descriptions of tables.
  - **DBA_CONSTRAINTS** – description of constraints.
  - **DBA_USERS** – information about the users of the system.
- Examples in Microsoft SQL Server 2008:
  - **sys.columns** – table and column definitions.
  - **sys.indexes** – table index information.
  - **sys.foreign_key_columns** – details about columns in foreign key constraints.

Routines and Triggers

- Routines:
  - Program modules that execute on demand.
- Functions:
  - Routines that return values and take input parameters.
- Procedures:
  - Routines that do not return values and can take input or output parameters.
- Triggers:
  - Routines that execute in response to a database event (**INSERT**, **UPDATE**, or **DELETE**).

Figure7-13: Triggers Contrasted with Stored Procedures (Based on Mullins 1995).



ROUTINE: Procedures are called explicitly.

Call Procedure_name (parameter_value:)

Explicit execution

Stored Procedure

code

returns value or performs routine

Database

TRIGGER: Triggers are event-driven.

Insert
Update
Delete

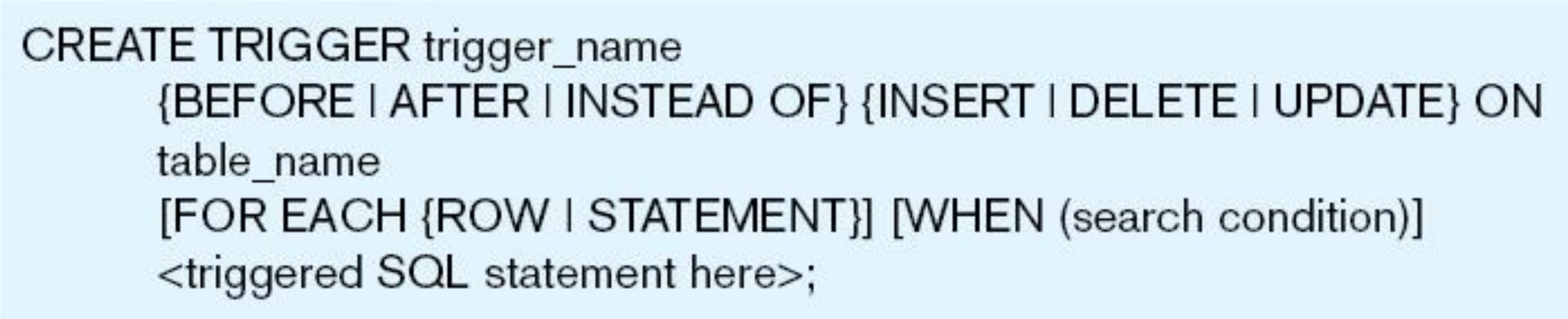
Implicit execution

Trigger

code

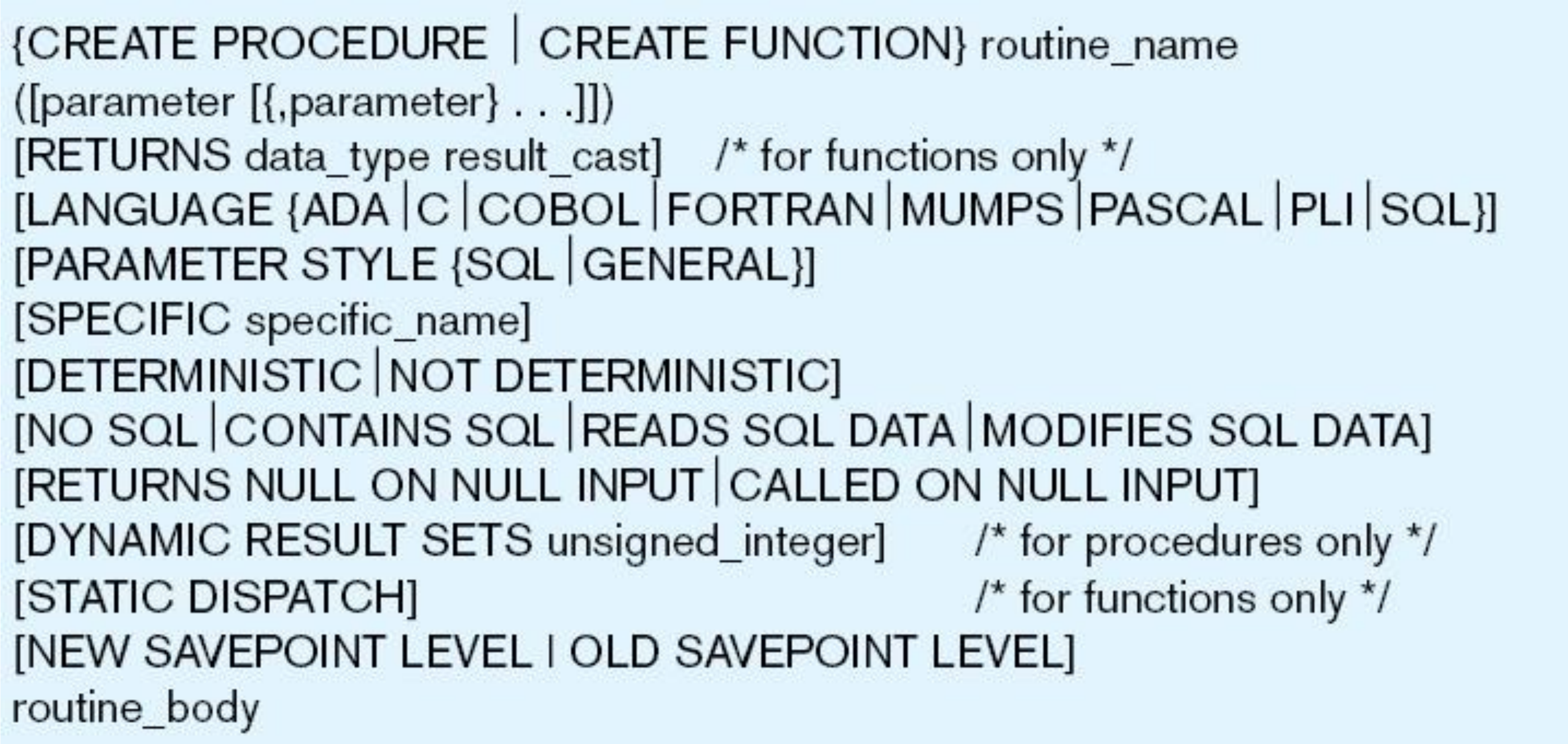
performs trigger action

Figure 7-14: Simplified Trigger Syntax (SQL:2008)



```
CREATE TRIGGER trigger_name
        {BEFORE | AFTER | INSTEAD OF} {INSERT | DELETE | UPDATE} ON
        table_name
        [FOR EACH {ROW | STATEMENT}] [WHEN (search condition)]
        <triggered SQL statement here>;
```

Figure 7-15: Syntax for Creating a Routine (SQL:2008)



```
{CREATE PROCEDURE | CREATE FUNCTION} routine_name
([parameter [{,parameter} . . .]])
[RETURNS data_type result_cast]    /* for functions only */
[LANGUAGE {ADA|C|COBOL|FORTRAN|MUMPS|PASCAL|PLI|SQL}]
[PARAMETER STYLE {SQL|GENERAL}]
[SPECIFIC specific_name]
[DETERMINISTIC|NOT DETERMINISTIC]
[NO SQL|CONTAINS SQL|READS SQL DATA|MODIFIES SQL DATA]
[RETURNS NULL ON NULL INPUT|CALLED ON NULL INPUT]
[DYNAMIC RESULT SETS unsigned_integer]      /* for procedures only */
[STATIC DISPATCH]                           /* for functions only */
[NEW SAVEPOINT LEVEL I OLD SAVEPOINT LEVEL]
routine_body
```

Table 7-2: Stored Procedures

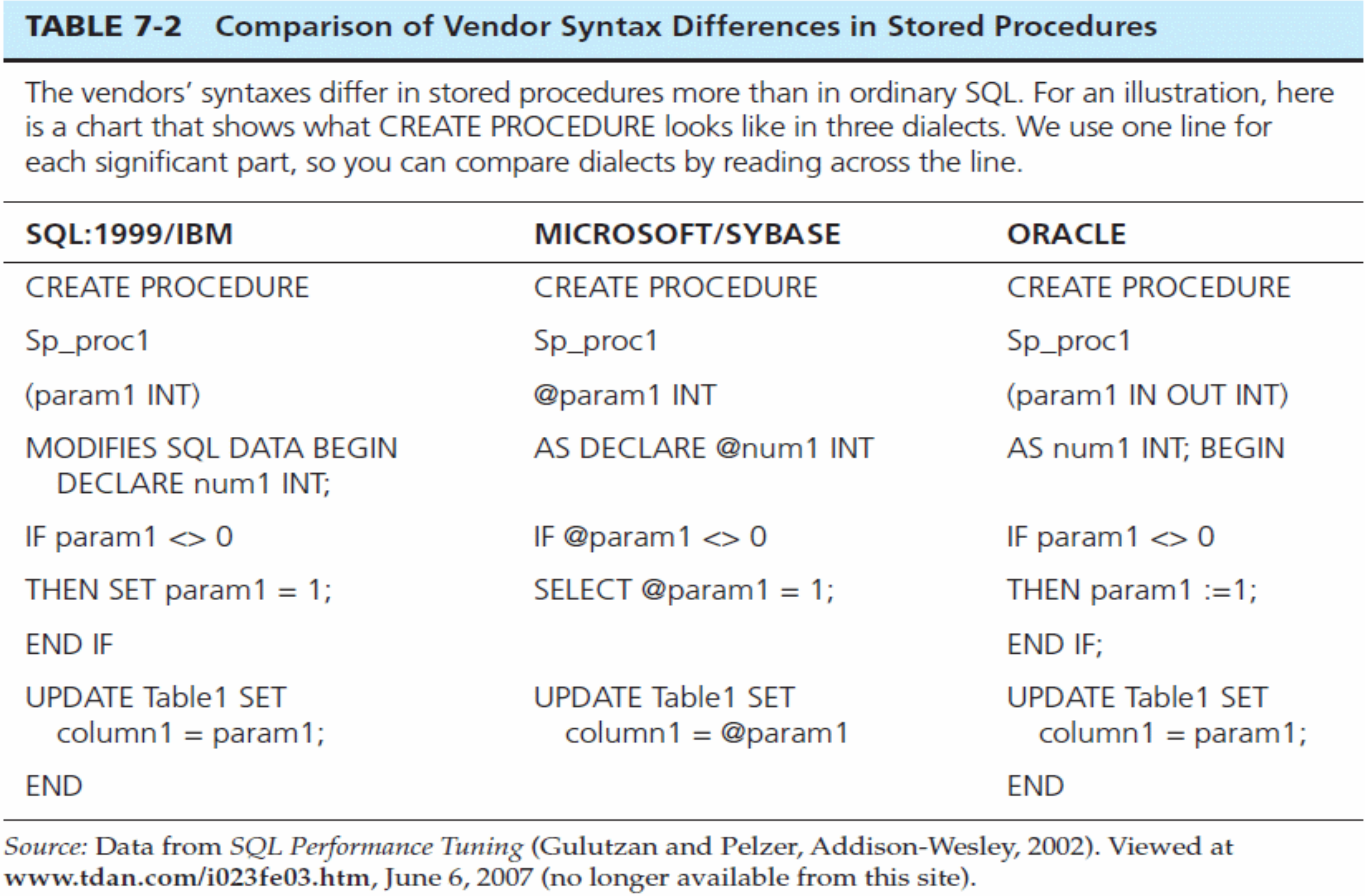


TABLE 7-2 Comparison of Vendor Syntax Differences in Stored Procedures

The vendors' syntaxes differ in stored procedures more than in ordinary SQL. For an illustration, here is a chart that shows what CREATE PROCEDURE looks like in three dialects. We use one line for each significant part, so you can compare dialects by reading across the line.

SQL:1999/IBM	MICROSOFT/SYBASE	ORACLE
CREATE PROCEDURE	CREATE PROCEDURE	CREATE PROCEDURE
Sp_proc1	Sp_proc1	Sp_proc1
(param1 INT)	@param1 INT	(param1 IN OUT INT)
MODIFIES SQL DATA BEGIN DECLARE num1 INT;	AS DECLARE @num1 INT	AS num1 INT; BEGIN
IF param1 <> 0	IF @param1 <> 0	IF param1 <> 0
THEN SET param1 = 1;	SELECT @param1 = 1;	THEN param1 :=1;
END IF		END IF;
UPDATE Table1 SET column1 = param1;	UPDATE Table1 SET column1 = @param1	UPDATE Table1 SET column1 = param1;
END		END

Source: Data from *SQL Performance Tuning* (Gulutzan and Pelzer, Addison-Wesley, 2002). Viewed at www.tdan.com/i023fe03.htm, June 6, 2007 (no longer available from this site).

Embedded and Dynamic SQL

- Embedded SQL:
  - Including hard-coded SQL statements in a program written in another language such as C or Java.
- Dynamic SQL:
  - Ability for an application program to generate SQL code on the fly, as the application is running.
- Reasons to embed SQL in 3GL:
  - Can create a more flexible, accessible interface for the user.
  - Possible performance improvement.
  - Database security improvement; grant access only to the application instead of users.