MDBM-Chapter 7 Advanced SQL

Main Knowledge Points

- Multiple table SQL queries
- Different types of joins
- Noncorrelated and correlated subqueries
- Referential integrity in SQL
- Triggers and stored procedures, etc.

Processing Multiple Tables: JOINs

- **JOIN:** a relational operation that causes two or more tables with a common domain to be combined into a single result table or view. The common columns in joined tables are often the primary key of the dominant table and the foreign key of the dependent table in 1:M relationships.
  - **JOIN implicitly**: using a WHERE clause, which is the traditional way and was the only means before SQL 1992.
  - **JOIN explicitly:** using the **JOIN…ON** commands. There are a number of join operations available by combining with different key words: **INNER, OUTER, FULL, LEFT, RIGHT, CROSS and UNION**. Not all DBMSes support all the key words.

EQUI-JOIN

EQUI-JOIN: a join in which the joining condition is based on **equality** between values in the common columns; common columns can appear redundantly in the result table. It is normally established by the key words INNER JOIN … ON

```
SELECT d.department_name, e.employee_name
    FROM   departments_t d
        INNER JOIN employees_t e
        ON d.department_id =  e.department_id
    WHERE  d.department_id >= 30
    ORDER BY d.department_name;
```

OUTER JOIN

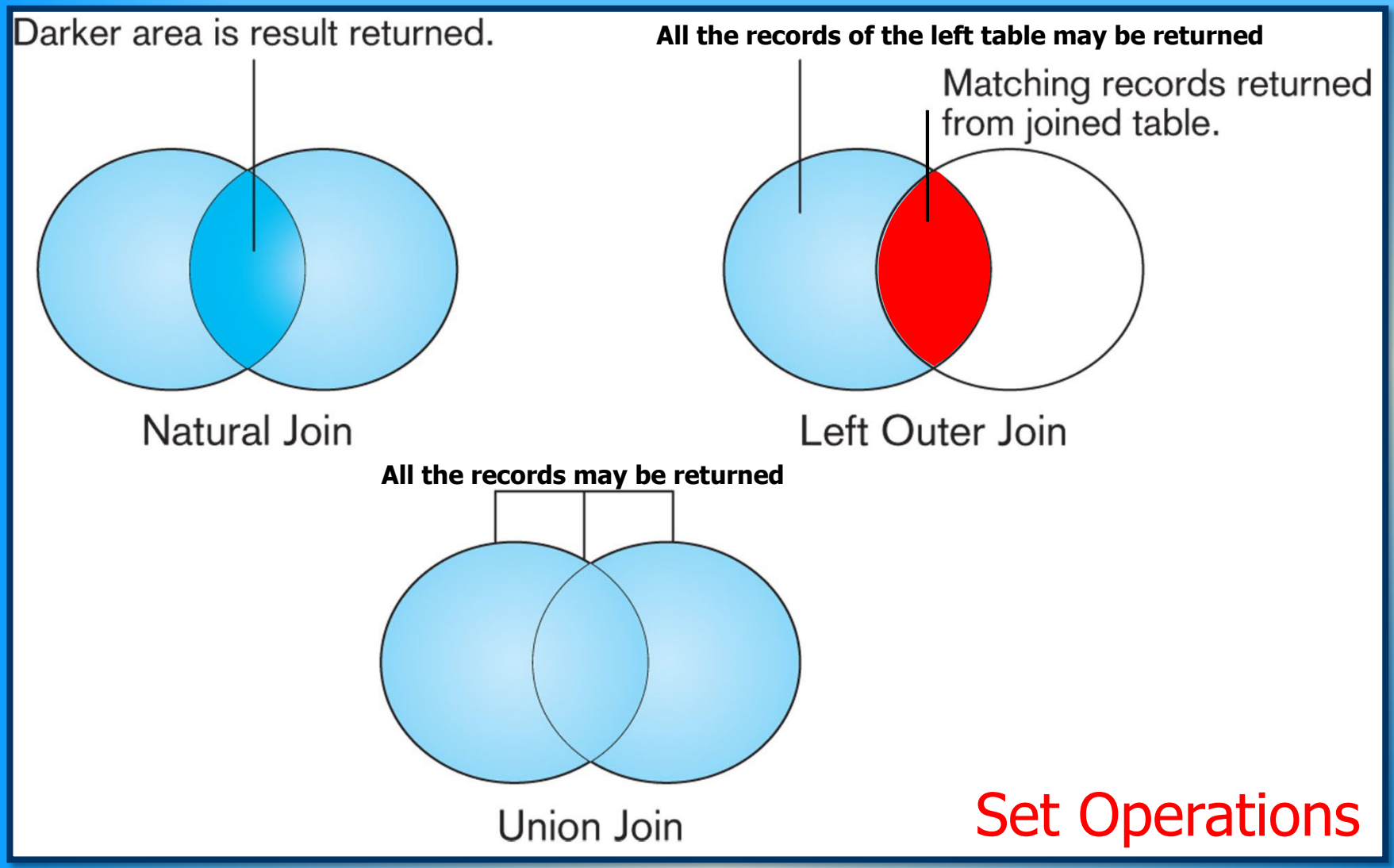
(A join in which rows that do not have matching values in common columns can also be included in the result table.)

- **LEFT OUTER JOIN:** a join in which rows from the table on the left side of the "LEFT OUTER JOIN" keywords that do not have matching values in common columns are also included in the result table; otherwise, the result will be the same as the INNER JOIN.
- **RIGHT OUTER JOIN:** a join in which rows from the table on the right side of the "RIGHT OUTER JOIN" keywords that do not have matching values in common columns are also included in the result table; otherwise, the result will be the same as the INNER JOIN.
- **FULL OUTER JOIN:** a join in which rows from the tables on the both sides of the "FULL OUTER JOIN" keywords that do not have matching values in common columns are also included in the result table; otherwise, the result will be the same as the LEFT OUTER JOIN or RIGHT OUTER JOIN.

UNION JOIN

- A join that includes all the data from each table in the join and an instance for each row of each table.
- The result table contains all the columns from each table; NULLs will be assigned to the columns where there are no values.
- This command is not available in Oracle, but it is easy to achieve the same result in Oracle.

Visualization of Different Join Types with Results Returned in Shaded Area

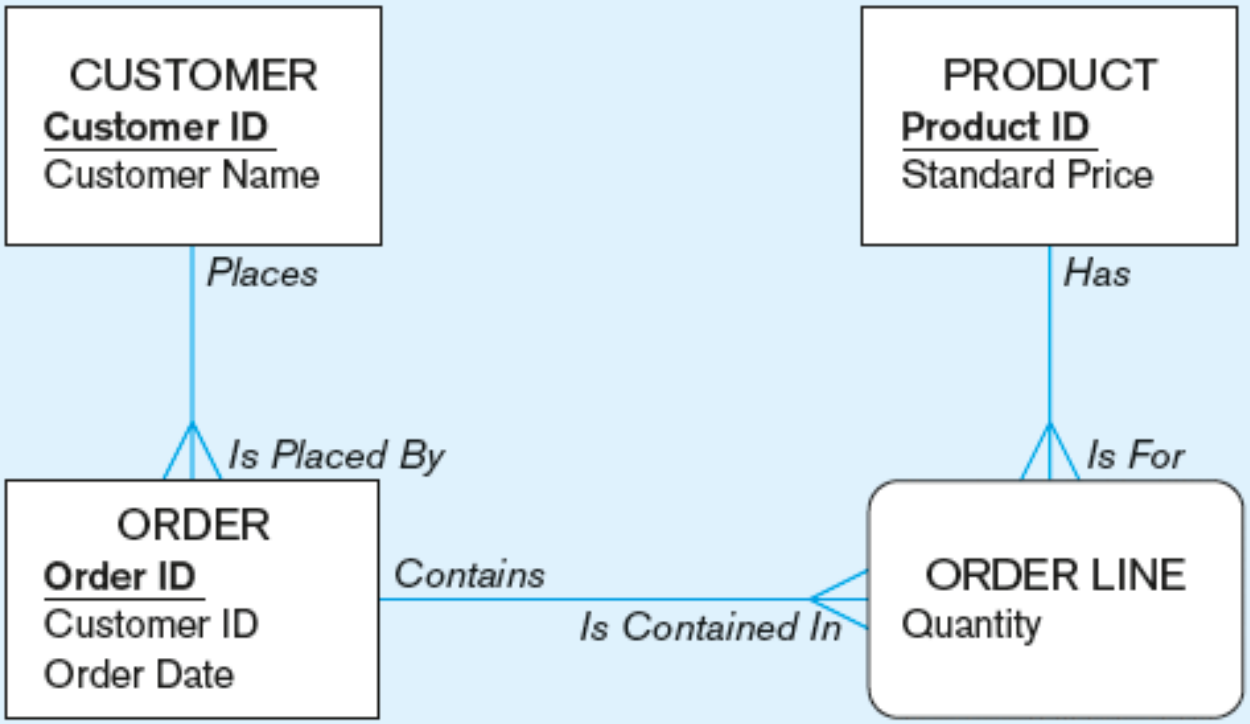


SELF-JOIN

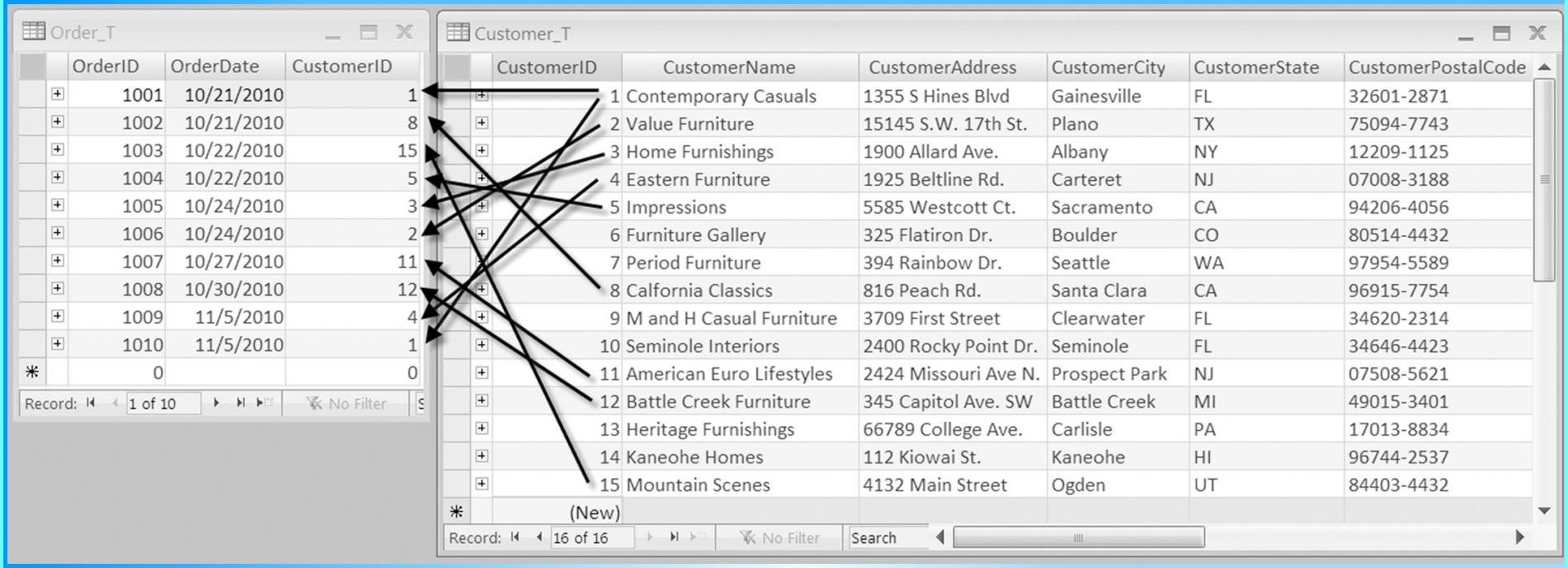
- Definition: A join that matches rows in a table with other rows in the same table.
- It often arises from a unary relationship.

JOIN Examples

The Following Slides will Use This Data Model



Pine Valley Furniture Company Customer_T and Order_T tables with Pointers from Customers to Their Orders



Order_T

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

Record: 1 of 10 No Filter

Customer_T

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	Calfornia 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)					

Record: 16 of 16 No Filter Search

EQUI-JOIN

- Using **WHERE**:

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

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

- Using **INNER JOIN … ON**:

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

(Note: In Oracle the keyword INNER is optional.)

Outer Join Example

List the customer ID, name, and order number for all customers. Include customer information even for those who have not placed an order.

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

(The LEFT OUTER JOIN clause causes customer data to appear in the result table even if there is no corresponding order data)

The Result: LEFT JOIN

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

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

16 rows selected

Multiple Table Join Example

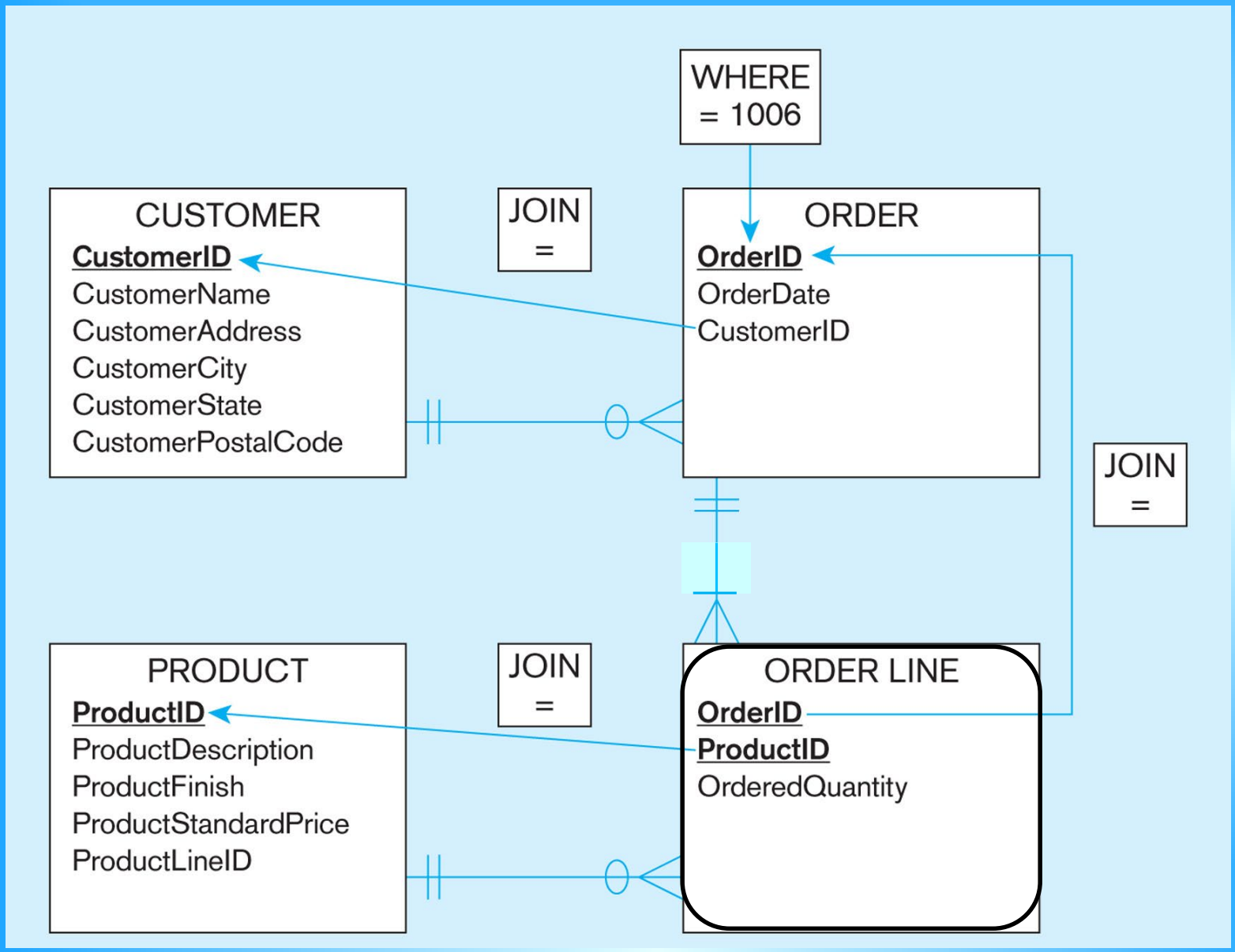
Assemble all information necessary to create an invoice for order number 1006

```
SELECT Customer_T.CustomerID, CustomerName, CustomerAddress
  CustomerCity, CustomerState, CustomerPostalCode, Order_T.OrderID,
  OrderDate, OrderedQuantity, ProductDescription, ProductStandardPrice,
  (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;
```

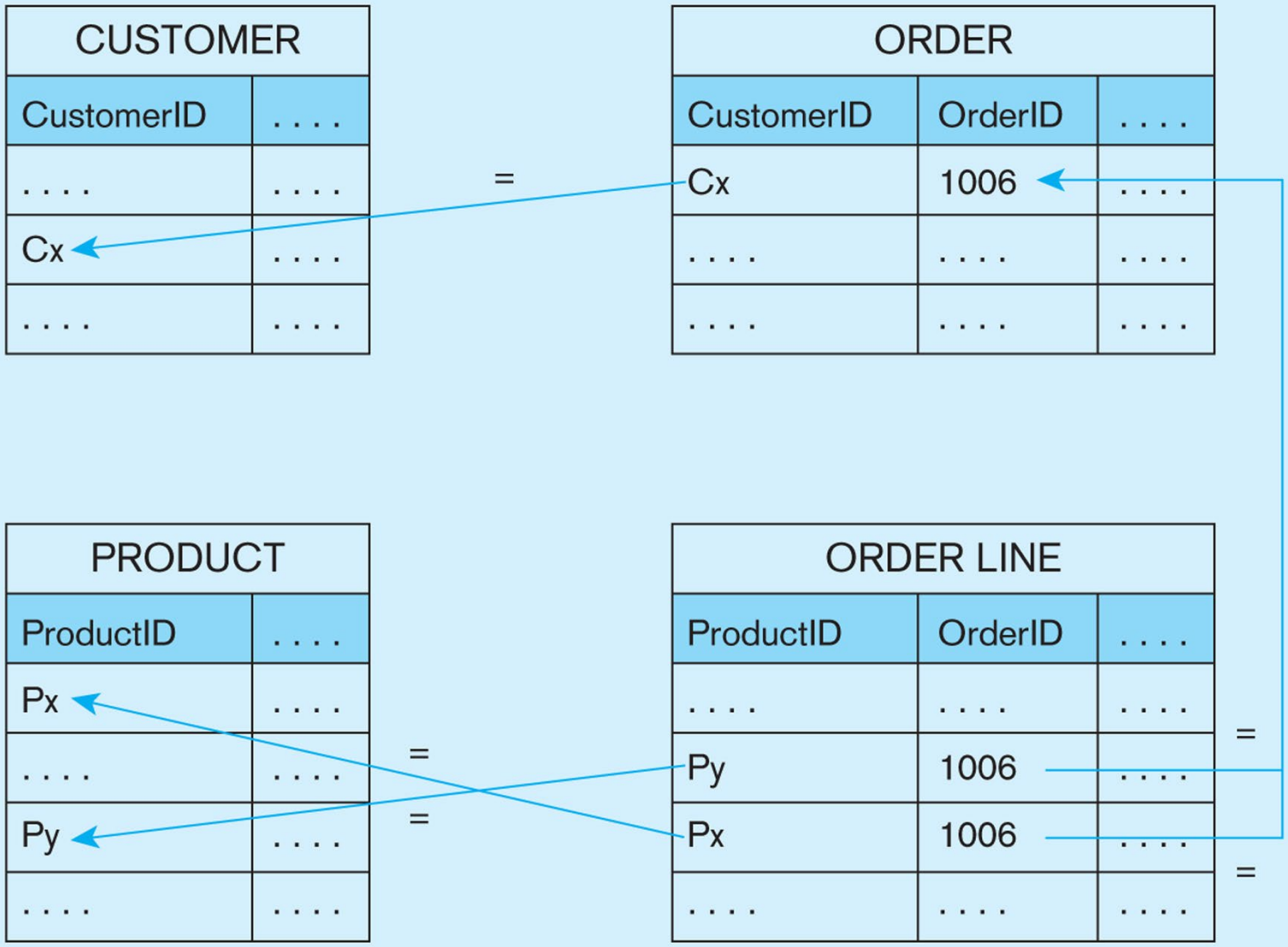
Four tables

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

Multiple Table Join Example



Multiple Table Join Example



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;
```

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

Result:

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

Self-joins are usually used on tables with unary relationships

Cartesian Join

(List all possible combinations of the rows of the two tables. For most times, you want to avoid this.)

- **Missing WHERE clause:**

```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
CustomerName, OrderID
FROM Customer_T, Order_T;
```

- **Specifying CROSS JOIN:**

```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
CustomerName, OrderID
FROM Customer_T CROSS JOIN Order_T;
```

Processing Multiple Tables Using Subqueries

- **Subquery**: a query (inner query) that is placed inside another query (outer query). Up to **16** levels of nesting are typically supported.
- Options:
  - In a condition of the WHERE clause
  - As a "table" of the FROM clause
  - Within the HAVING clause
- Two kinds of subqueries:
  - **Correlated**: executed once for each row returned by the outer query
  - **Noncorrelated**: executed once for the entire outer query

Subquery in the WHERE Clause

Show all customers who have placed an order

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

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

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

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.

Subquery in the FROM Clause

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

Subquery Within the HAVING Clause

Find the all the types of product finishes and their frequencies in the Product_T table except the most popular finish.

```
SELECT COUNT(*), ProductFinish
  FROM Product_T
    GROUP BY ProductFinish
      Having COUNT(*) <
        (SELECT MAX(COUNT(*))
          FROM Product_T
            GROUP BY ProductFinish);
```

Note: The HAVING clause works together with GROUP BY clause.

Correlated vs. Noncorrelated Subqueries

- **Noncorrelated subqueries**:
  - Do not depend on data from the outer query
  - Execute once for the entire outer query
- **Correlated subqueries**:
  - Depends on data from the outer query
  - Execute once for each row of the outer query
  - Often use the **EXISTS** operator in the WHERE clause of the outer query

Processing a Noncorrelated Subquery

What are the names of customers who have placed orders?

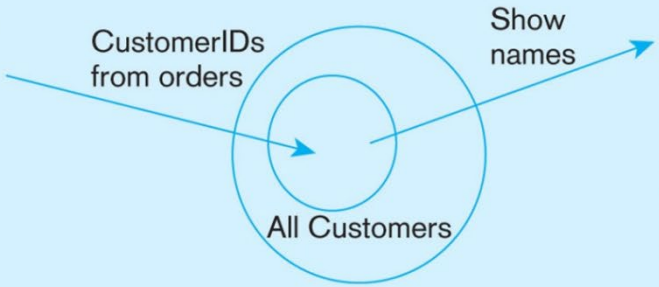
```
SELECT CustomerName
                FROM Customer_T
                        WHERE CustomerID IN
```

```
(SELECT DISTINCT CustomerID
FROM Order_T);
```

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.

A noncorrelated subquery processes completely before the outer query begins, i.e., the subqueries are processed **from the inside out**.

Correlated Subquery Example

Show all orders that include 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');
```

The EXISTS operator will return a TRUE value if the subquery resulted in **a non-empty set**, otherwise it returns a FALSE; this keyword is often an indication of a correlated subquery

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

Note: A correlated subquery always refers to an attribute from a table referenced in the outer query; a correlated subquery is processed **from the outside in**

Processing a Correlated Subquery

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

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.

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

ORDERID
1001
1002
1003
1007
1008
1009

Union Queries

Combining the outputs (union of multiple queries) together into a single result table

- Queries involved must output the same number of columns
- An output from each query for each column must have compatible data types
- It is safest to use the CAST command to control the data type conversion, e.g.,

```
SELECT CAST (OrderDate AS CHAR) FROM Order_T;
```

Union Query Example

You just need to define once for a column name

To combine two query results

```
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

Second query

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.

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

```
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.

The Result of the Example

CUSTOMERID	CUSTOMERNAME	ORDEREDQUANTUTY	QUANTUTY
1	Contemporary Casuals	1	Smallest Quantity
2	Value Furniture	1	Smallest Quantity
1	Contemporary Casuals	10	Largest Quantity

Some More Complicated SQL Queries

Question: For each salesperson, list his or her biggest selling product.

```
CREATE VIEW TSales_V AS
SELECT SalespersonName, ProductDescription,
      SUM(OrderedQuantity) Totorders
FROM Salesperson_T, OrderLine_T, Product_T, Order_T
      WHERE Salesperson_T.SalespersonID=Order_T.SalespersonID
      AND Order_T.OrderID=OrderLine_T.OrderID
      AND OrderLine_T.ProductID=Product_T.ProductID
      GROUP By SalespersonName, ProductDescription;

SELECT SalespersonName, ProductDescription
          FROM Tsales_V A
                    WHERE Totorders=(SELECT MAX(Totorders)
                              FROM Tsales_V B
                                        WHERE B.SalespersonName=
                                                  A.SalespersonName);
```

Guidelines for Query Design

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

Guidelines for Query Design (cont.)

- Understand how indexes are used in query processing.
- Keep optimizer statistics up-to-date (some DBMSes do it automatically but costly).
- Use compatible data types for fields and literals (in queries).
- Write simple queries (can use UNION to combine).
- Break complex queries into multiple simple parts.
- Try not to nest one query inside another query.
- Try not to combine a query with itself (self-joins) but use a temporary copy of the table.

Guidelines for Query Design (cont.)

- Create temporary tables for groups of queries.
  (CREATE GLOBAL TEMPORARY TABLE tempTable AS …;)
- Combine update operations (DBMS may do it in parallel).
- Retrieve only the data you need (do not use SELECT * when not retrieving all the fields).
- Try to avoid sorting without an index on the sort key field (often faster to do it outside a DBMS).
- Learn by tracking your DBMS process data (with the EXPLAIN PLAN FOR command).
- Consider the total time needed for ad hoc queries (just write logically correct queries).

Create a Temporary Table

Syntax:

```
CREATE GLOBAL TEMPORARY TABLE tempTableName
   ON COMMIT PRESERVE ROWS
   AS SELECT * FROM realTable;
```

Example:

```
CREATE GLOBAL TEMPORARY TABLE tempCustomer_t
   ON COMMIT PRESERVE ROWS
   AS SELECT * FROM customer_t;
```

Output:

```
Global temporary TABLE created.
```

Ensuring Transaction Integrity

- **Transaction**: A single unit of work that must be completely processed or not processed at all
  - It 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

An SQL Transaction Sequence (in Pseudocode)

Different DBMS may have different syntaxes.

Either all the rows are entered together or none of them should be entered; it is all or nothing at all.

```
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.

An Oracle Example

Create and populate a table:

```
CREATE TABLE Mytable_T
          (id          NUMBER       NOT NULL,
          description  VARCHAR2(50) NOT NULL);

INSERT INTO Mytable_T (id, description) VALUES (1, 'Description for 1');
INSERT INTO Mytable_T (id, description) VALUES (2, 'Description for 2');

BEGIN
  FOR i IN 3 .. 10 LOOP
    INSERT INTO Mytable_T (id, description)
    VALUES (i, 'Description for ' || i);
  END LOOP;
END;
/
```

Check table contents:

```
SELECT * FROM Mytable_T;
```

ID	DESCRIPTION
1	Description for 1
2	Description for 2
3	Description for 3
4	Description for 4
5	Description for 5
6	Description for 6
7	Description for 7
8	Description for 8
9	Description for 9
10	Description for 10

An Oracle Example (cont.)

Create and populate a table with ROLLBACK:

```
CREATE TABLE Mytable_T
          (id          NUMBER       NOT NULL,
          description  VARCHAR2(50) NOT NULL);

INSERT INTO Mytable_T (id, description) VALUES (1, 'Description for 1');
INSERT INTO Mytable_T (id, description) VALUES (2, 'Description for 2');

BEGIN
  FOR i IN 3 .. 10 LOOP
    INSERT INTO Mytable_T (id, description)
    VALUES (i, 'Description for ' || i);
  END LOOP;
ROLLBACK;
END;
/
```

The ROLLBACK statement removes all the insertions

Check table contents:

```
SELECT * FROM Mytable_T;
```

ID	DESCRIPTION

An Oracle Example (cont.)

Using SAVEPOINT and ROLLBACK:

```
CREATE TABLE Mytable_T
          (id          NUMBER       NOT NULL,
          description  VARCHAR2(50) NOT NULL);
INSERT INTO Mytable_T (id, description) VALUES (1, 'Description for 1');
INSERT INTO Mytable_T (id, description) VALUES (2, 'Description for 2');
SAVEPOINT a;
BEGIN
  FOR i IN 3 .. 10 LOOP
    INSERT INTO Mytable_T (id, description)
    VALUES (i, 'Description for ' || i);
  END LOOP;
ROLLBACK to a;
END;
/
```

The ROLLBACK statement removes all the insertions up to SAVEPOINT a

Check table contents:

```
SELECT * FROM Mytable_T;
```

ID	DESCRIPTION
1	Description for 1
2	Description for 2

An Oracle Example (cont.)

Using two SAVEPOINTs and ROLLBACK:

```
CREATE TABLE Mytable_T
          (id          NUMBER       NOT NULL,
          description  VARCHAR2(50) NOT NULL);
INSERT INTO Mytable_T (id, description) VALUES (1, 'Description for 1');
SAVEPOINT a;
INSERT INTO Mytable_T (id, description) VALUES (2, 'Description for 2');
SAVEPOINT b;
BEGIN
  FOR i IN 3 .. 10 LOOP
    INSERT INTO Mytable_T (id, description)
    VALUES (i, 'Description for ' || i);
  END LOOP;
ROLLBACK to a;
END;
/
```

You can use many SAVEPOINTs; the ROLLBACK statement can roll back to a specific SAVEPOINT

Check table contents:

```
SELECT * FROM Mytable_T;
```

ID	DESCRIPTION
1	Description for 1

An Oracle Example (cont.)

COMMIT and then ROLLBACK:

```
CREATE TABLE Mytable_T
          (id          NUMBER       NOT NULL,
          description  VARCHAR2(50) NOT NULL);

INSERT INTO Mytable_T (id, description) VALUES (1, 'Description for 1');
INSERT INTO Mytable_T (id, description) VALUES (2, 'Description for 2');
SAVEPOINT a;
BEGIN
  FOR i IN 3 .. 10 LOOP
    INSERT INTO Mytable_T (id, description)
    VALUES (i, 'Description for ' || i);
  END LOOP;
COMMIT;
ROLLBACK ;
END;
/
```

After issuing the COMMIT command, the insertions became final and the ROLLBACK statement cannot remove the insertions

Check table contents:

```
SELECT * FROM Mytable_T;
```

ID	DESCRIPTION
1	Description for 1
2	Description for 2
3	Description for 3
4	Description for 4
5	Description for 5
6	Description for 6
7	Description for 7
8	Description for 8
9	Description for 9
10	Description for 10

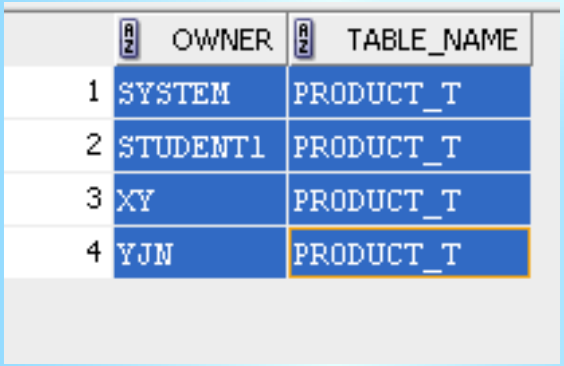
Data Dictionary Facilities

- **Data Dictionary**: System tables that store metadata (DB definition info), valuable info for users and DBAs.
- Users usually can view some of these tables by using SELECT statements.
- Users are restricted from updating them.
- Some examples in Oracle 11g (which has a total of 522 views for DBAs). You must be logged in as a DBA.
  - DBA_TABLES – descriptions of tables.
  - DBA_CONSTRAINTS – descriptions 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.

Some Oracle Examples

Find out the owner of Product_T table:

```
SELECT OWNER, TABLE_NAME
        FROM DBA_TABLES
                WHERE TABLE_NAME = 'PRODUCT_T';
```



	OWNER	TABLE_NAME
1	SYSTEM	PRODUCT_T
2	STUDENT1	PRODUCT_T
3	XY	PRODUCT_T
4	YJN	PRODUCT_T

Some Oracle Examples (cont.)

Find out all the information about all the tables in the database

```
SELECT *
    FROM ALL_TABLES;
```

	OWNER	TABLE_NAME	TABLESPACE_NAME	CLUSTER_NAME	IOT_NAME	STATUS	PCT_FREE	PCT_USED	INI_TRANS	MAX_TRANS
1	SYS	ICOL$	SYSTEM	C_OBJ#	(null)	VALID	0	0	0	0
2	SYS	CON$	SYSTEM	(null)	(null)	VALID	10	40	1	255
3	SYS	UNDO$	SYSTEM	(null)	(null)	VALID	10	40	1	255
4	SYS	PROXY_ROLE_DATA$	SYSTEM	(null)	(null)	VALID	10	40	1	255
5	SYS	FILE$	SYSTEM	(null)	(null)	VALID	10	40	1	255
6	SYS	UET$	SYSTEM	C_FILE#_BLOCK#	(null)	VALID	0	0	0	0
7	SYS	IND$	SYSTEM	C_OBJ#	(null)	VALID	0	0	0	0
8	SYS	SEG$	SYSTEM	C_FILE#_BLOCK#	(null)	VALID	0	0	0	0
9	SYS	COL$	SYSTEM	C_OBJ#	(null)	VALID	0	0	0	0
10	SYS	CLU$	SYSTEM	C_OBJ#	(null)	VALID	0	0	0	0
11	SYS	PROXY_DATA$	SYSTEM	(null)	(null)	VALID	10	40	1	255
12	SYS	TS$	SYSTEM	C_TS#	(null)	VALID	0	0	0	0
13	SYS	BOOTSTRAP$	SYSTEM	(null)	(null)	VALID	10	40	1	255
14	SYS	FET$	SYSTEM	C_TS#	(null)	VALID	0	0	0	0
15	SYS	CCOL$	SYSTEM	C_COBJ#	(null)	VALID	0	0	0	0
16	SYS	USER$	SYSTEM	C_USER#	(null)	VALID	0	0	0	0
17	SYS	OBJ$	SYSTEM	(null)	(null)	VALID	10	40	1	255
18	SYS	TAB$	SYSTEM	C_OBJ#	(null)	VALID	0	0	0	0
19	SYS	CDEF$	SYSTEM	C_COBJ#	(null)	VALID	0	0	0	0
20	SYS	OBJERROR$	SYSTEM	(null)	(null)	VALID	10	40	1	255
21	SYS	OBJAUTH$	SYSTEM	(null)	(null)	VALID	10	40	1	255

Some Oracle Examples (cont.)

Find out all the users of the database:

```
SELECT *
        FROM DBA_USERS;
```

	USERNAME	USER_ID	PASSWORD	ACCOUNT_STATUS	LOCK_DATE	EXPIRY_DATE	DEFAULT_TABLESPACE	TEMPORARY_TABLESPACE
1	SYSTEM	5	(null)	OPEN	(null)	15-JUN-12	SYSTEM	TEMP
2	SYS	0	(null)	OPEN	(null)	15-JUN-12	SYSTEM	TEMP
3	YJN	55	(null)	OPEN	(null)	21-AUG-12	SYSTEM	TEMP
4	MARVIN	49	(null)	OPEN	(null)	16-JUN-12	SYSTEM	TEMP
5	STUDENT1	54	(null)	OPEN	(null)	15-JUL-12	SYSTEM	TEMP
6	ANONYMOUS	35	(null)	OPEN	(null)	23-FEB-12	SYSAUX	TEMP
7	HR	43	(null)	OPEN	(null)	17-JUN-12	USERS	TEMP
8	XY	48	(null)	OPEN	(null)	15-JUN-12	USERS	TEMP
9	XIAOLONG	50	(null)	LOCKED(TIMED)	10-JAN-12	16-JUN-12	USERS	TEMP
10	APEX_PUBLIC_USER	45	(null)	LOCKED	27-AUG-11	23-FEB-12	SYSTEM	TEMP
11	APEX_040000	47	(null)	LOCKED	27-AUG-11	23-FEB-12	SYSAUX	TEMP
12	FLOWS_FILES	44	(null)	LOCKED	27-AUG-11	23-FEB-12	SYSAUX	TEMP
13	XS$NULL	2147483638	(null)	EXPIRED & LOCKED	27-AUG-11	27-AUG-11	SYSTEM	TEMP
14	OUTLN	9	(null)	EXPIRED & LOCKED	18-DEC-11	18-DEC-11	SYSTEM	TEMP
15	XDB	34	(null)	EXPIRED & LOCKED	27-AUG-11	27-AUG-11	SYSAUX	TEMP
16	CTXSYS	32	(null)	EXPIRED & LOCKED	18-DEC-11	18-DEC-11	SYSAUX	TEMP
17	MDSYS	42	(null)	EXPIRED & LOCKED	27-AUG-11	18-DEC-11	SYSAUX	TEMP

SQL:1999 and SQL:200N Enhancements/Extensions

- User-defined data types (UDT)
  - Users can define subclasses of standard types or an object type, which may contain functions/methods
- Analytical functions (for OLAP)
  - CEILING, FLOOR, SQRT, RANK, DENSE_RANK, ROLLUP, CUBE, SAMPLE,
  - WINDOW–improved numerical analysis capabilities
- New Data Types
  - BIGINT, MULTISET (collection), XML
- CREATE TABLE LIKE–create a new table similar to an existing one
- MERGE

SQL:1999 and SQL:200N Enhancements/Extensions (cont.)

- Persistent Stored Modules (SQL/PSM)
  - Capability to create and drop code modules
  - New statements:
    - CASE, IF, LOOP, FOR, WHILE, etc.
    - Makes SQL into a **procedural language**
- Oracle has proprietary version called **PL/SQL**, and Microsoft SQL Server has Transact/SQL

Oracle PL/SQL

PL/SQL stands for Procedural Language extension of SQL.

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 1990's to enhance the capabilities of SQL.

A PL/SQL Block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory).
- The Exception (or Error) Handling section (optional).

```
DECLARE
    Variable declaration
BEGIN
    Program Execution
EXCEPTION
    Exception handling
END;
```

An Oracle Example: Using RANK()

Find the employee with the highest salary:

```
SELECT *
FROM (
  SELECT employee_id, last_name, salary,
  RANK() OVER (ORDER BY salary DESC) EMPRANK
  FROM employees)
WHERE EMPRANK = 1;
```

EMPLOYEE_ID	LAST_NAME	SALARY	EMPRANK
1	Yang	2000000	1
2	Shen	2000000	1

An Oracle Example: Using PL/SQL

Calculate bonus for an employee:

```
DECLARE
        bonus NUMBER(8,2);
        emp_id NUMBER(6) := 1;
BEGIN
        SELECT salary * 0.10 INTO bonus
                FROM employees
                    WHERE employee_id = emp_id;
DBMS_OUTPUT.PUT_LINE (bonus);
END;
 /
```

PL/SQL procedure successfully completed.

200000

An Oracle Example: Using WHILE and LOOP Statements in PL/SQL

```
DECLARE
        done BOOLEAN; -- Initialize to NULL by default
        counter NUMBER := 0;
BEGIN
        done := FALSE; -- Assign literal value
        WHILE done != TRUE -- Compare to literal value
        LOOP
                counter := counter + 1;
                done := (counter > 500); -- Assign value of BOOLEAN expression
        END LOOP;
DBMS_OUTPUT.PUT_LINE ('Counter = '|| counter);
END;
/
```

PL/SQL procedure successfully completed.

Counter = 501

An Oracle Example: Using CASE Statement in PL/SQL

```
DECLARE
        grade CHAR(1) := 'B';
        appraisal VARCHAR2(20);
BEGIN
        appraisal :=
         CASE grade
            WHEN 'A' THEN 'Excellent'
            WHEN 'B' THEN 'Very Good'
            WHEN 'C' THEN 'Good'
            WHEN 'D' THEN 'Fair'
            WHEN 'F' THEN 'Poor'
            ELSE 'No such grade'
        END;
```

```
DBMS_OUTPUT.PUT_LINE
('Grade ' || grade || ' is ' || appraisal);
END;
/
```

PL/SQL procedure successfully completed.

Grade B is Very Good

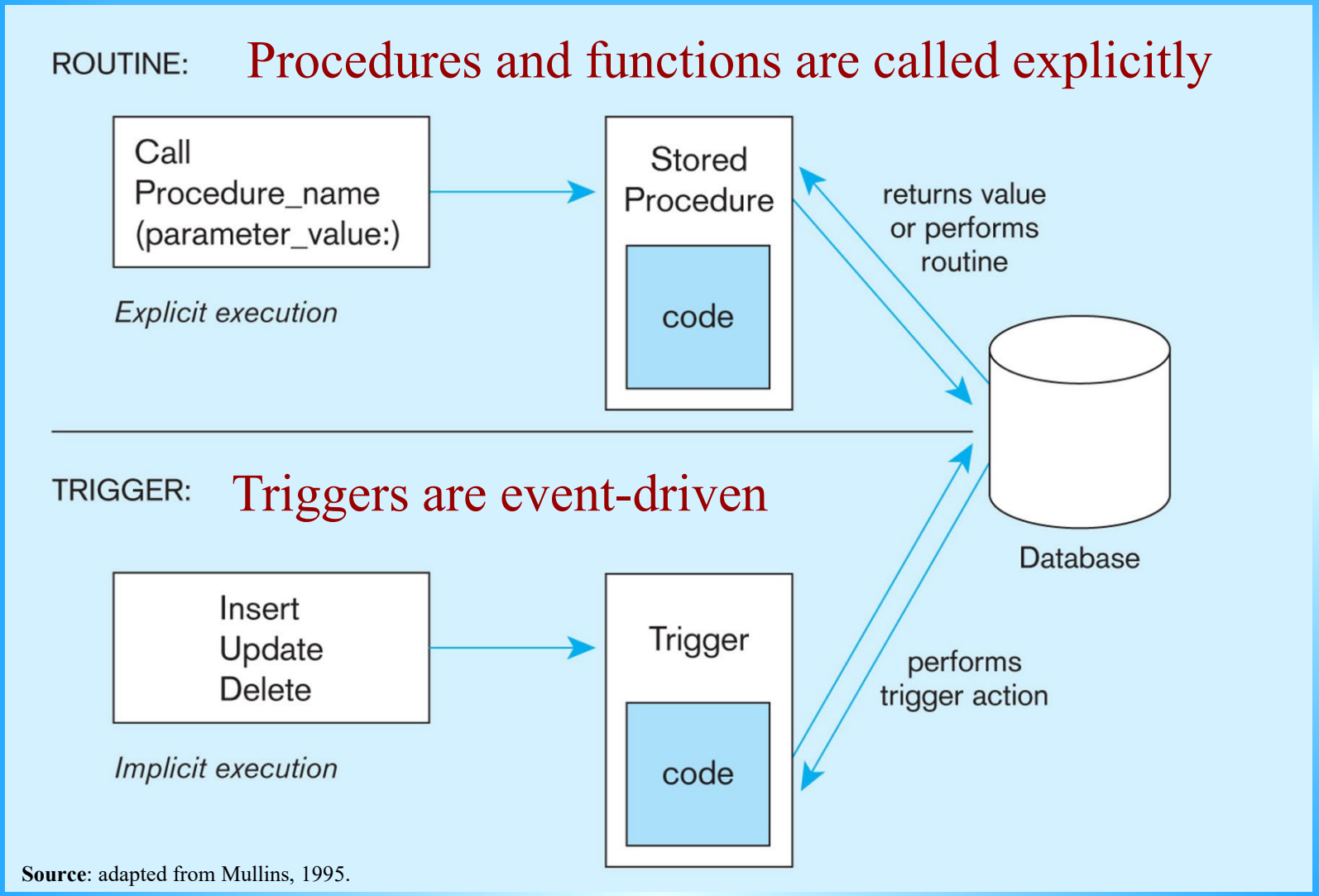
Routines and Triggers

- **Routines**:

  Stored program modules (or subprograms) that execute on demand (or must be called explicitly)

- **Function**: a routine that returns a value
- **Procedure**: a routine that performs a specific action but does not return a value
- **Trigger**: a named set of SQL statements that executes automatically in response to a database event (e.g., INSERT, UPDATE, or DELETE)

Triggers Contrasted with Stored Routines



Source: adapted from Mullins, 1995.

Syntax for Creating a Procedure or Function, SQL:200n

```
{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 | OLD SAVEPOINT LEVEL]
routine_body
```

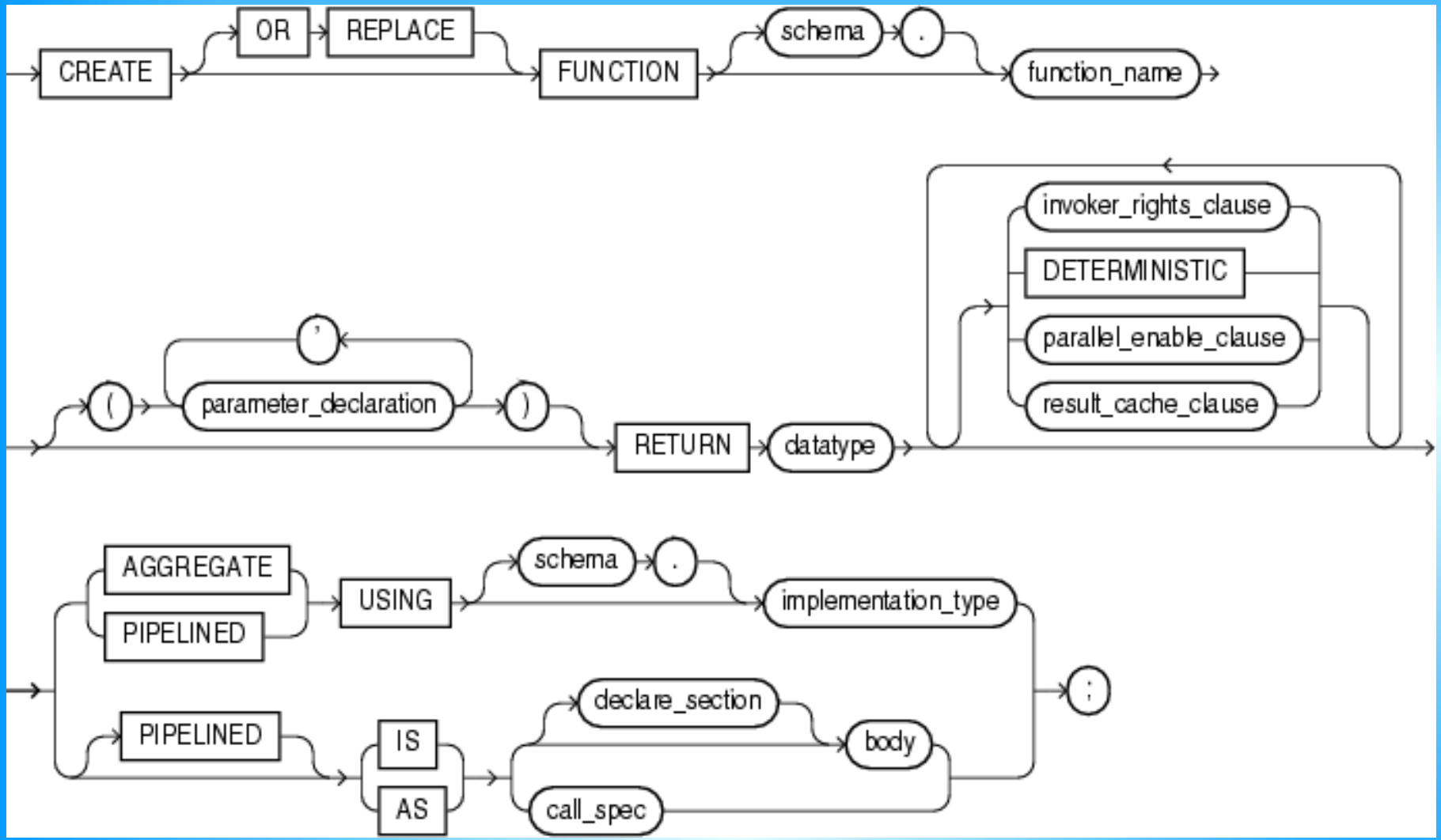
TABLE 7-2 Comparison of Vendor Syntax Differences in Stored Procedures

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.

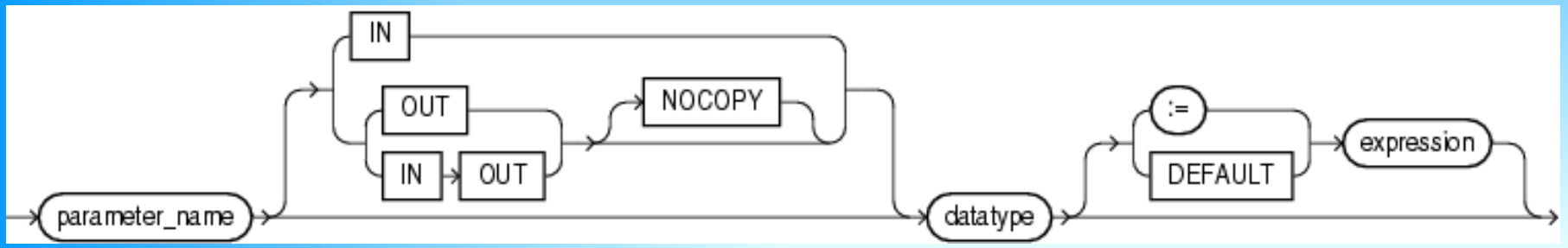
SQL:1999/IBM	MICROSOFT/SYBASE	ORACLE
CREATE PROCEDURE	CREATE PROCEDURE	CREATE PROCEDURE
Sp_proc1	Sp_proc1	Sp_proc1
(param1 INT)	@param1 INT	(param1 IN OUT INT)
MODIFIES SQL DATA BEGIN DECLARE num1 INT;	AS DECLARE @num1 INT	AS num1 INT; BEGIN
IF param1 <> 0	IF @param1 <> 0	IF param1 <> 0
THEN SET param1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

Oracle Syntax: Create a Function



Oracle Syntax: parameter_declaration



Example: The CREATE FUNCTION Statement in Oracle

Create the function **get_bal** on table Orders_T to return the balance of a specified account:

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
        RETURN NUMBER
        IS acc_bal NUMBER(11,2);
        BEGIN
                SELECT order_total
                INTO acc_bal
                FROM Orders_T
                WHERE customer_id = acc_no;
                RETURN(acc_bal);
        END;
/
```

To get the balance of the account number 165:

```
SELECT get_bal(165) FROM DUAL;
```

```
GET_BAL(165)
------------
        2519
```

Example: The CREATE PROCEDURE Statement in Oracle

To update the salary of an employee on table Employees_T:

```
CREATE PROCEDURE Update_salary (em_id NUMBER, new_salary
NUMBER) AS
        updated boolean;
        BEGIN
            UPDATE Employees_T
            SET Salary=new_salary
            WHERE Employees_T.EMPLOYEE_ID = em_id;
            updated := true;
        END;
/
```

Update the salary of employee # 100 to $500,000:

```
EXECUTE Update_salary(100, 500000);
```

Example: The CREATE PROCEDURE Statement without Parameters in Oracle

To calculate a sale price based on the standard price of the product:

```
CREATE PROCEDURE ProductLineSale AS
        BEGIN
            UPDATE Product_T
                SET SalePrice = 0.9*ProductStandardPrice
                WHERE ProductStandardPrice >= 400;
            UPDATE Product_T
                        SET SalePrice = 0.85*ProductStandardPrice
                        WHERE ProductStandardPrice < 400;
        END;
/
```

To run the procedure:

```
EXECUTE ProductLineSale;
```

Note: if the SalePrice column does not exist, run the following statement:

```
ALTER TABLE Product_T
ADD (SalePrice NUMBER (6,2));
```

CREATE TRIGGER syntax

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
[<ENABLE | DISABLE>]
<BEFORE | AFTER> <ACTION> [OR <ACTION> OR
<ACTION>]
ON <table_name>
[FOLLOWS <schema.trigger_name>]
[FOR EACH ROW]
DECLARE
 <variable definitions>
BEGIN
  <trigger_code>
EXCEPTION
  <exception clauses>
END <trigger_name>;
/
```

CREATE TRIGGER

```
CREATE OR REPLACE TRIGGER orders_after_update
AFTER UPDATE
    ON orders_t
    FOR EACH ROW
DECLARE
    v_username varchar2(10);
BEGIN
    -- Find username of a person and perform UPDATE into table
    SELECT user INTO v_username
    FROM dual;
    -- Insert record into an audit table created earlier
    INSERT INTO orders_audit_t
     ( order_id, quantity_before,  quantity_after, username )
    VALUES
     ( :new.order_id, :old.quantity, :new.quantity, v_username );
END;
/
```

Another Example: CREATE TRIGGER

```
CREATE OR REPLACE TRIGGER xy.salary_check
   BEFORE INSERT OR UPDATE OF salary, jobtitle ON xy.emp_t
   FOR EACH ROW
   BEGIN
      WHEN (new.jobtitle <> 'Data Scientist')
      CALL check_salary(:new.jobtitle, :new.salary, :new.empname);
   END;
/
```

Note: The check_salary() procedure has been implemented earlier.

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 (3rd Generation Languages)

- Can create a more flexible, accessible interface for the users.
- Possible performance improvement.
- Database security improvement; grant access only to the application instead of users.

Example: Embedded SQL in C

```
#include <stdio.h>

/* declare host variables */
char userid[12]="SCOTT/TIGER";
char emp_name[10];
void sql_error();

/*  include the SQL Communications Area */
#include<sqlca.h>

void main()
{
/* handle errors */
   EXEC SQL WHENEVER SQLERROR do sql_error("Oracle
error");

/* connect to Oracle */
   EXEC SQL CONNECT :userid;
   printf("Connected.\n");

/* declare a scrollable cursor */
   EXEC SQL DECLARE emp_cursor SCROLL CURSOR FOR
   SELECT ename FROM emp;

/* open the cursor and identify the active set */
    EXEC SQL OPEN emp_cursor;

 /* Fetch the last row */
    EXEC SQL FETCH LAST emp_cursor INTO
:emp_name;

 /* Fetch row number 5 */
     EXEC SQL FETCH ABSOLUTE 5 emp_cursor INTO
:emp_name;

/*  Fetch row number 10 */
      EXEC SQL FETCH RELATIVE 5 emp_cursor INTO
:emp_name;

/*  Fetch row number 7 */
      EXEC SQl FETCH RELATIVE -3 emp_cursor INTO
:emp_name;

/*  Fetch the first row */
      EXEC SQL FETCH FIRST emp_cursor INTO
:emp_name;

/*  Fetch row number 2*/
      EXEC SQL FETCH my_cursor INTO :emp_name;

/*  Fetch row number 3 */
      EXEC SQL FETCH NEXT my_cursor INTO :emp_name;

/*  Fetch row number 3 */
      EXEC SQL FETCH CURRENT my_cursor INTO
:emp_name;

/*  Fetch row number 2 */
       EXEC SQL FETCH PRIOR my_cursor INTO
:emp_name;
}
```