

COP 4710: Database Systems Spring 2013

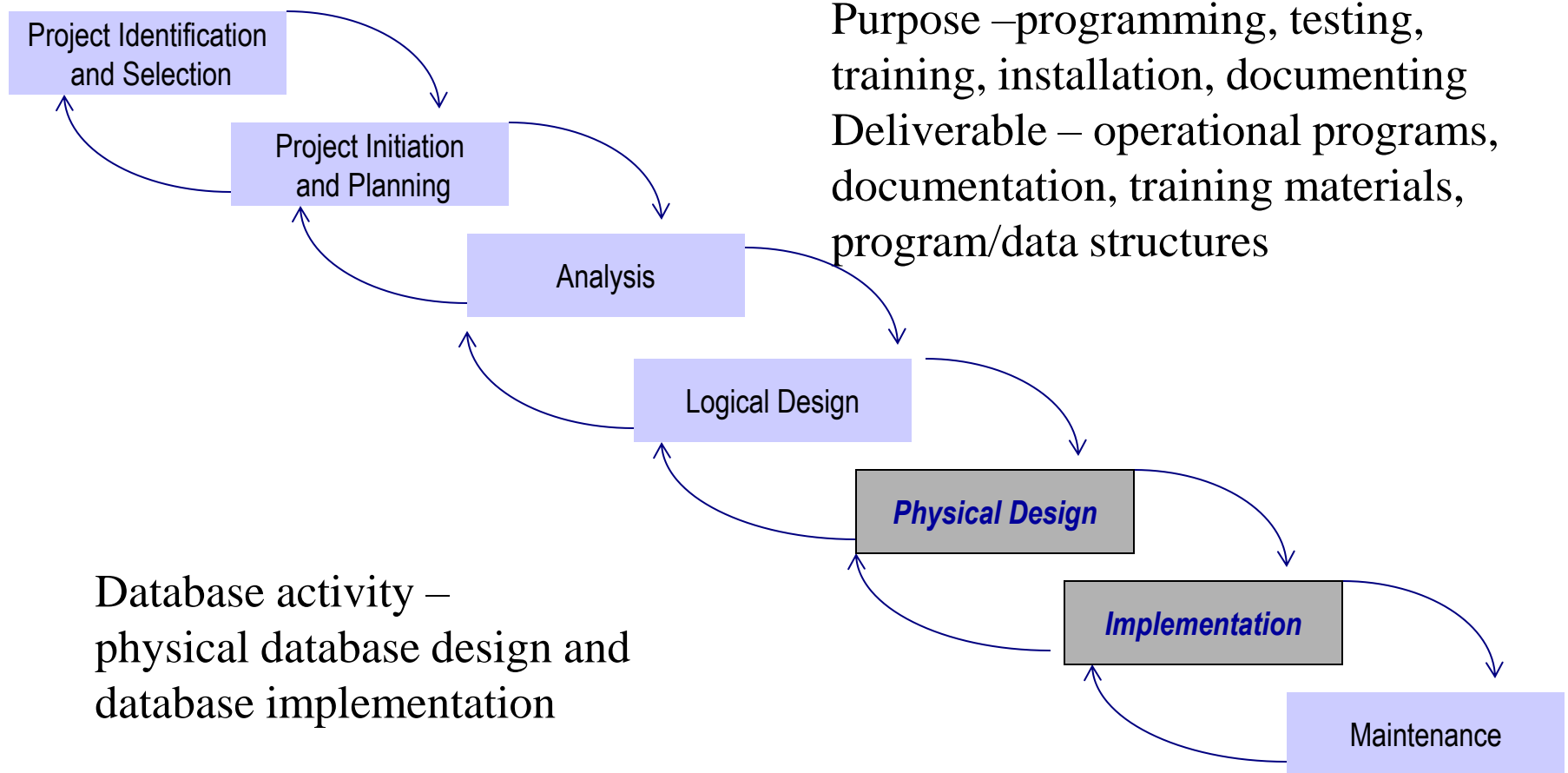
Chapter 5 – Introduction To SQL

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
<http://www.cs.ucf.edu/courses/cop4710/spr2013>

Department Of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida



The Physical Design Stage of SDLC



SQL Overview

- SQL \equiv Structured Query Language.
- The standard for relational database management systems (RDBMS).
- SQL: 2007 Standards – Purpose:
 - Specify syntax/semantics for data definition and manipulation.
 - Define data structures.
 - Enable portability.
 - Specify minimal (level 1) and complete (level 2) standards.
 - Allow for later growth/enhancement to standard.
- SQL: 20XX Standard



Benefits of a Standardized Relational Language

- Reduced training costs
- Productivity
- Application portability
- Application longevity
- Reduced dependence on a single vendor
- Cross-system communication

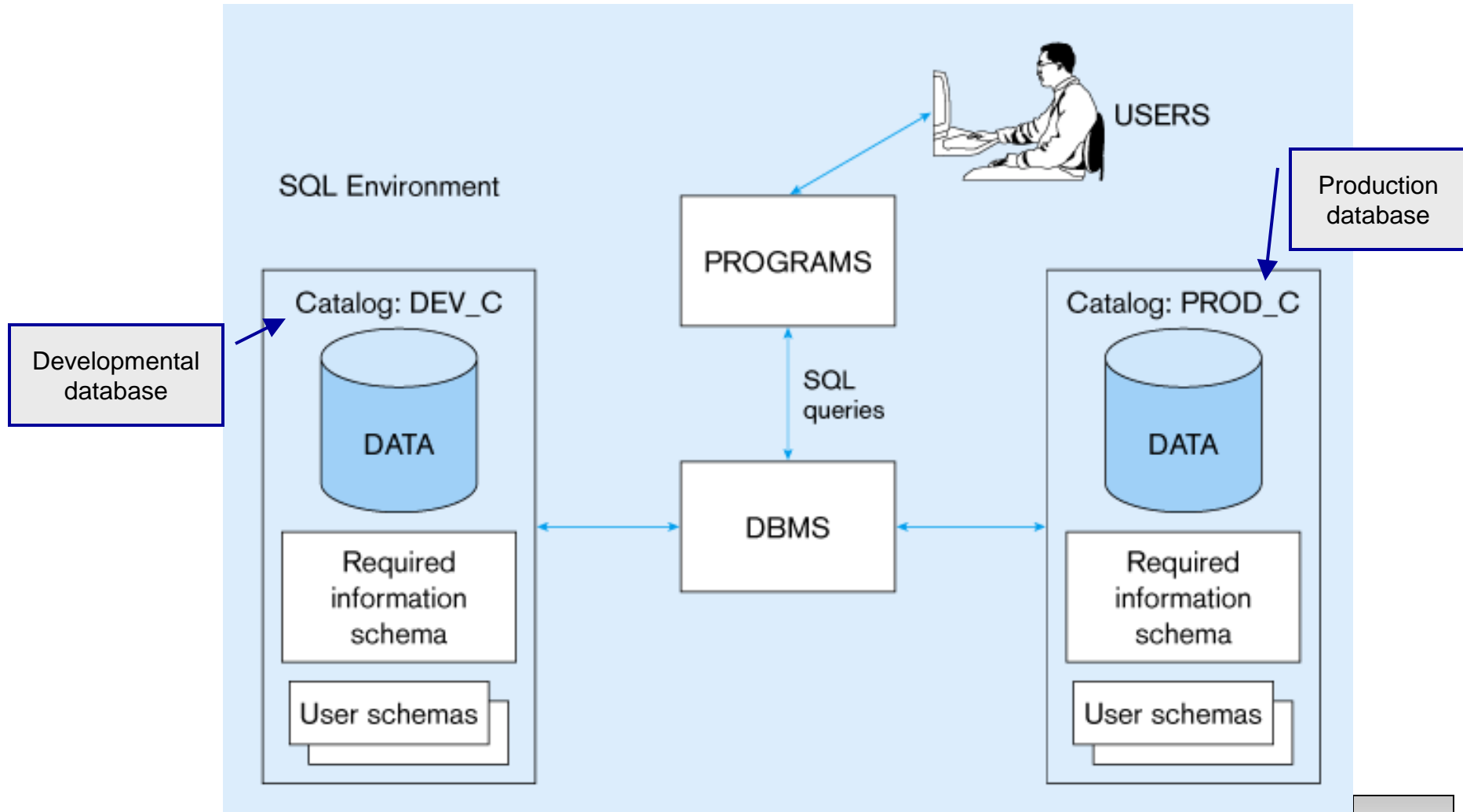


The SQL Environment

- Catalog
 - A set of schemas that constitute the description of a database.
- Schema
 - The structure that contains descriptions of objects created by a user (base tables, views, constraints).
- Data Definition Language (DDL)
 - Commands that define a database, including creating, altering, and dropping tables and establishing constraints.
- Data Manipulation Language (DML)
 - Commands that maintain and query a database.
- Data Control Language (DCL)
 - Commands that control a database, including administering privileges and committing data.



A simplified schematic of a typical SQL environment, as described by the SQL:20xx standard

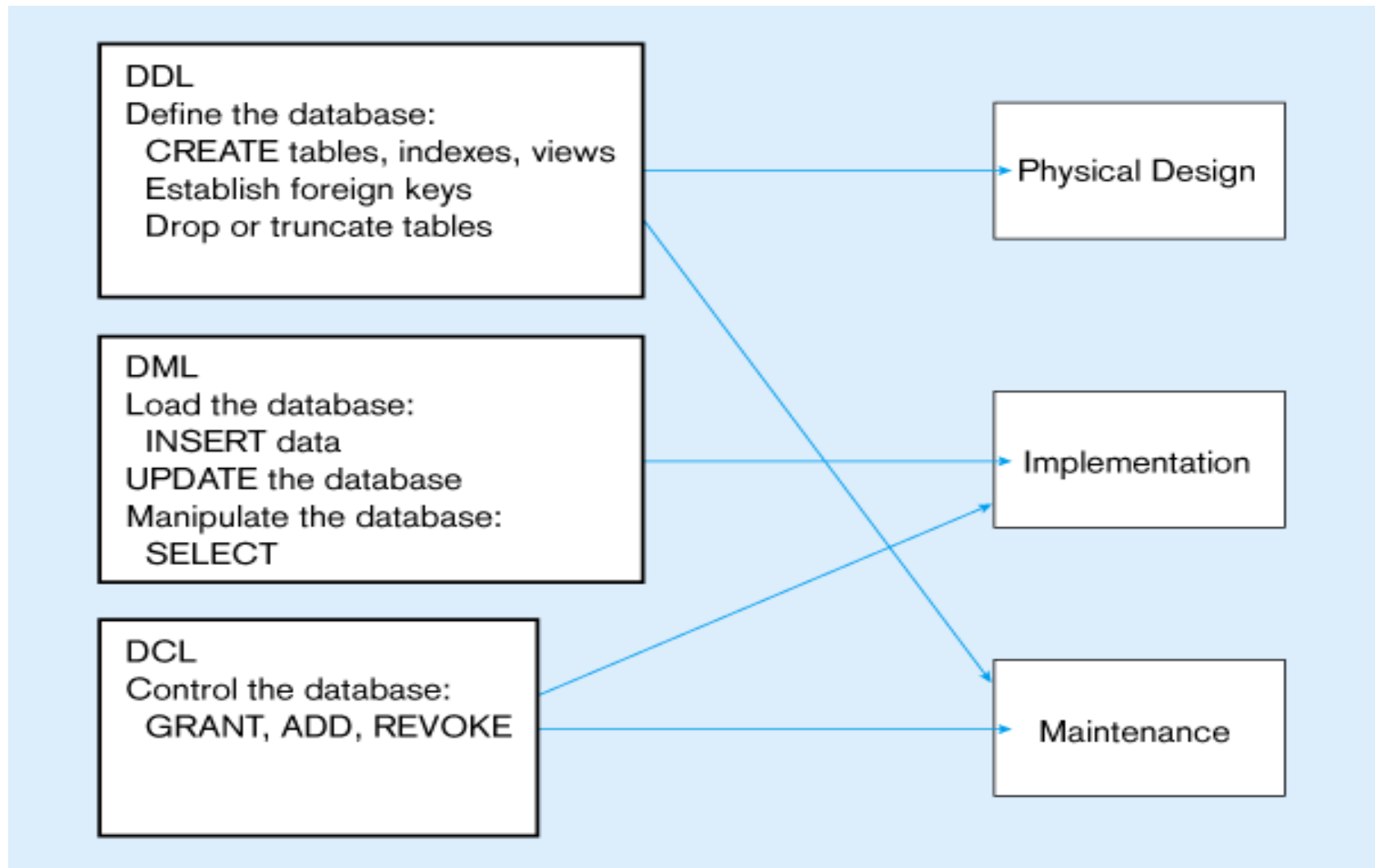


Some SQL Data Types (from Oracle 11g)

- String types
 - CHAR(n) – fixed-length character data, n characters long
Maximum length = 2000 bytes
 - VARCHAR2(n) – variable length character data, maximum 4000 bytes
 - LONG – variable-length character data, up to 4GB. Maximum 1 per table
- Numeric types
 - NUMBER(p,q) – general purpose numeric data type
 - INTEGER(p) – signed integer, p digits wide
 - FLOAT(p) – floating point in scientific notation with p binary digits precision
- Date/time type
 - DATE – fixed-length date/time in dd-mm-yy form



DDL, DML, DCL, and the database development process



SQL Database Definition

- Data Definition Language (DDL)
- Major CREATE statements:
 - CREATE SCHEMA – defines a portion of the database owned by a particular user.
 - CREATE TABLE – defines a table and its columns.
 - CREATE VIEW – defines a logical table from one or more views.
- Other CREATE statements: CHARACTER SET, COLLATION, TRANSLATION, ASSERTION, DOMAIN.



Table Creation

General syntax for CREATE TABLE

```
CREATE TABLE tablename  
( {column definition [table constraint] } . . .  
[ON COMMIT {DELETE | PRESERVE} ROWS] );
```

where *column definition* ::=

```
column_name  
    {domain name | datatype [(size)] }  
    [column_constraint_clause . . .]  
    [default value]  
    [collate clause]
```

and *table constraint* ::=

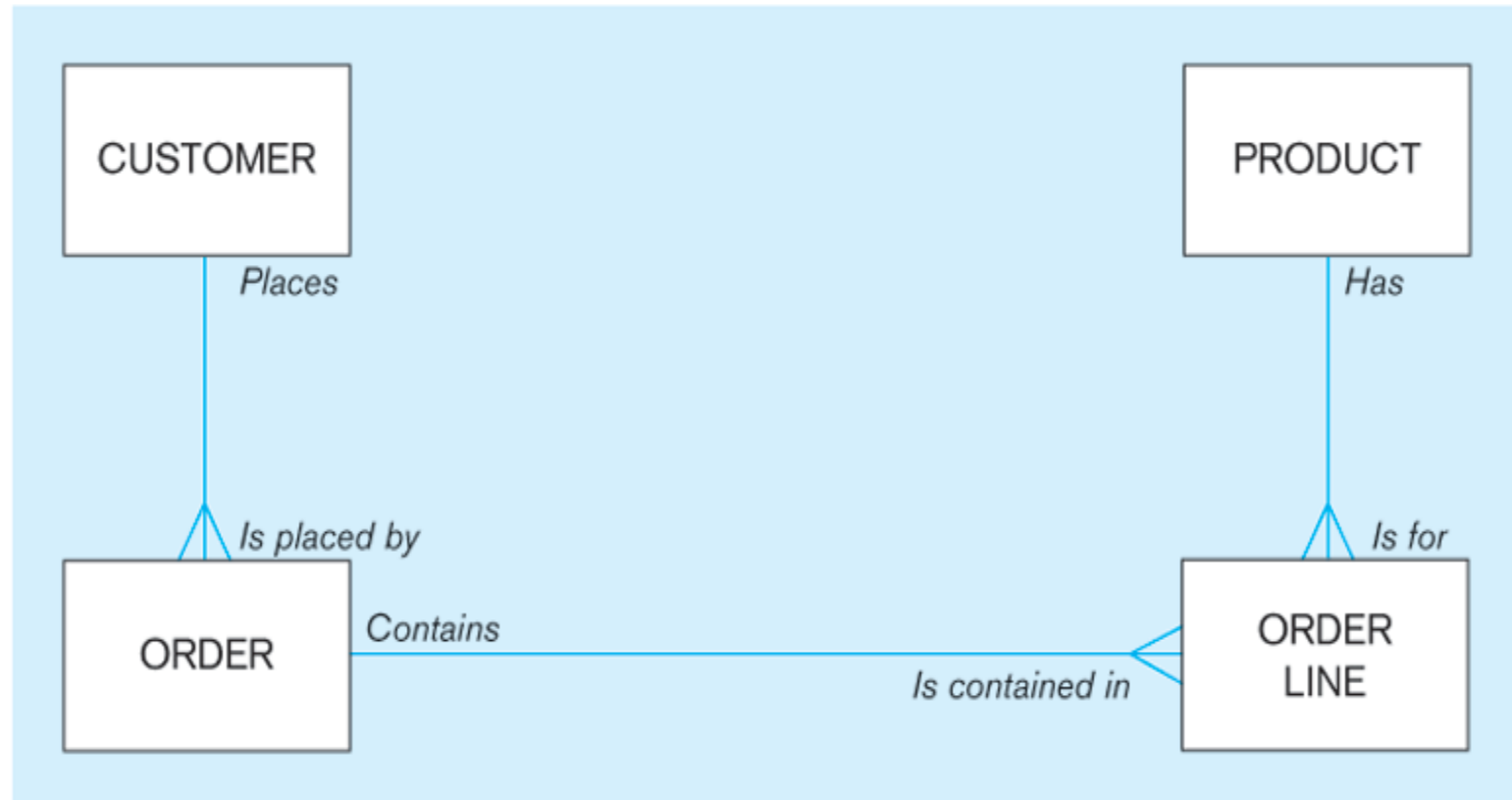
```
[CONSTRAINT constraint_name  
  Constraint_type [constraint_attributes]
```

Steps in table creation:

1. Identify data types for attributes
2. Identify columns that can and cannot be null
3. Identify columns that must be unique (candidate keys)
4. Identify primary key-foreign key mates
5. Determine default values
6. Identify constraints on columns (domain specifications)
7. Create the table and associated indexes



The following slides create tables for this enterprise data model



Examples of SQL database definition commands

```
CREATE TABLE CUSTOMER_T
(CUSTOMER_ID          NUMBER(11, 0) NOT NULL,
 CUSTOMER_NAME        VARCHAR2(25) NOT NULL,
 CUSTOMER_ADDRESS     VARCHAR2(30),
 CITY                 VARCHAR2(20),
 STATE                 VARCHAR2(2),
 POSTAL_CODE           VARCHAR2(9),
 CONSTRAINT CUSTOMER_PK PRIMARY KEY (CUSTOMER_ID));
```

```
CREATE TABLE ORDER_T
(ORDER_ID             NUMBER(11, 0) NOT NULL,
 ORDER_DATE           DATE          DEFAULT SYSDATE,
 CUSTOMER_ID          NUMBER(11, 0),
 CONSTRAINT ORDER_PK PRIMARY KEY (ORDER_ID),
 CONSTRAINT ORDER_FK FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER_T(CUSTOMER_ID));
```

```
CREATE TABLE PRODUCT_T
(PRODUCT_ID           INTEGER      NOT NULL,
 PRODUCT_DESCRIPTION   VARCHAR2(50),
 PRODUCT_FINISH        VARCHAR2(20)
                      CHECK (PRODUCT_FINISH IN ('Cherry', 'Natural Ash', 'White Ash',
                                                  'Red Oak', 'Natural Oak', 'Walnut')),
 STANDARD_PRICE        DECIMAL(6,2),
 PRODUCT_LINE_ID       INTEGER,
 CONSTRAINT PRODUCT_PK PRIMARY KEY (PRODUCT_ID));
```

```
CREATE TABLE ORDER_LINE_T
(ORDER_ID             NUMBER(11,0) NOT NULL,
 PRODUCT_ID           NUMBER(11,0) NOT NULL,
 ORDERED_QUANTITY     NUMBER(11,0),
 CONSTRAINT ORDER_LINE_PK PRIMARY KEY (ORDER_ID, PRODUCT_ID),
 CONSTRAINT ORDER_LINE_FK1 FOREIGN KEY(ORDER_ID) REFERENCES ORDER_T(ORDER_ID),
 CONSTRAINT ORDER_LINE_FK2 FOREIGN KEY (PRODUCT_ID) REFERENCES PRODUCT_T(PRODUCT_ID));
```



Defining attributes and their data types

```
CREATE TABLE PRODUCT_T
```

```
(PRODUCT_ID          INTEGER NOT NULL,  
  PRODUCT_DESCRIPTION VARCHAR2(50),  
  PRODUCT_FINISH      VARCHAR2(20)
```

Domain
constraint

```
    CHECK (PRODUCT_FINISH IN ('Cherry', 'Natural Ash', 'White Ash',  
                              'Red Oak', 'Natural Oak', 'Walnut')),
```

```
  STANDARD_PRICE      DECIMAL(6,2),  
  PRODUCT_LINE_ID     INTEGER,
```

```
  CONSTRAINT PRODUCT_PK PRIMARY KEY (PRODUCT_ID));
```



```
CREATE TABLE PRODUCT_T
```

```
(PRODUCT_ID
```

```
INTEGER
```

```
NOT NULL,
```

```
PRODUCT_DESCRIPTION
```

```
VARCHAR2(50),
```

```
PRODUCT_FINISH
```

```
VARCHAR2(20)
```

```
CHECK (PRODUCT_FINISH IN ('Cherry', 'Natural Ash', 'White Ash',  
'Red Oak', 'Natural Oak', 'Walnut')),
```

```
STANDARD_PRICE
```

```
DECIMAL(6,2),
```

```
PRODUCT_LINE_ID
```

```
INTEGER,
```

```
CONSTRAINT PRODUCT_PK PRIMARY KEY (PRODUCT_ID));
```

Non-null specification

Identifying primary key

Primary keys
can never have
NULL values



```
CREATE TABLE ORDER_LINE_T
```

Non-null specifications

```
(ORDER_ID          NUMBER(11,0) NOT NULL,
```

```
PRODUCT_ID        NUMBER(11,0) NOT NULL,
```

```
ORDERED_QUANTITY  NUMBER(11,0),
```

```
CONSTRAINT ORDER_LINE_PK PRIMARY KEY (ORDER_ID, PRODUCT_ID),
```

Primary key

```
CONSTRAINT ORDER_LINE_FK1 FOREIGN KEY (ORDER_ID) REFERENCES ORDER_T (ORDER_ID),
```

```
CONSTRAINT ORDER_LINE_FK2 FOREIGN KEY (PRODUCT_ID) REFERENCES PRODUCT_T (PRODUCT_ID));
```

**Some primary keys are composite –
composed of multiple attributes**



Controlling the values in attributes

```
CREATE TABLE ORDER_T
  (ORDER_ID          NUMBER(11, 0) NOT NULL,
   ORDER_DATE        DATE          DEFAULT SYSDATE,
   CUSTOMER_ID       NUMBER(11, 0),
  CONSTRAINT ORDER_PK PRIMARY KEY (ORDER_ID),
  CONSTRAINT ORDER_FK FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER_T(CUSTOMER_ID));

CREATE TABLE PRODUCT_T
  (PRODUCT_ID        INTEGER      NOT NULL,
   PRODUCT_DESCRIPTION VARCHAR2(50),
   PRODUCT_FINISH     VARCHAR2(20),
   CHECK (PRODUCT_FINISH IN ('Cherry', 'Natural Ash', 'White Ash',
                              'Red Oak', 'Natural Oak', 'Walnut')),
   STANDARD_PRICE     DECIMAL(6,2),
   PRODUCT_LINE_ID    INTEGER,
```

Default value

Domain constraint



Identifying foreign keys and establishing relationships

```
CREATE TABLE CUSTOMER_T
```

```
(CUSTOMER_ID          NUMBER(11, 0) NOT NULL,
```

```
  CUSTOMER_NAME        VARCHAR2(25) NOT NULL,
```

```
  CUSTOMER_ADDRESS     VARCHAR2(30),
```

```
  CITY                 VARCHAR2(20),
```

```
  STATE                VARCHAR2(2),
```

```
  POSTAL_CODE          VARCHAR2(9),
```

```
CONSTRAINT CUSTOMER_PK PRIMARY KEY (CUSTOMER_ID));
```

Primary key of
parent table

```
CREATE TABLE ORDER_T
```

```
(ORDER_ID             NUMBER(11, 0) NOT NULL,
```

```
  ORDER_DATE          DATE          DEFAULT SYSDATE,
```

```
  CUSTOMER_ID         NUMBER(11, 0),
```

```
CONSTRAINT ORDER_PK PRIMARY KEY (ORDER_ID),
```

```
CONSTRAINT ORDER_FK FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER_T(CUSTOMER_ID));
```

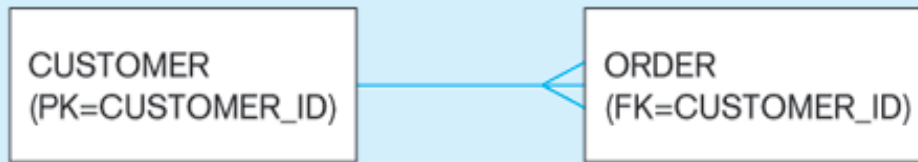
Foreign key of
dependent table



Data Integrity Controls

- **Referential integrity** – constraint that ensures that foreign key values of a table must match primary key values of a related table in 1:M relationships.
- **Restricting:**
 - Deletes of primary records.
 - Updates of primary records.
 - Inserts of dependent records.





Restricted Update: A customer ID can only be deleted if it is not found in ORDER table.

```
CREATE TABLE CUSTOMER_T
    (CUSTOMER_ID      INTEGER DEFAULT 'C999' NOT NULL,
     CUSTOMER_NAME    VARCHAR(40)         NOT NULL,
     ...
    CONSTRAINT CUSTOMER_PK PRIMARY KEY (CUSTOMER_ID),
    ON UPDATE RESTRICT);
```

Cascaded Update: Changing a customer ID in the CUSTOMER table will result in that value changing in the ORDER table to match.

```
... ON UPDATE CASCADE);
```

Set Null Update: When a customer ID is changed, any customer ID in the ORDER table that matches the old customer ID is set to NULL.

```
... ON UPDATE SET NULL);
```

Set Default Update: When a customer ID is changed, any customer ID in the ORDER tables that matches the old customer ID is set to a predefined default value.

```
... ON UPDATE SET DEFAULT);
```

Relational
integrity is
enforced via
the primary-
key to foreign-
key match



Changing and Removing Tables

- **ALTER TABLE** statement allows you to change column specifications:
 - ALTER TABLE CUSTOMER_T ADD (TYPE VARCHAR(2))
- **DROP TABLE** statement allows you to remove tables from your schema:
 - DROP TABLE CUSTOMER_T



Schema Definition

- Control processing/storage efficiency:
 - Choice of indexes
 - File organizations for base tables
 - File organizations for indexes
 - Data clustering
 - Statistics maintenance
- Creating indexes
 - Speed up random/sequential access to base table data
 - Example
 - `CREATE INDEX NAME_IDX ON CUSTOMER_T(CUSTOMER_NAME)`
 - This makes an index for the CUSTOMER_NAME field of the CUSTOMER_T table



Insert Statement

- Adds data to a table
- Inserting into a table
 - `INSERT INTO CUSTOMER_T VALUES (001, 'Contemporary Casuals', 1355 S. Himes Blvd.', 'Gainesville', 'FL', 32601);`
- Inserting a record that has some null attributes requires identifying the fields that actually get data
 - `INSERT INTO PRODUCT_T (PRODUCT_ID, PRODUCT_DESCRIPTION, PRODUCT_FINISH, STANDARD_PRICE, PRODUCT_ON_HAND) VALUES (1, 'End Table', 'Cherry', 175, 8);`
- Inserting from another table
 - `INSERT INTO CA_CUSTOMER_T SELECT * FROM CUSTOMER_T WHERE STATE = 'CA';`



Delete Statement

- Removes rows from a table.
- Delete certain rows
 - **DELETE FROM CUSTOMER_T WHERE STATE = 'HI';**
- Delete all rows
 - **DELETE FROM CUSTOMER_T;**



Update Statement

- Modifies data in existing rows
- `UPDATE PRODUCT_T SET UNIT_PRICE = 775
WHERE PRODUCT_ID = 7;`

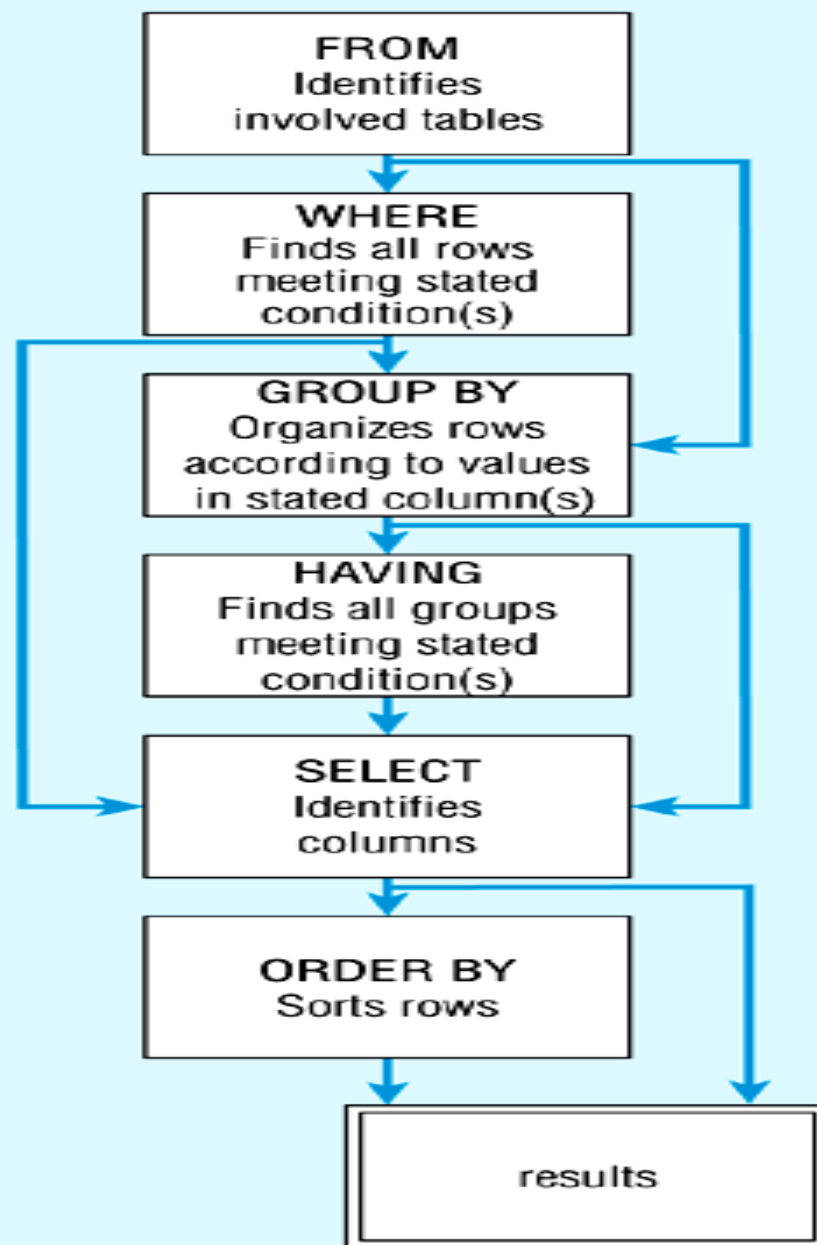


SELECT Statement

- Used for queries on single or multiple tables.
- Clauses of the SELECT statement:
 - **SELECT**
 - List the columns (and expressions) that should be returned from the query
 - **FROM**
 - Indicate the table(s) or view(s) from which data will be obtained
 - **WHERE**
 - Indicate the conditions under which a row will be included in the result
 - **GROUP BY**
 - Indicate categorization of results
 - **HAVING**
 - Indicate the conditions under which a category (group) will be included
 - **ORDER BY**
 - Sorts the result according to specified criteria



SQL statement processing order



SELECT Example

- Find products with standard price less than \$275

```
SELECT PRODUCT_NAME, STANDARD_PRICE  
FROM PRODUCT_V  
WHERE STANDARD_PRICE < 275;
```



SELECT Example using Alias

- Alias is an alternative column or table name.

```
SELECT CUST.CUSTOMER AS NAME,  
       CUST.CUSTOMER_ADDRESS  
FROM CUSTOMER_V CUST  
WHERE NAME = 'Home Furnishings';
```



SELECT Example Using a Function

- Using the COUNT *aggregate function* to find totals

```
SELECT COUNT(*) FROM ORDER_LINE_V  
WHERE ORDER_ID = 1004;
```

Note: with aggregate functions you can't have single-valued columns included in the SELECT clause



SELECT Example – Boolean Operators

- **AND**, **OR**, and **NOT** Operators for customizing conditions in WHERE clause

```
SELECT PRODUCT_DESCRIPTION, PRODUCT_FINISH,  
       STANDARD_PRICE  
FROM PRODUCT_V  
WHERE (PRODUCT_DESCRIPTION LIKE '%Desk'  
OR PRODUCT_DESCRIPTION LIKE '%Table')  
AND UNIT_PRICE > 300;
```

Note: the LIKE operator allows you to compare strings using wildcards. For example, the % wildcard in '%Desk' indicates that all strings that have any number of characters preceding the word "Desk" will be allowed



SELECT Example – Sorting Results with the ORDER BY Clause

- Sort the results first by STATE, and within a state by CUSTOMER_NAME

```
SELECT CUSTOMER_NAME, CITY, STATE  
FROM CUSTOMER_V  
WHERE STATE IN ('FL', 'TX', 'CA', 'HI')  
ORDER BY STATE, CUSTOMER_NAME;
```

Note: the IN operator in this example allows you to include rows whose STATE value is either FL, TX, CA, or HI. It is more efficient than separate OR conditions



SELECT Example –

Categorizing Results Using the GROUP BY Clause

- For use with aggregate functions
 - *Scalar aggregate*: single value returned from SQL query with aggregate function
 - *Vector aggregate*: multiple values returned from SQL query with aggregate function (via GROUP BY)

```
SELECT STATE, COUNT(STATE)
FROM CUSTOMER_V
GROUP BY STATE;
```

Note: you can use single-value fields with aggregate functions if they are included in the GROUP BY clause.



SELECT Example –

Qualifying Results by Category Using the HAVING Clause

- For use with GROUP BY

```
SELECT STATE, COUNT(STATE)
FROM CUSTOMER_V
GROUP BY STATE
HAVING COUNT(STATE) > 1;
```

Like a WHERE clause, but it operates on groups (categories), not on individual rows. Here, only those groups with total numbers greater than 1 will be included in final result



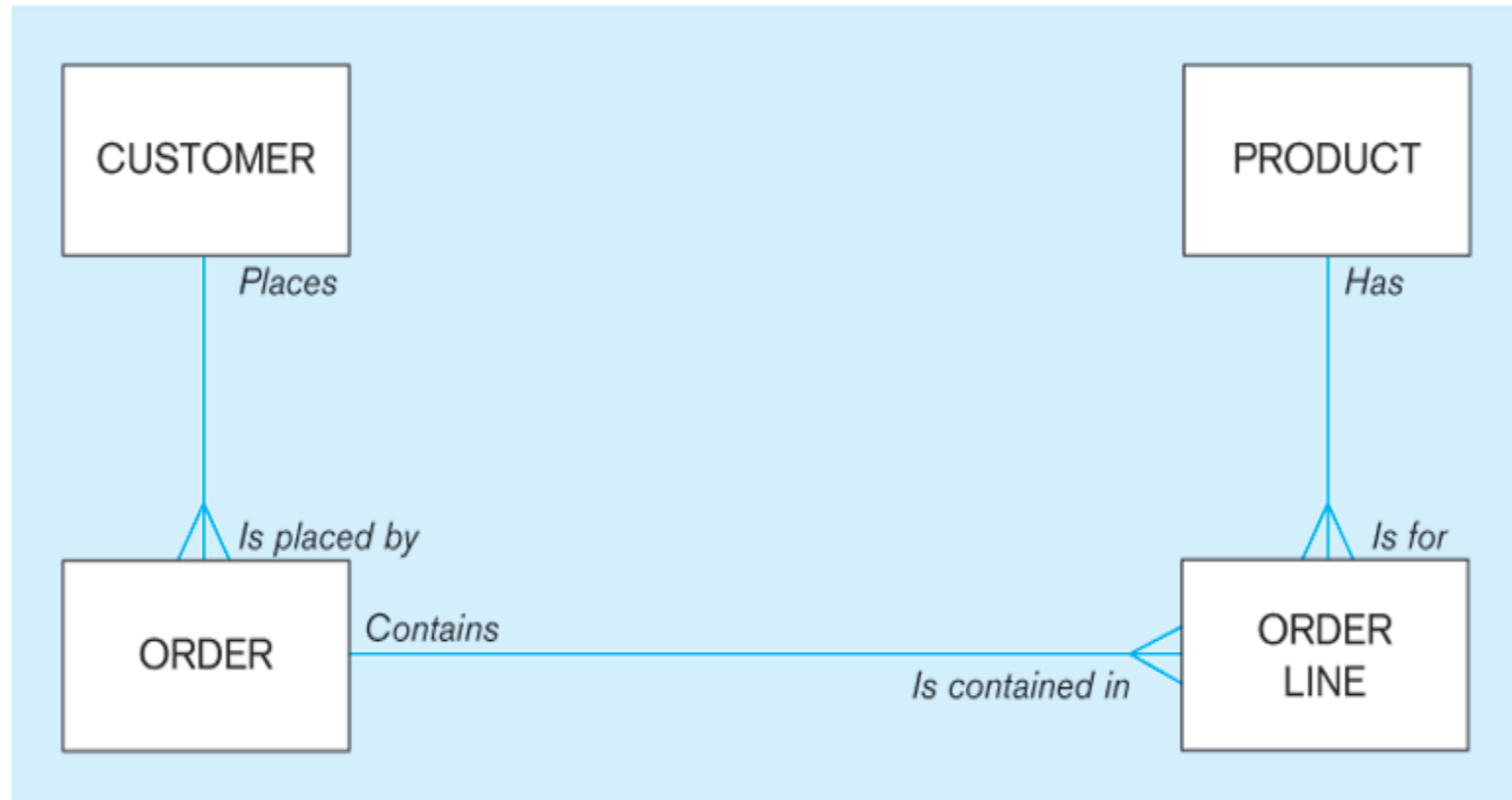
Processing Multiple Tables – Joins

- **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
- **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

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



Same Scenario As Page 11



Home Create External Data Database Tools Datasheet

Views Clipboard Font Rich Text Records Sort & Filter Window Find

Calibri 11

B I U

Refresh All New Save Delete Records

Filter

Size to Fit Form Switch Windows

Find

Navigation Pane

ORDER_t

Order_ID	Order_Date	Customer_ID	Add
1001	10/21/2008	1	
1010	11/5/2008	1	
1006	10/24/2008	2	
1005	10/24/2008	3	
1009	11/5/2008	4	
1004	10/22/2008	5	
1002	10/21/2008	8	
1007	10/27/2008	11	
1008	10/30/2008	12	
1003	10/22/2008	15	
*	0	0	

Record: 1 of 10

CUSTOMER_t

Customer_ID	Customer_Name	Customer_Address	City
1	Contemporary Casuals	1355 S Hines Blvd	Gainesville
2	Value Furniture	15145 S.W. 17th St.	Plano
3	Home Furnishings	1900 Allard Ave.	Albany
4	Eastern Furniture	1925 Beltline Rd.	Carteret
5	Impressions	5585 Westcott Ct.	Sacramento
6	Furniture Gallery	325 Flatiron Dr.	Boulder
7	Period Furniture	394 Rainbow Dr.	Seattle
8	California Classics	816 Peach Rd.	Santa Clara
9	M and H Casual Furniture	3709 First Street	Clearwater
10	Seminole Interiors	2400 Rocky Point Dr.	Seminole
11	American Euro Lifestyles	2424 Missouri Ave N.	Prospect Pa
12	Battle Creek Furniture	345 Capitol Ave. SW	Battle Creek
13	Heritage Furnishings	66789 College Ave.	Carlisle
14	Kaneohe Homes	112 Kiowai St.	Kaneohe
15	Mountain Scenes	4132 Main Street	Ogden
*	(New)		

Record: 4 of 15

These tables are used in queries that follow



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.CUSTOMER_ID, CUSTOMER_NAME, ORDER_ID  
FROM CUSTOMER_T, ORDER_T
```

```
WHERE CUSTOMER_T.CUSTOMER_ID = ORDER_T.CUSTOMER_ID;
```

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



Outer Join Example (Microsoft Syntax)

- 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.CUSTOMER_ID, CUSTOMER_NAME,  
       ORDER_ID  
FROM CUSTOMER_T, LEFT OUTER JOIN ORDER_T  
ON CUSTOMER_T.CUSTOMER_ID = ORDER_T.CUSTOMER_ID;
```

LEFT OUTER JOIN syntax with
ON keyword instead of WHERE
→ causes customer data to appear
even if there is no corresponding
order data



Results

CUSTOMER_ID	CUSTOMER_NAME	ORDER_ID
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 Furnishings	
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.

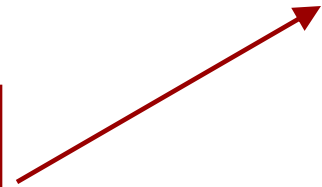


Outer Join Example (Oracle Syntax)

- 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.CUSTOMER_ID, CUSTOMER_NAME, ORDER_ID  
FROM CUSTOMER_T, ORDER_T  
WHERE CUSTOMER_T.CUSTOMER_ID = ORDER_T.CUSTOMER_ID(+);
```

Outer join in Oracle uses regular join syntax, but adds (+) symbol to the side that will have the missing data



Multiple Table Join Example

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

Four tables involved in this join

```
SELECT CUSTOMER_T.CUSTOMER_ID, CUSTOMER_NAME,  
       CUSTOMER_ADDRESS, CITY, STATE, POSTAL_CODE,  
       ORDER_T.ORDER_ID, ORDER_DATE, QUANTITY,  
       PRODUCT_NAME, UNIT_PRICE, (QUANTITY * UNIT_PRICE)  
FROM CUSTOMER_T, ORDER_T, ORDER_LINE_T, PRODUCT_T  
WHERE CUSTOMER_T.CUSTOMER_ID =  
       ORDER_LINE_T.CUSTOMER_ID   AND ORDER_T.ORDER_ID =  
       ORDER_LINE_T.ORDER_ID  
       AND ORDER_LINE_T.PRODUCT_ID =  
       PRODUCT_T.PRODUCT_ID  
       AND ORDER_T.ORDER_ID = 1006;
```

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



Results from a four-table join

From CUSTOMER_T table

CUSTOMER_ID	CUSTOMER_NAME	CUSTOMER_ADDRESS	CUSTOMER_CITY	CUSTOMER_ST	POSTAL_CODE
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

ORDER_ID	ORDER_DATE	ORDERED_QUANTITY	PRODUCT_NAME	STANDARD_PRICE	(QUANTITY* STANDARD_PRICE)
1006	24-OCT-04	1	Entertainment Center	650	650
1006	24-OCT-04	2	Writer's Desk	325	650
1006	24-OCT-04	2	Dining Table	800	1600

From ORDER_T table

From PRODUCT_T table



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.

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

```
SELECT CUSTOMER_NAME FROM CUSTOMER_T  
WHERE CUSTOMER_ID IN  
(SELECT DISTINCT CUSTOMER_ID 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



Correlated vs. 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.



Processing a noncorrelated subquery

1. The subquery executes and returns the customer IDs from the ORDER_T table
2. The outer query on the results of the subquery

```
SELECT CUSTOMER_NAME  
FROM CUSTOMER_T  
WHERE CUSTOMER_ID IN
```

(SELECT DISTINCT CUSTOMER_ID
FROM ORDER_T);

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

CUSTOMER_ID
1
8
15
5
3
2
11
12
4

9 rows selected.

No reference to data
in outer query, so
subquery executes
once only

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

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

9 rows selected.

These are the only
customers that have
IDs in the ORDER_T
table



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 ORDER_ID FROM ORDER_LINE_T
WHERE EXISTS
  (SELECT * FROM PRODUCT_T
   WHERE PRODUCT_ID = ORDER_LINE_T.PRODUCT_ID
   AND PRODUCT_FINISH = 'Natural ash');
```

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



Processing a correlated subquery

```
SELECT DISTINCT ORDER_ID FROM ORDER_LINE_T
WHERE EXISTS
  (SELECT *
   FROM PRODUCT_T
    WHERE PRODUCT_ID = ORDER_LINE_T.PRODUCT_ID
    AND PRODUCT_FINISH = 'Natural Ash');
```

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

	Order ID	Product ID	Ordered Quantity
1	1001	1	1
	1001	4	2
3	1002	3	1
	1002	3	1
	1004	6	4
	1005	6	4
	1006	4	1
	1006	6	1
	1007	1	1
	1007	2	1
	1008	3	1
	1008	6	1
	1009	4	1
	1009	7	1
	1010	6	1
	1010	6	1

	Product ID	Product Description	Product Finish	Standard Price	Product Line Id
1	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 ORDER_LINE_T: ORDER_ID = 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 ORDER_LINE_T: ORDER_ID = 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 303.

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



Another Subquery Example

- Show all products whose price is higher than the average

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 PRODUCT_DESCRIPTION, STANDARD_PRICE, AVGPRICE
FROM
  (SELECT AVG(STANDARD_PRICE) AVGPRICE FROM PRODUCT_T),
  PRODUCT_T
WHERE STANDARD_PRICE > AVG_PRICE;
```

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



SQL Join Operations

- The SQL join operations merge rows from two tables and returns the rows that:
 1. Have common values in common columns (natural join) or,
 2. Meet a given join condition (equality or inequality) or,
 3. Have common values in common columns or have no matching values (outer join).
- We've already examined the basic form of an SQL join which occurs when two tables are listed in the FROM clause and the WHERE clause specifies the join condition.
- An example of this basic form of the join is shown on the next page.



SQL Join Operations (cont.)

```
SELECT P_CODE, P_DESCRIPT, P_PRICE, V_NAME  
FROM PRODUCT, VENDOR  
WHERE PRODUCT.V_CODE = VENDOR.V_CODE;
```

- The FROM clause indicates which tables are to be joined. If three or more tables are specified, the join operation takes place two tables at a time, starting from left to right.
- The join condition is specified in the WHERE clause. In the example, a natural join is effected on the attribute V_CODE.
- The SQL join syntax shown above is sometimes referred to as an “old-style” join.
- The tables on pages 55 and 56, summarize the SQL join operations.



SQL Cross Join Operation

- A **cross join** in SQL is equivalent to a Cartesian product in standard relational algebra. The cross join syntax is:

```
SELECT column-list  
FROM table1, table2;
```

old style syntax

```
SELECT column-list  
FROM table1 CROSS JOIN table2;
```

new style syntax



SQL Natural Join Operation

- The **natural join** syntax is:

```
SELECT column-list
```

```
FROM table1 NATURAL JOIN table2;
```

new style syntax

- The natural join will perform the following tasks:
 - Determine the common attribute(s) by looking for attributes with identical names and compatible data types.
 - Select only the rows with common values in the common attribute(s).
 - If there are no common attributes, return the cross join of the two tables.



SQL Natural Join Operation (cont.)

- The syntax for the old-style natural join is:

```
SELECT column-list
FROM table1, table2
WHERE table1.C1 = table2.C2;
```

old style syntax

- One important difference between the natural join and the “old-style” syntax is that the natural join does not require the use of a table qualifier for the common attributes. The two SELECT statements shown on the next page are equivalent.



SQL Natural Join Operation (cont.)

```
SELECT CUS_NUM, CUS_LNAME,  
       INV_NUMBER, INV_DATE  
FROM   CUSTOMER, INVOICE  
WHERE  CUSTOMER.CUS_NUM = INVOICE.CUS_NUM;
```

old style
syntax

```
SELECT CUS_NUM, CUS_LNAME,  
       INV_NUMBER, INV_DATE  
FROM   CUSTOMER NATURAL JOIN INVOICE;
```

old style
syntax



Join With Using Clause

- A second way to express a join is through the USING keyword. This query will return only the rows with matching values in the column indicated in the USING clause. The column listed in the USING clause must appear in both tables.
- The syntax is:

```
SELECT column-list  
FROM table1 JOIN table2 USING (common-column);
```



Join With Using Clause (cont.)

- An example:

```
SELECT INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS,  
       LINE_PRICE  
FROM INVOICE JOIN LINE USING (INV_NUMBER)  
       JOIN PRODUCT USING (P_CODE);
```

- As was the case with the natural join command, the JOIN USING does not required the use of qualified names (qualified table names). In fact, Oracle 11g will return an error if you specify the table name in the USING clause.



Join On Clause

- Both the NATURAL JOIN and the JOIN USING commands use common attribute names in joining tables.
- Another way to express a join when the tables have no common attribute names is to use the JOIN ON operand. This query will return only the rows that meet the indicated condition. The join condition will typically include an equality comparison expression of two columns. The columns may or may not share the same name, but must obviously have comparable data types.
- The syntax is:

```
SELECT column-list
```

```
FROM table1 JOIN table2 ON join-condition;
```



Join On Clause (cont.)

- An example:

```
SELECT INVOICE.INV_NUMBER, P_CODE, P_DESCRIPT, LINE_UNITS, LINE_PRICE  
FROM INVOICE JOIN LINE ON INVOICE.INV_NUMBER = LINE.INV_NUMBER  
JOIN PRODUCT ON LINE.P_CODE = PRODUCT.P_CODE;
```

- Notice in the example query, that unlike the NATURAL JOIN and the JOIN USING operation, the JOIN ON clause requires the use of table qualifiers for the common attributes. If you do not specify the table qualifier you will get a “column ambiguously defined” error message.
- Keep in mind that the JOIN ON syntax allows you to perform a join even when the tables do not share a common attribute name.



Join On Clause (cont.)

- For example, to generate a list of all employees with the manager's name you can use the recursive query shown below which utilizes the JOIN ON clause.

```
SELECT E.EMP_MGR, M.EMP_LNAME, E.EMP_NUM, E.EMP_LNAME  
FROM EMP E JOIN EMP M ON E.EMP_MGR = M.EMP_NUM  
ORDER BY E.EMP_MGR;
```



Outer Joins

- We saw the forms for the LEFT OUTER JOIN and the RIGHT OUTER JOIN in the previous set of notes.
- There is also a FULL OUTER JOIN operation in SQL. A full outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column(s)), but also all the rows with unmatched values in either side table.
- The syntax of a full outer join is:

```
SELECT column-list  
FROM table1 FULL [OUTER] JOIN table2 ON join-condition;
```



Outer Joins (cont.)

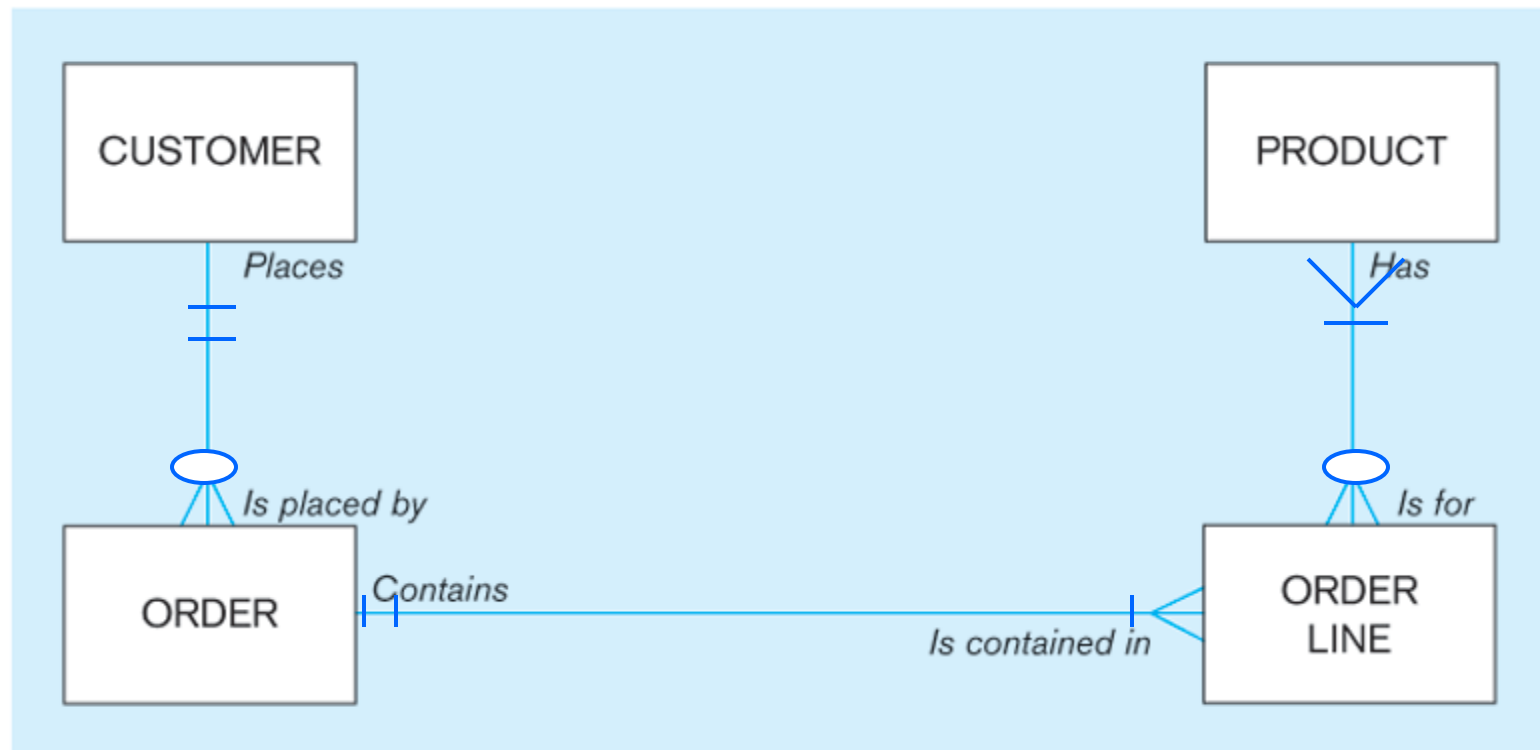
- The following example will list the product code, vendor code, and vendor name for all products and include all the product rows (products without matching vendors) and also all vendor rows (vendors without matching products):

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



Outer Joins (cont.)

- The following example will hopefully illustrate how outer joins work. Shown below is the ERD for the database:



Outer Joins (cont.)

- Suppose that we want to run the query: **For each customer, what is the customer's id, name and order number?**
- Notice that the ERD states that a customer may not place any orders, so when joining the customer table with the order table those customers without an order will not participate in the join if a natural join is used. However, a left outer join will include those customers who have not placed an order.
- Look at the instance data on the next page.



Outer Joins (cont.)

Order_T Table

	Order_ID ▾	Order_Date ▾	Customer_ID ▾	Add New Field
+	1001	10/21/2008	1	
+	1002	10/21/2008	8	
+	1003	10/22/2008	15	
+	1004	10/22/2008	5	
+	1005	10/24/2008	3	
+	1006	10/24/2008	2	
+	1007	10/27/2008	11	
+	1008	10/30/2008	12	
+	1009	11/5/2008	4	
+	1010	11/5/2008	1	
*	0		0	

Part of the Customer_T Table

	Customer_ID ▾	Customer_Name ▾	Customer_Address ▾	City
+	1	Contemporary Casuals	1355 S Hines Blvd	Gainesvi
+	2	Value Furniture	15145 S.W. 17th St.	Plano
+	3	Home Furnishings	1900 Allard Ave.	Albany
+	4	Eastern Furniture	1925 Beltline Rd.	Carteret
+	5	Impressions	5585 Westcott Ct.	Sacrame
+	6	Furniture Gallery	325 Flatiron Dr.	Boulder
+	7	Period Furniture	394 Rainbow Dr.	Seattle
+	8	California Classics	816 Peach Rd.	Santa Cl
+	9	M and H Casual Furniture	3709 First Street	Clearwat
+	10	Seminole Interiors	2400 Rocky Point Dr.	Seminol
+	11	American Euro Lifestyles	2424 Missouri Ave N.	Prospect
+	12	Battle Creek Furniture	345 Capitol Ave. SW	Battle Cr
+	13	Heritage Furnishings	66789 College Ave.	Carlisle
+	14	Kaneohe Homes	112 Kiowai St.	Kaneohe
+	15	Mountain Scenes	4132 Main Street	Ogden

Note that customer #'s 6, 7, 9, 10, 13, and 14 have not placed an order



Outer Joins (cont.)

The SQL Query

Query2

```
SELECT Customer_T.customer_id, customer_name, order_id  
FROM Customer_T, Order_T  
WHERE Customer_T.customer_id = Order_T.customer_id;
```

Result set

Customer_ID	Customer_Name	order_id
1	Contemporary Casuals	1001
1	Contemporary Casuals	1010
2	Value Furniture	1006
3	Home Furnishings	1005
4	Eastern Furniture	1009
5	Impressions	1004
8	California Classics	1002
11	American Euro Lifestyles	1007
12	Battle Creek Furniture	1008
15	Mountain Scenes	1003

Note that customer #'s 6, 7, 9, 10, 13, and 14 are not in the result set using a natural join



Outer Joins (cont.)

The SQL Query

Query1

```
SELECT Customer_T.customer_id, customer_name, order_id  
FROM Customer_T LEFT OUTER JOIN Order_T  
ON Customer_T.customer_id = Order_T.customer_id;
```

Result set

Query1

Customer_ID	Customer_Name	order_id
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 and 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

Note that customer #'s 6, 7, 9, 10, 13, and 14 are in the result set using a left outer join



Summary of SQL JOIN Operations

Join Classification	Join Type	SQL Syntax Example	Description
Cross	CROSS JOIN	SELECT * FROM T1, T2;	Old style. Returns the Cartesian product of T1 and T2
		SELECT * FROM T1 CROSS JOIN T2;	New style. 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.



Summary of SQL JOIN Operations (cont.)

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.



Subqueries and Correlated Queries

- The use of joins allows a RDBMS go get information from two or more tables. The data from the tables is processed simultaneously.
- It is often necessary to process data based on other processed data. Suppose, for example, that you want to generate a list of vendors who provide products. (Recall that not all vendors in the VENDOR table have provided products – some of them are only potential vendors.)
- The following query will accomplish our task:

```
SELECT V_CODE, V_NAME  
FROM VENDOR  
WHERE V_CODE NOT IN (SELECT V_CODE FROM PRODUCT);
```



Subqueries and Correlated Queries (cont.)

- A subquery is a query (SELECT statement) inside a query.
- A subquery is normally expressed inside parentheses.
- The first query in the SQL statement is known as the outer query.
- The second query in the SQL statement is known as the inner query.
- The inner query is executed first.
- The output of the inner query is used as the input for the outer query.
- The entire SQL statement is sometimes referred to as a nested query.



Subqueries and Correlated Queries (cont.)

- A subquery can return:
 1. One single value (one column and one row). This subquery can be used anywhere a single value is expected. For example, in the right side of a comparison expression.
 2. A list of values (one column and multiple rows). This type of subquery can be used anywhere a list of values is expected. For example, when using the IN clause.
 3. A virtual table (multi-column, multi-row set of values). This type of subquery can be used anywhere a table is expected. For example, in the FROM clause.
 4. No value at all, i.e., NULL. In such cases, the output of the outer query may result in an error or null empty set, depending on where the subquery is used (in a comparison, an expression, or a table set).



Correlated Queries

- A correlated query (really a subquery) is a subquery that contains a reference to a table that also appears in the outer query.
- A correlated query has the following basic form:

```
SELECT * FROM table1 WHERE col1 = ANY  
  (SELECT col1 FROM table2  
   WHERE table2.col2 = table1.col1);
```

- Notice that the subquery contains a reference to a column of `table1`, even though the subquery's **FROM** clause doesn't mention `table1`. Thus, query execution requires a look outside the subquery, and finds the table reference in the outer query.



WHERE Subqueries

- The most common type of subquery uses an inner SELECT subquery on the right hand side of a WHERE comparison expression.
- For example, to find all products with a price greater than or equal to the average product price, the following query would be needed:

```
SELECT P_CODE, P_PRICE  
FROM PRODUCT  
WHERE P_PRICE >= (SELECT AVG(P_PRICE)  
                  FROM PRODUCT);
```



WHERE Subqueries (cont.)

- Subqueries can also be used in combination with joins.
- The query below lists all the customers that ordered the product “Claw hammer”.

```
SELECT DISTINCT CUS_CODE, CUS_LNAME, CUYS_FNAME  
FROM CUSTOMER JOIN INVOICE USING (CUS_CODE)  
                JOIN LINE USING (INV_NUMBER)  
                JOIN PRODUCT USING (P_CODE)  
WHERE P_CODE = (SELECT P_CODE  
                FROM PRODUCT  
                WHERE P_DESCRIPT = “Claw hammer”);
```



WHERE Subqueries (cont.)

- Notice that the previous query could have been written as:

```
SELECT DISTINCT CUS_CODE, CUS_LNAME, CUYS_FNAME  
FROM CUSTOMER JOIN INVOICE USING (CUS_CODE)  
              JOIN LINE USING (INV_NUMBER)  
              JOIN PRODUCT USING (P_CODE)  
WHERE P_DESCRIPT = 'Claw hammer');
```

- However, what would happen if two or more product descriptions contain the string “Claw hammer”?
 - You would get an error message because only a single value is expected on the right hand side of this expression.



IN Subqueries

- To handle the problem we just saw, the IN operand must be used.
- The query below lists all the customers that ordered any kind of hammer or saw.

```
SELECT DISTINCT CUS_CODE, CUS_LNAME, CUYS_FNAME
FROM CUSTOMER JOIN INVOICE USING (CUS_CODE)
              JOIN LINE USING (INV_NUMBER)
              JOIN PRODUCT USING (P_CODE)
WHERE P_CODE IN (SELECT P_CODE
                  FROM PRODUCT
                  WHERE P_DESCRIPT LIKE '%hammer%'
                  OR P_DESCRIPT LIKE '%saw%');
```



HAVING Subqueries

- It is also possible to use subqueries with a HAVING clause.
- Recall that the HAVING clause is used to restrict the output of a GROUP BY query by applying a conditional criteria to the grouped rows.
- For example, the following query will list all products with the total quantity sold greater than the average quantity sold.

```
SELECT DISTINCT P_CODE, SUM(LINE_UNITS)
FROM LINE
GROUP BY P_CODE
HAVING SUM(LINE_UNITS) > (SELECT AVG(LINE_UNITS)
                           FROM LINE);
```



Multi-row Subquery Operators: ANY and ALL

- The IN subquery uses an equality operator; that is, it only selects those rows that match at least one of the values in the list. What happens if you need to do an inequality comparison of one value to a list of values?
- For example, suppose you want to know what products have a product cost that is greater than all individual product costs for products provided by vendors from Florida.

```
SELECT P_CODE, P_ONHAND*P_PRICE
FROM PRODUCT
WHERE P_ONHAND*P_PRICE > ALL (SELECT P_ONHAND*P_PRICE
                                FROM PRODUCT
                                WHERE V_CODE IN (SELECT V_CODE
                                                    FROM VENDOR
                                                    WHERE V_STATE= 'FL'));
```



FROM Subqueries

- In all of the cases of subqueries we've seen so far, the subquery was part of a conditional expression and it always appeared on the right hand side of an expression. This is the case for WHERE, HAVING, and IN subqueries as well as for the ANY and ALL operators.
- Recall that the FROM clause specifies the table(s) from which the data will be drawn. Because the output of a SELECT statement is another table (or more precisely, a “virtual table”), you could use a SELECT subquery in the FROM clause.
- For example, suppose that you want to know all customers who have purchased products 13-Q2/P2 and 23109-HB. Since all product purchases are stored in the LINE table, it is easy to find out who purchased any given product just by searching the P_CODE attribute in the LINE table. However, in this case, you want to know all customers who purchased both, not just one.
- The query on the next page accomplishes this task.



FROM Subqueries (cont.)

```
SELECT DISTINCT CUSTOMER.CUS_CODE      , CUSTOMER.LNAME
FROM CUSTOMER, (SELECT INVOICE.CUS_CODE
                 FROM INVOICE NATURAL JOIN LINE
                 WHERE P_CODE = '13-Q2/P2') CP1,
              (SELECT INVOICE.CUS_CODE
               FROM INVOICE NATURAL JOIN LINE
               WHERE P_CODE = '23109-HB') CP2
WHERE CUSTOMER.CUS_CODE = CP1.CUS_CODE
      AND CP1.CUS_CODE = CP2.CUS_CODE;
```



Conditional Expressions Using Case Syntax

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

CASE conditional syntax

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



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



An SQL Transaction sequence (in pseudocode)

```
BEGIN transaction
```

```
INSERT Order_ID, Order_date, Customer_ID into Order_t;
```

```
INSERT Order_ID, Product_ID, Quantity into Order_line_t;
```

```
INSERT Order_ID, Product_ID, Quantity into Order_line_t;
```

```
INSERT Order_ID, Product_ID, Quantity into Order_line_t;
```

```
END transaction
```

Valid information inserted.
COMMIT work

All changes to data
are made permanent.

Invalid Product_ID entered

Transaction will be ABORTED.
ROLLBACK all changes made to Order_t

All changes made to Order_t
and Order_line_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
- 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
 - SYSCOLUMNS – table and column definitions
 - SYSDEPENDS – object dependencies based on foreign keys
 - SYSPERMISSIONS – access permissions granted to users



SQL:2007

Enhancements/Extensions

- User-defined data types (UDT)
 - Subclasses of standard types or an object type
- Analytical functions (for OLAP)
- 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 propriety version called PL/SQL, and Microsoft SQL Server has Transact/SQL



Routines and Triggers

- **Routines**

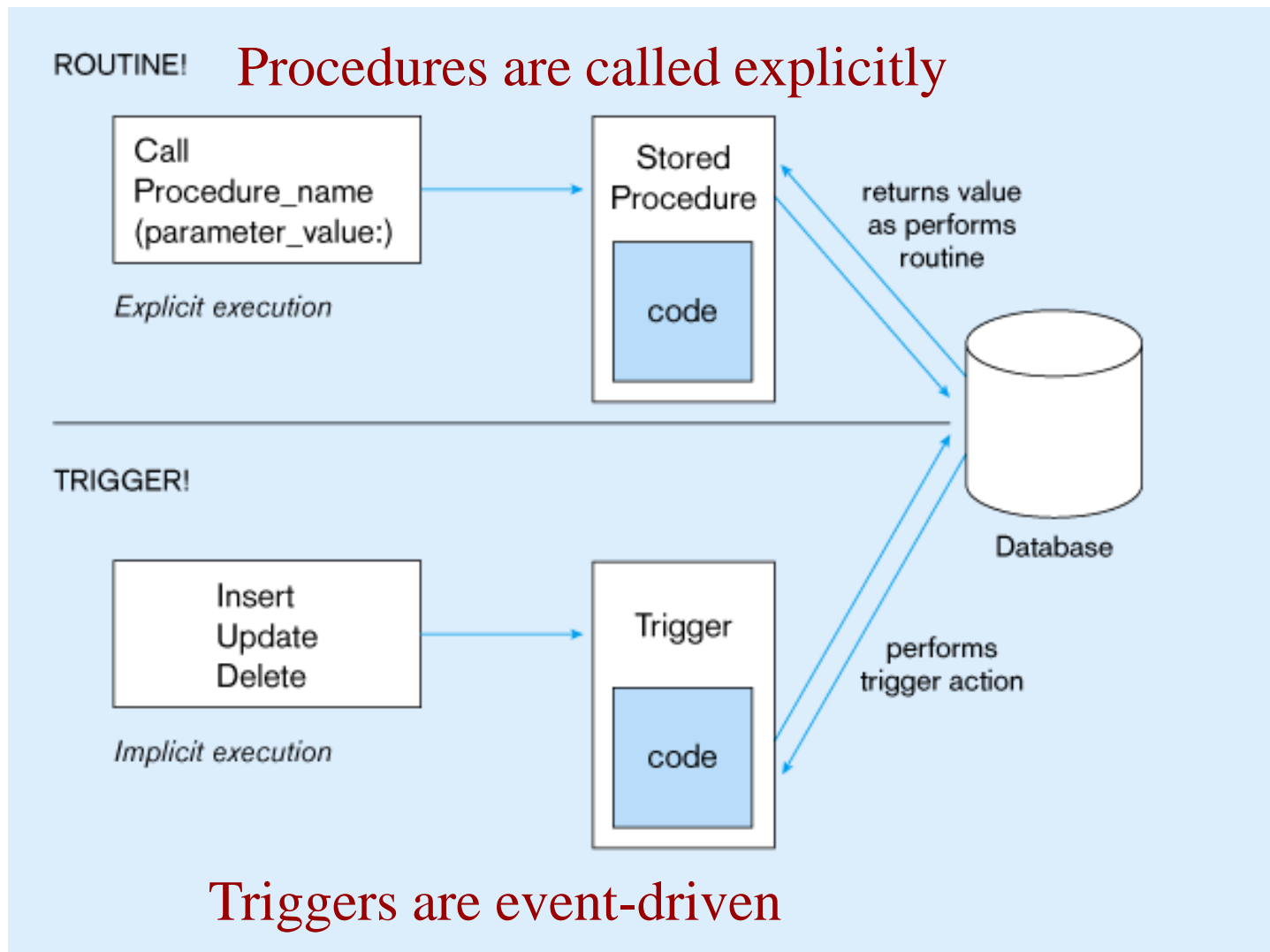
- Program modules that execute on demand
- **Functions** – routines that return values and take input parameters
- **Procedures** – routines that do not return values and can take input or output parameters

- **Triggers**

- Routines that execute in response to a database event (INSERT, UPDATE, or DELETE)



Triggers contrasted with stored procedures



Oracle PL/SQL trigger syntax

```
CREATE [OR REPLACE] TRIGGER trigger_name
  {BEFORE AFTER} {INSERT | DELETE | UPDATE} ON table_name
  [FOR EACH ROW [WHEN (trigger_condition)]]
  trigger_body_here;
```

SQL:2007 Create routine syntax

```
{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]
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
[DYNAMIC RESULT SETS unsigned_integer] /* for procedures only */
[STATIC DISPATCH] /* for functions only */
routine_body
```



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

