CHAPTER 5:
INTRODUCTION TO SQL

Shakeel Ahmad

Modern Database Management

13th Edition

Jeff Hoffer, Ramesh Venkataraman, Heikki Topi

If some figures have "6-#", please treat them as "5-#"

#70: "SEVEN SINS" // #71: Flowchart of SELECT //
#73-81: The new SELECT-clause examples

OBJECTIVES

Focus: Syntax

- Define terms
- Interpret history and role of SQL
- Define a database using SQL data definition language
- Write single table queries using SQL
  - No subquery, yet
- Establish referential integrity using SQL

SQL OVERVIEW

- Structured Query Language – often pronounced "Sequel"
- The standard for relational database management systems (RDBMS)
- RDBMS: A database management system that manages data as a collection of tables in which all relationships are represented by common values in related tables

HISTORY OF SQL

- 1970–E. F. Codd develops relational database concept
- 1974-1979–System R with Sequel (later SQL) created at IBM Research Lab
- 1979–Oracle markets first relational DB with SQL
- 1981 – SQL/DS first available RDBMS system on DOS/VSE
- Others followed: INGRES (1981), IDM (1982), DG/SGL (1984), Sybase (1986)
- 1986–ANSI SQL standard released
- 1989, 1992, 1999, 2003, 2006, 2008, 2011–Major ANSI standard updates
- Current–SQL is supported by most major database vendors

PURPOSE OF SQL STANDARD

- Specify syntax/semantics for data definition and manipulation
- Define data structures and basic operations
- Enable portability of database definition and application modules
- Specify minimal (level 1) and complete (level 2) standards
- Allow for later growth/enhancement to standard (referential integrity, transaction management, user-defined functions, extended join operations, national character sets)

SQL ENVIRONMENT

Course priority:
2, 1, 3

- 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)

1. Data Definition Language (DDL)
   1. Commands that define a database, including creating, altering, and dropping tables and establishing constraints
2. Data Manipulation Language (DML)
   1. Commands that maintain and query a database
3. Data Control Language (DCL)
   1. Commands that control a database, including administering privileges and committing data

FIG 5-2 QUERY SYNTAX P.212

Display order of columns specified here

```
1  SELECT [ALL/DISTINCT] column_list
2  FROM table_list
3  [WHERE conditional expression]
4  [GROUP BY group_by_column_list]
5  [HAVING conditional expression]
6  [ORDER BY order_by_column_list]
```

Sequence of clauses!

No comma between clauses; semi-colon at the end

- SELECT Major, AVERAGE(GPA)
- FROM STUDENT
- WHERE ExpGraduateYr = “2018”
- GROUP BY Major
- HAVING AVERAGE(GPA) >=3.0
- ORDER BY Major;

Line breaks NOT needed!

FIG 6-2 QUERY SYNTAX

Clause	Role	Note: Necessity; Roles
SELECT	Fields to display	Must have; can be queries (Subquery)
FROM	Table(s) to get fields	Must have: data source; can be Subquery
[WHERE	Condi to get rows	1-Get rows; 2-join table; 3-pass parameters
[GROUP BY	Field	Can work w WHERE&HAVING but diffrnt
[HAVING	Condi for groups	MUST have Grp By to use; AFTER Grp By
[ORDER BY	Sorting	Last clause; primary & secondary sorts

- SELECT Major, AVG(GPA) , AVG(YrGrad-YrAdm) AS Duration, AdvsrName
- FROM STUDENT, FACULTY AS FAC
- WHERE Major="ACCT" AND FAC.FacID = STUDENT.FacID
- GROUP BY Major, AdvsrName
- HAVING AVG(GPA) >=3.0
- ORDER BY AVG(GPA) DESC, AdvsrName;

More than 85% of usage of SELECT-clause is here

SQL DATA TYPES

TABLE 5-2 Sample ANSI SQL Data Types

String	CHARACTER(n) or CHAR(n)	Stores string values containing any characters in a character set. CHAR is defined to be a fixed length, for example, CHAR(2).
	CHARACTER VARYING(n) or CHAR VARYING(n)	Stores string values containing any characters in a character set using space only for the actual length of the string. In Oracle, VARCHAR2, for example, VARCHAR2(30).
Binary	BINARY LARGE OBJECT (BLOB)	Stores binary string values in hexadecimal format. BLOB is defined to be a variable length. (Oracle also has CLOB and NCLOB as well as BFILE for storing unstructured data outside the database.)
Number	NUMERIC(p,s) or DECIMAL (p,s)	Stores exact numbers with a defined precision and scale. In Oracle, NUMBER (precision, scale), for example, NUMBER (12,2).
	INTEGER or INT	Stores exact numbers with a predefined precision and scale of zero.
Temporal	TIMESTAMP TIMESTAMP WITH LOCAL TIME ZONE	Stores a moment an event occurs, using a definable fraction-of-a-second precision. Value adjusted to the user's session time zone (available fully in DB2 and Oracle).
Boolean	BOOLEAN	Stores truth values: TRUE, FALSE, or UNKNOWN.

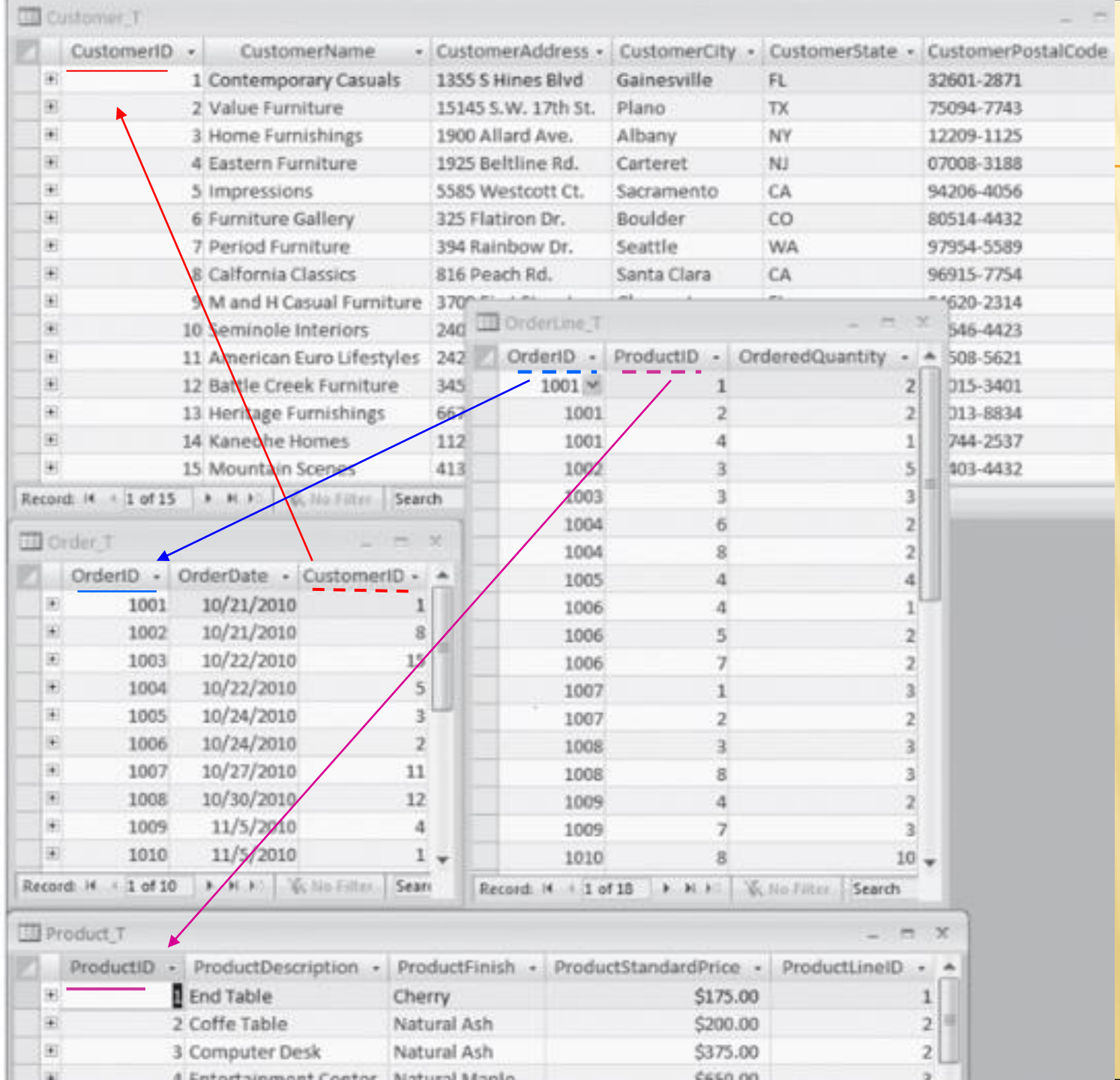
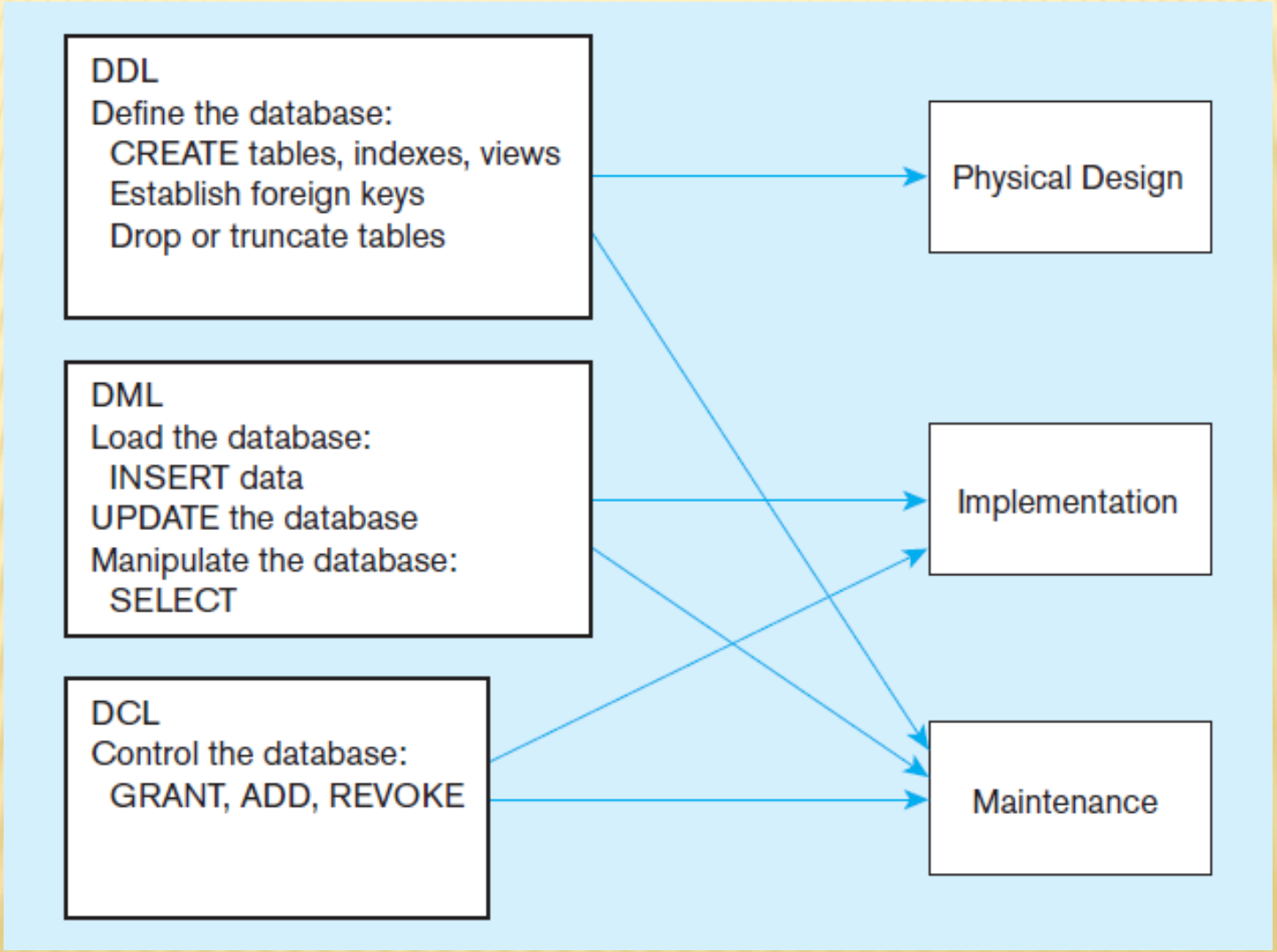


FIGURE 6-3: SAMPLE PVFC DATA

Figure 5-4 P.215
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 new table and its columns
  - CREATE VIEW–defines a logical table from one or more tables or views
- Other CREATE statements: CHARACTER SET, COLLATION, TRANSLATION, ASSERTION, DOMAIN

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

DEFINING A DATABASE IN SQL

Because most systems allocate storage space to contain base tables, views, constraints, indexes, and other database objects when a database is created, you may not be allowed to create a database. Because of this, the privilege of creating databases may be reserved for the database administrator, and you may need to ask to have a database created. Students at a university may be assigned an account that gives access to an existing database, or they may be allowed to create their own database in a limited amount of allocated storage space (sometimes called *perm space* or *table space*). In any case, the basic syntax for creating a database is:

```
CREATE SCHEMA schema_name AUTHORIZATION owner_userid
```

The database will be owned by the authorized user, although it is possible for other specified users to work with the database or even to transfer ownership of the database. Physical storage of the database is dependent on both the hardware and the software environment and is usually the concern of the system administrator. The amount of control over physical storage that a database administrator is able to exert depends on the RDBMS being used. Little control is possible when using Microsoft Access, but Microsoft SQL Server 2008 and later versions allow for more control of the physical database. A database administrator may exert considerable control over the placement of data, control files, index files, schema ownership, and so forth, thus improving the ability to tune the database to perform more efficiently and to create a secure database environment. You will learn more about these topics in Chapter 8.

Figure 6-5 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]
```

THE FOLLOWING SLIDES CREATE TABLES FOR THIS ENTERPRISE DATA MODEL

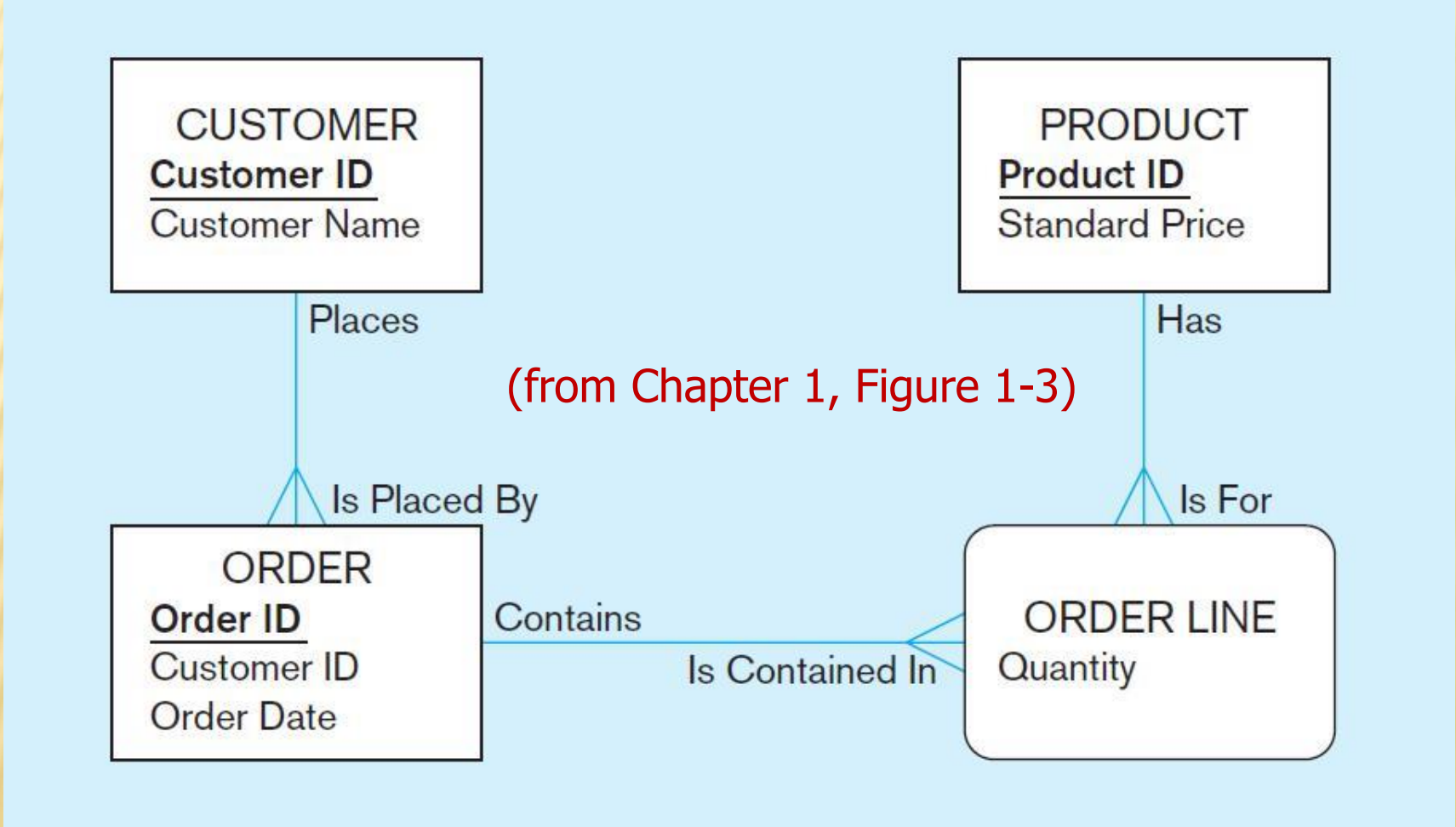


Figure 6-6 SQL database definition commands for PVF Company (Oracle 12c)

```
CREATE TABLE Customer_T
                (CustomerID                    NUMBER(11,0)        NOT NULL,
                 CustomerName                  VARCHAR2(25)        NOT NULL,
                 CustomerAddress               VARCHAR2(30),
                 CustomerCity                  VARCHAR2(20),
                 CustomerState                 CHAR(2),
                 CustomerPostalCode            VARCHAR2(9),
CONSTRAINT Customer_PK PRIMARY KEY (CustomerID));
```

```
CREATE TABLE Order_T
                (OrderID                       NUMBER(11,0)        NOT NULL,
                 OrderDate                     DATE DEFAULT SYSDATE,
                 CustomerID                    NUMBER(11,0),
CONSTRAINT Order_PK PRIMARY KEY (OrderID),
CONSTRAINT Order_FK FOREIGN KEY (CustomerID) REFERENCES Customer_T(CustomerID));
```

```
CREATE TABLE Product_T
                (ProductID                     NUMBER(11,0)        NOT NULL,
                 ProductDescription            VARCHAR2(50),
                 ProductFinish                 VARCHAR2(20)
                        CHECK (ProductFinish IN ('Cherry', 'Natural Ash', 'White Ash'
                                                 'Red Oak', 'Natural Oak', 'Walnut')),
                 ProductStandardPrice          DECIMAL(6,2),
                 ProductLineID                 INTEGER,
CONSTRAINT Product_PK PRIMARY KEY (ProductID));
```

```
CREATE TABLE OrderLine_T
                (OrderID                       NUMBER(11,0)        NOT NULL,
                 ProductID                     INTEGER             NOT NULL,
                 OrderedQuantity               NUMBER(11,0),
CONSTRAINT OrderLine_PK PRIMARY KEY (OrderID, ProductID),
CONSTRAINT OrderLine_FK1 FOREIGN KEY (OrderID) REFERENCES Order_T(OrderID),
CONSTRAINT OrderLine_FK2 FOREIGN KEY (ProductID) REFERENCES Product_T(ProductID));
```

Overall table definitions

Interpreta-tions next 5 slides

Defining attributes and their data types

```
CREATE TABLE Product_T
    (ProductID                  NUMBER(11,0)        NOT NULL,
     ProductDescription         VARCHAR2(50),
     ProductFinish              VARCHAR2(20)
                                CHECK (ProductFinish IN ('Cherry', 'Natural Ash', 'White Ash',
                                'Red Oak', 'Natural Oak', 'Walnut')),
     ProductStandardPrice       DECIMAL(6,2),
     ProductLineID              INTEGER,
CONSTRAINT Product_PK PRIMARY KEY (ProductID));
```

Non-nullable specification

Primary keys can never have NULL values

```
CREATE TABLE Product_T
        (ProductID                  NUMBER(11,0)    NOT NULL,
         ProductDescription         VARCHAR2(50),
         ProductFinish              VARCHAR2(20)
                                    CHECK (ProductFinish IN ('Cherry', 'Natural Ash', 'White Ash',
                                           'Red Oak', 'Natural Oak', 'Walnut')),
         ProductStandardPrice       DECIMAL(6,2),
         ProductLineID              INTEGER,
CONSTRAINT Product_PK PRIMARY KEY (ProductID));
```

Identifying primary key

Constraint name **Cnstr Type** **Field**

Non-nullable specifications

```
CREATE TABLE OrderLine_T
        (OrderID                NUMBER(11,0)    NOT NULL,
         ProductID              INTEGER         NOT NULL,
         OrderedQuantity        NUMBER(11,0),
CONSTRAINT OrderLine_PK PRIMARY KEY (OrderID, ProductID),
CONSTRAINT OrderLine_FK1 FOREIGN KEY (OrderID) REFERENCES Order_T(OrderID),
CONSTRAINT OrderLine_FK2 FOREIGN KEY (ProductID) REFERENCES Product_T(ProductID));
```

Composite key

Some primary keys are composite–composed of multiple attributes

Controlling the values in attributes

```
CREATE TABLE Order_T
                (OrderID                    NUMBER(11,0)        NOT NULL,
                 OrderDate                  DATE DEFAULT SYSDATE,
                 CustomerID                 NUMBER(11,0),
CONSTRAINT Order_PK PRIMARY KEY (OrderID),
CONSTRAINT Order_FK FOREIGN KEY (CustomerID) REFERENCES Customer_T(CustomerID));

CREATE TABLE Product_T
                (ProductID                  NUMBER(11,0)        NOT NULL,
                 ProductDescription         VARCHAR2(50),
                 ProductFinish              VARCHAR2(20)
                    CHECK (ProductFinish IN ('Cherry', 'Natural Ash', 'White Ash',
                                             'Red Oak', 'Natural Oak', 'Walnut')),
                 ProductStandardPrice       DECIMAL(6,2),
                 ProductLineID              INTEGER,
CONSTRAINT Product_PK PRIMARY KEY (ProductID));
```

Default value

Domain constraint validation rule

Identifying foreign keys and establishing relationships

```
CREATE TABLE Customer_T
                (CustomerID                NUMBER(11,0)        NOT NULL,
                 CustomerName              VARCHAR2(25)        NOT NULL,
                 CustomerAddress           VARCHAR2(30),
                 CustomerCity              VARCHAR2(20),
                 CustomerState             CHAR(2),
                 CustomerPostalCode        VARCHAR2(9),
CONSTRAINT Customer_PK PRIMARY KEY (CustomerID));

CREATE TABLE Order_T
                (OrderID                   NUMBER(11,0)        NOT NULL,
                 OrderDate                 DATE DEFAULT SYSDATE,
                 CustomerID                NUMBER(11,0),
CONSTRAINT Order_PK PRIMARY KEY (OrderID),
CONSTRAINT Order_FK FOREIGN KEY (CustomerID) REFERENCES Customer_T(CustomerID));
```

Primary key of parent table

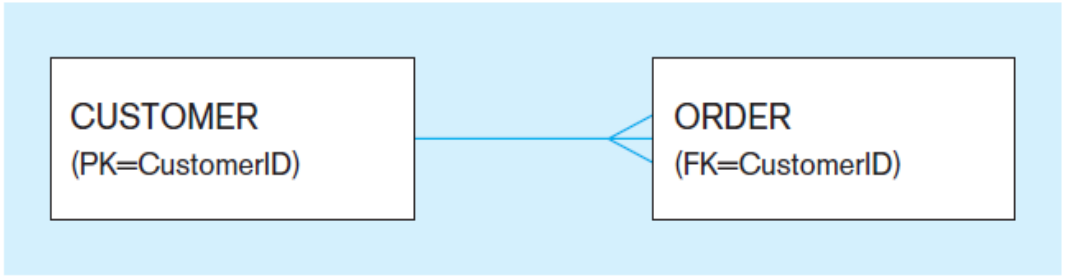
Foreign key of dependent table

Constraint name **Cnstrnt Type** **Field** **REFERENCES** **Tbl(Key)**

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

Figure 6-7 Ensuring data integrity through updates



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

```
CREATE TABLE CustomerT
                (CustomerID          INTEGER DEFAULT '999'     NOT NULL,
                 CustomerName        VARCHAR(40)               NOT NULL,
                 . . .
CONSTRAINT Customer_PK PRIMARY KEY (CustomerID),
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

The CREATE TABLE statement includes a clause to specify **how updates and deletes are processed** when there are dependent tables.

If a DELETE request comes for a record with dependent records in another table, The DBMS could **restrict** the delete, which means to disallow it. Or, it could **cascade** the delete, so that dependent records with matching foreign keys will also be deleted.

Or it could **set null**, which means that deleting the primary key record will result in setting all corresponding foreign keys to be set to null (this would imply an optional one cardinality in the relationship).

CHANGING TABLES - ALTER TABLE-CLAUSE

- ALTER TABLE statement allows you to change column specifications:

```
ALTER TABLE table_name alter_table_action;
```

- Table Actions:

```
ADD [COLUMN] column_definition
ALTER [COLUMN] column_name SET DEFAULT default-value
ALTER [COLUMN] column_name DROP DEFAULT
DROP [COLUMN] column_name [RESTRICT] [CASCADE]
ADD table_constraint
```

- Example (adding a new column with a default value):

The ALTER command will be done after tables have already been created.
For example, if you have an existing database, even one with actual data in it, you can modify tables by adding or changing columns, removing columns adding constraints, etc.

If data in the tables violate the constraints, you will be prevented from setting these constraints until after changing the data.

So, whereas CREATE TABLE is mostly a process that takes place during implementation, ALTER TABLE often takes place during maintenance.

```
ALTER TABLE CUSTOMER_T
ADD COLUMN CustomerType VARCHAR2 (10) DEFAULT "Commercial";
```

ADD COLUMN Name **Type** Value **Default** Value

REMOVING TABLES

- DROP TABLE statement allows you to remove tables from your schema:
  - DROP TABLE CUSTOMER_T

Tables will not be dropped if there are other tables that depend on them. This means that if any table has a foreign key to the table being dropped, the drop will fail. Therefore, it makes a difference which order you drop the tables in.

INSERT STATEMENT

Two methods: full-row & partial-row insertions

- Adds one or more **rows** to a table
- Inserting into a table

INSERT INTO *Table* VALUES (*value-list*)

```
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 (ProductID,
ProductDescription, ProductFinish, ProductStandardPrice)
    VALUES (1, 'End Table', 'Cherry', 175, 8);
```

(field-list)
VALUES
(value-list)

- Inserting from another table

```
INSERT INTO CaCustomer_T
SELECT * FROM Customer_T
    WHERE CustomerState = 'CA';
```

A SUBSET from another table

CREATING TABLES WITH IDENTITY COLUMNS

```
CREATE TABLE Customer_T
(CustomerID INTEGER GENERATED ALWAYS AS IDENTITY
     (START WITH 1
     INCREMENT BY 1
     MINVALUE 1
     MAXVALUE 10000
     NO CYCLE),
CustomerName          VARCHAR2(25) NOT NULL,
CustomerAddress       VARCHAR2(30),
CustomerCity          VARCHAR2(20),
CustomerState         CHAR(2),
CustomerPostalCode    VARCHAR2(9),
CONSTRAINT Customer_PK PRIMARY KEY (CustomerID);
```

Introduced with SQL:2008

Inserting into a table does not require explicit customer ID entry or field list

```
INSERT INTO CUSTOMER_T VALUES ( 'Contemporary Casuals',
'1355 S. Himes Blvd.', 'Gainesville', 'FL', 32601);
```

DELETE STATEMENT

- Removes rows from a table
- Delete certain row(s)
  - DELETE FROM CUSTOMER_T
    WHERE CUSTOMERSTATE = 'HI';
  - DELETE FROM CUSTOMER_T
    WHERE CUSTOMER_ID = ‘C123';
- Delete all rows
  - DELETE FROM CUSTOMER_T;
  - *Careful!!!*

UPDATE STATEMENT

- Modifies data in existing rows

UPDATE *Table-name*
SET *Attribute = Value*
WHERE *Criteria-to-apply-the-update*

For this UPDATE, we know that it will affect only one record in the table. How do we know this?

Answer: Because ProductID is the primary key, which must be unique. So, there can be only one product with ProductID = 7.

However, many times updates and deletes affect many records. For example,

DELETE FROM CUSTOMER_T WHERE CUSTOMERSTATE = 'HI'; affects all customers from Hawaii.

```
UPDATE Product_T
SET ProductStandardPrice = 775
    WHERE ProductID = 7;
```

- Note: the WHERE clause may be a subquery (Chap 7)

COMPARISON OF ALTER, INSERT, AND UPDATE

- ALTER TABLE: change the columns [structure] of the table
  - ALTER TABLE CUSTOMER_T ADD *column*…
- INSERT: add records based on the existing table [did not change/alter table]
  - INSERT INTO CUSTOME_T VALUES (…)
- UPDATE: change the values of some **fields** in **SOME** existing records [did not add a record]
  - UPDATE CUSTOMER_T SET *field* = *value* …**WHERE**…

COMPARISON OF ALTER, INSERT, AND UPDATE (CONT)

- Degree of impact:
  - ALTER TABLE – wide (whole table); deep (nature/structure)
  - INSERT – local (add rows; only make a table longer)
  - UPDATE – very minor (only change values of some fields of some rows)

Common confusion

Decrease

MERGE STATEMENT

```
MERGE INTO Product_T AS PROD
USING
(SELECT ProductID, ProductDescription, ProductFinish,
ProductStandardPrice, ProductLineID FROM Purchases_T) AS PURCH
    ON (PROD.ProductID = PURCH.ProductID)
WHEN MATCHED THEN UPDATE
    PROD.ProductStandardPrice = PURCH.ProductStandardPrice
WHEN NOT MATCHED THEN INSERT
    (ProductID, ProductDescription, ProductFinish, ProductStandardPrice,
    ProductLineID)
    VALUES(PURCH.ProductID, PURCH.ProductDescription,
    PURCH.ProductFinish, PURCH.ProductStandardPrice,
      PURCH.ProductLineID);
```

Interpretation?

Makes it easier to update a table…allows combination of Insert and Update in one statement

Useful for updating master tables with new data

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(CUSTOMERNAME)
    - This makes an index for the CUSTOMERNAME field of the CUSTOMER_T table

Data manipulation

QUERY: SELECT STATEMENT

- This is the **CENTER** section of the chapter, and the **CENTER** of this course
- Many slides contain multiple points of concepts/skills – do NOT miss the text, and pay attention to color code

SELECT STATEMENT

- Used for queries on single or multiple tables
- Clauses of the SELECT statement:

Memorizing clause order: 6-39

1. **SELECT**
   1. List the columns (& expressions) to be returned from the query
2. **FROM**
   1. Indicate the table(s) or view(s) from which data will be obtained
3. **WHERE**
   1. Indicate the conditions under which a row will be included in the result
4. **GROUP BY**
   1. Indicate categorization of results (*collapsed* into groups)
5. **HAVING**
   1. Indicate the conditions under which a category (group) will be included
6. **ORDER BY**
   1. Sorts the result according to specified criteria on named fields

"No GROUP BY, no HAVING!!!!"; HAVING always AFTER Grp By

FIG 6-2 QUERY SYNTAX

Clause	Role	Note: Necessity; Roles
SELECT	Fields to display	Must have; can be queries (Subquery)
FROM	Table(s) to get fields	Must have: data source; can be Subquery
[WHERE	Condi to get rows	1-Get rows; 2-join table; 3-pass parameters
[GROUP BY	Field	Can work w WHERE&HAVING but diffrnt
[HAVING	Condi for groups	MUST have Grp By to use; AFTER Grp By
[ORDER BY	Sorting	Last clause; primary & secondary sorts

- SELECT Major, AVG(GPA) , AVG(YrGrad-YrAdm) AS Duration, AdvsrName
- FROM STUDENT, FACULTY AS FAC —— [Missing table join here]
- WHERE Major="ACCT" AND FAC.FacID = STUDENT.FacID
- GROUP BY Major, AdvsrName
- HAVING AVG(GPA) >=3.0
- ORDER BY AVG(GPA) DESC, AdvsrName;

More than 80% of usage of SELECT-clause is here

HAVING VS WHERE; GROUP BY AND HAVING

- HAVING and WHERE are both condition-clause: states conditions/criteria for selection
  - WHERE states cri for INDIVIDUAL rows
  - HAVING states cri for GROUPS
- Because HAVING states cri for groups,
  - It must be preceded by GROUP BY:
    - No GROUP BY, no HAVING
    - HAVING always comes after GROUP BY
    - "A lower-level bullet w big fonts"

Remember slide 6-38

>1/4 made this mistake

```
SELECT [ALL/DISTINCT] column_list
FROM table_list
[WHERE conditional expression]
[GROUP BY group_by_column_list]
[HAVING conditional expression]
[ORDER BY order_by_column_list]
```

Figure 5-2
General syntax of the SELECT statement used in DML

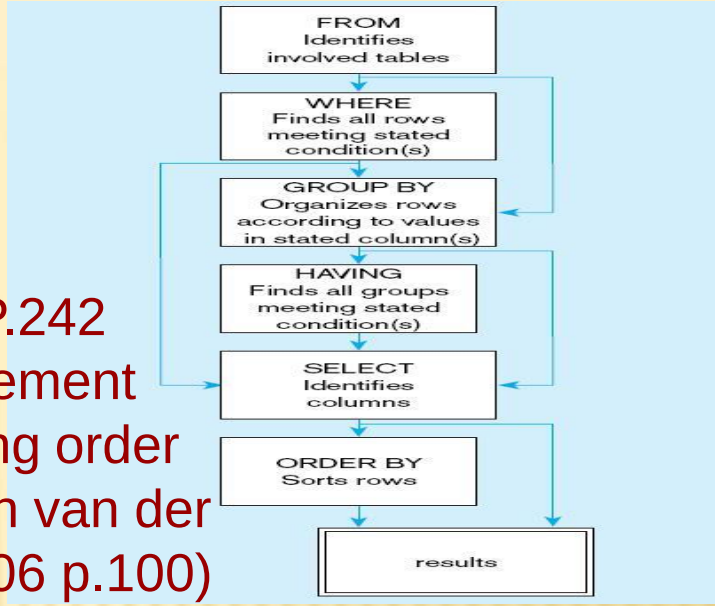


Figure 5-10 P.242
SQL statement processing order (based on van der Lans, 2006 p.100)

1st SELECT
2nd FROM
3rd Row condition
4th Grouping
5th Group condition
6th Finally display order

1st SELECT
2nd FROM tables
3rd Row condition – row first
4th Grouping – *then group*
5th No grping, no group cond
6th Finally **display** order – this does NOT involve/affect logic!

Insert #71 here

SELECT EXAMPLE

- Find products with standard price less than $275

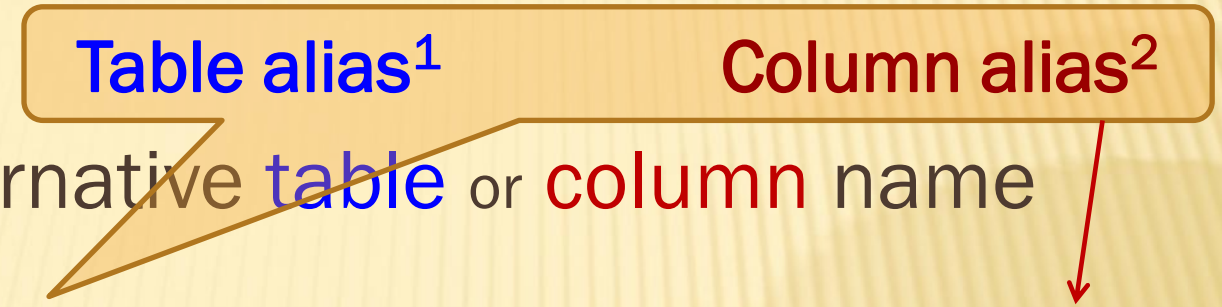
```
SELECT ProductDescription, ProductStandardPrice
   FROM Product_T
      WHERE ProductStandardPrice < 275;
```

P. 226

Every SESLECT statement returns a result table - Can be used as part of another query - subquery

SELECT **DISTINCT**: no duplicate rows
SELECT * : all columns selected

SELECT EXAMPLE USING ALIAS P. 227



- Alias is an alternative table or column name

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

Used in SELECT while defined in FROM

Avoid !**

Note1: Specifying source table

Note2: Column alias is used as display heading ONLY ;
NOT for comparison/calculation, Grp, Order (PP. 227~228)

USING EXPRESSIONS PP. 228~9

- Example:
  - ProductStandardPrice*1.1 AS Plus10Percent
- Which contents does it look like that we learned in Access (in IS 312)?
  - YearInBiz:(NOW()-DateOpened)/365
  - Observe the result on P. 229

SELECT EXAMPLE USING A FUNCTION

More functions: P. 229

- Using the COUNT ***aggregate function*** to find totals

```
SELECT COUNT(*) FROM ORDERLINE_T
    WHERE ORDERID = 1004;
```

Functions that can ONLY be used w numeric columns?

Note: COUNT (*) and COUNT – different

+ COUNT(*) counts num rows
+ COUNT(field) counts rows that have not-null val for the field

- What is average standard price for each [*examine!!*] product in inventory?

```
SELECT AVG (STANDARD_PRICE) AS AVERAGE
FROM PROCUCT_V
GROUP BY Product_ID;
```

Can be w or w/o GROUP BY, but w different meanings

+ Meaning *IF* w/o GROUP BY clause?
+ - w/o GROUP BY, the AVG is performed over the whole table.

USING A FUNCTION (CONT)

```
SELECT COUNT(*)
    FROM OoderLine_T
        WHERE OrderID = 1004;
```

Output:

COUNT(*)
2

Note: With aggregate functions ("set values") you can't have single-valued columns ("row values") included in the SELECT clause

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

PP. 230~1

*** Examine the error message on P. 230

A field name in SELECT must be either (1) in aggr func, or (2) the GROUP BY field

USING A FUNCTION (CONT)

Insert #72 here

With aggregate functions ("set values" – AVG, SUM, COUNT, MIN/MAX, etc) you can't have single-valued columns ("row values") included in the SELECT clause

- More example (error):

```
SELECT ProductStandardPrice –
   AVG(ProductStandardPrice)
FROM Product_T;
```

Row value

Set value

*** !!! "Row values and set values do NOT belong to the same SELECT clause"

If both really need to be in the same SELECT statement, use subquery (P. 231, top-middle)

SELECT INVOLVING AGGREGATE FUNCTION

- If no aggr func in SELECT: nothing to worry
- If ALL aggregate function, nothing to worry
- If there's a mixture of fields & agg func(s) in SELECT, error msg will result.
  - In this case (fields, w agg func) if a field (attribute) is to be in SELECT, then that attribute must be
    1. either in an aggregate function,
    2. or it is the GROUP BY field
- Note:

A GROUP BY field doesn't have to be in SELECT

ISSUES ABOUT ROW VALUE AND AGGREGATES

- **Top of P. 231**: SQL cannot return both a row value (such as Product_ID) and a set value (such as COUNT/AVG/SUM of a group);
  - users must run two separate queries, one that returns row info and one that returns set info

1. SELECT S_ID FROM STUDENT - Row
2. SELECT AVG(GPA) FROM STUDENT - Set
3. SELECT S_ID, AVG(GPA) FROM STUDENT – row & set: **ERROR!**

ROW VALUE AND AGGREGATES (CONT)

- SELECT S_ID, MAJOR, GPA
  FROM STUDENT
- SELECT S_ID, MAJOR, AVG(GPA)
  FROM STUDENT
- SELECT S_ID, MAJOR, AVG(GPA)
  FROM STUDENT
  GROUP BY MAJOR
- SELECT COUNT(S_ID), MAJOR, AVG(GPA)
  FROM STUDENT
  GROUP BY MAJOR

See middle of P. 230, SQL Server error message

WILDCARDS; COMPARISON OPERATORS; NULL VALUES

- Wildcards: P. 232
  - LIKE '%Desk'
  - LIKE '*Desk'
- Comparison operators:
- NULL values:
  - IS NULL
  - IS NOT NULL

TABLE 6-3 Comparison Operators in SQL

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
!=	Not equal to

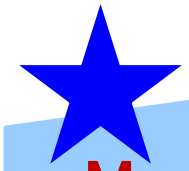
IS NULL

- IS NULL: the field has a null value – not 0, not space (_), but no value
- Useful in scenarios as follows:
  - Sold_date in a real estate property table
  - Paid_date in a student registration table
  - Transaction_amount in a auction table
  - …
- Syntax: WHERE *field_name* IS NULL
- The opposite is: WHERE *field_name* IS NOT NULL

SELECT EXAMPLE–BOOLEAN OPERATORS

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

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
   FROM Product_T
      WHERE ProductDescription LIKE '%Desk'
         OR ProductDescription LIKE '%Table'
         AND ProductStandardPrice > 300;
```



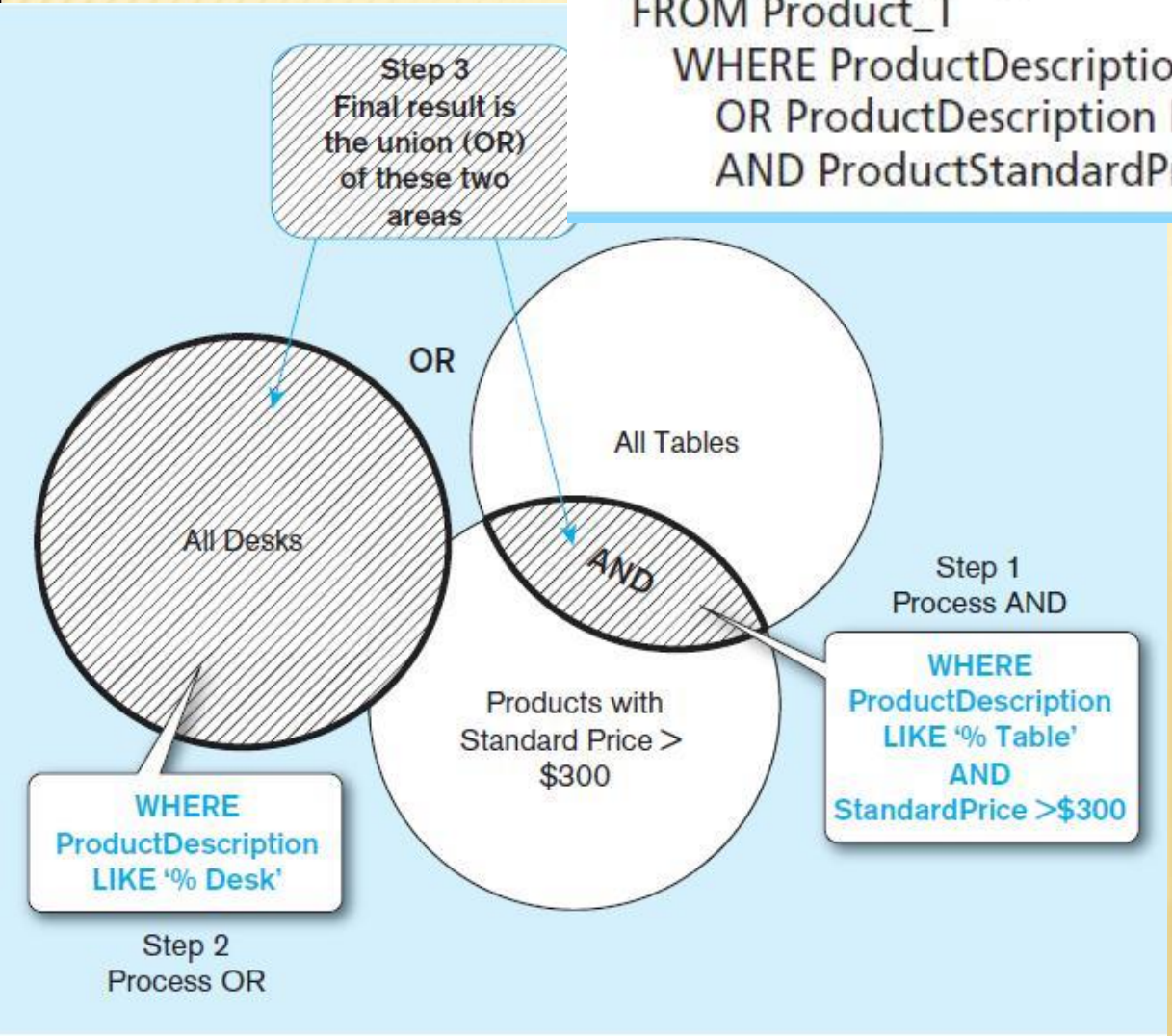
Must be very, very, very, VERY careful about Boolean operations

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.

In MS Access wildcard is "*"

Figure 6-8 Boolean query A without use of parentheses

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
  FROM Product_T
    WHERE ProductDescription LIKE '%Desk'
      OR ProductDescription LIKE '%Table'
      AND ProductStandardPrice > 300;
```



By default, processing order of Boolean operators is NOT, then AND, then OR

SELECT EXAMPLE–BOOLEAN OPERATORS

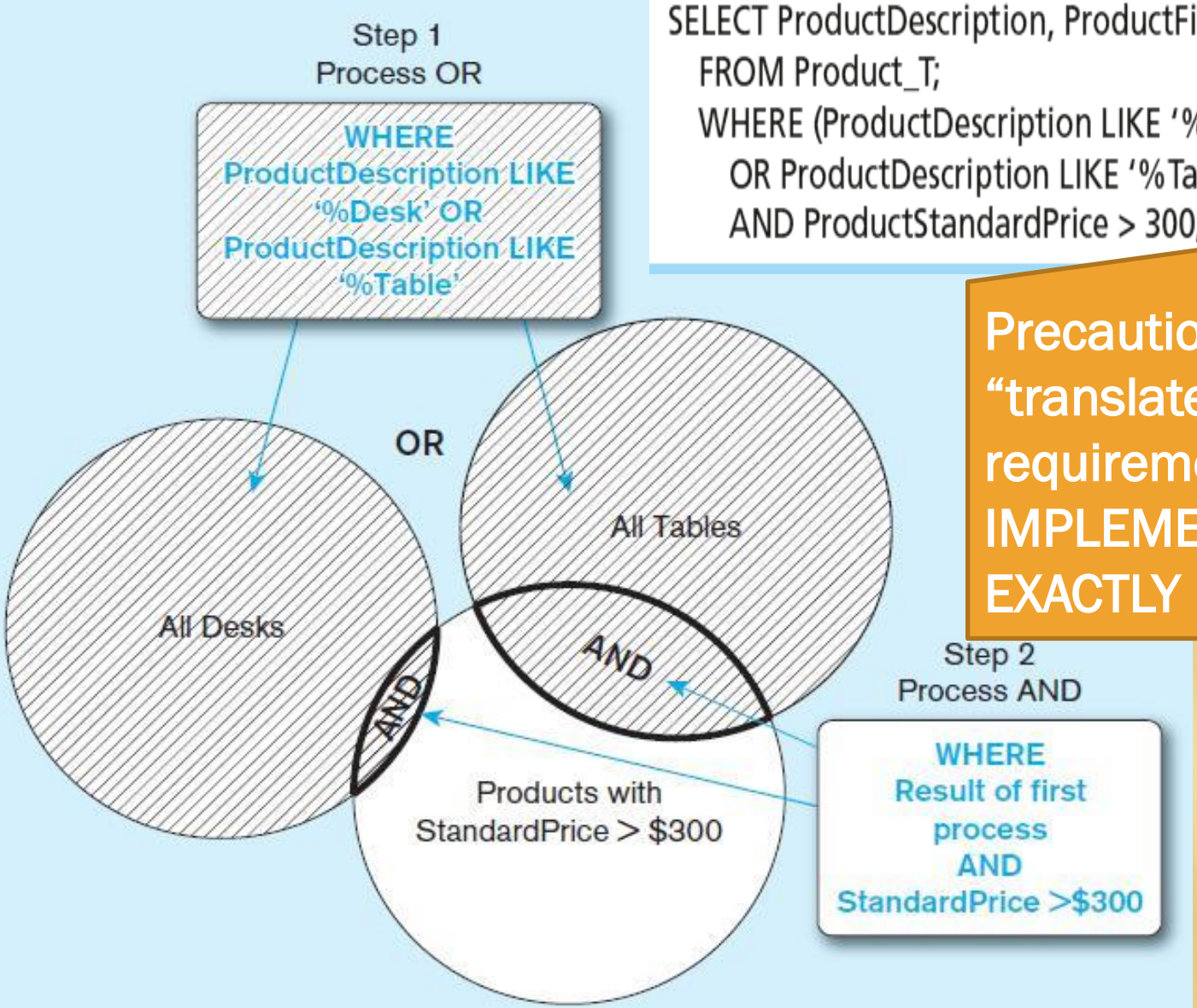
- **With parentheses…**these override the normal precedence of Boolean operators

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
   FROM Product_T;
   WHERE (ProductDescription LIKE '%Desk'
      OR ProductDescription LIKE '%Table')
      AND ProductStandardPrice > 300;
```

With parentheses, you can override normal precedence rules. In this case parentheses make the OR take place before the AND.

Figure 6-9 Boolean query B with use of parentheses

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
  FROM Product_T;
  WHERE (ProductDescription LIKE '%Desk'
    OR ProductDescription LIKE '%Table')
    AND ProductStandardPrice > 300;
```



Precaution: Do NOT try to "translate" the logical requirements – IMPLEMENT the logic EXACTLY

USING DISTINCT VALUES

- Compare:

```
SELECT ORDER_ID
    FROM ORDER_LINE_V;
```

and –

```
SELECT DISTINCT ORDER_ID
    FROM ORDER_LINE_V;
```

But:

```
SELECT DISTINCT ORDER_ID, ORDER_QUANTITY
    FROM ORDER_LINE_V;
```

Compare three figures on 272 & 273

USING IN AND NOT IN WITH LISTS

- WHERE field IN (value list)
  - 1 value use =, many values use IN
- Sets the condition that the value of the field
- Can be of any value that is in the list in the parentheses

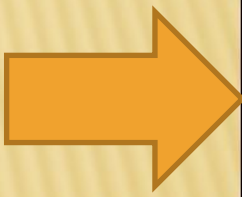
- WHERE Major IN ('ACCT', 'CIT', 'IS')
- Means the major can be either ACCT, or CIT, or IS
- It is more efficient than separate OR conditions.

- There is NEVER such code as follows:
- WHERE Major = ('ACCT', 'CIT', 'IS')

USING IN AND NOT IN WITH LISTS (2)

- When there is ONE value in the comparison:
  - WHERE Field = value [Note: ONE value! Same below]
  - WHERE Field > value
  - WHERE Field <= value
  - WHERE Field <> value (<> -- "not equal")
- When there are M values in comparison:
  - WHERE field IN (Value1, Value2, Value3, ...)

Compare:



COMPARISON OF IN AND =

- IN means “among the values of”, “can be one of the values in the list”
- = means “exactly that value”, “exactly the value of that variable”.
- So,
  - **IN** (list of values)
  - = one_value, = one_variable
  - NEVER **=** (list of variables)

IN (‘CA’, ‘OR’, ‘WA’)
= ‘CA’
= (‘CA’, ‘OR’, ‘WA’)

SORTING RESULTS WITH ORDER BY CLAUSE

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

Primary; secondary

(1) Not "SORT"! (2) Don't confuse w GROUP BY!!

```
SELECT CustomerName, CustomerCity, CustomerState
   FROM Customer_T
      WHERE CustomerState IN ('FL', 'TX', 'CA', 'HI')
         ORDER BY CustomerState, CustomerName;
```

Can also use column positions

Note: The IN operator in this example allows you to include rows whose CustomerState value is either FL, TX, CA, or HI.

IN; NOT IN

CATEGORIZING RESULTS USING GROUP BY CLAUSE

Collapses records into groups – no more records

- For use with aggregate functions
  - *Scalar aggregate*: single value returned from SQL query with aggregate function ("aggregate of the whole table")
  - *Vector aggregate*: multiple values returned from SQL query with aggregate function (via GROUP BY) ("agg. of the groups")

```
SELECT CustomerState, COUNT (CustomerState)
   FROM Customer_T
      GROUP BY CustomerState;
```

You can use single-value fields with aggregate functions if they (the fields) are included in the GROUP BY clause

Example: DNC students – 7K stud, 10 majors

ROW VALUE AND AGGREGATES (W GROUP)

- SELECT S_ID, MAJOR, GPA
  FROM STUDENT

What? How many rows?

- SELECT MAJOR, AVG(GPA)
  FROM STUDENT
  GROUP BY MAJOR

What? How many rows?

- SELECT T S_ID, MAJOR, AVG(GPA)
  FROM STUDENT
  GROUP BY MAJOR

Discuss

- SELECT COUNT(S_ID), MAJOR, AVG(GPA)
  FROM STUDENT
  GROUP BY MAJOR

Discuss

AGGREGATE FUNCTION W AND W/O GROUP BY

- Without GROUP BY, the aggregate functions (SUM, AVG, COUNT etc) are sum/avg/count of the WHOLE TABLE
- With GROUP BY, …
- It is important to know the differences between the two, so one can correctly and properly
  - Interpret the query outcomes
  - Design the query
- Example:
  - STUDENT table; GROUP BY Major
  - PRODUCT table; GROUP BY ProductLine

QUALIFYING RESULTS BY CATEGORIES USING THE HAVING CLAUSE

- For use with GROUP BY

```
SELECT CustomerState, COUNT (CustomerState)
   FROM Customer_T
      GROUP BY CustomerState
      HAVING COUNT (CustomerState) > 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.

A QUERY WITH BOTH WHERE AND HAVING

```
SELECT ProductFinish, AVG (ProductStandardPrice)
   FROM Product_T
      WHERE ProductFinish IN ('Cherry', 'Natural Ash', 'Natural Maple',
      'White Ash')
         GROUP BY ProductFinish
            HAVING AVG (ProductStandardPrice) < 750
               ORDER BY ProductFinish;
```

Discussion: Switch?

ProductID	ProductDescription	ProductFinish	ProductStandardPrice	ProductLineID
1	End Table	Cherry	$175.00	1
2	Coffee Table	Natural Ash	$200.00	2
3	Computer Desk	Natural Ash	$375.00	2
4	Entertainment Cente	Natural Maple	$650.00	3
5	Writers Desk	Cherry	$325.00	1
6	8-Drawer Desk	White Ash	$750.00	2
7	Dining Table	Natural Ash	$800.00	2
8	Computer Desk	Walnut	$250.00	3

Result:

PRODUCTFINISH	AVG(PRODUCTSTANDARDPRICE)
Cherry	250
Natural Ash	458.333333
Natural Maple	650

Discussions

QUERY WITH BOTH WHERE & HAVING (2)

- WHERE screens rows
  - those products whose finish are among (…)
  - Criteria applied to ALL rows
  - Before the grouping
- HAVING screens groups
  - those groups whose average price is …
  - Criteria applied to groups
  - After the grouping
- Observe the SELECT-clause: why ProductFinish can be there?

USING AND DEFINING VIEWS

- Views provide users controlled access to tables
- Base Table–table containing the raw data
- Dynamic View
  - A “virtual table” created dynamically upon request by a user
  - No data actually stored; instead data from base table made available to user
  - Based on SQL SELECT statement on base tables or other views
- Materialized View
  - Copy or replication of data
  - Data actually stored
  - Must be refreshed periodically to match corresponding base tables

Views

- Syntax of CREATE VIEW:
- CREATE VIEW *view-name* AS

  SELECT (*that provides the rows and columns of the view*)
- Example:
  - CREATE VIEW ORDER_TOTALS_V AS

    SELECT PRODUCT_ID **PRODUCT**,

    SUM(STANDARD_PRICE*QUANTITY) **TOTAL**

    FROM INVOICE_V

    GROUP BY PRODUCT_ID;

SAMPLE CREATE VIEW

```
CREATE VIEW ExpensiveStuff_V
  AS
    SELECT ProductID, ProductDescription, ProductStandardPrice
     FROM Product_T
       WHERE ProductStandardPrice > 300
       WITH CHECK OPTION;
```

- View has a name.
- View is based on a SELECT statement.
- CHECK_OPTION works only for updateable views and prevents updates that would create rows not included in the view.

ADVANTAGES OF VIEWS

- Simplify query commands
- Assist with data security (but don't rely on views for security, there are more important security measures)
- Enhance programming productivity
- Contain most current base table data
- Use little storage space
- Provide customized view for user
- Establish physical data independence

DISADVANTAGES OF VIEWS

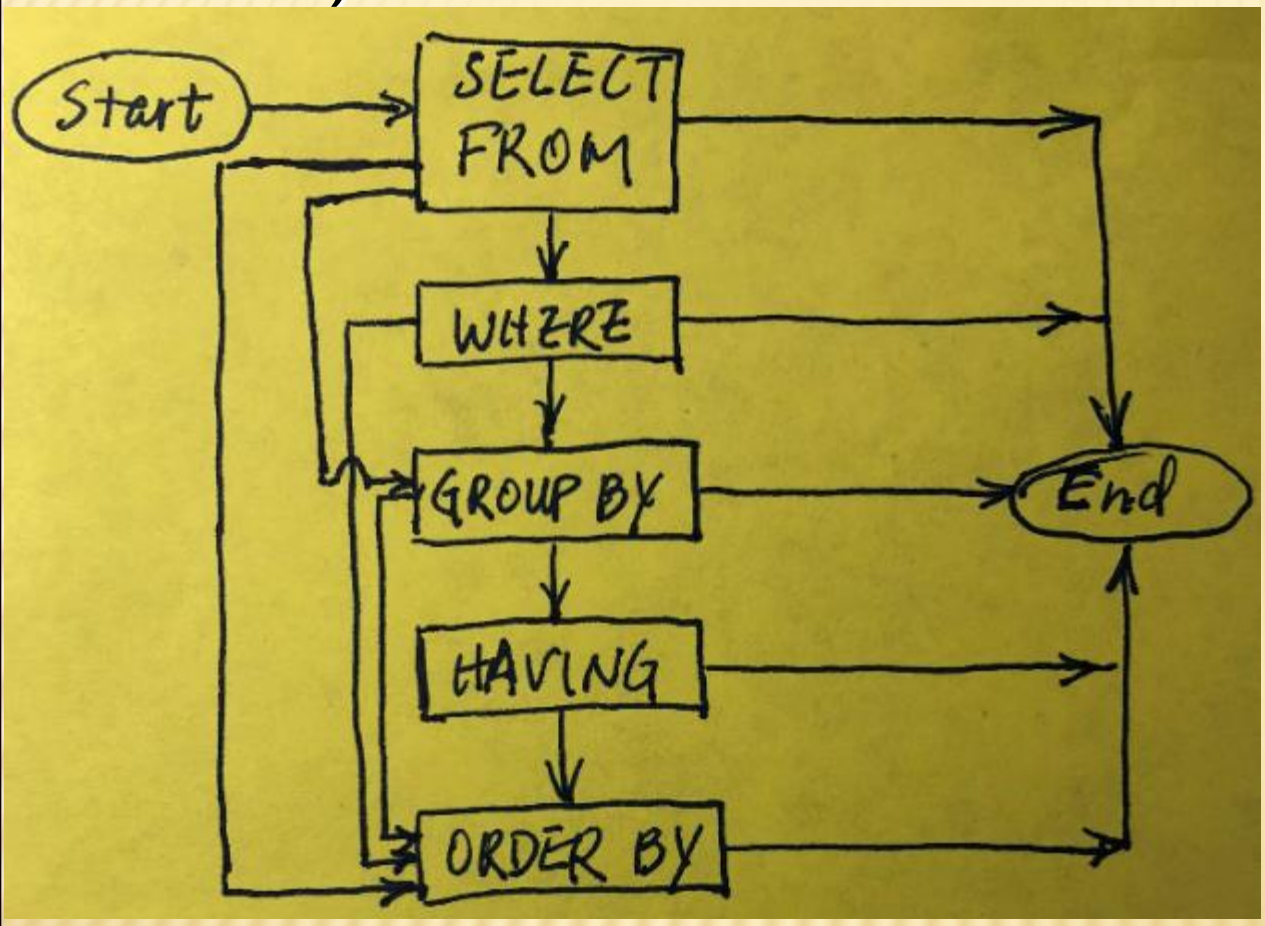
- Use processing time each time view is referenced
- May or may not be directly updateable

"SEVEN SINS" IN SELECT-STATEMENT
(POINT-EARNING OR POINT-SAVING TIPS)

1. Mix row values and set values (aggr functions)
   - The only allowable "mixture" is when the row value is in the GROUP-BY field
2. Use WHERE with aggr functions ("WHERE AVG...>...") – WHERE is condition for rows!
3. Use HAVING when there's no GROUP BY
4. HAVING appears before GROUP BY
5. Use column alias in conditions or Grp/Odr By
6. ORDERE BY: (1) Not "SORT"! (2) Don't confuse w GROUP BY!!
7. Put comma between clauses

Added 8/3/19

FLOWCHART OF SELECT STATEMENT (SHOULD GO TO #35-38)



1. All can end
2. All can go to ORDER BY
3. Only GROUP-BY can go to HAVING
4. ORDER BY is the last clause, *if* it is used

SELECT – FROM are must;
all others are optional

SQL BASICS – 1: SELECT STATEMENT

- Manipulates the data file
- Performs
  - Selection (filter rows)
  - Projection (filter columns)
  - Join (relate 2+ tables – getting data from 2^+ tables)
- Basic syntax:
- SELECT *field_list* FROM *table_list* WHERE *criteria* GROUP BY *field_list* HAVING *criteria* ORDER BY *field_list*;

SQL BASICS – 2: CLAUSES IN SELECT STATEMENT

	SELECT Clauses	Roles
1	SELECT *field_list*	What fields/columns to display - project
2	FROM *table_list*	What tables to use; join when 2+ tables
3	WHERE *criteria*	What rows to display – select; cri can be connected w AND or OR
4	GROUP BY *field_list*	Performs the logic of Total Query in Access
5	HAVING *criteria*	Same logic as 3, but now for groups (not rows)
6	ORDER BY *field_list*;	Display order/sequence; no logic function – does NOT have any real effect except look

Note: SELECT is ONE statement that ends with ";"
You do NOT have to break them into lines:
breaking them into lines is only for ease of reading

SQL BASICS – 3: EXAMPLE 1

	SELECT clauses	Example
1	SELECT *field_list*	**SELECT StudID, SLName, SFName, Major**
2	FROM *table_list*	**FROM STUDENT_t;**
3	WHERE *criteria*	
4	GROUP BY *field_list*	
5	HAVING *criteria*	
6	ORDER BY *field_list*;	

SQL BASICS – 4: EXAMPLE 2

	SELECT Clauses	Example
1	SELECT *field_list*	SELECT StudID, SLName, SFName, Major
2	FROM *table_list*	FROM STUDENT_t
3	WHERE *criteria*	WHERE Major = “Accounting”;
4	GROUP BY *field_list*	
5	HAVING *criteria*	
6	ORDER BY *field_list*;	

SQL BASICS – 5: EXAMPLE 3

	SELECT Clauses	Example
1	SELECT *field_list*	SELECT StudID, SLName, SFName, Major
2	FROM *table_list*	FROM STUDENT_t
3	WHERE *criteria*	WHERE Major = “Accounting” AND GPA >=3.0;
4	GROUP BY *field_list*	
5	HAVING *criteria*	
6	ORDER BY *field_list*;	

SQL BASICS – 6: EXAMPLE 4

	SELECT Clauses	Example
1	SELECT *field_list*	SELECT StudID, SLName, SFName, Major
2	FROM *table_list*	FROM STUDENT_t
3	WHERE *criteria*	WHERE Major = “Accounting” AND GPA >=3.0
4	GROUP BY *field_list*	
5	HAVING *criteria*	
6	ORDER BY *field_list*;	ORDER BY SLName, SFName;

SQL BASICS – 7: EXAMPLE 5 – GROUP BY

	SELECT Clauses	Example
1	SELECT *field_list*	SELECT COUNT(StudID), Major, AVG(GPA)
2	FROM *table_list*	FROM STUDENT_t
3	WHERE *criteria*	WHERE Major = "ECON" OR Major = "IS"
4	GROUP BY *field_list*	GROUP BY Major;
5	HAVING *criteria*	
6	ORDER BY *field_list*;	

Added 8/3

Field BooleanOpr Value AND **Field** BooleanOpr Value OR **Field** BooleanOpr Value – the "Field" must be repeated!!!

SQL BASICS – 8: EXAMPLE 6 – GROUP BY W ORDER

	SELECT Clauses	Example
1	SELECT *field_list*	SELECT COUNT(StudID), Major, AVG(GPA)
2	FROM *table_list*	FROM STUDENT_t
3	WHERE *criteria*	WHERE Major = “ECON” OR Major = “IS”
4	GROUP BY *field_list*	GROUP BY Major
5	HAVING *criteria*	
6	ORDER BY *field_list*;	ORDER BY Major, AVG(GPA);

Don’t confuse

Two fields

SQL BASICS – 9: EXAMPLE 7 – GROUP BY WITH HAVING

	SELECT Clauses	Example
1	SELECT *field_list*	SELECT COUNT(StudID), Major, AVG(GPA) AS AvGPA
2	FROM *table_list*	FROM STUDENT_t
3	WHERE *criteria*	WHERE Major = "ECON" OR Major = "IS"
4	GROUP BY *field_list*	GROUP BY Major
5	HAVING *criteria*	HAVING AVG(GPA) >=2.5
6	ORDER BY *field_list*;	ORDER BY Major, AVG(GPA);

Watch use of alias; 8/3

IMPORTANT CLARIFICATION RE THREE CLAUSES (SUMMER 2019 ADDED)

- SELECT-Clause: if there is a mixture of row values and set values,
  - the row (field) must be in GROUP BY
  - The opposite is not true: GROUP BY-field does NOT need to be in SLECT! Does NOT need!!!
  - (discuss the logic)
- GROUP BY and HAVING:
  - HAVING needs GROUP BY (and appears *AFTER* GROUP BY)
  - GROUP BY does NOT need HAVING: you can have GROUP BY w/o HAVING

Continue:

IMPORTANT CLARIFICATION RE THREE CLAUSES (CONTD; SUMMER 2019 ADDED)

- SELECT-clause and WHERE-clause
- SELECT-clause and HAVING-clause

Both are conditions

- SELECT and WHERE: the relationship of a field in SELECT-clause and in WHERE-clause?

- SELECT and HAVING: the relationship of a field in SELECT-clause and in HAVING-clause?