Lecture 5: Link | Syllabus
*Class Week 5/15

Database Systems by Coronel & Morris

Chapter 7: Introduction to Structured Query Language (SQL)

* 7.1 *Introduction to SQL*

Ideally, a database language allows you to create database and table structures, perform basic data management chores (add, delete, and modify), and perform complex queries designed to transform the raw data into useful information. Moreover, a database language must perform such basic functions with minimal user effort, and its command structure and syntax must be easy to learn. Finally, it must be portable; that is, it must conform to some basic standard so a person does not have to relearn the basics when moving from one RDBMS to another. SQL meets those ideal database language requirements well

SQL functions fit into two broad categories:

- it is a *data definition Language (DDL)* - SQL includes commands to create database objects such as tables, indexes, and views, as well as commands to define access rights to those database objects. Some common data definition commands you will learn are listed in Table 7.1
- It is a *data manipulation language (DML)* - SQL includes commands to insert, update, delete, and retrieve data within the database tables. The data manipulation commands you will learn in this chapter are listed in Table 7.2

SQL DATA DEFINITION COMMANDS

COMMAND OR OPTION	DESCRIPTION
CREATE SCHEMA AUTHORIZATION	Creates a database schema
CREATE TABLE	Creates a new table in the user's database schema
NOT NULL	Ensures that a column will not have null values
UNIQUE	Ensures that a column will not have duplicate values
PRIMARY KEY	Defines a primary key for a table
FOREIGN KEY	Defines a foreign key for a table
DEFAULT	Defines a default value for a column (when no value is given)
CHECK	Validates data in an attribute
CREATE INDEX	Creates an index for a table
CREATE VIEW	Creates a dynamic subset of rows and columns from one or more tables (see Chapter 8, Advanced SQL)
ALTER TABLE	Modifies a table's definition (adds, modifies, or deletes attributes or constraints)
CREATE TABLE AS	Creates a new table based on a query in the user's database schema
DROP TABLE	Permanently deletes a table (and its data)
DROP INDEX	Permanently deletes an index
DROP VIEW	Permanently deletes a view

SQL DATA MANIPULATION COMMANDS

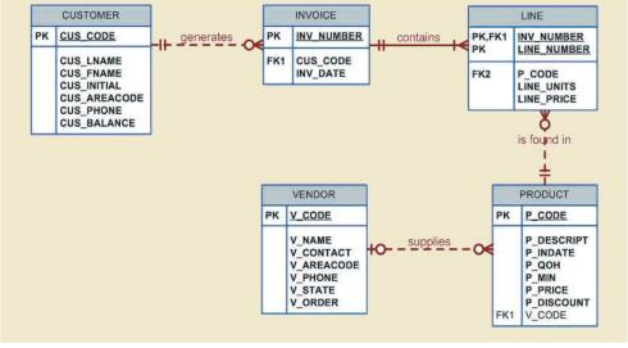
COMMAND OR OPTION	DESCRIPTION
INSERT	Inserts row(s) into a table
SELECT	Selects attributes from rows in one or more tables or views
WHERE	Restricts the selection of rows based on a conditional expression
GROUP BY	Groups the selected rows based on one or more attributes
HAVING	Restricts the selection of grouped rows based on a condition
ORDER BY	Orders the selected rows based on one or more attributes
UPDATE	Modifies an attribute's values in one or more table's rows
DELETE	Deletes one or more rows from a table
COMMIT	Permanently saves data changes
ROLLBACK	Restores data to its original values
Comparison operators	
=, <, >, <=, >=, <>, !=	Used in conditional expressions
Logical operators	
AND/OR/NOT	Used in conditional expressions
Special operators	Used in conditional expressions
BETWEEN	Checks whether an attribute value is within a range
IS NULL	Checks whether an attribute value is null
LIKE	Checks whether an attribute value matches a given string pattern
IN	Checks whether an attribute value matches any value within a value list
EXISTS	Checks whether a subquery returns any rows
DISTINCT	Limits values to unique values
Aggregate functions	Used with SELECT to return mathematical summaries on columns
COUNT	Returns the number of rows with non-null values for a given column
MIN	Returns the minimum attribute value found in a given column
MAX	Returns the maximum attribute value found in a given column
SUM	Returns the sum of all values for a given column
AVG	Returns the average of all values for a given column

* The American National Standards Institute (ANSI) prescribes a standard SQL. The ANSI SQL standards are also accepted by the International Organization for Standardization (ISO), a consortium composed of national standards bodies of over 150 countries. ed application from one RDBMS to another without making some changes. However, even though there are several different SQL "dialects," their differences are minor. Whether you use Oracle, Microsoft SQL Server, MySQL, IBM's DB2, Microsoft Access.

* 7.2-a *The Database Model*

A simple database composed of the following tables is used to illustrate the SQL commands in this chapter: CUSTOMER, INVOICE, LINE, PRODUCT, and VENDOR. This database model is shown in Figure 7.1

FIGURE 7.1 THE DATABASE MODEL



* The database model in Figure 7.1 reflects the following business rules:

- A customer may generate many invoices. Each invoice is generated by one customer.
- An invoice contains one or more invoice lines. Each invoice line is associated with one invoice.
- Each invoice line references one product. A product may be found in many invoice lines. (You can sell more than one hammer to more than one customer.)
- A vendor *may* supply many products. Some vendors do not yet supply products. For example, a vendor list may include potential vendors.
- If a product is vendor-supplied, it is supplied by only a single vendor.
- Some products are not supplied by a vendor. For example, some products may be produced in-house or bought on the open market.

* 7.2-c *The Database Schema*

In the SQL environment, a schema is a logical group of database objects—such as tables and indexes—that are related to each other. Usually, the schema belongs to a single user or application. A single database can hold multiple schemas that belong to different users

or applications. Schemas are useful in that they group tables by owner (or function) and enforce a first level of security by allowing each user to see only the tables that belong to that user

For most RDBMSs, the CREATE SCHEMA AUTHORIZATION command is optional, which is why this chapter focuses on the ANSI SQL commands required to create and manipulate tables.

* 7.2-d *Data Types*

Here is a table from figure 7.1 with their attributes and their data-types

TABLE 7.3

DATA DICTIONARY FOR THE CH07_SALECO DATABASE

TABLE NAME	ATTRIBUTE NAME	CONTENTS	TYPE	FORMAT	RANGE	REQUIRED	PK OR FK	FK REFERENCED TABLE
PRODUCT	P_CODE	Product code	VARCHAR(10)	XXXXXXXXXX	NA	Y	PK	
	P_DESCRIPT	Product description	VARCHAR(35)	Xxxxxxxxxxx	NA	Y		
	P_INDATE	Stocking date	DATE	DD-MON-YYYY	NA	Y		
	P_QOH	Units available	SMALLINT	####	0–9999	Y		
	P_MIN	Minimum units	SMALLINT	####	0–9999	Y		
	P_PRICE	Product price	NUMBER(8,2)	####.##	0.00–9999.00	Y		
	P_DISCOUNT	Discount rate	NUMBER(5,2)	0.##	0.00–0.20	Y		
	V_CODE	Vendor code	INTEGER	###	100–999		FK	VENDOR
VENDOR	V_CODE	Vendor code	INTEGER	####	1000–9999	Y	PK	
	V_NAME	Vendor name	VARCHAR(35)	Xxxxxxxxxxxxx	NA	Y		
	V_CONTACT	Contact person	VARCHAR(25)	Xxxxxxxxxxxxx	NA	Y		
	V_AREACODE	Area code	CHAR(3)	999	NA	Y		
	V_PHONE	Phone number	CHAR(8)	999–9999	NA	Y		
	V_STATE	State	CHAR(2)	XX	NA	Y		
	V_ORDER	Previous order	CHAR(1)	X	Y or N	Y		

FK = Foreign key
PK = Primary key
CHAR = Fixed-length character data, 1 to 255 characters
VARCHAR = Variable-length character data, 1 to 2,000 characters. VARCHAR is automatically converted to VARCHAR2 in Oracle.
NUMBER = Numeric data. NUMBER(9,2) is used to specify numbers that have two decimal places and are up to nine digits long, including the decimal places. Some RDBMSs permit the use of a MONEY or a CURRENCY data type.
NUMERIC = Numeric data. DBMSs that do not support the NUMBER data type typically use NUMERIC instead.
INT = Integer values only. INT is automatically converted to NUMBER in Oracle.
SMALLINT = Small integer values only. SMALLINT is automatically converted to NUMBER in Oracle.
DATE formats vary. Commonly accepted formats are DD-MON-YYYY, DD-MON-YY, MM/DD/YYYY, and MM/DD/YY.

*Not all the ranges shown here will be illustrated in this chapter. However, you can use these constraints to practice writing your own.

* Data types for table 7.3 explained:

- P_PRICE clearly requires some kind of numeric data type; defining it as a character field is not acceptable.
- Just as clearly, a vendor name is an obvious candidate for a character data type. For example, VARCHAR(35) fits well because vendor names are var-length character strings, and such strings may be up to 35 characters long.
- At first glance, it might seem logical to select a numeric data type for V_AREACODE because it contains only digits. So selecting a character data type is more appropriate. This is true for many common attributes in business data models. Even zip codes contain all digits, they must be defined as character data because some zip codes begin with the digit zero (0), and a numeric data type would cause the leading zero to be dropped.
- Selecting P_INDATE to be a (Julian) DATE field rather than a character field is desirable because Julian dates allow you to make simple date comparisons and perform date arithmetic.

* 7.2-e *Creating Table Structures*

Now you are ready to implement the PRODUCT and VENDOR table structures with the help of SQL, using the CREATE TABLE syntax shown

```
CREATE TABLE tablename (
    column1      data type     [constraint] [,
    column2      data type     [constraint] ] [,
    PRIMARY KEY  (column1      [, column2]) ] [,
    FOREIGN KEY  (column1      [, column2]) REFERENCES tablename] [,
    CONSTRAINT   constraint ] );
```

* To make the SQL code more readable, most SQL programmers use one line per column (attribute) definition. In addition, spaces are used to line up the attribute characteristics and constraints. Finally, both table and attribute names are fully capitalized. Those conventions are used in the following examples that create VENDOR and PRODUCT tables and subsequent tables throughout the book

* 7.2-f *SQL Constraints*

A column constraint applies to just one column; a table constraint may apply to many columns. Those constraints are supported at varying levels of compliance by enterprise RDBMSs'. In this chapter, Oracle is used to illustrate SQL constraints. For example, note that the following SQL command sequence uses the DEFAULT and CHECK constraints to define the table named CUSTOMER.

```
CREATE TABLE CUSTOMER (
CUS_CODE        NUMBER          PRIMARY KEY,
CUS_LNAME       VARCHAR(15)     NOT NULL,
CUS_FNAME       VARCHAR(15)     NOT NULL,
CUS_INITIAL     CHAR(1),
CUS_AREACODE    CHAR(3)         DEFAULT '615'     NOT NULL
                                CHECK(CUS_AREACODE IN
                                ('615','713','931')),
CUS_PHONE       CHAR(8)         NOT NULL,
CUS_BALANCE     NUMBER(9,2)     DEFAULT 0.00,
CONSTRAINT CUS_UI1 UNIQUE (CUS_LNAME, CUS_FNAME));
```

* Besides the PRIMARY KEY and FOREIGN KEY constraints, the ANSI SQL standard also defines the following constraints:

- The NOT NULL constraint ensures that a column does not accept nulls.
- The UNIQUE constraint ensures that all values in a column are unique.
- The DEFAULT constraint assigns a value to an attribute when a new row is added to a table. The end user may, of course, enter a value other than the default value.
- The CHECK constraint is used to validate data when an attribute value is entered. The CHECK constraint does precisely what its name suggests: it checks to see that a specified condition exists.

* 7.2-g *SQL Index*

The ability to create indexes quickly and efficiently is important. Using the CREATE INDEX command, SQL indexes can be created on the basis of any selected attribute.

```
CREATE [UNIQUE]INDEX indexname ON tablename(column1 [, column2])
```

For example, based on the attribute P_INDATE stored in the PRODUCT table, the following command creates an index named P_INDATEX:

```
CREATE INDEX P_INDATEX ON PRODUCT(P_INDATE);
```

To delete an index, use the **DROP INDEX** command:

```
DROP INDEX indexname
```

For example, if you want to eliminate the PROD_PRICEX index, type:

```
DROP INDEX PROD_PRICEX;
```

* 7.3-a *Adding Table Rows*

SQL requires the use of the INSERT command to enter data into a table.

```
INSERT INTO tablename VALUES (value1, value2, …, valuen)
```

* Normal example of INSERT Command

```
INSERT INTO PRODUCT(P_CODE, P_DESCRIPT) VALUES ('BRT-345','Titanium drill bit');
```

* Inserting Rows with optional Attributes

*** 7.3-b** *Saving Table Changes*

Any changes made to the table contents are not saved on disk until you close the data base, close the program you are using, or use the COMMIT command. ---> COMMIT[*WORKS*] or COMMIT;

*** 7.3-c** *Listing Table Rows*

The SELECT command is used to list the contents of a table.

```
SELECT    columnlist    FROM    tablename
```

```
SELECT * FROM PRODUCT;
```

* A wildcard character is a symbol that can be used as a general substitute for other characters or commands.

*** 7.3-d** *Updating Table Rows*

Use the UPDATE command to modify data in a table.

```
UPDATE      tablename
SET         columnname = expression [, columnname = expression]
[WHERE      conditionlist ];
```

For example, if you want to change P_INDATE from December 13, 2015, to January 18, 2016, in the second row of the PRODUCT table (see Figure 7.3), use the primary key (13-Q2/P2) to locate the correct row. Therefore, type:

```
UPDATE      PRODUCT
SET         P_INDATE = '18-JAN-2016'
WHERE       P_CODE = '13-Q2/P2';
```

*** 7.3-e** *Restoring Table Contents*

If you have not yet used the COMMIT command to store the changes permanently in the database, you can restore the database to its previous condition with the ROLLBACK command. ROLLBACK undoes any changes since the last COMMIT command and brings all of the data back to the values that existed before the changes were made. ---> ROLLBACK;

*** 7.3-f** *Deleting Table Rows*

It is easy to delete a table row using the DELETE statement.

For example, if you want to delete the product you added earlier whose code (P_CODE) is 'BRT-345', use the following command:

```
DELETE FROM     PRODUCT
WHERE           P_CODE = 'BRT-345';
```

*** 7.3-g** *Inserting Table Rows with a Select Subquery*

You learned in Section 7-3a how to use the INSERT statement to add rows to a table. In that section, you added rows one at a time. In this section, you will learn how to add multiple rows to a table, using another table as the source of the data.

```
INSERT INTO tablename   SELECT columnlist   FROM tablename;
```

In this case, the INSERT statement uses a SELECT subquery. A **subquery**, also known as a **nested query** or an **inner query**, is a query that is embedded (or nested) inside another query. The inner query is always executed first by the RDBMS. Given

*** 7.4** *Select Queries*

In this section, you will learn how to fine-tune the SELECT command by adding restrictions to the search criteria. When coupled with appropriate search conditions, SELECT is an incredibly powerful tool that enables you to transform data into information.

*** 7.4-a** *Select Rows with Conditional Restrictions*

You can select partial table contents by placing restrictions on the rows to be included in the output. Use the WHERE clause to add conditional restrictions to the SELECT statement that limit the rows returned by the query.

```
SELECT      columnlist
FROM        tablelist
[WHERE      conditionlist ];
```

* You can use the WHERE command along with a different comparisons operators in the chart to the right to retrieve table data

COMPARISON OPERATORS	
SYMBOL	MEANING
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

* You can use the WHERE command along with a different comparisons operators in the chart to the right to retrieve table data

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<> or !=	Not equal to

* 7.4-b *Arithmetic Operators: The Rule of Precedence*

As you saw in the previous example, you can use arithmetic operators with table attributes in a column list or in a conditional expression. In fact, SQL commands are often used in conjunction with the arithmetic operators shown below

THE ARITHMETIC OPERATORS	
OPERATOR	DESCRIPTION
+	Add
-	Subtract
*	Multiply
/	Divide
^	Raise to the power of (some applications use ** instead of ^)

* As you perform mathematical operations on attributes, remember the mathematical rules of precedence. As the name suggests, the rules of precedence are the rules that establish the order in which computations are completed.

For example, note the order of the following computational sequence:

1. Perform operations within parentheses
2. Perform power operations
3. Perform multiplications and divisions
4. Perform additions and subtractions

* 7.4-c *Logical Operators: AND, OR, and NOT*

SQL allows you to include multiple conditions in a query through the use of logical operators. The logical operators are AND, OR, and NOT.

```
SELECT    P_DESCRIPT, P_INDATE, P_PRICE, V_CODE
FROM      PRODUCT
WHERE     (P_PRICE < 50 AND P_INDATE > '15-Jan-2016')
OR        V_CODE = 24288;
```

* 7.4-d *Special Operators*

ANSI-standard SQL allows the use of special operators in conjunction with the WHERE clause.

These special operators include…

- BETWEEN : Used to check whether an attribute value is within a range
- IS NULL : Used to check whether an attribute value is null
- LIKE : Used to check whether an attribute value matches a given string pattern
- IN : Used to check whether an attribute value matches any value within a value list
- EXISTS : Used to check whether a subquery returns any rows

* 7.5 *Additional Data Definition Commands*

All changes in the table structure are made by using the ALTER TABLE command followed by a keyword that produces the specific change you want to make. Three options are available: ADD, MODIFY, and DROP. You use ADD to add a column, MODIFY to change column characteristics, and DROP to delete a column from a table

```
ALTER TABLE tablename
    {ADD | MODIFY} ( columnname datatype [ {ADD | MODIFY}
    columnname datatype] );
```

The ALTER TABLE command can also be used to add table constraints. In those cases, the syntax would be:

```
ALTER TABLE tablename
    ADD constraint [ ADD constraint ];
```

* 7.5-a *Changing a Column's Data Type*

Using the ALTER syntax, the integer V_CODE in the PRODUCT table can be changed to a character V_CODE by using the following command:

```
ALTER TABLE PRODUCT
    MODIFY (V_CODE CHAR(5));
```

* 7.5-b *Changing a Column's Data Characteristics*

If the column to be changed already contains data, you can make changes in the column's characteristics if those changes do not alter the data type. For example, if you want to increase the width of the P_PRICE column to nine digits, use the following command:

```
ALTER TABLE PRODUCT
    MODIFY (P_PRICE DECIMAL(9,2));
```

If you now list the table contents, you can see that the column width of P_PRICE has increased by one digit.

*** 7.5-c** *Adding a Column*

You can alter an existing table by adding one or more columns

```
ALTER TABLE PRODUCT
    ADD (P_SALECODE CHAR(1));
```

*** 7.5-d** *Dropping a Column*

Occasionally, you might want to modify a table by deleting a column. Suppose that you want to delete the V_ORDER attribute from the VENDOR table. You would use the following command:

```
ALTER TABLE VENDOR
    DROP COLUMN V_ORDER;
```

Again, some RDBMSs impose restrictions on attribute deletion. For example, you may not drop attributes that are involved in foreign key relationships, nor may you delete an attribute if it is the only one in a table.

*** 7.5-e** *Advanced Data Updates*

To make changes to data in the columns of existing rows, use the UPDATE command. Do not confuse the INSERT and UPDATE commands: INSERT creates new rows in the table, while UPDATE changes rows that already exist.

```
UPDATE      PRODUCT
SET         P_SALECODE = '2'
WHERE       P_CODE = '1546-QQ2';
```

*** 7.5-f** *Copying Parts of Tables*

As you will discover in later chapters on database design, sometimes it is necessary to break up a table structure into several component parts (or smaller tables). Fortunately, SQL allows you to copy the contents of selected table columns so that the data need not be re-entered manually into the newly created table(s). For example, if you want to copy P_CODE, P_DESCRIPT, P_PRICE, and V_CODE from the PRODUCT table to a new table named PART, you create the PART table structure first, as follows:

```
CREATE TABLE PART(
PART_CODE           CHAR(8),
PART_DESCRIPT       CHAR(35),
PART_PRICE          DECIMAL(8,2),
V_CODE              INTEGER,
PRIMARY KEY (PART_CODE));
```

*** 7.5-g** *Adding Primary and Foreign Key Designations*

When you create a new table based on another table, the new table does not include integrity rules from the old table. In particular, there is no primary key. To define the primary key for the new PART table, use the following command :

```
ALTER TABLE     LINE
    ADD         PRIMARY KEY (INV_NUMBER, LINE_NUMBER)
    ADD         FOREIGN KEY (INV_NUMBER) REFERENCES INVOICE
    ADD         FOREIGN KEY (P_CODE) REFERENCES PRODUCT;
```

*** 7.5-h** *Deleting a Table from the Database*

A table can be deleted from the database using the DROP TABLE command. For example, you can delete the PART table you just created with the following command :

```
DROP TABLE PART;
```

You can drop a table only if it is not the "one" side of any relationship. If you try to drop a table otherwise, the RDBMS will generate an error message indicating that a foreign key integrity violation has occurred.

*** 7.6** *Additional SELECT Query Keywords*

One of the most important advantages of SQL is its ability to produce complex free form queries. The logical operators that were introduced earlier to update table contents work just as well in the query environment. In addition, SQL provides useful functions

that count, find minimum and maximum values, calculate averages, and so on.

*** 7.6-a** *Ordering a Listing*

The ORDER BY clause is especially useful when the listing order is important to you. The syntax is:

```
SELECT      columnlist
FROM        tablelist
[WHERE      conditionlist ]
[ORDER BY   columnlist [ASC | DESC] ];
```

*** 7.6-b** *Ordering a Listing*

How many different vendors are currently represented in the PRODUCT table? A simple listing (SELECT) is not very useful if the table contains several thousand rows and you have to sift through the vendor codes manually. Fortunately, SQL's DISTINCT clause produces a list of only those values that are different from one another. For example, the command

```
SELECT      DISTINCT V_CODE
FROM        PRODUCT;
```

*** 7.6-c** *Aggerate Functions*

SQL can perform various mathematical summaries for you, such as counting the number of rows that contain a specified condition, finding the minimum or maximum values for a specified attribute, summing the values in a specified column, and averaging the values in a specified column.

SOME BASIC SQL AGGREGATE FUNCTIONS

FUNCTION	OUTPUT
COUNT	The number of rows containing non-null values
MIN	The minimum attribute value encountered in a given column
MAX	The maximum attribute value encountered in a given column
SUM	The sum of all values for a given column
AVG	The arithmetic mean (average) for a specified column

7.6-d *Grouping Data*

In the previous examples, the aggregate functions summarized data across all rows in the given tables. Sometimes, however, you do not want to treat the entire table as a single collection of data for summarizing. Rows can be grouped into smaller collections quickly and easily using the GROUP BY clause within the SELECT statement

```
SELECT      columnlist
FROM        tablelist
[WHERE      conditionlist ]
[GROUP BY   columnlist ]
[HAVING     conditionlist ]
[ORDER BY   columnlist [ASC | DESC] ];
```

7.7 *Joining Database Tables*

The ability to combine, or join, tables on common attributes is perhaps the most important distinction between a relational database and other databases. A join is performed when data is retrieved from more than one table at a time. To join tables, you simply list the tables in the FROM clause of the SELECT statement. The DBMS will create the Cartesian product of every table in the FROM clause. However, to get the correct result—that is, a natural join—you must select only the rows in which the common attribute values match. To do this, use the WHERE clause to indicate the common attributes used to link the tables; this WHERE clause is sometimes referred to as the join condition.

CREATING LINKS THROUGH FOREIGN KEYS

TABLE	ATTRIBUTES TO BE SHOWN	LINKING ATTRIBUTE
PRODUCT	P_DESCRIPT, P_PRICE	V_CODE
VENDOR	V_NAME, V_CONTACT, V_AREACODE, V_PHONE	V_CODE

7.7-a *Joining Tables with an Alias*

An alias may be used to identify the source table from which the data is taken. The aliases P and V are used to label the PRODUCT and VENDOR tables in the next command sequence

```
SELECT      P_DESCRIPT, P_PRICE, V_NAME, V_CONTACT, V_AREACODE,
            V_PHONE
FROM        PRODUCT P, VENDOR V
WHERE       P.V_CODE = V.V_CODE
ORDER BY    P_PRICE;
```

7.7-b *Recursive Joins*

An alias is especially useful when a table must be joined to itself in a recursive query. For example, suppose that you are working with the EMP table shown in Figure 7.30. Using the data in the EMP table, you can generate a list of all employees with their managers' names by joining the EMP table to itself. In that case, you would also use aliases to differentiate the table from itself. The SQL command sequence would look like this:

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

Chapter 8: Advanced SQL

* 8.1 *SQL Join Operators*

The relational join operation merges rows from two tables and returns the rows with one of the following conditions :

- Have common values in common columns (natural join).
- Meet a given join condition (equality or inequality).
- Have common values in common columns or have no matching values (outer join).

In Chapter 7, you learned how to use the SELECT statement in conjunction with the WHERE clause to join two or more tables. For example, you can join the PRODUCT and VENDOR tables through their common V_CODE by writing the following:

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

* Generally, the join condition will be an equality comparison of the primary key in one table and the related foreign key in the second table. Join operations can be as inner joins and outer joins. The inner join is the join in which only rows that meet a given criterion are selected. The join criterion can be an equality condition or an inequality condition. An outer join returns not only the matching rows but the rows with unmatched attribute values for one table or both tables to be joined. More example of SQL join commands --->

SQL JOIN EXPRESSION STYLES

JOIN CLASSIFICATION	JOIN TYPE	SQL SYNTAX EXAMPLE	DESCRIPTION
CROSS	CROSS JOIN	SELECT * FROM T1, T2	Returns the Cartesian product of T1 and T2 (old style)
		SELECT * FROM T1 CROSS JOIN T2	Returns the Cartesian product of T1 and T2
INNER	Old-style JOIN	SELECT * FROM T1, T2 WHERE T1.C1=T2.C1	Returns only the rows that meet the join condition in the WHERE clause (old style); only rows with matching values are selected
	NATURAL JOIN	SELECT * FROM T1 NATURAL JOIN T2	Returns only the rows with matching values in the matching columns; the matching columns must have the same names and similar data types
	JOIN USING	SELECT * FROM T1 JOIN T2 USING (C1)	Returns only the rows with matching values in the columns indicated in the USING clause
	JOIN ON	SELECT * FROM T1 JOIN T2 ON T1.C1=T2.C1	Returns only the rows that meet the join condition indicated in the ON clause
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

* 8.1-a *Cross Join*

A cross join performs a relational product (also known as the Cartesian product) of two tables.

SELECT *column-list* FROM *table1* CROSS JOIN *table2*

For example, the following command:

SELECT * FROM INVOICE CROSS JOIN LINE;

performs a cross join of the INVOICE and LINE tables that generates 144 rows. (There are 8 invoice rows and 18 line rows, yielding 8 × 18 = 144 rows.)

You can also perform a cross join that yields only specified attributes. For example, you can specify:

```
SELECT      INVOICE.INV_NUMBER, CUS_CODE, INV_DATE, P_CODE
FROM        INVOICE CROSS JOIN LINE;
```

The results generated through that SQL statement can also be generated by using the following syntax:

```
SELECT      INVOICE.INV_NUMBER, CUS_CODE, INV_DATE, P_CODE
FROM        INVOICE, LINE;
```

* 8.1-b *Natural Join*

A natural join returns all rows with matching values in the matching columns and eliminates duplicate columns. This style of query is used when the tables share one or more common attributes with common names.

SELECT *column-list* FROM *table1* NATURAL JOIN *table2*

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 relational product of the two tables.

The following example performs a natural join of the CUSTOMER and INVOICE tables and returns only selected attributes:

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

* 8.1-c *JOIN USING Clause*

A second way to express a join is through the USING keyword. The query returns only the rows with matching values in the column indicated in the USING clause—and that column must exist in both tables.

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

To see the JOIN USING query in action, perform a join of the INVOICE and LINE tables by writing the following:

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

* 8.1-d *JOIN ON Clause*

The previous two join styles use common attribute names in the joining tables. Another way to express a join when the tables have no common attribute names is to use the JOIN ON operand. The query will return only the rows that meet the indicated join 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 obviously they must have comparable data types.)

```
SELECT    INVOICE.INV_NUMBER, PRODUCT.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;
```

* 8.1-e *Outer Join*

An outer join returns not only the rows matching the join condition (that is, rows with matching values in the common columns), it returns the rows with unmatched values. The ANSI standard defines three types of outer joins: left, right, and full. The left and right designations reflect the order in which the tables are processed by the DBMS. Remember that join operations take place two tables at a time. The first table named in the FROM clause will be the left side, and the second table named will be the right side. If three or more tables are being joined, the result of joining the first two tables becomes the left side, and the third table becomes the right side.

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

For example, the following query lists the product code, vendor code, and vendor name for all products and includes those vendors with no matching products:

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

* 8.2 *Subqueries and Correlated Queries*

You already know that a subquery is based on the use of the SELECT statement to return one or more values to another query, but subqueries have a wide range of uses. For example, you can use a subquery within a SQL data manipulation language (DML) statement such as INSERT, UPDATE, or DELETE, in which a value or list of values (such as multiple vendor codes or a table) is expected.

SELECT SUBQUERY EXAMPLES

SELECT SUBQUERY EXAMPLES	EXPLANATION
INSERT INTO PRODUCT SELECT * FROM P;	Inserts all rows from Table P into the PRODUCT table. Both tables must have the same attributes. The subquery returns all rows from Table P.
UPDATE PRODUCT SET P_PRICE = (SELECT AVG(P_PRICE) FROM PRODUCT) WHERE V_CODE IN (SELECT V_CODE FROM VENDOR WHERE V_AREACODE = '615')	Updates the product price to the average product price, but only for products provided by vendors who have an area code equal to 615. The first subquery returns the average price; the second subquery returns the list of vendors with an area code equal to 615.
DELETE FROM PRODUCT WHERE V_CODE IN (SELECT V_CODE FROM VENDOR WHERE V_AREACODE = '615')	Deletes the PRODUCT table rows provided by vendors with an area code equal to 615. The subquery returns the list of vendor codes with an area code equal to 615.

* Note that the subquery is always on the right side of a comparison or assigning expression. Also, a subquery can return one or more values. To be precise, the subquery can return the following; One single value (one column and one row), A list of values (one column and multiple rows), a virtual table (multicolumn, multirow, set of values). Also, note that a subquery can return no values at all; it is a NULL. In such cases, the output of the outer query might result in an error or a null empty set, depending on where he subquery is used (in a comparison, an expression, or a table set).

* 8.2-a *WHERE Subqueries*

The most common type of subquery uses an inner SELECT subquery on the right side of a WHERE comparison expression. For example, to

find all products with a price greater than or equal to the average product price, you write the following query :

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

* 8.2-b *IN Subqueries*

There are multiple occurrences of products that contain "saw" in their product descriptions, including saw blades and jigsaws.
In such cases, you need to compare the P_CODE not to one product code (a single value), but to a list of product code values. When you want to compare a single attribute to a list of values, you use the IN operator. When the P_CODE values are not known beforehand, but they can be derived using a query, you must use an IN subquery.

```
SELECT      DISTINCT CUS_CODE, CUS_LNAME, CUS_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%');
```

* 8.2-c *HAVING Subqueries*

Just as you can use subqueries with the WHERE clause, you can use a subquery with a HAVING clause. The HAVING clause is used to restrict the output of a GROUP BY query by applying conditional criteria to the grouped rows. For example, to list all products with a total quantity sold greater than the average quantity sold, you would write the following query

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

* 8.2-d *Multirow Subquery Operators: ANY and ALL*

So far, you have learned that you must use an IN subquery to compare a value to a list of values. However, the IN subquery uses an equality operator; that is, it selects only those rows that are equal to at least one of the values in the list. What happens if you need to make an inequality comparison (> or <) of one value to a list of values

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

* 8.2-e *FROM Subqueries*

So far you have seen how the SELECT statement uses subqueries within WHERE, HAVING, and IN statements, and how the ANY and ALL operators are used for multirow subqueries. In all of those cases, the subquery was part of a conditional expression, and it always appeared at the right side of the expression. But subqueries can also be created in the FROM clause

```
SELECT      DISTINCT CUSTOMER.CUS_CODE, CUSTOMER.CUS_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;
```

* 8.2-f *Attribute List Subqueries*

The SELECT statement uses the attribute list to indicate what columns to project in the resulting set. Those columns can be attributes of base tables, computed attributes, or the result of an aggregate function. The attribute list can also include a subquery expression, also known as an inline subquery. A subquery in the attribute list must return one value; otherwise, an error code is raised. For example, a simple inline query can be used to list the difference between each product's price and the average product price

```
SELECT      P_CODE, P_PRICE, (SELECT AVG(P_PRICE) FROM PRODUCT)
            AS AVGPRICE,
            P_PRICE – (SELECT AVG(P_PRICE) FROM PRODUCT) AS DIFF
FROM        PRODUCT;

SELECT      P_CODE, SALES, ECOUNT, SALES/ECOUNT AS CONTRIB
FROM        (SELECT P_CODE, SUM(LINE_UNITS * LINE_PRICE) AS SALES,
                    (SELECT COUNT(*) FROM EMPLOYEE) AS ECOUNT
            FROM        LINE
            GROUP BY    P_CODE);
```

* In this case, you are actually using two subqueries. The subquery in the FROM clause executes first and returns a virtual table with three columns: P_CODE, SALES, and ECOUNT. The FROM subquery contains an inline subquery that returns the number of employees as ECOUNT. Because the outer query receives the output of the inner query, you can now refer to the columns in the outer subquery by using the column aliases

* 8.2-g *Correlated Subqueries*

A correlated subquery is a subquery that executes once for each row in the outer query. The process is similar to the typical nested loop in a programming language.

1. Compute the average units sold for a product.
2. Compare the average computed in Step 1 to the units sold in each sale row, and then select only the rows in which the number of units sold is greater.

The following correlated query completes the preceding two-step process:

```
SELECT      INV_NUMBER, P_CODE, LINE_UNITS
FROM        LINE LS
WHERE       LS.LINE_UNITS > (SELECT AVG(LINE_UNITS)
                             FROM LINE LA
                             WHERE LA.P_CODE = LS.P_CODE);
```

* 8.3 *SQL Functions*

SQL functions are very useful tools. You'll need to use functions when you want to list all employees ordered by year of birth, or when your marketing department wants you to generate a list of all customers ordered by zip code and the first three digits of their telephone numbers. In both of these cases, you'll need to use data elements that are not present as such in the database. Instead, you will need a SQL function that can be derived from an existing attribute. Functions always use a numerical, date, or string value. The value may be part of the command itself (a constant or literal) or it may be an attribute located in a table. Therefore, a function may appear anywhere in a SQL statement where a value or an attribute can be used

There are many types of SQL functions, such as arithmetic, trigonometric, string, date, and time functions. This section will not explain all of these functions in detail, but it will give you a brief overview of the most useful ones

* 8.4 *Relational Set Operators*

SQL data manipulation commands are set oriented; that is, they operate over entire sets of rows and columns (tables) at once. You can combine two or more sets to create new sets (or relations). That is precisely what the UNION, INTERSECT, and EXCEPT (MINUS) statements do. In relational database terms, you can use the words sets, relations, and tables interchangeably because they all provide a conceptual view of the data set as it is presented to the relational database user.

UNION, INTERSECT, and EXCEPT (MINUS) work properly only if relations are union-compatible, which means that the number of attributes must be the same and their corresponding data types must be alike. In practice, some RDBMS vendors require the data types to be compatible but not exactly the same.

* 8.4-a *UNION*

The UNION statement can be used to unite more than just two queries. For example, assume that you have four union-compatible queries named T1, T2, T3, and T4. With the UNION statement, you can combine the output of all four queries into a single result set. The SQL statement will be similar to this:

```
SELECT column-list FROM T1
UNION
SELECT column-list FROM T2
UNION
SELECT column-list FROM T3
UNION
SELECT column-list FROM T4;
```

* 8.4-b *UNION ALL*

If SaleCo's management wants to know how many customers are on both the CUSTOMER and CUSTOMER_2 lists, a UNION ALL query can be used to produce a relation that retains the duplicate rows. Therefore, the following query will keep all rows from both queries (including the duplicate rows) and return 17 rows

```
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE,
            CUS_PHONE
FROM        CUSTOMER
UNION ALL
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE,
            CUS_PHONE
FROM        CUSTOMER_2;
```

* 8.4-c *INTERSECT*

If SaleCo's management wants to know which customer records are duplicated in the CUSTOMER and CUSTOMER_2 tables, the INTERSECT statement can be used to combine rows from two queries, returning only the rows that appear in both sets.

query INTERSECT *query*

To generate the list of duplicate customer records, you can use the following commands:

```
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE,
            CUS_PHONE
FROM        CUSTOMER
```

```
INTERSECT
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE,
            CUS_PHONE
FROM        CUSTOMER_2;
```

*** 8.4**-d *EXCEPT (MINUS)*

The EXCEPT statement in SQL combines rows from two queries and returns only the rows that appear in the first set but not in the second. The syntax for the EXCEPT statement in MS SQL Server and the MINUS statement in Oracle is

query EXCEPT *query*

and

query MINUS *query*

For example, if the SaleCo managers want to know which customers in the CUSTOMER table are not found in the CUSTOMER_2 table, they can use the following commands in Oracle:

```
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE,
            CUS_PHONE
FROM        CUSTOMER
MINUS
SELECT      CUS_LNAME, CUS_FNAME, CUS_INITIAL, CUS_AREACODE,
            CUS_PHONE
FROM        CUSTOMER_2;
```

*** 8.4**-e *Syntax Alternatives*

If your DBMS does not support the INTERSECT or EXCEPT (MINUS) statements, you can use IN and NOT IN subqueries to obtain similar results. For example, the following query will produce the same results as the INTERSECT query shown in Section 8-4c:

```
SELECT      CUS_CODE FROM CUSTOMER
WHERE       CUS_AREACODE = '615' AND
            CUS_CODE IN (SELECT DISTINCT CUS_CODE FROM INVOICE);
```

*** 8.5** *Virtual Tables: Creating a View*

A view is a virtual table based on a SELECT query. The query can contain columns, computed columns, aliases, and aggregate functions from one or more tables. The tables on which the view is based are called base tables

You can create a view by using the CREATE VIEW command :

CREATE VIEW *viewname* AS SELECT *query*

```
CREATE VIEW PROD_STATS AS
SELECT      V_CODE, SUM(P_QOH*P_PRICE) AS TOTCOST, MAX(P_QOH)
            AS MAXQTY, MIN(P_QOH) AS MINQTY, AVG(P_QOH) AS
            AVGQTY
FROM        PRODUCT
GROUP BY    V_CODE;
```

* Views may also be used as the basis for reports. For example, if you need a report that shows a summary of total product cost and quantity-on-hand statistics grouped by vendor, you could create a PROD_STATS view as:

*** 8.6** *Sequences*

The basic syntax to create a sequence is as follows :

CREATE SEQUENCE *name* [START WITH *n*] [INCREMENT BY *n*] [CACHE | NOCACHE]

* The sequence statement includes….

- name is the name of the sequence.
- n is an integer value that can be positive or negative.
- START WITH specifies the initial sequence value. (The default value is 1.)
- INCREMENT BY determines the value by which the sequence is incremented. (The default increment value is 1. The sequence increment can be positive or negative to enable you to create ascending or descending sequences.)
- The CACHE or NOCACHE/NO CACHE clause indicates whether the DBMS will pre-allocate sequence numbers in memory. Oracle uses NOCACHE as one word and pre-allocates 20 values by default. SQL Server uses NO CACHE as two words. If a cache size is not specified in SQL Server, then the DBMS will determine a default cache size that is not guaranteed to be consistent across different databases

*** 8.7** *Procedural SQL*

To remedy the lack of procedural functionality in SQL and to provide some standardization within the many vendor offerings, the SQL-99 standard defined the use of persistent stored modules. A persistent stored module (PSM) is a block of code containing standard SQL statements and procedural extensions that is stored and executed at the DBMS server.

Procedural Language SQL (PL/SQL) is a language that makes it possible to use and store procedural code and SQL statements within the database and to merge SQL and traditional programming constructs, such as variables, conditional processing (IF-THEN-ELSE), basic loops (FOR and WHILE loops), and error trapping. The procedural code is executed as a unit by the DBMS when it is invoked (directly or indirectly) by the end user.

*** 8.7-a** *Triggers*

A trigger is procedural SQL code that is automatically invoked by the RDBMS upon the occurrence of a given data manipulation event.
It is useful to remember that :

- A trigger is invoked before or after a data row is inserted, updated, or deleted.
- A trigger is associated with a database table.
- Each database table may have one or more triggers.
- A trigger is executed as part of the transaction that triggered it.
- Triggers are critical to proper database operation and management. For example:
  - Triggers can be used to enforce constraints that cannot be enforced at the DBMS design and implementation levels.
  - Triggers add functionality by automating critical actions and providing appropriate warnings and suggestions for remedial action. In fact, one of the most common uses for triggers is to facilitate the enforcement of referential integrity.
  - Triggers can be used to update table values, insert records in tables, and call other stored procedures.

```
CREATE OR REPLACE TRIGGER trigger_name
[BEFORE / AFTER] [DELETE / INSERT / UPDATE OF column_name] ON table_name
[FOR EACH ROW]
[DECLARE]
[variable_namedata type[:=initial_value] ]
BEGIN
PL/SQL instructions;
…
END;
```

* A trigger can have many parts, the above example is from Oracle.

- The *triggering timing*: BEFORE or AFTER. This timing indicates when the trigger's PL/SQL code executes—in this case, before or after the triggering statement is completed.
- The triggering event: The statement that causes the trigger to execute (INSERT, UPDATE, or DELETE).
  - The triggering level: The two types of triggers are statement-level triggers and row level triggers. A statement-level trigger is assumed if you omit the FOR EACH ROW keywords. This type of trigger is executed once, before or after the triggering statement is completed. This is the default case.
  - A row-level trigger requires use of the FOR EACH ROW keywords. This type of trigger is executed once for each row affected by the triggering statement. (In other words, if you update 10 rows, the trigger executes 10 times.)
- The triggering action: The PL/SQL code enclosed between the BEGIN and END keywords. Each statement inside the PL/SQL code must end with a semicolon (;)

* 8.8 *Embedded SQL*

Embedded SQL is a term used to refer to SQL statements contained within an application programming language such as Visual Basic .NET, C#, COBOL, or Java. The program being developed might be a standard binary executable in Windows or Linux, or it might be a web application designed to run over the Internet. No matter what language you use, if it contains embedded SQL statements, it is called the host language. Embedded SQL is still the most common approach to maintaining procedural capabilities in DBMS-based applications.

However, mixing SQL with procedural languages requires that you understand some key differences between the two

- *Run-time mismatch* Remember that SQL is a nonprocedural, interpreted language; that is, each instruction is parsed, its syntax is checked, and it is executed one instruction at a time. (The authors are particularly grateful for the thoughtful comments provided by Emil T. Cipolla.) All of the processing takes place at the server side. Meanwhile, the host language is generally a binary-executable program (also known as a compiled program). The host program typically runs at the client side in its own memory space, which is different from the DBMS environment.
- *Processing mismatch* Conventional programming languages (COBOL, ADA, FORTRAN, Pascal, C++, and PL/I) process one data element at a time. Although you can use arrays to hold data, you still process the array elements one row at a time. This is especially true for file manipulation, where the host language typically manipulates data one record at a time. However, newer programming environments such as Visual Studio .NET have adopted several object-oriented extensions that help the programmer manipulate data sets in a cohesive manner.
- *Data type mismatch* SQL provides several data types, but some of them might not match data types used in different host languages (for example, the DATE and VARCHAR2 data types)