Unit - 9 : Introduction to SQL

Structure of the Unit

9.0 Objective

9.1 Introduction

9.2 What is SQL?

- 9.2.1 Characteristics of SQL

9.3 Types of SQL

- 9.3.1 The Relational database
- 9.3.2 Client/Server Technology
- 9.3.3 Web-Based Database System

9.4 Advantage of SQL

9.5 SQL Data types and Literals

- 9.5.1 String types
- 9.5.2 Numeric types
- 9.5.3 Date and time types

9.6 Large Object Types

- 9.6.1 Numeric Types
- 9.6.2 Decimal Types
- 9.6.3 Integers
- 9.6.4 Floating-Point Decimals
- 9.6.5 Date and Time Types
- 9.6.6 NULL Data Types
- 9.6.7 User-Defined Types

9.7 Types of SQL Commands

- 9.7.1 Data Definition Language (DDL)
- 9.7.2 Data Manipulation Language (DML)
- 9.7.3 Data Query Language (DQL)
- 9.7.4 Data Control Language (DCL)
- 9.7.5 Transactional Control Commands

9.8 Summary

9.9 Self-Assessment Questions

9.0 Objective

At the end of this unit, you should be able to -

- Describe the various types of SQL
- Describe the different data types and use of SQL
- Describe the DML, DML, DQL and DCL commands of SQL
- Describe the data administrations and TCL commands of SQL
- Describe the various queries of SQL to manipulate data

9.1 Introduction

SQL is a relational database data sublanguage. It is not a complete programming language, but depends on the I/O and control facilities of a host language. It is both a dejure and a de facto standard. ANSI (American Nationa Standards Institute) has published three generations of SQL, as has ISO (International Organization for Standardization). X/Open, a consortium, has also published an SQL specification. Chamberlin and Boyce (1974) published the first paper on what became SQL, based on Codd's mathematical foundation for logical representation and manipulation of data (Codd 1974). In 1978, ANSI began to standardize a data definition language for the network database language then being designed by CODASYL; Technical Committee X3H2 was formed for this project, which soon evolved to encompass the entire network database language, published in 1986 as Databae Language NDL. X3H2 recognized the importance of the relational model and intitated a project based on Chamberlin's work. In cooperation with the corresponding ISO group, the SQL specification was developed and published in 1986. SQL-86 omitted support for referential integrity, but SQL-89 added basic referential integrity. SQL-86 and SQL-89 were rightly criticiezed as inadequate for real applications. In 1922, a major new version, SQL-92, was published, containing features that allowed signifcant applications without vendor extensions (ANSI 1992 ; ISO 1992).

SQL has proved key in the success of relational database management systems and is central to many areas, ragning from traditional MIS applications to scientific reserach. The fourth generation of SQL is currently being prepared. SQL 3 adds significant new facilities, including support for object tehcnology, and is partioned into several parts that can progress independently. See Melton and Simon (1993) for a comprehensive introduction to the SQL language.

9.2 What is SQL?

Structured Query Language (SQL) is the standard language used to communicate with a relational database. The prototype was originally developed by IBM using Dr. E.F. Codd's paper ("A Relational Model of Data for Large Shared Data Banks") as a model. In 1979, not long after IBM's prototype, the first SQL product, ORACLE, was released by Relational Software, Incorporated (which was later renamed Oracle Corporation). Today it is one of the distinguished leaders in relational database technologies.

The American National Standards Institute (ANSI) is an organization that approves certain standards in many different industries. SQL has been deemed the standard language in relational database communication, originally approved in 1986 based on IBM's implementation. In 1987, the ANSI SQL standard was accepted as the international standard by the International Standards Organization (ISO). The standard was revised again in 1992 (SQL-92) and once again in 1999 (SQL-99). The newest standard is now called SQL-2008, which was officially adopted in July of 2008.

Characteristics of SQL

SQL databases tend to be mysterious when it comes to the information that is stored within them. The SQL Database Engine is the core service for storing, processing, and extracting data. The SQL Database Engine provides access and rapid transaction processing to meet the requirements of applications.

The SQL Database Engine can be used to create relational databases for online transaction processing or online analytical processing data. It is likely that you have a number of applications that put information into your various SQL database engines. From this information set, you need to be able to retrieve meaningful information and that is where this course comes in.

9.3 Types of SQL

9.3.1 The Relational Database

A relational database is a database divided into logical units called tables, where tables are related to one another within the database. A relational database allows data to be broken down into logical, smaller, manageable units, enabling easier maintenance and providing more optimal database performance according to the level of organization. In Figure 1, you can see that tables are related to one another through a common key (data value) in a relational database.

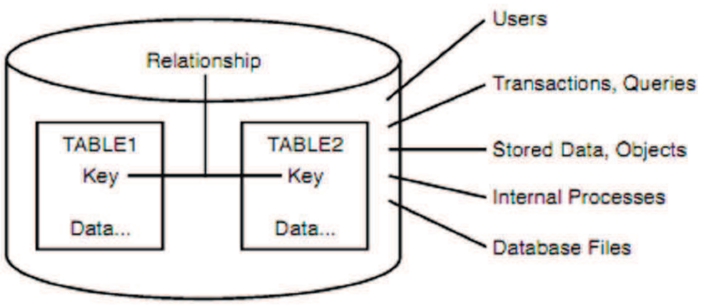


Figure 1. The relational database

Again, tables are related in a relational database, allowing adequate data to be retrieved in a single query (although the desired data may exist in more than one table). By having common keys, or fields, among relational database tables, data from multiple tables can be joined to form one large set of data. As you venture deeper into this book, you see more of a rela- tional database's advantages, including overall performance and easy data access.

9.3.2 Client/Server Technology

In the past, the computer industry was predominately ruled by mainframe computers—large, powerful systems capable of high storage capacity and high data processing capabilities. Users communicated with the mainframe through dumb terminals—terminals that did not think on their own but relied solely on the mainframe's CPU, storage, and memory. Each terminal had a data line attached to the mainframe. The mainframe environment definitely served its purpose and does today in many businesses, but a greater technology was soon to be introduced: the client/server model.

In the client/server system, the main computer, called the server, is accessible from a network—typically a local area network (LAN) or a wide area network (WAN). The server is normally accessed by personal computers (PCs) or by other servers, instead of dumb terminals. Each PC, called a client, is provided access to the network, allowing communication between the client and the server, thus explaining the name client/server. The main difference between client/server and mainframe environments is that the user's PC in a client/server environment is capable of thinking on its own, capable of running its own processes using its own CPU and memory, but readily accessible to a server computer through a network. In most cases, a client/server system is much more flexible for today's overall business needs and is much preferred.

Modern database systems reside on various types of computer systems with various operating systems. The most common types of operating systems are Windows-based systems, Linux, and command-line systems such as UNIX. Databases reside mainly in client/server and web environments. A lack of training and experience is the main reason for failed implementations of database systems. Nevertheless, an understanding of the client/server model and web-based systems, which will be explained in the next section, is imperative with the rising (and sometimes unreasonable) demands placed on today's businesses as well as the development of Internet technologies and network computing. Figure 2 illustrates the concept of client/server technology.

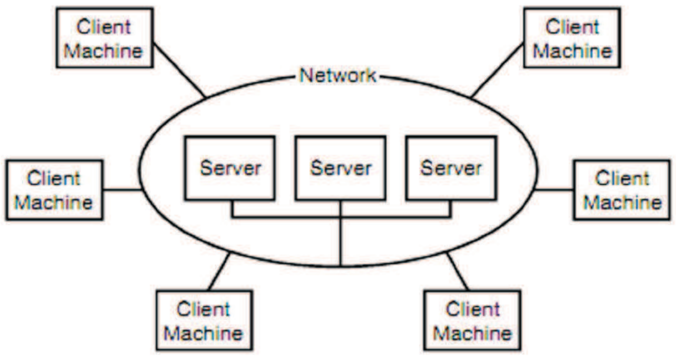


Figure 2. The client/server model.

9.3.3 Web-Based Database System

Business information systems are moving toward web integration. Databases are now accessible through the Internet, meaning that customers' access to an organization's information is enabled through an Internet browser such as Internet Explorer or Firefox. Customers (users of data) are able to order merchandise, check on inventories, check on the status of orders, make administrative changes to accounts, transfer money from one account to another, and so forth. A customer simply invokes an Internet browser, goes to the organization's website, logs in (if required by the organization), and uses an application built into the organization's web page to access data. Most organizations require users to register with them and issue a login and password to the customer. Of course, many things occur behind the scenes when a database is being accessed via a web browser. SQL, for instance, can be executed by the web application. This executed SQL is used to access the organization's database, return data to the web server, and then return that data to the customer's Internet browser.

The basic structure of a web-based database system is similar to that of a client-server system from a user's standpoint (refer to Figure 1.3). Each user has a client machine, which has a connection to the Internet and contains a web browser. The network in Figure 1.3 (in the case of a web-based database) just happens to be the Internet, as opposed to a local network. For the most part, a client is still accessing a server for information. It doesn't matter that the server might exist in another state or even another country. The main point of web-based database systems is to expand the potential customer base of a database system that knows no physical location bounds, thus increasing data availability and an organization's customer base.

9.4 Advantages of SQL

Programming using static SQL requires less effort than using embedded dynamic SQL. Static SQL statements are simply embedded into the host language source file, and the precompiler handles the necessary conversion to database manager run-time services API calls that the host language compiler can process.

Because the authorization of the person binding the application is used, the end user does not require direct privileges to execute the statements in the package. For example, an application could allow a user to update parts of a table without granting an update privilege on the entire table. This can be achieved by restricting the static SQL statements to allow updates only to certain columns or to a range of values.

Static SQL statements are persistent, meaning that the statements last for as long as the package exists.

Dynamic SQL statem ents are cached until they are either invalidated, freed for space management reasons, or the database is shut down. If required, the dynamic SQL statements are recompiled implicitly by the DB2[(R)] SQL compiler whenever a cached statement becomes invalid.

The key advantage of static SQL, with respect to persistence, is that the static statements exist after a particular database is shut down, whereas dynamic SQL statements cease to exist when this occurs. In addition, static SQL does not have to be compiled by the DB2 SQL compiler at run time, while dynamic SQL must be explicitly compiled at run time (for example, by using the PREPARE statement). Because DB2 caches dynamic SQL statements, the statements do not need to be compiled often by DB2, but they must be compiled at least once when you execute the application.

There can be performance advantages to static SQL. For simple, short-running SQL programs, a static SQL statement executes faster than the same statement processed dynamically because the overhead of preparing an executable form of the statement is done at precompile time instead of at run time.

9.5 SQL Data Types and Literals

The following sections discuss the basic data types supported by ANSI SQL. Data types are characteristics of the data itself, whose attributes are placed on fields within a table. For example, you can specify that a field must contain numeric values, disallowing the entering of alphanumeric strings. After all, you would not want to enter alphabetic characters in a field for a dollar amount. Defining each field in the database with a data type eliminates much of the incorrect data found in a database due to data entry errors. Field definition (data type definition) is a form of data validation that controls the type of data that may be entered into each given field. Depending on your implementation of relational database management sys- tem (RDBMS), certain data types can be converted automatically to other data types depending upon their format. This type of conversion in known as an implicit conversion, which means that the database handles the con- version for you. An example of this is taking a numeric value of 1000.92 from a numeric field and inputting it into a string field. Other data types cannot be converted implicitly by the host RDBMS and therefore must undergo an explicit conversion. This usually involves the use of an SQL function, such as CAST or CONVERT. For example

SELECT CAST('12/27/1974' AS DATETIME) AS MYDATE

The very basic data types, as with most other languages, are

- ✓ String types
- ✓ Numeric types
- ✓ Date and time types

Fixed-Length Strings :

Constant characters, those strings that always have the same length, are stored using a fixed-length data type. The following is the standard for an SQL fixed-length character:

Character (n) :

n represents a number identifying the allocated or maximum length of the particular field with this definition.

Some implementations of SQL use the CHAR data type to store fixed-length data. You can store alphanumeric data in this data type. An example of a constant length data type would be for a state abbreviation because all state abbreviations are two characters.

Spaces are normally used to fill extra spots when using a fixed-length data type; if a field's length was set to 10 and data entered filled only 5 places, the remaining 5 spaces would be recorded as spaces. The padding of spaces ensures that each value in a field is a fixed length.

Varying-Length Strings :

SQL supports the use of varying-length strings, strings whose length is not constant for all data. The following is the standard for an SQL varying- length character:

Character Varying (n) :

n represents a number identifying the allocated or maximum length of the particular field with this definition.

Common data types for variable-length character values are the VARCHAR, VARBINARY, and VARCHAR2 data types. VARCHAR is the ANSI standard, which Microsoft SQL Server and MySQL use; Oracle uses both VARCHAR and VARCHAR2. The data stored in a character-defined column can be alphanumeric, which means that the data value may contain numeric characters. VARBINARY is similar to VARCHAR and VARCHAR2 except that it contains a variable length of bytes. Normally, you would use a type such as this to store some kind of digital data such as possibly an image file.

Remember that fixed-length data types typically pad spaces to fill in allocated places not used by the field. The varying-length data type does not work this way. For instance, if the allocated length of a varying-length field is 10, and a string of 5 characters is entered, the total length of that particular value would be only 5. Spaces are not used to fill unused places in a column.

9.6 Large Object Types

Some variable-length data types need to hold longer lengths of data than what is traditionally reserved for a VARCHAR field. The BLOB and TEXT data types are two examples of such data types in modern database implementations. These data types are specifically made to hold large sets of data. The BLOB is a binary large object, so its data is treated as a large binary string (a byte string). A BLOB is especially useful in an implementation that needs to store binary media files in the database, such as images or MP3s. The TEXT data type is a large character string data type that can be treated as a large VARCHAR field. It is often used when an implementation needs to store large sets of character data in the database. An example of this would be storing HTML input from the entries of a blog site. Storing this type of data in the database enables the site to be dynamically updated.

9.6.1 Numeric Types

Numeric values are stored in fields that are defined as some type of number, typically referred to as NUMBER, INTEGER, REAL, DECIMAL, and so on. The following are the standards for SQL numeric values:

- BIT(n)
- BIT VARYING(n)
- DECIMAL(p,s)
- INTEGER
- SMALLINT
- BIGINT
- FLOAT(p,s)
- DOUBLE PRECISION(p,s)
- REAL(s)

p represents a number identifying the allocated or maximum length of the particular field for each appropriate definition.

s is a number to the right of the decimal point, such as 34.ss.

A common numeric data type in SQL implementations is NUMERIC, which accommodates the direction for numeric values provided by ANSI. Numeric values can be stored as zero, positive, negative, fixed, and floating-point numbers. The following is an example using NUMERIC:

NUMERIC(5)

This example restricts the maximum value entered in a particular field to 99999. Note that all the database implementations that we use for the examples support the NUMERIC type but implement it as a DECIMAL.

9.6.2 Decimal Types

Decimal values are numeric values that include the use of a decimal point. The standard for a decimal in SQL follows, where p is the precision and s is the decimal's scale:

DECIMAL(p,s)

The precision is the total length of the numeric value. In a numeric defined DECIMAL(4,2), the precision is 4, which is the total length allocated for a numeric value. The scale is the number of digits to the right of the decimal point. The scale is 2 in the previous DECIMAL(4,2) example. If a value has more places to the right side of the decimal point than the scale allows, the value is rounded; for instance, 34.33 inserted into a DECIMAL(3,1) is typically rounded to 34.3. If a numeric value was defined as the following data type, the maximum value allowed would be 99.99:

DECIMAL(4,2)

The precision is 4, which represents the total length allocated for an associated value. The scale is 2, which represents the number of places, or bytes, reserved to the right side of the decimal point. The decimal point does not count as a character.

Allowed values for a column defined as DECIMAL(4,2) include the following:

- 12
- 12.4
- 2.44
- 12.449

The last numeric value, 12.449, is rounded off to 12.45 upon input into the column. In this case, any numbers between 12.445 and 12.449 would be rounded to 12.45.

9.6.3 Integers

An integer is a numeric value that does not contain a decimal, only whole numbers (both positive and negative).

Valid integers include the following:

- 1
- 0
- –1
- 99
- –99
- 199

9.6.4 Floating-Point Decimals

Floating-point decimals are decimal values whose precision and scale are variable lengths and virtually without limit. Any precision and scale is acceptable. The REAL data type designates a column with single-precision, floating-point numbers. The DOUBLE PRECISION data type designates a column that contains double-precision, floating-point numbers. To be considered a single-precision floating point, the precision must be between 1 and 21 inclusive. To be considered a double-precision floating point, the precision must be between 22 and 53 inclusive. The following are examples of the FLOAT data type:

- FLOAT
- FLOAT(15)
- FLOAT(50)

9.6.5 Date and Time Types

Date and time data types are quite obviously used to keep track of information concerning dates and time. Standard SQL supports what are called DATETIME data types, which include the following specific data types:

- DATE
- TIME
- DATETIME
- TIMESTAMP

The elements of a DATETIME data type consist of the following:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND

9.6.6 NULL Data Types

As you should know from Hour 1, a NULL value is a missing value or a column in a row of data that has not been assigned a value. NULL values are used in nearly all parts of SQL, including the creation of tables, search conditions for queries, and even in literal strings.

The following are two methods for referencing a NULL value:

- **NULL (the keyword NULL itself)**

The following does not represent a NULL value, but a literal string containing the characters N-U-L-L:

'NULL' When using the NULL data type, it is important to realize that data is not required in a particular field. If data is always required for a given field, always use NOT NULL with a data type. If there is a chance that there might not always be data for a field, it is better to use NULL.

9.6.7 User-Defined Types

A user-defined type is a data type that the user defines. User-defined types allow users to customize their own data types to meet data storage needs and are based on existing data types. User-defined data types can assist the developer by providing greater flexibility during database application development because they maximize the number of possibilities for data storage. The CREATE TYPE statement is used to create a user-defined type.

For example, you can create a type as follows in both MySQL and Oracle:

```
CREATE TYPE PERSON AS OBJECT
(NAME      VARCHAR (30),
 SSN       VARCHAR (9));
```

You can reference your user-defined type as follows:

```
CREATE TABLE EMP_PAY
(EMPLOYEE   PERSON,
 SALARY     DECIMAL(10,2),
 HIRE_DATE  DATE);
```

Notice that the data type referenced for the first column EMPLOYEE is PERSON. PERSON is the user-defined type you created in the first example.

9.7 Types of SQL Commands

The following sections discuss the basic categories of commands used in SQL to perform various functions. These functions include building database objects, manipulating objects, populating database tables with data, updating existing data in tables, deleting data, performing database queries, controlling database access, and overall database administration.

The main categories are

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Query Language (DQL)
- Data Control Language (DCL)
- Data administration commands
- Transactional control commands

9.7.1 Data Definition Language (DDL)

Data Definition Language (DDL) is the part of SQL that enables a database user to create and restructure database objects, such as the creation or the deletion of a table.

Some of the most fundamental DDL commands discussed during the following hours include

- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- CREATE INDEX
- ALTER INDEX
- DROP INDEX
- CREATE VIEW
- DROP VIEW

1. **<u>Create Table:</u>** This command is used to create a new table in a database.

The SQL syntax for CREATE TABLE is

```
CREATE TABLE “table_name”
(“column 1” “data_type_for_column_1”,
”column 2" “data_type_for_column_2”,
... )
```

So, if we are to create the customer table specified as above, we would type in

```
CREATE TABLE customer
(First_Name char(50),
Last_Name char(50),
Address char(50),
City char(50),
Country char(25),
Birth_Date date)
```

2. **<u>Alter table:</u>**

The ALTER TABLE statement is used to add or drop columns in an existing table.

```
ALTER TABLE table_name
ADD column_name datatype
```

```
ALTER TABLE table_name
DROP COLUMN column_name
```

Person:

LastName	**FirstName**	**Address**
Pettersen	Kari	Storgt 20

Example

To add a column named "City" in the "Person" table:

```
ALTER TABLE Person ADD City varchar(30)
```

Result:

LastName	**FirstName**	**Address**	**City**
Pettersen	Kari	Storgt 20	

3. **Drop Table:** This command is used to drop a table from a database.

The syntax for **drop table** is

DROP TABLE "table_name"

4. **Create Index:**

Indices are created in an existing table to locate rows more quickly and efficiently. It is possible to create an index on one or more columns of a table, and each index is given a name. The users cannot see the indexes, they are just used to speed up queries.

Note: Updating a table containing indexes takes more time than updating a table without, this is because the indexes also need an update. So, it is a good idea to create indexes only on columns that are often used for a search.

A Unique Index

Creates a unique index on a table. A unique index means that two rows cannot have the same index value.

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name)
```

The "column_name" specifies the column you want indexed.

A Simple Index

Creates a simple index on a table. When the UNIQUE keyword is omitted, duplicate values are allowed.

```
CREATE INDEX index_name
ON table_name (column_name)
```

The "column_name" specifies the column you want indexed.

Example

This example creates a simple index, named "PersonIndex", on the LastName field of the Person table:

```
CREATE INDEX PersonIndex
ON Person (LastName)
```

If you want to index the values in a column in descending order, you can add the reserved word DESC after the column name:

```
CREATE INDEX PersonIndex
ON Person (LastName DESC)
```

If you want to index more than one column you can list the column names within the parentheses, separated by commas:

```
CREATE INDEX PersonIndex
ON Person (LastName, FirstName)
```

5. Drop Index

You can delete an existing index in a table with the DROP INDEX statement.

Syntax for Microsoft SQLJet (and Microsoft Access):

```
DROP INDEX index_name ON table_name
```

Syntax for MS SQL Server:

```
DROP INDEX table_name.index_name
```

Syntax for IBM DB2 and Oracle:

```
DROP INDEX index_name
```

Syntax for MySQL:

```
ALTER TABLE table_name DROP INDEX index_name
```

6. Create View

In SQL, a VIEW is a virtual table based on the result-set of a SELECT statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database. You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from a single table.

Note: The database design and structure will NOT be affected by the functions, where, or join statements in a view.

Syntax

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

Note: The database does not store the view data. The database engine recreates the data, using the view's SELECT statement, every time a user queries a view.

Using Views

A view could be used from inside a query, a stored procedure, or from inside another view. By adding functions, joins, etc., to a view, it allows you to present exactly the data you want to the user. The sample database Northwind has some views installed by default. The view "Current Product List" lists all active products (products that are not discontinued) from the Products table. The view is created with the following SQL:

```
CREATE VIEW [Current Product List] AS
SELECT ProductID,ProductName
FROM Products
WHERE Discontinued=No
```

We can query the view above as follows:

```
SELECT * FROM [Current Product List]
```

Another view from the Northwind sample database selects every product in the Products table that has a unit price that is higher than the average unit price:

```
        CREATE VIEW [Products Above Average Price] AS
        SELECT ProductName,UnitPrice
        FROM Products
WHERE UnitPrice>(SELECT AVG(UnitPrice) FROM Products)
```

We can query the view above as follows:

```
        SELECT * FROM [Products Above Average Price]
```

Another example view from the Northwind database calculates the total sale for each category in 1997. Note that this view select its data from another view called “Product Sales for 1997”:

```
        CREATE VIEW [Category Sales For 1997] AS
        SELECT DISTINCT CategoryName,Sum(ProductSales) AS CategorySales
        FROM [Product Sales for 1997]
        GROUP BY CategoryName
```

We can query the view above as follows:

```
        SELECT * FROM [Category Sales For 1997]
```

We can also add a condition to the query. Now we want to see the total sale only for the category “Beverages”:

```
        SELECT * FROM [Category Sales For 1997]
WHERE CategoryName=’Beverages’
```

9.7.2 Data Manipulation Language (DML)

Data Manipulation Language (DML) is the part of SQL used to manipulate data within objects of a relational database.

The three basic DML commands are

- INSERT
- UPDATE
- DELETE

1. <u>INSERT Statement</u>

The INSERT Statement adds one or more rows to a table. It has two formats:

INSERT INTO table-1 [(column-list)] VALUES (value-list)

and,

INSERT INTO table-1 [(column-list)] (query-specification)

The first form inserts a single row into table-1 and explicitly specifies the column values for the row. The second form uses the result of query-specification to insert one or more rows into table-1. The result rows from the query are the rows added to the insert table.

Both forms have an optional column-list specification. Only the columns listed will be assigned values. Unlisted columns are set to null, so unlisted columns must allow nulls. The values from the VALUES Clause (first form) or the columns from the query-specification rows (second form) are assigned to the corresponding column in column-list in order.

If the optional column-list is missing, the default column list is substituted. The default column list contains all columns in table-1 in the order they were declared in CREATE TABLE, or CREATE VIEW.

VALUES Clause

The VALUES Clause in the INSERT Statement provides a set of values to place in the columns of a new row. It has the following general format:

VALUES (value-1 [, value-2] ...)

value-1 and value-2 are Literal Values or Scalar Expressions involving literals. They can also specify NULL.

The values list in the VALUES clause must match the explicit or implicit column list for INSERT in degree (number of items). They must also match the data type of corresponding column or be convertible to that data type.

INSERT Examples

INSERT INTO p (pno, color) VALUES ('P4', 'Brown')

Before				**After**		
pno	**descr**	**color**		**pno**	**descr**	**color**
P1	Widget	Blue		P1	Widget	Blue
P2	Widget	Red	=>	P2	Widget	Red
P3	Dongle	Green		P3	Dongle	Green
				P4	NULL	Brown

INSERT INTO sp

SELECT s.sno, p.pno, 500

FROM s, p

WHERE p.color='Green' AND s.city='London'

Before				**After**		
sno	**pno**	**qty**		**sno**	**pno**	**qty**
S1	P1	NULL		S1	P1	NULL
S2	P1	200	=>	S2	P1	200
S3	P1	1000		S3	P1	1000
S3	P2	200		S3	P2	200
				S2	P3	500

2. UPDATE Statement

The UPDATE statement modifies columns in selected table rows. It has the following general format:

UPDATE table-1 SET set-list [WHERE predicate]

The optional WHERE Clause has the same format as in the SELECT Statement. See WHERE Clause. The WHERE clause chooses which table rows to update. If it is missing, all rows are in table-1 are updated.

The set-list contains assignments of new values for selected columns.

The SET Clause expressions and WHERE Clause predicate can contain subqueries, but the subqueries cannot reference table-1. This prevents situations where results are dependent on the order of processing.

SET Clause

The SET Clause in the UPDATE Statement updates (assigns new value to) columns in the selected table rows. It has the following general format:

SET column-1 = value-1 [, column-2 = value-2] ...

column-1 and column-2 are columns in the Update table. value-1 and value-2 are expressions that can reference columns from the update table. They also can be the keyword — NULL, to set the column to null.

Since the assignment expressions can reference columns from the current row, the expressions are evaluated first. After the values of all Set expressions have been computed, they are then assigned to the referenced columns. This avoids results dependent on the order of processing.

UPDATE Examples

UPDATE sp SET qty = qty + 20

Before

sno.	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

=>

After

sno.	pno	qty
S1	P1	NULL
S2	P1	220
S3	P1	1020
S3	P2	220

UPDATE s

SET name = 'Tony', city = 'Milan'

WHERE sno = 'S3'

Before

sno	name	city
S1	Pierre	Paris
S2	John	London
S3	Mario	Rome

=>

After

sno	name	city
S1	Pierre	Paris
S2	John	London
S3	Tony	Milan

3. <u>DELETE Statement</u>

The DELETE Statement removes selected rows from a table. It has the following general format:

DELETE FROM table-1 [WHERE predicate]

The optional WHERE Clause has the same format as in the SELECT Statement. See WHERE Clause. The WHERE clause chooses which table rows to delete. If it is missing, all rows are in table-1 are removed.

The WHERE Clause predicate can contain subqueries, but the subqueries cannot reference table-1. This prevents situations where results are dependent on the order of processing.

DELETE Examples

DELETE FROM sp WHERE pno = 'P1'

Before

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

=>

After

sno	pno	qty
S3	P2	200

DELETE FROM p WHERE pno NOT IN (SELECT pno FROM sp)

Before

pno	descr	color
P1	Widget	Blue
P2	Widget	Red
P3	Dongle	Green

=>

After

pno	descr	color
P1	Widget	Blue
P2	Widget	Red

9.7.3 Data Query Language (DQL)

Though comprised of only one command, Data Query Language (DQL) is the most concentrated focus of SQL for modern relational database users. The base command is SELECT.

This command, accompanied by many options and clauses, is used to compose queries against a relational database. A query is an inquiry to the database for information. A query is usually issued to the database through an application interface or via a command-line prompt. You can easily create queries, from simple to complex, from vague to specific.

The SELECT statement is used to select data from a table. The tabular result is stored in a result table (called the result-set).

Syntax

```
        SELECT column_name(s)
FROM table_name
```

Note: SQL statements are not case sensitive. SELECT is the same as select.

SQL SELECT Example

To select the content of columns named "LastName" and "FirstName", from the database table called "Persons", use a SELECT statement like this:

```
        SELECT LastName,FirstName FROM Persons
```

The database table "Persons":

LastName	**FirstName**	**Address**	**City**
Hansen Ola	Timoteivn 10	Sandnes	
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

The result

LastName	**FirstName**
Hansen Ola	
Svendson	Tove
Pettersen	Kari

Select All Columns

To select all columns from the "Persons" table, use a * symbol instead of column names, like this:

```
        SELECT * FROM Persons
```

Result

LastName	**FirstName**	**Address**	**City**
Hansen Ola	Timoteivn 10	Sandnes	
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

9.7.4 Data Control Commands

Data control commands in SQL enable you to control access to data within the database. These Data Control Language (DCL) commands are normally used to create objects related to user access and also control the distribution of privileges among users. Some data control commands are as follows:

- ALTER PASSWORD
- GRANT

- REVOKE
- CREATE SYNONYM

You will find that these commands are often grouped with other commands and might appear in a number of lessons throughout this book. Data Administration Commands Data administration commands enable the user to perform audits and perform analyses on operations within the database. They can also be used to help analyze system performance. Two general data administration commands are as follows:

- START AUDIT
- STOP AUDIT

Do not get data administration confused with database administration. Database administration is the overall administration of a database, which envelops the use of all levels of commands. Data administration is much more specific to each SQL implementation than are those core commands of the SQL language.

SQL-Transaction Statements

SQL-Transaction Statements control transactions in database access. This subset of SQL is also called the Data Control Language for SQL (SQL DCL).

There are 2 SQL-Transaction Statements:

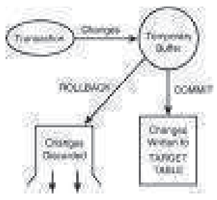
- COMMIT Statement — commit (make persistent) all changes for the current transaction
- ROLLBACK Statement — roll back (rescind) all changes for the current transaction

9.7.5 Transactional Control Commands

Transactional control is the ability to manage various transactions that may occur within a relational database management system. When a transaction is executed and completes successfully, the target table is not immediately changed, although it may appear so according to the output. When a transaction successfully completes, there are transactional control commands that are used to finalize the transaction, either saving the changes made by the transaction to the database or reversing the changes made by the transaction.

There are three commands used to control transactions:

- COMMIT
- ROLLBACK
- SAVEPOINT
- The SET TRANSACTION Command
- Transactional control commands are only used with the DML commands INSERT, UPDATE, and DELETE. For example, you do not issue a COMMIT statement after creating a table. When the table is created, it is automatically committed to the database. Likewise, you cannot issue a ROLLBACK to replenish a table that was just dropped.
- When a transaction has completed, the transactional information is stored either in an allocated area or in a temporary rollback area in the database. All changes are held in this temporary rollback area until a transactional control command is issued. When a transactional control command is issued, changes are either made to the database or discarded; then, the temporary rollback area is emptied. Figure 6.1 illustrates how changes are applied to a relational database.



The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMITcommand saves all transactions to the database since the last COMMIT or ROLLBACK command.

COMMIT [WORK];

The keyword COMMIT is the only mandatory part of the syntax, along with the character or command used to terminate a statement according to each implementation. WORK is a keyword that is completely optional; its only purpose is to make the command more user-friendly.

The ROLLBACK Command

The ROLLBACK command is the transactional control command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for the ROLLBACK command is as follows:

rollback [work];

Once again, the COMMIT statement, the WORK keyword is an optional part of the ROLLBACK syntax.

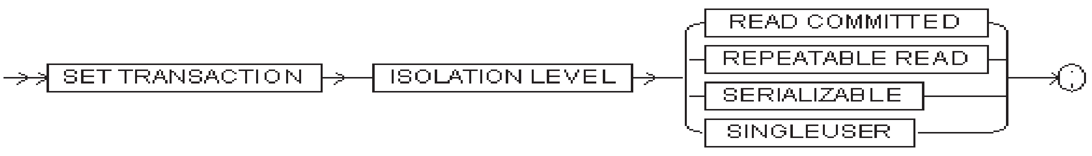
The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for the SAVEPOINT command is

This command serves only in the creation of a SAVEPOINT among transactional statements. The ROLLBACK command is used to undo a group of transactions. The SAVEPOINT is a way of managing transactions by breaking large numbers of transactions into smaller, more manageable groups.

The SET TRANSACTION Command



The SET TRANSACTION Command Establishes the isolation level of the current transaction. If you use a SET TRANSACTION statement, it must be the first statement in your transaction. However, a transaction need not have a SET TRANSACTION statement.

9.8 Summary

SQL (Structured Query Language) is a database sublanguage for querying and modifying relational databases. It was developed by IBM Research in the mid 70's and standardized by ANSI in 1986. SQL is a version of Relational Calculus. The basic structure in SQL in the statement. Semicolons separate multiple SQL statements.

9.9 Self-Assessment Questions

1. How DDL commands are different from DML? Explain with the help of suitable example.
2. Explain the different data types of SQL with the help of example.
3. What do you understand by sub queries? Give example and explain.
4. Explain the use of join in RDBMS with the help of example.
5. Discuss the advantages of SQL over old database software packages.

Unit - 10 : More on SQL

Structure of the Unit

10.0 Objective

10.1 Introduction

10.2 Aggregate Functions

10.3 Group By Clause

10.4 Having Clause

10.5 Order By Clause

10.6 Join

10.6.1 Inner join

10.6.2 Outer join

10.6.3 Self join

10.7 Set Operations

10.7.1 Union

10.7.2 Union All

10.7.3 Intersection

10.7.4 Minus

10.8 Summary

10.9 Self Assessment Questions

10.0 Objective

At the end of this unit, you should be able to -

- Describe the various aggregate functions of SQL
- Describe the join in SQL
- Describe the set operations in SQL

10.1 Introduction

In previous section we have only focused on queries that refer to exactly one table. Furthermore, conditions in a where were restricted to simple comparisons. A major feature of relational databases, however, is to combine (join) tuples stored in different tables in order to display more meaningful and complete information.

10.2 Aggregate Functions

SQL has a lot of built-in functions for counting and calculations.

Function Syntax

The syntax for built-in SQL functions is:

SELECT function(column) FROM table

Aggregate functions operate against a collection of values, but return a single value.

Note : If used among many other expressions in the item list of a SELECT statement, the SELECT must have a GROUP BY clause!!

Five important aggregate functions: SUM, AVG, MAX, MIN, and COUNT.

They are calledaggregate functions because they summarize the results of a query, rather than listing all of the rows.

- SUM () gives the total of all the rows, satisfying any conditions, of the given column, where the givencolumn is numeric.
- AVG () gives the average of the given column.
- MAX () gives the largest figure in the given column.
- MIN () gives the smallest figure in the given column.
- COUNT(*) gives the number of rows satisfying the conditions.

Examples 1 :

```
SELECT SUM(SALARY), AVG(SALARY)
FROM EMPLOYEESTABLE;
```

This query shows the total of all salaries in the table, and the average salary of all of the entries in the table.

```
SELECT MIN(BENEFITS)
FROM EMPLOYEESTABLE
WHERE POSITION = 'Manager';
```

This query gives the smallest figure of the Benefits column, of the employees who are Managers, which is 12500.

```
SELECT COUNT(*)
FROM EMPLOYEESTABLE
WHERE POSITION = 'Staff';
```

This query tells you how many employees have Staff status.

Aggregate functions (like SUM) often need an added GROUP BY functionality.

10.3 GROUP BY Clause

GROUP BY Clause is added to SQL because aggregate functions (like SUM) return the aggregate of all column values every time they are called, and without the GROUP BY function it was impossible to find the sum for each individual group of column values.

The syntax for the GROUP BY function is:

```
SELECT column,SUM(column) FROM table GROUP BY column
```

GROUP BY Example

This "Sales" Table :

Company	Amount
TVS	5500
IBM	4500
TVS	7100

And This SQL :

```
SELECT Company, SUM(Amount) FROM Sales
```

Returns this result :

Company	SUM(Amount)
TVS	17100
IBM	17100
TVS	17100

The above code is invalid because the column returned is not part of an aggregate. A GROUP BY clause will solve this problem:

SELECT Company,SUM(Amount) FROM Sales

GROUP BY Company

Returns this result :

Company	SUM(Amount)
TVS	12600
IBM	4500

10.4 HAVING Clause

Having clause can also be added to SQL because the WHERE keyword could not be used against aggregate functions (like SUM), and without HAVING, it would be impossible to test for result conditions.

The syntax for the HAVING function is:

SELECT column,SUM(column) FROM table

GROUP BY column

HAVING SUM(column) condition value

This "Sales" Table :

Company	Amount
TVS	5500
IBM	4500
TVS	7100

This SQL :

SELECT Company,SUM(Amount) FROM Sales

GROUP BY Company

HAVING SUM(Amount)>10000

Returns this result :

Company	SUM(Amount)
TVS	12600

10.5 ORDER BY Clause

The ORDER BY clause is optional. If used, it must be the last clause in the SELECT statement. The ORDER BY clause requests sorting for the results of a query.

When the ORDER BY clause is missing, the result rows from a query have no defined order (they are *unordered*). The ORDER BY clause defines the ordering of rows based on columns from the SELECT clause. The ORDER BY clause has the following general format:

ORDER BY column-1 [ASC|DESC] [column-2 [ASC|DESC]] ...

column-1, *column-2*, ... are column names specified (or implied) in the select list. If a select column is renamed (given a new name in the select entry), the new name is used in the ORDER BY list. ASC and DESC request ascending or descending sort for a column. ASC is the default.

ORDER BY sorts rows using the ordering columns in left-to-right, major-to-minor order. The rows are sorted first on the first column name in the list. If there are any duplicate values for the first column, the duplicates are sorted on the second column (within the first column sort) in the Order By list, and so on. There is no defined inner ordering for rows that have duplicate values for all Order By columns.

Database *nulls* require special processing in ORDER BY. A *null* column sorts higher than all regular values; this is reversed for DESC.

In sorting, *nulls* are considered duplicates of each other for ORDER BY. Sorting on *hidden* information makes no sense in utilizing the results of a query. This is also why SQL only allows select list columns in ORDER BY.

For convenience when using expressions in the select list, select items can be specified by number (starting with 1). Names and numbers can be intermixed.

Example queries :

SELECT * FROM sp ORDER BY 3 DESC

sno	**pno**	**qty**
S1	P1	NULL
S3	P1	1000
S3	P2	200
S2	P1	200

SELECT name, city FROM s ORDER BY name

name	**city**
John	London
Mario	Rome
Pierre	Paris

SELECT * FROM sp ORDER BY qty DESC, sno

sno	pno	qty
S1	P1	NULL
S3	P1	1000
S2	P1	200
S3	P2	200

Orders table :

Company	OrderNumber
Sega	3412
ABC Shop	5678
TVS	2312
TVS	6798

Example :

To display the company names in alphabetical order:

SELECT Company, OrderNumber FROM Orders

ORDER BY Company ASC (asending)

Result:

Company	OrderNumber
ABC Shop	5678
Sega	3412
TVS	6798
TVS	2312

Example :

To display the company names in alphabetical order AND the OrderNumber in numerical order :

SELECT Company, OrderNumber FROM Orders

ORDER BY Company, OrderNumber

Result:

Company	OrderNumber
ABC Shop	5678
Sega	3412
TVS	2312
TVS	6798

Example :

To display the company names in reverse alphabetical order:

SELECT Company, OrderNumber FROM Orders

ORDER BY Company DESC

Result:

Company	OrderNumber
TVS	6798
TVS	2312
Sega	3412
ABC Shop	5678

Example :

To display the company names in reverse alphabetical order AND the OrderNumber in numerical order:

SELECT Company, OrderNumber FROM Orders

ORDER BY Company DESC, OrderNumber ASC

Result:

Company	OrderNumber
TVS	2312
TVS	6798
Sega	3412
ABC Shop	5678

Notice that there are two equal company names (TVS) in the result above. The only time you will see the second column in ASC order would be when there are duplicated values in the first sort column, or a handful of nulls.The ORDER BY keyword is used to sort the result.

Sort the Rows :

The ORDER BY clause is used to sort the rows.

Orders :

Company	OrderNumber
Sega	3412
ABC Shop	5678
TVS	2312
TVS	6798

Example :

To display the company names in alphabetical order:

SELECT Company, OrderNumber FROM Orders

ORDER BY Company

Result :

Company	OrderNumber
ABC Shop	5678
Sega	3412
TVS	6798
TVS	2312

Example :

To display the company names in alphabetical order AND the OrderNumber in numerical order:

SELECT Company, OrderNumber FROM Orders

ORDER BY Company, OrderNumber

Result :

Company	OrderNumber
ABC Shop	5678
Sega	3412
TVS	2312
TVS	6798

Example :

To display the company names in reverse alphabetical order:

SELECT Company, OrderNumber FROM Orders

ORDER BY Company DESC

Result :

Company	OrderNumber
TVS	6798
TVS	2312
Sega	3412
ABC Shop	5678

Example :

To display the company names in reverse alphabetical order AND the OrderNumber in numerical order:

SELECT Company, OrderNumber FROM Orders

ORDER BY Company DESC, OrderNumber ASC

Result :

Company	OrderNumber
TVS	2312
TVS	6798
Sega	3412
ABC Shop	5678

Notice that there are two equal company names (TVS) in the result above. The only time you will see the second column in ASC order would be when there are duplicated values in the first sort column, or a handful of nulls.

10.6 Join

The FROM clause allows more than 1 table in its list, however simply listing more than one table will very rarely produce the expected results. The rows from one table must be correlated with the rows of the others. This correlation is known as joining.

s Table

sno	name	city
S1	Pierre	Paris
S2	John	London
S3	Mario	Rome

sp Table

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

An example can best illustrate the rationale behind joins. The following query:

SELECT * FROM sp, p

Produces :

sno	pno	qty	pno	descr	color
S1	P1	NULL	P1	Widget	Blue
S1	P1	NULL	P2	Widget	Red
S1	P1	NULL	P3	Dongle	Green
S2	P1	200	P1	Widget	Blue
S2	P1	200	P2	Widget	Red
S2	P1	200	P3	Dongle	Green
S3	P1	1000	P1	Widget	Blue
S3	P1	1000	P2	Widget	Red
S3	P1	1000	P3	Dongle	Green
S3	P2	200	P1	Widget	Blue
S3	P2	200	P2	Widget	Red
S3	P2	200	P3	Dongle	Green

Each row in sp is arbitrarily combined with each row in p, giving 12 result rows (4 rows in sp X 3 rows in p.) This is known as a cartesian product.

A more usable query would correlate the rows from sp with rows from p, for instance matching on the common column — pno:

SELECT *

FROM sp, p

WHERE sp.pno = p.pno

This Produces :

sno	pno	qty	pno	descr	color
S1	P1	NULL	P1	Widget	Blue
S2	P1	200	P1	Widget	Blue
S3	P1	1000	P1	Widget	Blue
S3	P2	200	P2	Widget	Red

Rows for each part in p are combined with rows in sp for the same part by matching on part number (pno). In this query, the WHERE Clause provides the join predicate, matching pno from p with pno from sp.

The join in this example is known as an innerequi-join. equi meaning that the join predicate uses = (equals) to match the join columns. Other types of joins use different comparison operators. For example, a query might use a greater-than join.

The term inner means only rows that match are included. Rows in the first table that have no matching rows in the second table are excluded and vice versa (in the above join, the row in p with pno P3 is not included in the result.) An outer join includes unmatched rows in the result.

More than 2 tables can participate in a join. This is basically just an extension of a 2 table join. 3 tables — a, b, c, might be joined in various ways:

- a joins b which joins c
- a joins b and the join of a and b joins c
- a joins b and a joins c

Plus several other variations. With inner joins, this structure is not explicit. It is implicit in the nature of the join predicates. With outer joins, it is explicit;

This query performs a 3 table join:

SELECT name, qty, descr, color

FROM s, sp, p

WHERE s.sno = sp.sno

AND sp.pno = p.pno

It joins *s* to *sp* and *sp*to *p*, producing :

name	qty	descr	color
Pierre	NULL	Widget	Blue
John	200	Widget	Blue
Mario	1000	Widget	Blue
Mario	200	Widget	Red

Note that the order of tables listed in the FROM clause should have no significance, nor does the order of join predicates in the WHERE clause.

10.6.1 Inner Join

An inner join is a join in which the values in the columns being joined are compared using a comparison operator.

Inner joins can be specified in either the FROM or WHERE clause.. Inner joins specified in the WHERE clause are known as old-style inner joins..

SQL INNER JOIN Syntax

```
SELECT column_name(s)
FROM table_name1
INNER JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

The INNER JOIN keyword return rows when there is at least one match in both tables Let's assume that we have the following two tables,

Table Store_Information

store_name	**Sales**	**Date**
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

Table Geography

region_name	**store_name**
East	Boston
East	New York
West	Los Angeles
West	San Diego

We want to find out sales by store, and we only want to see stores with sales listed in the report. To do this, we can use the following SQL statement using INNER JOIN:

```
SELECT A1.store_name STORE, SUM(A2.Sales) SALES
FROM Geography A1
INNER JOIN Store_Information A2
ON A1.store_name = A2.store_name
GROUP BY A1.store_name
```

Result:

STORE	**SALES**
Los Angeles	$1800
San Diego	$250
Boston	$700

By using INNER JOIN, the result shows 3 stores, even though we are selecting from theGeography table, which has 4 rows. The row "New York" is not selected because it is not present in the Store_Information table.

10.6.2 Outer Join

An inner join excludes rows from either table that don't have a matching row in the other table. An outer join provides the ability to include unmatched rows in the query results. The outer join combines the unmatched row in one of the tables with an artificial row for the other table. This artificial row has all columns set to null.

The outer join is specified in the FROM clause and has the following general format:

table-1 { LEFT | RIGHT | FULL } OUTER JOIN table-2 ON predicate-1

Predicate -1 : is a join predicate for the outer join. It can only reference columns from the joined tables. The LEFT, RIGHT or FULL specifiers give the type of join:

- LEFT — only unmatched rows from the left side table (table-1) are retained
- RIGHT — only unmatched rows from the right side table (table-2) are retained
- FULL — unmatched rows from both tables (table-1 and table-2) are retained

Outer join example:

SELECT pno, descr, color, sno, qty

FROM p LEFT OUTER JOIN sp ON p.pno = sp.pno

pno	descr	color	sno	qty
P1	Widget	Blue	S1	NULL
P1	Widget	Blue	S2	200
P1	Widget	Blue	S3	1000
P2	Widget	Red	S3	200
P3	Dongle	Green	NULL	NULL

10.6.3 Self Join

A query can join a table to itself. Self joins have a number of real world uses. For example, a self join can determine which parts have more than one supplier :

SELECT DISTINCT a.pno

FROM sp a, sp b

WHERE a.pno = b.pno

AND a.sno<>b.sno

As illustrated in the above example, self joins use correlation names to distinguish columns in the select list and where predicate. In this case, the references to the same table are renamed - a and b. Self joins are often used in sub queries.

10.7 Set Operations

Set Operators : Set operators combines results of two queries into a single result. Set operations are generally performed on two Lists obtained from distinct tables.

10.7.1 Union

The UNION command is used to select related information from two tables, much like the JOIN command. However, when using the UNION command all selected columns need to be of the same data type.

Note : With UNION, only distinct values are selected.

SQL Statement 1

UNION

SQL Statement 2

Employees_Norway :

E_ID	E_Name
01	Hansen, Ola
02	Svendson, Tove
03	Svendson, Stephen
04	Pettersen, Kari

Employees_USA :

E_ID	E_Name
01	Turner, Sally
02	Kent, Clark
03	Svendson, Stephen
04	Scott, Stephen

Using the UNION Command

Example :

List all different employee names in Norway and USA:

```
SELECT E_Name FROM Employees_Norway
UNION
SELECT E_Name FROM Employees_USA
```

Result :

E_Name
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Scott, Stephen

Note : This command cannot be used to list all employees in Norway and USA. In the example above we have two employees with equal names, and only one of them is listed. The UNION command only selects distinct values.

10.7.2 Union All

The UNION ALL command is equal to the UNION command, except that UNION ALL selects all values.

```
SQL Statement 1
UNION ALL
SQL Statement 2
```

Using the UNION ALL Command

Example :

List all employees in Norway and USA:

```
SELECT E_Name FROM Employees_Norway
UNION ALL
SELECT E_Name FROM Employees_USA
```

Result :

E_Name
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Svendson, Stephen
Scott, Stephen

10.7.3 Intersection

The Intersect operator is used to return the rows returned by both queries. The following command displays the rows that are common in the results of first and second queries.

```
SELECT                  member_id
FROM                    members
WHERE                   category ='F'
INTERSECT
SELECT DISTINCT         member_id
FROM bookissue
ORDER BY                member_id;
```

```
MEMBER_I
D
--------------
---
2
4
5
7
```

10.7.4 Minus

The Minus operator is used to return rows from the result of the first query that are not available in the result of the second query.

```
SELECT              member_id
FROM                members
WHERE               category ='F'
MINUS
SELECT DISTINCT     member_id
FROM                bookissue
ORDER BY            member_id;
```

pno
P1

MEMBER_ID ------------------
1
6

- Set Operator combine the result of two queries into one
- All set operators have equal precedence.
- When multiple set operators are present in the same query they are evaluated from left to right.
- The datatype of resulting columns should match in both queries.
- The resultant column name would be the column name of first query.

10.8 Summary

Using different aggregate functions, joins and different set operations we can perform more complex queries on DBMS and get desired outputs in different formats.

10.9 Self Assessment Questions

1. What do you understand by aggregate functions in SQL? Explain.
2. Explain the basic difference between where and having clause of select statement.
3. Explain the different set operations of SQL with the help of suitable example.
4. Explain the use of order by clause of select statement.

Unit - 11 : Queries and Subqueries

Structure of the Unit

11.0 Objective

11.1 Introduction

11.2 Queries with join

 11.2.1 Equi Joins

 11.2.2 Cartesian Join

 11.2.3 Outer Join

 11.2.4 Self Join

11.3 Set Operators

 11.3.1 Union

 11.3.2 Intersect

 11.3.3 Minus

11.4 Sub Queries

11.5 Summary

11.6 Self Assessment Questions

11.0 Objective

In general, a *query* is a form of questioning, in a line of inquiry. This unit covers the in depth of SQL query that run on Oracle 10.

- Queries
- Joins
- Sub queries with exists. any, some and all operators.

11.1 Introduction

Tables used in queries :

EMP	Contains information about the employees of the sample company
DEPT	Contains information about the departments in the company

Structure of EMP Table:

EMPNO	NUMBER(4)	Employee number
ENAME	VARCHAR(20)	Employee name
JOB	CHAR (10)	Designation
MGR	NUMBER (4)	Respective manager's EMPNO
HIREDATE	DATE	Date of joining
SAL	NUMBER (9,2)	Basic salary
COMM	NUMBER (7,2)	Commission
DEPTNO	NUMBER (2)	Department number

Structure of DEPT Table :

Column names	**Types**	**Description**
DEPTNO	NUMBER(2)	Department number
DNAME	VARCHAR2 (20)	Name of the department
LOC	VARCHAR2 (10)	Location of the department

11. 2 Queries with join

- Joins are used to combine columns from different tables
- The connection between tables is established through the WHERE clause
- Types of joins: Equi Joins, Cartesian Joins, Outer Joins, Self Joins

One of the most important features of SQL is the ability to define relationships between multiple tables and draw information from them in terms of these relationship, all within a single command. With joins, the information from any number of tables can be related.

To join two tables, the retrieval criteria will typically specify the condition that a column in the first table (which is defined as foreign key) is equal to a column in the second table (which is the primary key referenced by the foreign key) A join's where clause may contain additional conditions. In a join, the table names are listed in the FROM clause, separated by commas.

```
SELECT    < select - list >
FROM      < table1 > < table 2 >,...............< tableN >
WHERE     < table1. column1> = < table2. column 2 and
< table2 column 3 =  < tableN. columnN >
....................................
additional - conditions
```

<u>The variables are defined as follows:-</u>

<select-list> is the set of columns and expressions from **<table1>** through **<tableN>**

<table1> through **<tableN>** are the tables from which column values are retrieved.

<column1> through **<columnN>** are the columns in **<table>** through **<tableN>**

<u>Additional conditions are optional query criteria.:-</u>

We introduce another table, **INCR**, which holds the information about the salary increments of the employee. The structure of the INCR table is:

EMPNO	**NUMBER (4)**
AMT	**NUMBER (7,2)**
DATEINCR	**DATE**

11.2.1 Equi Join

When two tables are joined together using equality of values in one or more columns, they make an **Equi Join.** Table prefixes are utilized to prevent ambiguity and the WHERE clause specifies the columns being joined.

Examples:-

List the employee numbers, names, department numbers and the department name:

```
SELECT empno, ename, emp. deptno, dname FROM emp, dept
WHERE emp. deptno = dept. deptno;
```

Here the deptno column exists in both the tables. To avoid ambiguity, the column name should be qualified with the table name (or with an alias).

Both the table names need to be specifed (emp and dept.) The WHERE clause defines the joining condition i.e., joining the deptno of emp table to the deptno of dept table. Here it checks for the equality values in these columns.

```
EMPNO.      ENAME       EMP. DEPTNO    DNAME
..........  ..........  ...........    ...........
7369        SMITH       20             RESEARCH
7499        ALLEN       30             SALES
7521        WARD        30             SALES
7566        JONES       20             RESEARCH
7654        MARTIN      30             SALES
7698        BLAKE       30             SALES
7782        CLARK       10             ACCOUNTING
7788        SCOTT       20             RESEARCH
7839        KING        10             ACCOUNTING
7844        TURNER      30             SALES
7876        ADAMS       20             RESEARCH
7900        JAMES       30             SALES
7902        FORD        20             RESEARCH
7934        MILLER      10             ACCOUNTING
7945        ALLEN       20             RESEARCH
7526        MARTIN      20             RESEARCH
7985        SCOTT       30             SALES
```

14 rows selected.

Using Table Aliases:-

It can be very tedious to type table mames repeatedly. Temporary labels (or aliases) can be used in the FROM clause. These temporary names are valid only for the current select statement.Table aliases should also specified in the select clause. Table aliases can be up to 30 character in length but the shorter they are the better.

Note: The advantages of using table aliases in that it effectively speeds up the query.

```
SELECT e. empno, e.ename, e.deptno, d.dname FROM emp e, dept d
WHERE e. deptno = d. deptno;
```

Self Learning Exercise :-

1. Why do we need to specify the table prefix when using equi-join ?

11.2.2 Cartesian Join

When no WHERE clause is specified, each row of one table matches every row of the other table. This results in a Cartesian product.

```
SELECT empno, ename,  dname, loc FROM emp, dept;
```

If the umber of rows are 14 and 4 in emp and dept tables respectively, then the total number of rows produced is 56.

Cartesian product is finding out all the possible combination of columns from different tables.

Example:-

Consider the following tables and the data present:

Tab1	: Holds the principal amount.
Tab2	: Holds year and rate of interest.

Tab1	Tab2	
PRINCIPAL	YEAR	RATE
....	....	....
1000	1	10
2000	2	11
3000	3	11.5
	4	12

Finding the possible combinations of calculation of amount, a Cartesian join of the Tab1 and Tab2 is required. The formula for calulation of Amount is Princopal* (1+(rate/100)^{year}

SELECT PRINCIPAL, YAER, RATE, PRINCIPAL* POWER (1+RATE/100), YEAR) FROM TAB1, TAB2;

Will produce the following output

```
PRINCIPAL         YEAR         RATE   PRINCIPAL* POWER (1+RATE/100), YEAR)
....................  ............  ..........  ..........................................................................
1000              1            10                                     1100
2000              1            10                                     2200
3000              1            10                                     3300
1000              2            11                                     1232.1
2000              2            11                                     2464.2
3000              2            11                                     3696.3
1000              3            11.5                                   2772.1959
2000              3            11.5                                   2772.3918
3000              3            11.5                                   4158.5876
1000              4            12                                     1573.5194
2000              4            12                                     3147.0387
3000              4            12                                     4720.5581
```

11.2.3 Outer Join

If there are any values in one table that do not have corresponding values (s) in the other, in an equi join that row will not be selected. Such rows can be forcefully selected by using the outer join symbol (+) The corresponding columns for that row will have NULLs.

Example:-

In the emp table, no record of the employees belonging to the department 40 is present. Therefore, in the example above for equi join, the row of department 40 from the dept table will not be displayed. Dispaly the list of the employees working in each department Display the department information even if no empployee belongs to that department:

SELECT empno, ename, emp. deptno, dnamo, loc FROM emp, dept

WHERE emp. deptno (+) = demp. deptno;

```
EMPNO            ENAME            EMP. DEPTNDNAME
................ ............... ........................ ..................
                 -
7369             SMITH            20               RESEARCH
7499             ALLEN            30               SALES
7521             WARD             30               SALES
7566             JONES            20               RESEARCH
7654             MARTIN           30               SALES
7698             BLAKE            30               SALES
7782             CLARE            10               ACCOUNTING
7788             SCOTT            20               RESEARCH
7839             KING             10               ACCOUNTING
7844             TURNER           30               SALES
7876             ADAMS            20               RESEARCH
7900             JAMES            30               SALES
7902             FORD             20               RESEARCH
7934             MILLER           10               ACCOUNTING
7945             ALLEN            20               RESEARCH
7526             MARTIN           20               RESEARCH
7985             SCOTT            30               SALES
                                  40               OPERATIONS
14 rows selected.
```

If the symbol (+) is placed on the other side of the equaticn then all the employee details with no corresponding department name and location, will be displayed with NULL values in DNAME and LOC column.

Rules to place (+) operator:-

- The outer join symbol (+) can not be on both the side.
- We can not "outer join" the some table to more than one other table in a single SELECT statement.
- A condition involving an outer join may not use the IN operator or be linked to another condition by the OR operator.

11.2.4 Self Join

To join a table to itself means taht each row of the table is combined with itself and with every other row of the table. The self join can be viewed as a join of two copies of the same table. The table is not actually copied, but SQL performs the command as though it were.

The syntax of the command for joining a table to itself is almost the same as that for joinning two different table. To distinguish the column names from one another, aliases for the actual the table name are used, since both the table have the same name. Table name aliases are defined in the FROM

clause of the query.

To define the alias, one space is left after the table name and the alias.

Example:-

*** EMP TABLE**

EMPNO	ENAME	MGR
7839	KING	
7566	JONES	7839
7876	ADAMS	7788
7934	MILLER	7782
....	....	....

Consider the emp table shown above. Primary key of the emp table is empno. Details of each employee's manager is just another row in the EMP table whose EMPNO is stored in MGR column of some other row. So every employee except manager has a Manager. Therefore MGR is a foreign key taht reference empno. To list out the names of the manager with the employee record one will have to join EMP with ltself.

SELECT WORKER. ename, MANAGER. ename 'Manager'

FROM emp WORKER, emp MANAGER

WHERE WORKER. mgr = MANAGER. empno;

Where WORKER and MANAGER are two aliases for the EMP table and as a virtual

EMP TABLE

EMPNO	ENAME	MGR
7839	KING	
7566	JONES	7839
7876	ADAMS	7788
7934	MILLER	7782
....	....	....

WORKER

EMPNO	ENAME	MGR
7839	KING	
7566	JONES	7839
7876	ADAMS	7788
7934	MILLER	7782
....	....	....

MANAGER

EMPNO	ENAME	MGR
7839	KING	
7788	SCOTT	7566
7782	CLARK	7839
7934	MILLER	7782
....	....	....

The output will be:

```
ENAME                   MANAGER
........................ ..................
SCOTT                   JONES
FORD                    JONES
ALLEN                   BLAKE
WARD                    BLAKE
JAMES                   BLAKE
TURNER                  BLAKE
MARTIN                  BLAKE
MILLER                  CLARK
ADAMS                   SCOTT
JONES                   KING
CLARK                   KING
BLAKE                   KING
SMITH                   FORD
13 rows selected.
```

Self Learning Exercise:-

2. In the previous query, only 13 rows (Not 14) have been retrieved. Why ?
3. List all employees who joined the company before their manager.

```
SELECT e. ename, e.hiredate, m.ename manager, m.hiredate
FROM emp e, emp m
WHERE e.mgr = m.empno
and e. hiredate < m.hiredate;
```

```
ename        hiredate     manager      hiredate
............ ................ ................ ...............
ALLEN        15-AUG-83    BLAKE        11-JUN-84
WARD         26-MAR-84    BLAKE        11-JUN-84
MARTIN       05-DEC-83    BLAKE        11-JUN-84
TURNER       04-JUN-84    BLAKE        11-JUN-84
MILLER       21-NOV-83    CLARK        14-MAY-84
JONES        31-OCT-83    KING         09-JUN-84
BLAKE        11-JUN-84    KING         09-JUL-84
CLARK        14-MAY-84    KING         09-JUL-84
SMITH        13-JUN-83    FORD         05-DEC-83
```

11.3 Set Operators

- SET Operator are used to combine information of similar type from one or more than one table.
- Datatype of correponding columns must be the same
- The types of SET operator in ORACLE are :

UNION	: Rows of first query plus rows of second query, less duplicate rows
INTERSECT	: Common rows from all the queries
MINUS	: Rows unique to the first query

SET operator combine two or more queries into one result.

Suppose we want following three details from dept table

- List of all the different designations in department 20 and 30
- List the jobs common to department 20 and 30
- List the jobs unique to department 20 :

To get these combination of information the SET operators UNION, INTERSECT and MINUS are used.

11.3.1 Union

The UNION clause merges the outputs of two of more queries into a single set of rows and columns.

The syntax of UNION operator is

```
select <stmt1>
union
select <stmt2>
{order-by-clause}
```

The variables are defined as follows:

select stmt1 and select stmt 2 are valid **SELECT** statement.

order-by-clause is optional and it references the columns by number rather than by name.

The queries are all executed independently, but their output is merged. Only the final query ends with a semicolon.

Examples:-

- Dispaly the different designations in department 20 and 30:

```
SELECT job FROM emp
WHERE deptno = 20
UNION
SELECT job FROM emp
WHERE deptno = 30;
```

The output will be:

```
JOB
.......................
CLERK
SALESMAN
MANAGER
ANALYST
```

Points to be kept in mind while using UNION operator.

- The two select statement may not contain an ORDER BY clause; hawever, the final result of the entire UNION operation can be ordered.
- The number of columns retrieved by first select must be equal to number of columns retrieved by second select.
- The date types of columns retrieved by the select statements should be same.
- The optional order by clause differs from the usual ORDER BY clause an a SELECT statement because the columns used for ordering must be reference by a number rather than name. The reason that the columns must be referenced by number is the SQL does not require that the column name retrieved by first select be identical to theolumns names retrieved by second select.

Example:-

```
select empno, ename from emp
where deptno = 10
UNION
select empno, ename from emp
where deptno = 30
order by 1;
EMPNO           ENAME
...............  ...............
7499            ALLEN
7521            WARD
7654            MARTIN
7698            BLAKE
7839            KING
7844            TURNER
7900            JAMES
7934            MILLER
```

11.3.2 Intersect

The intersect operator returns the rows that are common between two sets of rows.

The syntax of INTERSECT operator is same as UNION operator. Only UNION key word is replaced by INTERSECT.

```
select stmt1
INTERSECT
select stmt2
{order-by clause}
```

Example:-

List the jobs common to department 20 and 30:

SELECT job FROM emp WHERE deptno = 20

```
INTERSECT
SELECT job FROM emp WHERE deptno = 30,
Job
...........
CLERK
MANAGER
```

11.3.3 Minus

Minus operator returns the rows unique to first query. The syntax using the MINUS operator resembles the syntax for the union operator:

```
select stmt1
MINUS
select stmt2
{order-by clause}
```

The requirements and considerations for using the MINUS operator are essentially the same as those for the INTERSECT and UNION operator. To illustrate the use of the MINUS operator, consider the following example.

List the jobs unique to department 20:

```
SELECT job FROM emp
WHERE deptno = 20
MINUS
SELECT job FROM emp
WHERE deptno = 10
MINUS
SELECT job FROM emp
WHERE deptno = 30;
          Job
          ...........
          ANALYST
```

Classroom Exercise:-

Can we rewrite the query to find jobs that are unique to department 20 as:

```
SELECT job FROM emp WHERE deptno = 20
MINUS
SELECT job FROM emp WHERE deptno IN (10, 30);
```

11.4 Sub Queries

- The result of inner query is dynamically substituted in the condition of outer query
- There is no practical limitation to the level of nesting of queries in Oracle 9
- When using relational operators, ensure that the sub query returns a single column output
- In some cases, the DISTINCT clause cab be used to ensure single valued output

SQL has an ability to nest queries within one another. A subquery is a SELECT statement that is nested within another SELECT statement and which returns intermediate results. SQL frist evaluates the inner query (or sub query) within the WHERE clause. The inner query generates values that are tested in the predict of the outer query, determining when it will be true. The return value of inner query is then substituted in the condition of the outer query.

Advantages of Nested queries

- Subqueries allows a developer to build powerful commands out of simple ones.
- The nested subquery is very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

Example:-

List the employees belonging to the department of MILLER:

Here we do not know the department to which MILLER belongs. So, we have to determine the epartment of MILLER and use that department number to find out the other employees of that department.

```
SELECT deptno FROM emp
WHERE ename = 'MILLER';
DEPTNO
....................
10
SELECT ename FROM emp
WHERE deptno = 10;
        ENAME
        ...........
        KING
        CLARK
        MILLER
```

Combining the above two queries:

```
SELECT ename FROM emp
WHERE deptno = (SELECT deptno FROM emp WHERE ename = 'MILLER');
```

* list the names of the employee drawing the highest salary:

```
SELECT ename FROM emp
WHERE sal = (SELECT MAX (sal) FROM emp),;
```

Using Aggregate Functions In Subqueries

Aggregate function produces single value for any number of rows. We want to see all employee details whose salary is greater than avarege salary of employees whose hiredata is before'01-04-81' For this we need to use aggregate function in inner query.

```
SELECT * from emp
where sal >
        (select avg (sal) from emp
        where hiredata < '01-APR- 81');
```

Subqueries in Having

We can also use subqueries within the Having clause. These subqueries can use their own aggregate functions as long as do not produce multiple values or use GROUP BY or HAVING themselves.

List the employee number, name, total number of increments and total increments amount for the employee who has got maximum number of increments:

```
SELECT incr. empno, ename, COUNT (*), SUM (amt) FROM emp, incr
WHERE incr. empno = emp. empno
GROUP BY incr. empno, ename
HAVING COUNT (*) =        (SELECT MAX (COUNT (*) from incr
                          GROUP BY empno);
```

The output will be:

```
EMPNO        ENAME        COUNT(*)     SUM (AMT)
...............  ...............  ....................  ......................
7369         SMITH        3            500
```

List the job with highest average salary.

```
SELECT job, AVG (sal)
FROM emp
GROUP BY job
HAVING AVG (sal)    =      (SELECT MAX (AVG (sal)
                           FROM emp
                           GROUP BY job);
```

The output will be:

```
JOB              AVG (SAL)
..........       ....................
PRESIDENT        5000
```

The inner query first finds the average salary for each different job group, and the MAX function picks the highest average salary. That value (5000) is used in the HAVING clause.The GROUP BY clause in the main query is needed because the main query's SELECT list contains both an aggregate and non-aggregate column.

Distinct Clause with Subqueries

Distinct clause is used in some cases to force a subquery to generate a single value. Suppose we want to find the details of the department whose manager's empcode '7698'.The query for this is shown below.

```
select * from dept
where deptno = (select distinct deptno from emp where mgr = '7698');
```

The inner query will give deptno whose manager's empcode is '7698' Without distinct clause the inner query would have returned more than one row as there are more than one employee whose manager's empcode is '7698'.

Subqueries that return more than one row

When a query returns more than one row we need to use multirow comparision operator.

Example:-

List the names of the employees, who have got an increment:

```
SELECT ename FROM emp
WHERE empno IN (SELECT empno FROM incr);
```

Here, the inner query returns multiple values, hence the IN operator is used instead of a relational operator.

List the names of the employees, who earn lowest salary in each department:

```
SELECT ename, sal, deptno FROM emp
WHERE sal IN (SELECT MIN (sal) FROM emp GROUP BY deptno);
```

Here the inner query has a GROUP BY clause. This means it may return more than one value. In this case, the IN operator must be used because it expects a list of values.

The following points should be kept in mind while writting subqueries:

1. The inner query must be enclosed in parentheses.
2. The inner query must on the right hand side of the condition.
3. The subquery may not have an order by clause.
4. The ORDER BY clause appears at the end of the main select statement.
5. Subqueries are always executed from the most deeply nested to the least deeply nested, unless they are correlated subqueries.

Correlated Subquery

A correlated subquery is a nested subquery which is executed once for each 'candidate row' considered by the main query and which executed uses a value from a column in the outer query. In a correlated subquery, the column value used in inner sub query refers to the column value present in the outher query forming a subquery. The subquery is executed repeatedly once for each row of the main (outer) query table.

List the employee numbers and names, who have got more than 1 increments:

```
SELECT empno ename FROM emp
WHERE 1 <
        (SELECT COUNT (*) FROM incr
        WHERE empno = emp. empno);
```

```
EMPNO        ENAME
.............  ...............
7369         SMITH
7788         SCOTT
7900         JAMES
7934         MILLER
```

List employee details who earn salary greater than the average salary for their department.

```
SELECT empno, ename, sal, deptno
FROM emp e
WHERE  sal > (select AVG (sal) FROM emp WHERE deptno = e. deptno);
```

```
EMPNO        ENAME        SAL         DEPTNO
.............  ................  ..........  ..................
7839         KING         5000        10
7566         JONES        2975        20
7788         SCOTT        3000        20
7902         FORD         1600        30
7698         BLAKE        2850        30
5 rows selected.
```

Remember, a correlated subquery is signalled by a column name, a table name or table alias in the WHERE clause that refers to the value of a column in each candidate row ofthe outer select. Also the correlated subquery executes repeatedly for each candidate row in the main query. Correlated subquery is used to answer multipart questions whose answer depends on the value of each row of the parent query. The inner select is normaly executed once for each candidate row.

Self Learning Exercise:-

4. How are nested queries different from joined queries ?

Using Special Operators in Subqureies:-

Some Special operators used in subqueries are:

EXISTS

ANY

SOME

ALL Operators

EXISTS

This operator is used to check the existence of values

This operator produces a Boolean result

It takes a subquery as an argument and evaluates it to True, if it produces any output or

False, if it does not

ANY, SOME and ALL

Used along with the relational operators

Similar to IN operator, but only used in subqueries

The SOME and ANY operator can be used interchangeably

EXISTS operator

The EXISTS operator is frequenly used with correlated subqueries. It tests whether a value is there (NOT EXISTS ensure for nonexistence of values.) If the value exists it returns TRUE, if it does not exists it returns FALSE.

NOT EXISTS operator is more relible if the subquery returns any NULL values.

Examples:-

List all employee who have atleast one person reporting to them.

```
SELECT empno, ename, job, deptno
FROM emp e
```

```
        WHERE EXISTS    (SELECT empno from emp
                        WHERE emp. mgr = e. empno)
        ORDER BY empno;
empno       ename       job             deptno
..........  ..........  ........        ...........
7566        JONES       MANAGER         20
7698        BLAKE       MANAGER         30
7782        CLARK       MANAGER         10
7788        SCOTT       ANALYST         20
7839        KING        PRESISENT       10
7902        FORD        ANALYST         20
```

list the employee details if and only if more than 10 employees are present in department number 10:

```
SELECT * FROM  emp
WHERE DEPTNO = 10 AND EXISTS (SELECT COUNT (*) FROM emp
                        WHERE deptno = 10
                        GROUP BY deptno
                        HAVING COUNT (*) > 10);
```

list the name of employee from the employee table where the increment amount is greater than 1000 and the number of employees receiving the same increment is greater than 5:

```
SELECT  ename FROM emo
WHERE empno IN    (SELECT empno FROM incr
                  WHERE amt > 1000
                  AND EXISTS (SELECT COUNT (*)
                        FROM incr GROUP BY amt
                        HAVING count (*) > 5));
```

List all the employees datails who do not manage any one.

```
SELECT  ename, job from emp e
where not exists (select mgr frm emp where mgr = e. empno);
```

The output is

```
        ename       job
        ..........  .......
        SMITH       CLERK
        ADAMS       CLERK
        ALLEN       SALESMAN
        WARD        SALESMAN
        MARTIN      SELESMAN
        TURNER      SELESMAN
        JAMES       CLERK
        MILLER      CLERK
```

Self Learning Exercise

5. What will be the output if we use NOT IN operator instead of NOT EXISTS in the above query?

ANY operator

The ANY operator compares the lowest value from the set.

List the employee names whose salary is greater than the lowest of an employeebelonging to department number 20:

SELECT ename FROM emp

WHERE sal > ANY (SELECT sal FROM emp WHERE deptno = 21);

List the employee details of those employees whose salary is greater than any of the managers:

SELECT EMPNO, ENAME SAL FROM EMP WHERE SAL > ANY (SELECT SAL FROM EMP WHERE JOB = 'MANAGER');

ALL Operator

In case of All operator the predicate is true if every value selected by the subquery satisfies the condition in the predicate of the outer query.

Example:-

List the employee names whose salary is greater than the highest salary of all employee belonging to department number 20:

SELECT ename FROM emp

WHERE sal > ALL (SELECT sal FROM emp WHERE deptno = 20);

The inner query return salary of all employees who belong to department number 20. The outer query selects employee name of that employee whose salary is greater than all the employees' salary who belong to department number 20.

List the details of the employee earning more than the highest paid MANAGER:

SELECT empno, ename, sal FROM emp WHERE sal > ALL (SELECT sal FROM emp WHERE job = 'MANAGER');

11.5 Summary

- Joins are used to combine columns from different tables.
- The different types of joins are: equi joins, cartesian joins, outer joins, self joins and nonequi joins.
- Joing condition is specified in the WHERE clause of the SELECT statement.
- When two tables are joined together using equality of values in one or more columns, they make an **Equi Join.**

1. Without any joining condition the join becomes a cartesian join.
2. Join a table with itself is know is self join.
3. Self Join is possible by providing table name aliases for the table.
4. With joins, the names of all the table to be joined, need to be specified.
5. To select a row forcfully which cannot be selected using equi join outer join symbol (+). is used.
6. Set operator are used to combine result form different queries. the operator used are UNION, INTERSECT and MINUS.
7. The UNION clause merges the outputs of two or more queries into a single set of rows and columms.