Sql(Structured Query Language)

Introduction

- SQL is a standard language for accessing databases.
- SQL lets you access and manipulate databases.
- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

1.MySQL Data Types

In MySQL there are three main types : text, number, and Date/Time types.

Text types:

Data type	Description
CHAR(size)	Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. Can store up to 255 characters
VARCHAR(size)	Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. Can store up to 255 characters. **Note:** If you put a greater value than 255 it will be converted to a TEXT type
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT	Holds a string with a maximum length of 65,535 characters
BLOB	For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data
ENUM(x,y,z,etc.)	Let you enter a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. **Note:** The values are sorted in the order you enter them. You enter the possible values in this format: ENUM('X','Y','Z')

SET	Similar to ENUM except that SET may contain up to 64 list items and can store more than one choice

Number types:

Data type	Description
TINYINT(size)	-128 to 127 normal. 0 to 255 UNSIGNED*. The maximum number of digits may be specified in parenthesis
SMALLINT(size)	-32768 to 32767 normal. 0 to 65535 UNSIGNED*. The maximum number of digits may be specified in parenthesis
MEDIUMINT(size)	-8388608 to 8388607 normal. 0 to 16777215 UNSIGNED*. The maximum number of digits may be specified in parenthesis
INT(size)	-2147483648 to 2147483647 normal. 0 to 4294967295 UNSIGNED*. The maximum number of digits may be specified in parenthesis
BIGINT(size)	-9223372036854775808 to 9223372036854775807 normal. 0 to 18446744073709551615 UNSIGNED*. The maximum number of digits may be specified in parenthesis
FLOAT(size,d)	A small number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter
DOUBLE(size,d)	A large number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter
DECIMAL(size,d)	A DOUBLE stored as a string , allowing for a fixed decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter

*The integer types have an extra option called UNSIGNED. Normally, the integer goes from an negative to positive value. Adding the UNSIGNED attribute will move that range up so it starts at zero instead of a negative number.

Date types:

Data type	Description
DATE()	A date. Format: YYYY-MM-DD **Note:** The supported range is from '1000-01-01' to '9999-12-31'
DATETIME()	*A date and time combination. Format: YYYY-MM-DD HH:MI:SS **Note:** The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'
TIMESTAMP()	*A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD HH:MI:SS

	Note: The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC
TIME()	A time. Format: HH:MI:SS **Note:** The supported range is from '-838:59:59' to '838:59:59'
YEAR()	A year in two-digit or four-digit format. **Note:** Values allowed in four-digit format: 1901 to 2155. Values allowed in two-digit format: 70 to 69, representing years from 1970 to 2069

2. Some of The Most Important SQL Commands

CREATE DATABASE - creates a new database

```
create database databasename;
```

CREATE TABLE - creates a new table

```
CREATE TABLE table_name
  (
 column_name1 data_type(size),
column_name2 data_type(size),
column_name3 data_type(size),
   ....
);
```

The column_name parameters specify the names of the columns of the table.
The data_type parameter specifies what type of data the column can hold (e.g. varchar, integer, decimal, date, etc.).
The size parameter specifies the maximum length of the column of the table.

Example:

```
CREATE TABLE suppliers
( supplier_id number(10) NOT NULL,
  supplier_name varchar2(50) NOT NULL,
 contact_name varchar2(50));
```

The ALTER TABLE Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

SQL ALTER TABLE Syntax

To add the column: To add a column in a table, use the following syntax:

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

Example:

```
ALTER TABLE supplier
  ADD (supplier_name varchar2(50),
       city varchar2(45));
```

To delete the column: To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

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

Example:

```
ALTER TABLE supplier
  DROP COLUMN supplier_name;
```

To modify the column: To change the data type of a column in a table, use the following syntax:

SQL Server / MS Access:

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype
```

My SQL / Oracle (prior version 10G):

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype
```

Example:

```
ALTER TABLE supplier
  MODIFY supplier_name varchar2(100) not null;
```

Oracle 10G and later:

```
ALTER TABLE table_name
MODIFY column_name datatype
```

Rename column in table

Syntax

To rename a column in an existing table, the SQL ALTER TABLE syntax is:

```
ALTER TABLE table_name
  RENAME COLUMN old_name to new_name;
```

For example:

```
ALTER TABLE supplier
  RENAME COLUMN supplier_name to sname;
```

The DROP TABLE Statement

The DROP TABLE statement is used to delete a table.

```
Drop  table tablename;
```

The DROP DATABASE Statement

The DROP DATABASE statement is used to delete a database.

```
DROP DATABASE database_name
```

The TRUNCATE TABLE Statement

What if we only want to delete the data inside the table, and not the table itself?

Then, use the TRUNCATE TABLE statement:

```
TRUNCATE TABLE table_name
```

The SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

SQL INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two forms.

The first form does not specify the column names where the data will be inserted, only their values:

```
INSERT INTO table_name
```

```
VALUES (value1,value2,value3,...);
```

Example:

```
INSERT INTO suppliers
(supplier_id, supplier_name)
VALUES
(24553, 'IBM');
```

The SQL SELECT Statement

The SELECT statement is used to select data from a database.

The result is stored in a result table, called the result-set.

SQL SELECT Syntax

```
SELECT column_name,column_name
FROM table_name;
```

and

```
SELECT * FROM table_name;
```

Example:

```
SELECT supplier_name, city, state
FROM suppliers
```

The SQL WHERE Clause

The WHERE clause is used to extract only those records that fulfill a specified criterion.

SQL WHERE Syntax

```
SELECT column_name,column_name
FROM table_name
WHERE column_name operator value;
```

Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

Operator	Description
=	Equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
>	Greater than
<	Less than

>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

The SQL AND & OR Operators

The AND operator displays a record if both the first condition AND the second condition are true.

The OR operator displays a record if either the first condition OR the second condition is true.

```
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 AND age < 25;
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 AND age is less tan 25 years.

```
 SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000 OR age < 25;
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 OR age is less tan 25 years.

SQL LIKE Operator:

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern;
```

3. WildCards Used In Like Operator:

SQL Wildcard Characters

In SQL, wildcard characters are used with the SQL LIKE operator.

SQL wildcards are used to search for data within a table.

With SQL, the wildcards are:

Wildcard	Description

%	A substitute for zero or more characters
_	A substitute for a single character
[*charlist*]	Sets and ranges of characters to match
[^*charlist*] or [!*charlist*]	Matches only a character NOT specified within the brackets

```
SELECT
employeeNumber, lastName, firstName
FROM
employees
WHERE
firstName LIKE 'a%';
```

MySQL scans the whole employees table to find employee whose first name starts with character a and followed by any number of characters.

To search for employee whose last name ends with on e.g., Patterson, Thompson, you can use the % wildcard at the beginning of the pattern as the following query:

```
SELECT
employeeNumber, lastName, firstName
FROM
employees
WHERE
lastName LIKE '%on';
```

To find all employees whose last names contain on string, you use the following query with pattern %on %

```
SELECT
employeeNumber, lastName, firstName
FROM
employees
WHERE
lastname LIKE '%on%';
```

MySQL LIKE with underscore(_) wildcard

To find employee whose first name starts with T, ends with m and contains any single character between e.g., Tom , Tim, you use the underscore wildcard to construct the pattern as follows:

```
SELECT
employeeNumber, lastName, firstName
FROM
employees
WHERE
firstname LIKE 'T_m';
```

The **[**charlist**]** WILDCARDS are used to represent any single character within a charlist.

To get all rows from the table 'agents' with following condition -

the 'agent_name' must begin with the letter 'a' or 'b' or 'i'

the following sql statement can be used :

```
SELECT*
FROM agents
WHERE agent_name LIKE '[abi]%';
```

The SQL UPDATE Statement

The UPDATE statement is used to update existing records in a table.

SQL UPDATE Syntax

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE some_column=some_value;
```

Example

```
UPDATE customers
SET last_name = 'Anderson'
WHERE customer_id = 5000;
```

SQL SELECT DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values.

SQL SELECT DISTINCT Syntax

```
SELECT DISTINCT column_name,column_name
FROM table_name;
```

The following SQL statement selects only the distinct values from the "City" columns from the "Customers" table:

Example

```
SELECT DISTINCT City FROM Customers;
```

The SQL SELECT TOP Clause

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause can be very useful on large tables with thousands of records. Returning a large number of records can impact on performance.

```
SELECT TOP number|percent column_name(s)
FROM table_name;
```

Example

```
SELECT TOP 10 distinct *
FROM people
WHERE names='SMITH'
```

From this,MySQL will get the first 10 distinct rows of a table.

SQL BETWEEN Operator

The BETWEEN operator is used to select values within a range.The values can be numbers, text, or dates.

SQL BETWEEN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Example

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

The IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

SQL IN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2,...);
```

Example:

```
SELECT employeeid, lastname, salary
FROM employee_info
WHERE lastname IN ('Hernandez', 'Jones', 'Roberts', 'Ruiz');
```

4.SQL Aliases:

SQL aliases are used to temporarily rename a table or a column heading.

```
SELECT column_name AS alias_name
FROM table_name;
```

Example:

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person]
FROM Customers;
```

```
SELECT CustomerName, Address+', '+City+', '+PostalCode+', '+Country AS Address
FROM Customers;
```

```
SELECT CustomerName, CONCAT(Address,', ',City,', ',PostalCode,', ',Country) AS Address
FROM Customers;
```

Aliases can be useful when:

- There are more than one table involved in a query
- Functions are used in the query
- Column names are big or not very readable
- Two or more columns are combined together

Alias Example for Tables

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we have used aliases to make the SQL shorter):

Example:

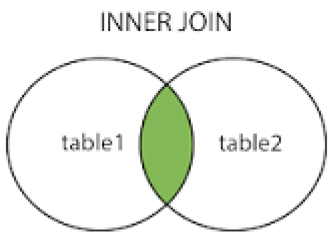
```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName="Around the Horn" AND c.CustomerID=o.CustomerID;
```

5.SQL Joins:

SQL joins are used to combine rows from two or more tables,based on a common field between them.

SQL INNER JOIN (simple join).

An SQL INNER JOIN return all rows from multiple tables where the join condition is met.



Syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name=table2.column_name;
```

Example:

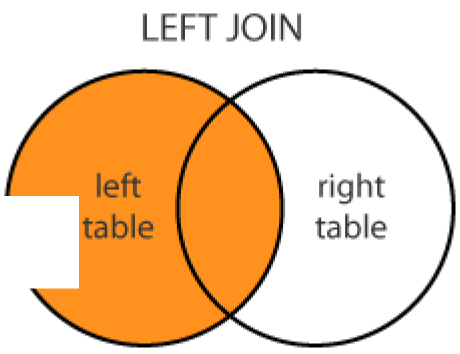
```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers
ON Orders.CustomerID=Customers.CustomerID;
```

SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

SQL LEFT JOIN Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name=table2.column_name;
```



Example:

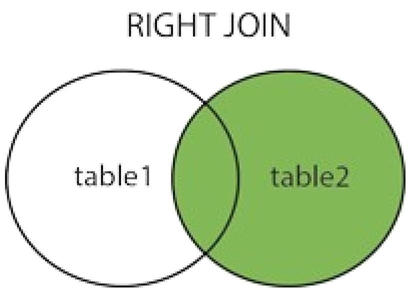
```
SELECT OrderNumber, TotalAmount, FirstName, LastName, City, Country  FROM Customer C LEFT
JOIN [Order] O    ON O.CustomerId = C.Id
```

SQL RIGHT JOIN :

The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.

SQL **RIGHT** JOIN Syntax

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name=table2.column_name;
```



Example:

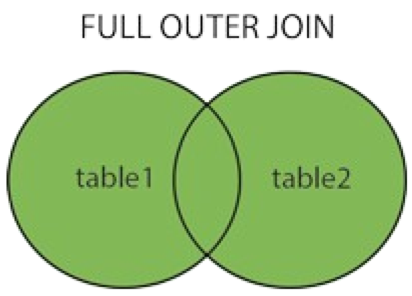
```
SELECT OrderNumber, TotalAmount, FirstName, LastName, City, Country  FROM Customer C
RIGHT JOIN [Order] O    ON O.CustomerId = C.Id;
```

SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword returns all rows from the left table (table1) and from the right table (table2).The FULL OUTER JOIN keyword combines the result of both LEFT and RIGHT joins.

Syntax:

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name=table2.column_name;
```

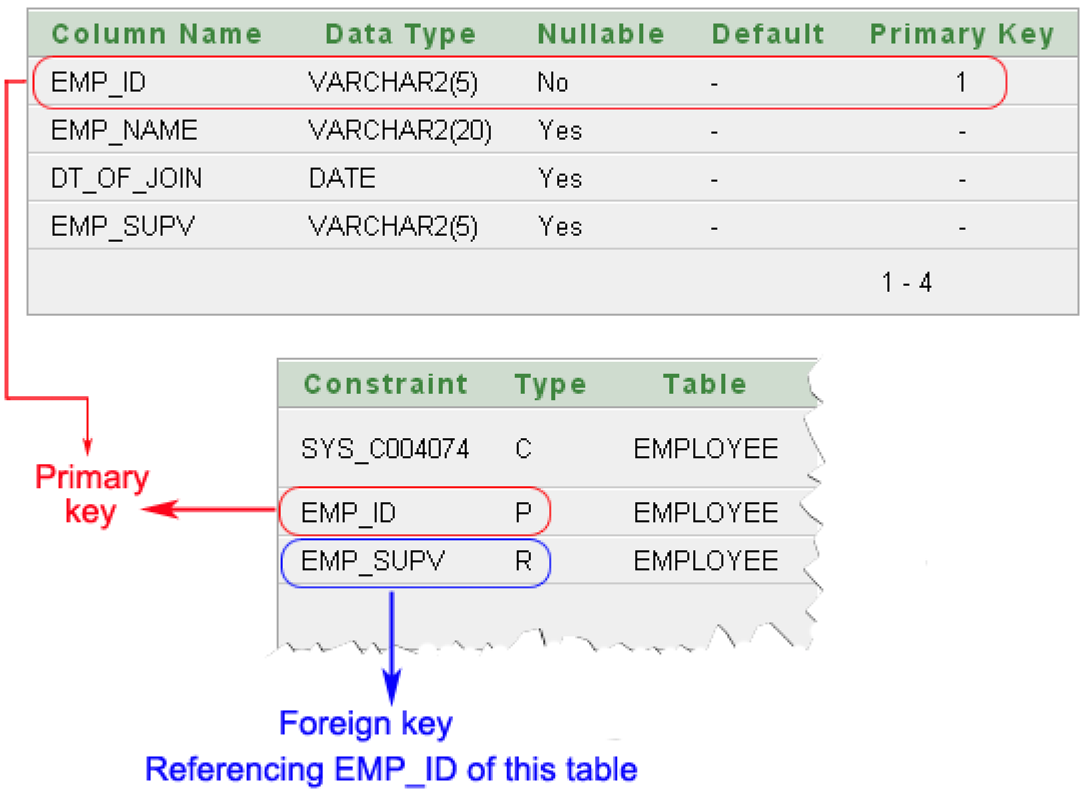


Example:

SELECT OrderNumber, TotalAmount, FirstName, LastName, City, Country FROM Customer C FULL OUTER JOIN [Order] O ON O.CustomerId = C.Id;

SELF JOIN:
It is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.**Self**-joins are used to compare values in a column with other values in the same column in the same table.

Example:



Column Name	Data Type	Nullable	Default	Primary Key
EMP_ID	VARCHAR2(5)	No	-	1
EMP_NAME	VARCHAR2(20)	Yes	-	-
DT_OF_JOIN	DATE	Yes	-	-
EMP_SUPV	VARCHAR2(5)	Yes	-	-
				1 - 4

Constraint	Type	Table
SYS_C004074	C	EMPLOYEE
EMP_ID	P	EMPLOYEE
EMP_SUPV	R	EMPLOYEE

In the EMPLOYEE table displayed above, emp_id is the primary key. emp_supv is the foreign key (this is the supervisor's employee id).

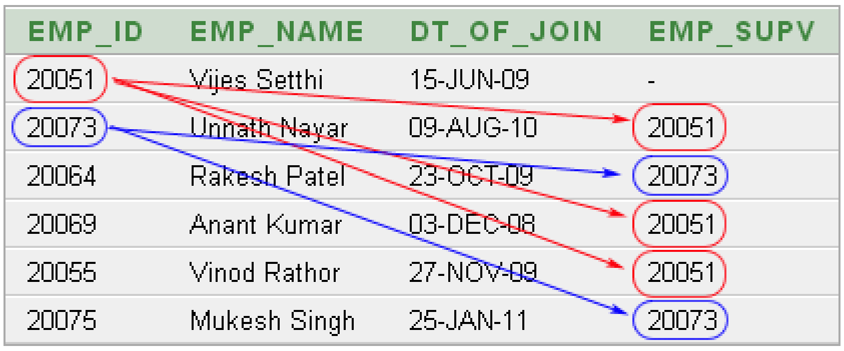
If we want a list of employees and the names of their supervisors, we'll have to JOIN the EMPLOYEE table to itself to get this list.

Unary relationship to employee

How the employees are related to themselves :

- An employee may report to another employee (supervisor).
- An employee may supervise himself (i.e. zero) to many employee (subordinates).

We have the following data into the table EMPLOYEE.



EMP_ID	EMP_NAME	DT_OF_JOIN	EMP_SUPV
20051	Vijes Setthi	15-JUN-09	-
20073	Unnath Nayar	09-AUG-10	20051
20064	Rakesh Patel	23-OCT-09	20073
20069	Anant Kumar	03-DEC-08	20051
20055	Vinod Rathor	27-NOV-09	20051
20075	Mukesh Singh	25-JAN-11	20073

The above data shows :

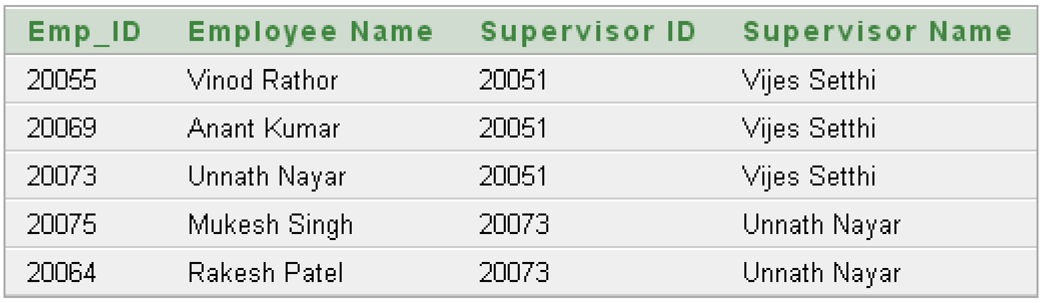
- Unnath Nayar's supervisor is Vijes Setthi
- Anant Kumar and Vinod Rathor can also report to Vijes Setthi.
- Rakesh Patel and Mukesh Singh are under supervison of Unnith Nayar.

In the following example we will use the table EMPLOYEE twice and in order to do this we will use the alias of the table.

To get the list of employees and their supervisor the following sql statement has used :

```
SELECT a.emp_id AS "Emp_ID",a.emp_name AS "Employee Name",
b.emp_id AS "Supervisor ID",b.emp_name AS "Supervisor Name"
FROM employee a, employee b
WHERE a.emp_supv = b.emp_id;
```

Output:



Emp_ID	Employee Name	Supervisor ID	Supervisor Name
20055	Vinod Rathor	20051	Vijes Setthi
20069	Anant Kumar	20051	Vijes Setthi
20073	Unnath Nayar	20051	Vijes Setthi
20075	Mukesh Singh	20073	Unnath Nayar
20064	Rakesh Patel	20073	Unnath Nayar

CARTESIAN (CROSS)JOIN:

- Returns the Cartesian product of the sets of records from the two or more joined tables.
- The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table.

Example:

Table 1: GameScores

PlayerName	DepartmentId	Scores
Jason	1	3000
Irene	1	1500
Jane	2	1000
David	2	2500
Paul	3	2000
James	3	2000

Table 2: Departments

DepartmentId	DepartmentName
1	IT
2	Marketing
3	HR

SELECT* FROM GameScores **CROSS JOIN** Departments**;**

Result:

PlayerName	DepartmentId	Scores	DepartmentId	DepartmentName
Jason	1	3000	1	IT
Irene	1	1500	1	IT
Jane	2	1000	1	IT
David	2	2500	1	IT
Paul	3	2000	1	IT
James	3	2000	1	IT
Jason	1	3000	2	Marketing
Irene	1	1500	2	Marketing
Jane	2	1000	2	Marketing
David	2	2500	2	Marketing
Paul	3	2000	2	Marketing
James	3	3000	2	Marketing
Jason	1	3000	3	HR
Irene	1	1500	3	HR

Jane	2	1000	3	HR
David	2	2500	3	HR
Paul	3	2000	3	HR
James	3	3000	3	HR

6.SQL Constraints:

SQL constraints are used to specify rules for the data in a table.

Constraints can be specified when the table is created (inside the CREATE TABLE statement) or after the table is created (inside the ALTER TABLE statement).

SQL CREATE TABLE + CONSTRAINT Syntax

```
CREATE TABLE table_name
(
column_name1 data_type(size) constraint_name,
column_name2 data_type(size) constraint_name,
column_name3 data_type(size) constraint_name,
....
);
```

In SQL, we have the following constraints:

SQL NOT NULL Constraint:

The NOT NULL constraint enforces a column to NOT accept NULL values.

The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

Example:

```
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NOT NULL.
```

SQL UNIQUE Constraint

- The UNIQUE constraint uniquely identifies each record in a database table.
- The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.
- Note that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

Example:

```
CREATE TABLE supplier
        ( supplier_id numeric(10) NOT NULL,
         supplier_name varchar2(50) NOT NULL,
        contact_name varchar2(50),
          CONSTRAINT supplier_unique UNIQUE (supplier_id)
        );
```

SQL PRIMARY KEY Constraint

- The PRIMARY KEY constraint uniquely identifies each record in a database table.
- Primary keys must contain UNIQUE values.
- A primary key column cannot contain NULL values.
- Most tables should have a primary key, and each table can have only ONE primary key.

Example:

```
CREATE TABLE employees
        ( employee_id INT PRIMARY KEY,
          last_name VARCHAR(50) NOT NULL,
         first_name VARCHAR(50) NOT NULL,
         salary MONEY
        );
```

SQL FOREIGN KEY Constraint

A FOREIGN KEY in one table points to a PRIMARY KEY in another table.

Example:

```
CREATE TABLE products
        ( product_id numeric(10) not null,
          supplier_id numeric(10) not null,
          CONSTRAINT fk_supplier
          FOREIGN KEY (supplier_id)
          REFERENCES supplier(supplier_id)
        );
```

SQL CHECK Constraint

- The CHECK constraint is used to limit the value range that can be placed in a column.

- If you define a CHECK constraint on a single column it allows only certain values for this column.
- If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.
- **Example:**

```
CREATE TABLE employees
        ( employee_id INT NOT NULL,
          last_name VARCHAR(50) NOT NULL,
          first_name VARCHAR(50),
          salary MONEY,
        CONSTRAINT check_salary
          CHECK (salary > 0)
        );
```

SQL DEFAULT Constraint

The DEFAULT constraint is used to insert a default value into a column.
The default value will be added to all new records, if no other value is specified.

Example:

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255) DEFAULT 'Sandnes'
)
```

SQL UNION Operator

- The SQL UNION operator combines the result of two or more SELECT statements.
- Notice that each SELECT statement within the UNION must have the same number of columns.
- The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.
- **Example:**

```
SELECT City FROM Customers
UNION
```

```
SELECT City FROM Suppliers
```

- To allow duplicate values, use the ALL keyword with UNION.

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
```

SQL ORDER BY Keyword

- The ORDER BY keyword is used to sort the result-set.
- The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can use the DESC keyword.

Example:

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'IBM'
ORDER BY supplier_city;
```

7. SQL Functions:

7.1 SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column.

Useful aggregate functions:

- AVG() - Returns the average value
- COUNT()- Returns the number of rows
- FIRST() - Returns the first value
- LAST() - Returns the last value
- MAX() - Returns the largest value
- MIN() - Returns the smallest value
- SUM() - Returns the sum

7.2 SQL Scalar functions

SQL scalar functions return a single value, based on the input value.

Useful scalar functions:

- UCASE() - Converts a field to upper case
- LCASE() - Converts a field to lower case
- MID() - Extract characters from a text field
- LEN() - Returns the length of a text field
- ROUND() - Rounds a numeric field to the number of decimals specified
- NOW() - Returns the current system date and time
- FORMAT() - Formats how a field is to be displayed

SQL GROUP BY Statement

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

Example:

```
SELECT department, SUM(sales) AS "Total sales"
FROM order_details
GROUP BY department;
```

The HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

```
SELECT department, SUM(sales) AS "Total sales"
FROM order_details
GROUP BY department
HAVING SUM(sales) > 1000;
```

8.SQL Procedure

A **procedure** or in simple a proc is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

Procedures: Passing Parameters

We can pass parameters to procedures in three ways.
1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters

A procedure may or may not return any value.

General Syntax to create a procedure is:

```
CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters]

      IS

     Declaration section

    BEGIN
   Execution section
   EXCEPTION
  Exception section
  END;
```

- **IS -** marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.
- The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.
- The optional parameter list contains name, mode and types of the parameters.
- IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example:

```
Declare
a number;
b number;
c number;
```

```
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;

BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;

CREATE or REPLACE PROCEDURE pro1(no in number,temp out emp1%rowtype)
IS
BEGIN
    SELECT * INTO temp FROM emp1 WHERE eno = no;
END;
```

How to execute a Stored Procedure?

There are two ways to execute a procedure.

1) From the SQL prompt.

```
EXECUTE [or EXEC] procedure_name;
```

2) Within another procedure – simply use the procedure name.

```
procedure_name;
```

9.Triggers:

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Syntax of Triggers

Syntax for Creating a Trigger

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
  --- sql statements
END;
```

- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *{BEFORE | AFTER | INSTEAD OF }* - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- *{INSERT [OR] | UPDATE [OR] | DELETE}* - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- *[OF col_name]* - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *[ON table_name]* - This clause identifies the name of the table or view to which the trigger is associated.
- *[REFERENCING OLD AS o NEW AS n]* - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- *[FOR EACH ROW]* - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- *WHEN (condition)* - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For Example: The price of a product changes constantly. It is important to maintain the history of the prices of the products.

We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'product_price_history' table

```
CREATE TABLE product_price_history
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );


CREATE TABLE product
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );
```

2) Create the price_history_trigger and execute it.

```
CREATE or REPLACE TRIGGER price_history_trigger
BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES
(:old.product_id,
 :old.product_name,
 :old.supplier_name,
 :old.unit_price);
END;
```

3) update the price of a product.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.