Joining Tables

Breaking data into multiple tables enables more efficient storage, easier manipulation, and greater scalability. To retrieve that data with a single SELECT statement we use a join.

Foreign key A column in one table that contains the primary key values from another table, thus defining the relationships between tables.

Cartesian product (cross join) The results returned by a table relationship without a join condition.The number of rows retrieved is the number of rows in the first table multiplied by the number of rows in the second table.

```
SELECT vend_name, prod_name, prod_price
FROM vendors, products
ORDER BY vend_name, prod_name;
```

Creating a Join:

Creating a join is simple. You must specify all the tables to be included and how they are related to each other.

```
SELECT vend_name, prod_name, prod_price
FROM vendors, products
WHERE vendors.vend_id = products.vend_id
ORDER BY vend_name;
```

The tables are correctly joined with a WHERE clause that instructs MariaDB to match vend_id in the vendors table with vend_id in the products table.

Note :
This fully qualified column name is required here because if you just specified vend_id, MariaDB cannot tell which vend_id columns you are referring to (as there are two of them, one in each table).

The Importance of the WHERE Clause :

When you join two tables, what you are actually doing is pairing every row in the first table with every row in the second table. The WHERE clause acts as a filter to only include rows that match the specified filter condition—the join condition, in this case. Without the WHERE clause, every row in the first table is paired with every row in the second table

Joining Multiple Tables

```
SELECT prod_name, vend_name, prod_price, quantity
FROM orderitems, products, vendors
WHERE products.vend_id = vendors.vend_id AND orderitems.prod_id = products.prod_id AND
order_num = 20005;
```

Using Table Aliases

To shorten the SQL syntax.To enable multiple uses of the same table within a single SELECT Statement

```
SELECT cust_name, cust_contact
FROM customers AS c, orders AS o, orderitems AS oi
WHERE c.cust_id = o.cust_id
AND oi.order_num = o.order_num
AND prod_id = 'TNT2';
```

Views

Views are virtual tables. Unlike tables that contain data, views simply contain queries that dynamically retrieve data when used.

It does not contain any actual columns or data as a table would. Instead, it contains a SQL query

Use of Views:

- To reuse SQL statements.
- To simplify complex SQL operations. After the query is written, it can be reused easily, without having to know the details of the underlying query itself.
- To expose parts of a table instead of complete tables.
- To secure data. Users can be given access to specific subsets of tables instead of entire tables.

After views are created, they can be used in the same way as tables. You can perform SELECT operations, filter and sort data, join views to other views or tables, and possibly even add and update data (There are some restrictions)

Views contain no data themselves, so the data they return is retrieved from other tables. When data is added or changed in those tables, the views will return that changed data.

View Rules and Restrictions:

- Like tables, views must be uniquely named. (They cannot be named with the name of any other table or view.)
- There is no limit to the number of views that can be created.
- To create views, you must have security access. This is usually granted by the database administrator.
- Views can be nested; that is, a view may be built using a query that retrieves data from another view.

Using Views

- Views are created using the CREATE VIEW statement.
- To view the statement used to create a view, use *SHOW CREATE VIEW viewname;*.
- To remove a view, the DROP statement is used. The syntax is simply
  DROP VIEW viewname;.
- To update a view you may use the DROP statement and then the CREATE statement again, or just use *CREATE OR REPLACE VIEW,* which creates the view if it does not exist and replaces it if it does.

Creating view

```
CREATE VIEW productcustomers AS
SELECT cust_name, cust_contact, prod_id
FROM customers, orders, orderitems
WHERE customers.cust_id = orders.cust_id
AND orderitems.order_num = orders.order_num;
```

Using view :

```
SELECT cust_name, cust_contact
FROM productcustomers
WHERE prod_id = 'TNT2';
```

Working with Stored Procedures

Stored procedures are simply collections of one or more MariaDB statements saved for future use.

Executing Stored Procedures

MariaDB refers to stored procedure execution as calling, and so the MariaDB statement to execute a stored procedure is simply CALL. CALL takes the name of the stored procedure and any parameters that need to be passed to it.

```
CALL productpricing(@pricelow,
@pricehigh,
@priceaverage);
```

Creating Stored Procedures

```
CREATE PROCEDURE productpricing()
BEGIN
    SELECT Avg(prod_price) AS priceaverage
    FROM products;
END;
```

Calling:

```
Call productpricing()
```

Dropping Stored Procedures

DROP PROCEDURE productpricing;

DROP PROCEDURE IF EXISTS productpricing;

Working with Parameters

```
CREATE PROCEDURE productpricing(
OUT pl DECIMAL(8,2)
)
BEGIN
SELECT Min(prod_price)
INTO pl
FROM products;
End;
```

Calling :

```
CALL productpricing(@pricelow);
Select @pricelow;
```

Show all stored procedures:

SHOW PROCEDURE STATUS

Triggers

Triggers used when you want a statement (or statements) to be executed automatically when events occur.

Note

Only Tables Triggers are supported only on tables, not on views (and not on temporary tables).

A trigger is a MariaDB statement (or a group of statements enclosed within BEGIN and END statements) that are automatically executed by MariaDB in response to any of these statements: Update, Insert,Delete

Creating Triggers

When creating a trigger you need to specify four pieces of information:

- The unique trigger name
- The table to which the trigger is to be associated
- The action that the trigger should respond to (DELETE, INSERT, or UPDATE)
- When the trigger should be executed (before or after processing)

CREATE TRIGGER newproduct AFTER INSERT ON products

FOR EACH ROW
BEGIN
END;

The trigger then specifies FOR EACH ROW and the code to be executed for each inserted row.

Dropping Triggers:

DROP TRIGGER newproduct;

Types of Triggers in MySQL

We can define the maximum Six types of actions or events in the form of triggers:

- **Before Insert**: It is activated before the insertion of data into the table.
- **After Insert:** It is activated after the insertion of data into the table.
- **Before Update**: It is activated before the update of data in the table.
- **After Update**: It is activated after the update of the data in the table.
- **Before Delete:** It is activated before the data is removed from the table.
- **After Delete**: It is activated after the deletion of data from the table.

When we use a statement that does not use INSERT, UPDATE or DELETE query to change the data in a table, the triggers associated with the trigger will not be invoked.

```
Create Trigger before_insert_emp workinghours    BEFORE INSERT ON employee
 FOR EACH ROW
BEGIN
IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;
END IF;
END
```

Functions :

MariaDB function is a stored program that is used to pass parameters into them and return a value.

MariaDB Create Function

```
create function occupation_age(age int)
returns varchar(20) deterministic
begin
   declare occupation varchar(20);
   if(age < 10) then set occupation="KID";
```

```
  elseif(age >= 10 and age <= 18) then set occupation="Teen";
 elseif(age > 18) then set occupation=”Adult”
  end if;
  return (occupation);
End
```

Stored Function calling :

```
select name,age,occupation_age(age) from students;
```

Import/Export :

Import to MariaDB Database:

To import an existing dump file into MySQL or MariaDB, you will have to create a new database. This database will hold the imported data.

```
mysql -u username -p new_database < data-dump.sql
```

Exporting MariaDB Database :

```
mysqldump -u root -p database_name > database_name.sql
```