SESSION 8 | INTRODUCTION TO JOINS

SESSION OVERVIEW:

By the end of this session students will be able to:

- Understand the different constraints used in SQL.
- Understand how the relationship between tables is generated. (ERDs)
- Understand the fundamentals of different JOINs which are majorly used in SQL.

KEY TOPICS AND EXAMPLES:

Understanding the different constraints used in SQL:

1. **Uses of constraints used in the database:**

    - **Data Integrity**: Constraints ensure that data stored in the database follows predefined rules, such as not allowing NULL values in certain columns, ensuring uniqueness of values, and maintaining referential integrity between related tables. This helps to maintain the accuracy and consistency of the data.
    - **Data Validation:** Constraints can be used to validate data before it is inserted into the database. For example, a constraint can ensure that a date is in the correct format, or that a numeric value falls within a certain range.
    - **Data Security:** Constraints can help improve data security by preventing certain types of malicious or accidental data manipulation. For example, a constraint can prevent users from deleting records that are referenced by other records.
    - **Query Optimization:** Constraints can be used by the database engine to optimize query execution. For example, if a column is marked as unique, the database can use an index to quickly check for duplicate values.

2. **How to specify a constraint to the columns in the table:**

    We can specify constraints at the time of creating the table using the CREATE TABLE statement. We can also specify the constraints after creating a table using the ALTER TABLE statement.

Syntax using CREATE TABLE:
This helps us specify the constraints while creating the table.

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    ...
);
```

NOTE:

- *table_name is the name of the table you're creating.*
- *column1, column2, etc., are the names of the columns in the table.*
- *datatype is the data type of the column.*
- *constraint is the constraint you want to apply to that column.*

For example, if you want to create a table called Users with columns ID and Name, and you want both columns to not allow NULL values, you would write:

```
CREATE TABLE Users (
    ID INT NOT NULL,
    Name VARCHAR(255) NOT NULL
);
```

This creates a table called Users with two columns: ID and Name, both of which do not allow NULL values.

Syntax using ALTER statement:
The alter statement also helps us to modify the table. In a previously taught session, we learned how to change the datatype of the columns in a table. Here we will learn how to add or change a constraint in the columns.

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name constraint_type (column1, column2, ...)
```

Note:

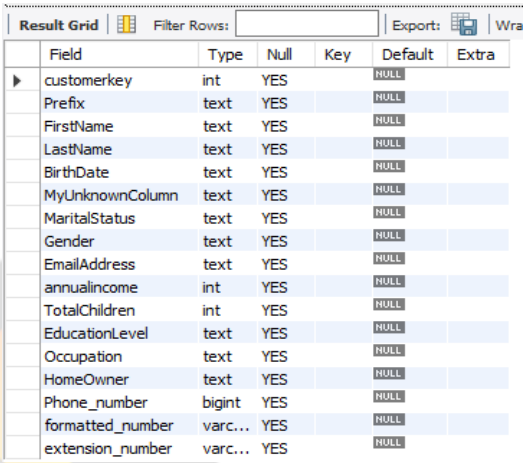
- *table_name is the name of the table to which you want to add the constraint.*
- *constraint_name is the name you give to the constraint. This is optional, and if not specified, the database system will generate a name.*
- *constraint_type is the type of constraint you want to add, such as PRIMARY KEY, FOREIGN KEY, CHECK, etc.*
- *(column1, column2, ...) specifies the column or columns to which the constraint applies.*

3. **Types of constraints in SQL:**
    a. **Concepts on NOT NULL constraints:**

    Suppose we specify a field in a table to be NOT NULL. Then the field will never accept a null value. That is, you will not be allowed to insert a new row in the table without specifying any value to this field.

Let’s take an example at the customers table on which we were working in our previous session. The first step that we are going to follow is to describe the table to check if the table already contains NOT NULL constraints in any of the columns.

```
DESC customers;
```



Result Grid | Filter Rows: | Export: | Wra

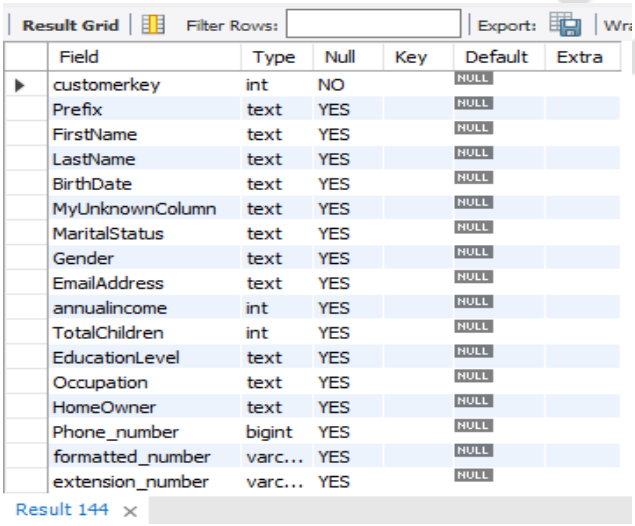
Field	Type	Null	Key	Default	Extra
customerkey	int	YES		NULL	
Prefix	text	YES		NULL	
FirstName	text	YES		NULL	
LastName	text	YES		NULL	
BirthDate	text	YES		NULL	
MyUnknownColumn	text	YES		NULL	
MaritalStatus	text	YES		NULL	
Gender	text	YES		NULL	
EmailAddress	text	YES		NULL	
annualincome	int	YES		NULL	
TotalChildren	int	YES		NULL	
EducationLevel	text	YES		NULL	
Occupation	text	YES		NULL	
HomeOwner	text	YES		NULL	
Phone_number	bigint	YES		NULL	
formatted_number	varc...	YES		NULL	
extension_number	varc...	YES		NULL	

But to our surprise, we noticed that the table does not have any column with a NOT NULL constraint in it which means that each column in customers table will accept null values and won't return any error if we keep any cell empty or blank.

Now we realize that the customerKey column should never be NULL in this table. Thus we use the ALTER TABLE statement to set the column as NOT NULL.

```
ALTER TABLE customers
MODIFY customerkey INT NOT NULL;
```

Now if we describe the table, let's see the difference between the previously described table and this one.



Result Grid | Filter Rows: | Export: | Wra

Field	Type	Null	Key	Default	Extra
customerkey	int	NO		NULL	
Prefix	text	YES		NULL	
FirstName	text	YES		NULL	
LastName	text	YES		NULL	
BirthDate	text	YES		NULL	
MyUnknownColumn	text	YES		NULL	
MaritalStatus	text	YES		NULL	
Gender	text	YES		NULL	
EmailAddress	text	YES		NULL	
annualincome	int	YES		NULL	
TotalChildren	int	YES		NULL	
EducationLevel	text	YES		NULL	
Occupation	text	YES		NULL	
HomeOwner	text	YES		NULL	
Phone_number	bigint	YES		NULL	
formatted_number	varc...	YES		NULL	
extension_number	varc...	YES		NULL	

Result 144 ×

We can clearly see the customerkey column in the customers table the NULL column says NO which means the column will not accept NULL values.

b. Concepts on UNIQUE constraints:

This constraint helps to uniquely identify each row in the table. i.e. for a particular column, all the rows should have unique values. We can have more than one UNIQUE column in a table.

Syntax using CREATE TABLE statement:

```
CREATE TABLE table_name (
    column1 datatype UNIQUE,
    column2 datatype,
    ...
);
```

Alternative syntax:
This alternative method helps us to specify the constraint to multiple columns at a time.

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    UNIQUE (column1, column2)
);
```

Syntax using ALTER TABLE statements:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name UNIQUE (column1, column2, ...);
```

NOTE:

- *table_name is the name of the table to which you want to add the unique constraint.*
- *constraint_name is an optional name for the constraint. If not specified, the database system will generate a name.*
- *(column1, column2, ...) specifies the column or columns to which the unique constraint applies.*

```
CREATE TABLE Student
(
ID int(6) NOT NULL UNIQUE,
NAME varchar(10),
ADDRESS varchar(20)
);
```

c. **Concepts on PRIMARY KEY:**

In MySQL, a primary key is a column or a combination of columns in a table that uniquely identifies each row in that table. Primary keys are used to ensure data integrity and enforce entity integrity in a relational database management system (RDBMS).

Here are some key points about primary keys in MySQL:

- **Uniqueness:** Each value in the primary key column(s) must be unique across all rows in the table. No two rows can have the same primary key value.
- **Non-null:** Primary key columns cannot contain NULL values. This constraint

ensures that every row in the table can be identified uniquely.

- **Single vs. Composite:** A primary key can be a single column or a combination of multiple columns (known as a composite primary key).

Syntax using CREATE TABLE statement:

```
CREATE TABLE table_name (
    column1 datatype PRIMARY KEY,
    column2 datatype,
    ...
);
```

If you want to use a composite primary key:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...,
    PRIMARY KEY (column1, column2)
);
```

Syntax using ALTER TABLE statement:
We can use the ALTER TABLE statement to add a primary constraint in an existing table.

```
ALTER TABLE table_name
ADD PRIMARY KEY (column1, column2);
```

Primary keys play a crucial role in establishing relationships between tables in a relational database. They are typically used as foreign keys in other tables to create relationships and maintain referential integrity.

d. Concepts on FOREIGN KEY:

In MySQL, a foreign key is a column or a combination of columns in a table that references the primary key of another table. Foreign keys are used to establish and enforce relationships between tables in a relational database management system (RDBMS).

Here are some important concepts related to foreign keys in MySQL:

- **Referential Integrity:** Foreign keys are used to maintain referential integrity, which ensures that the data in related tables is consistent and valid. A foreign key constraint ensures that the values in the foreign key column(s) must either match an existing value in the referenced primary key column(s) or be NULL (if nulls are allowed).
- **Parent-Child Relationship:** The table containing the primary key is often referred to as the parent table or the referenced table, while the table containing the foreign key is called the child table or the referencing table.
- **Cascading Actions:** When defining a foreign key constraint, you can specify

cascading actions that determine what should happen when a related row in the parent table is updated or deleted. Common cascading actions include:

- **RESTRICT**: Prevents the operation if there are any related rows in the child table.
- **CASCADE**: Automatically updates or deletes the related rows in the child table.
- **SET NULL**: Sets the foreign key column(s) in the child table to NULL when the referenced row is deleted.

Syntax using SELECT statement:

```
CREATE TABLE child_table (
    column1 datatype,
    column2 datatype,
    ...,
    FOREIGN KEY (column1, column2, ...)
        REFERENCES parent_table(parent_column1, parent_column2, ...)
        [ON DELETE and/or ON UPDATE action]
);
```

Syntax using ALTER TABLE statement:

```
ALTER TABLE child_table
ADD FOREIGN KEY (column1, column2, ...)
    REFERENCES parent_table(parent_column1, parent_column2, ...)
    [ON DELETE and/or ON UPDATE action];
```

Example:
This example is a hypothetical example for understanding purposes and isn't related to the dataset that we have been using.

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
        ON DELETE CASCADE
);
```

NOTE:
In this example, the customer_id column in the orders table is a foreign key that references the customer_id column (primary key) in the customers table. If a customer is deleted from the customers table, the ON DELETE CASCADE action will automatically delete all related orders for that customer from the orders table.

e. **Concepts on CHECK:**

A **CHECK constraint** in MySQL is used to specify a condition that values in a column must satisfy. When a CHECK constraint is applied to a column, MySQL ensures that all values in that column meet the specified condition. If the condition is violated, the database rejects the data modification.

NOTE: The examples mentioned below are hypothetical examples for understanding purposes and aren't related to the dataset that we have been using.

Syntax using CREATE statement:

```
CREATE TABLE table_name (
    column_name datatype,
    ...
    CONSTRAINT constraint_name CHECK (condition)
);
```

Example:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    quantity INT,
    price DECIMAL(10, 2),
    CONSTRAINT check_positive_amount CHECK (quantity * price > 0)
);
```

Syntax using ALTER statement:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name CHECK (condition);
```

Example:

```
ALTER TABLE employees
ADD CONSTRAINT check_age CHECK (age >= 18);
```

f. **Concepts on DEFAULT:**

In MySQL, the DEFAULT constraint is used to assign a default value to a column when no value is explicitly provided during an INSERT operation. The DEFAULT value is applied automatically, ensuring that the column has a value if none is supplied.

Syntax using CREATE statement:

```
CREATE TABLE table_name (
    column_name datatype DEFAULT default_value,
```

```
    ...
);
```

Example:

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT DEFAULT 25
);
```

Syntax using ALTER statement:

```
ALTER TABLE table_name
MODIFY column_name datatype DEFAULT default_value;
```

Example:

```
ALTER TABLE employees
MODIFY age INT DEFAULT 30;
```

Understanding relationships between the tables and their importance:

1. **What is Entity Relationship Diagrams?**

   In SQL, Entity-Relationship Diagrams (ERDs) are graphical representations of the logical structure of a database, showing how different entities (tables) relate to each other. ERDs use various symbols and notation conventions to depict entities, attributes (columns), primary keys, foreign keys, and relationships between entities. They help in visualizing and designing database schemas before implementing them in SQL.

2. **Uses of ERDs in SQL:**

   Entity-relationship diagrams (ERDs) are used in database design and development to:

   - **Visualize Database Structure:** ERDs provide a visual representation of the database schema, showing tables, columns, and relationships between entities. This helps developers and stakeholders understand the database structure easily.

   - **Design Databases:** ERDs are used in the initial stages of database design to plan the structure of the database. They help identify entities, attributes, and relationships, which are essential for creating an efficient and well-organized database.

   - **Identify Relationships:** ERDs help identify the relationships between entities, such as one-to-one, one-to-many, and many-to-many relationships. This is crucial for designing the database schema and ensuring data integrity.

- **Normalize Data:** ERDs help in normalizing the database schema, which involves organizing data in a way that reduces redundancy and dependency. Normalization helps improve database performance and data integrity.

- **Database Documentation:** ERDs serve as documentation for the database structure. They can be used to explain the database schema to stakeholders, developers, and database administrators.

- **Identify Key Constraints:** ERDs help identify primary keys, foreign keys, and other key constraints in the database schema. This is important for ensuring data integrity and enforcing referential integrity between tables.

3. **Types of relationships:**

Several types of relationships can exist between tables or entities. These relationships are defined based on cardinality (the maximum number of occurrences of one entity that can be associated with a single instance of another entity) and participation (whether the existence of an entity instance is dependent on its relationship with another entity).

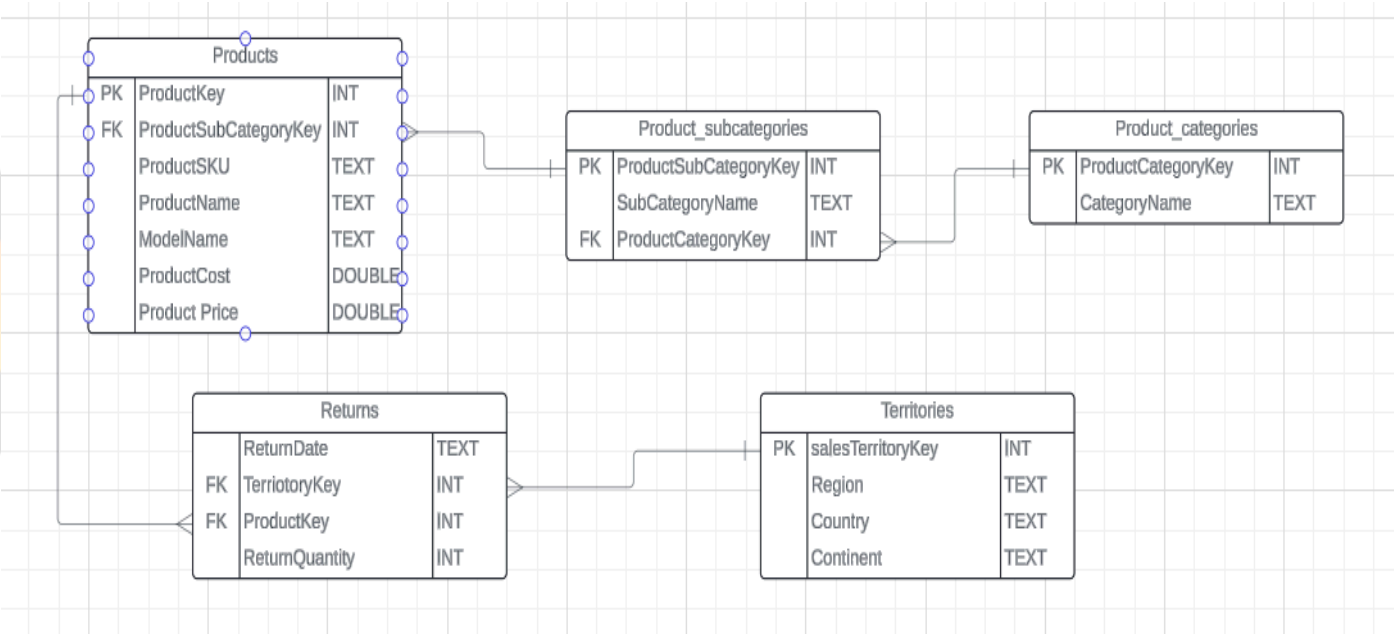
1. **One-to-One (1:1) Relationship:**
   In a one-to-one relationship, one instance of an entity is associated with at most one instance of another entity, and vice versa.
   Example: A person can have one passport, and a passport can belong to one person.

2. **One-to-Many (1:M) Relationship:**
   In a one-to-many relationship, one instance of an entity can be associated with multiple instances of another entity, but one instance of the other entity can be associated with at most one instance of the first entity.
   Example: A department can have many employees, but an employee can belong to only one department.

3. **Many-to-One (M:1) Relationship:**
   This is the reverse of the one-to-many relationship, where multiple instances of an entity can be associated with a single instance of another entity.
   Example: Many students can enroll in one course.

4. **Many-to-Many (M: M) Relationship:**
   In a many-to-many relationship, multiple instances of an entity can be associated with multiple instances of another entity.
   Example: Students can enroll in multiple courses, and courses can have multiple students enrolled.
   To implement a many-to-many relationship in SQL, an additional table (called a junction, intersection, or bridge table) is required to associate the two entities.

4. **Examples of ERDs:**

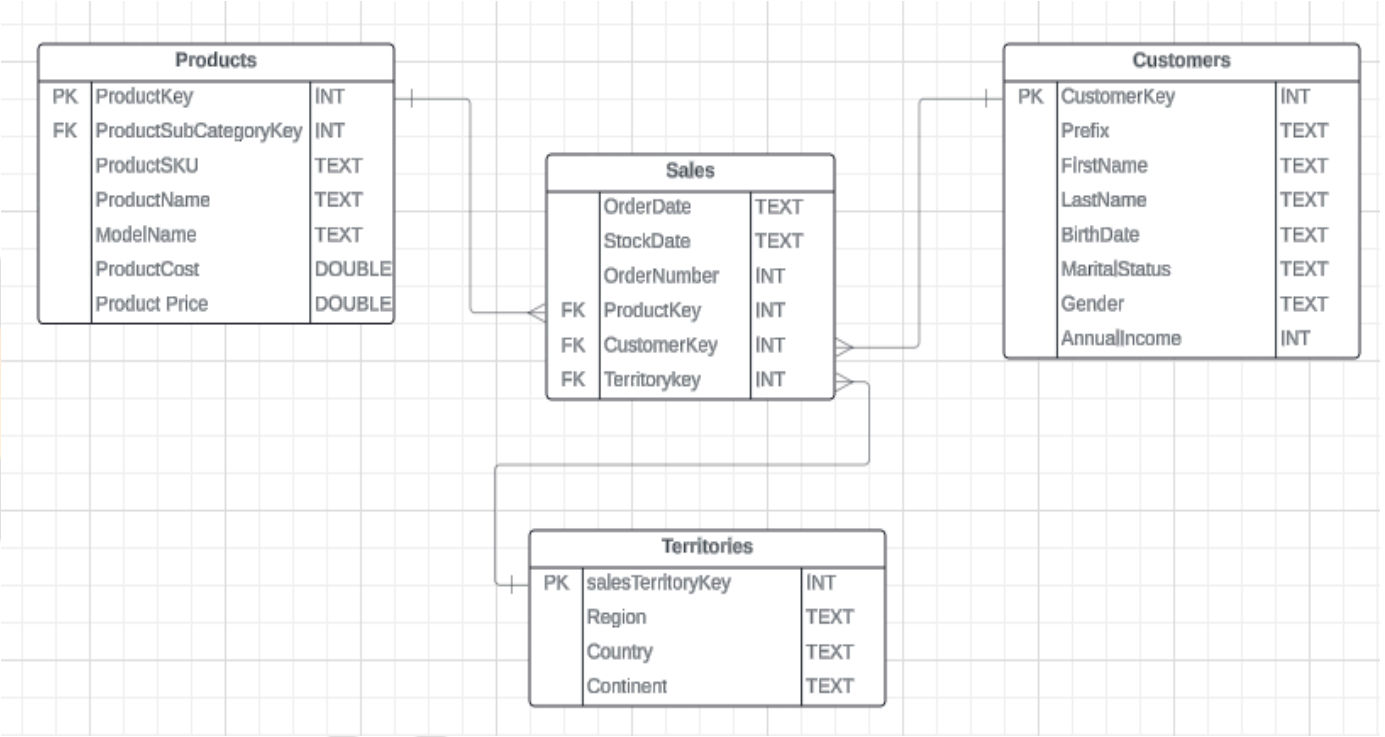


Explanation of the above ERD:

- **Product Hierarchy:** This ERD shows a hierarchical structure for products. Products belong to a subcategory, and subcategories belong to a category. This hierarchy allows for better organization and categorization of products based on their characteristics or types.

- **Product-Territory Relationship:** The relationship between Products and Territories entities indicates that products can be sold across different territories, which could represent geographical regions, countries, or sales areas. This relationship allows tracking product sales and returns based on the territory.

- **Returns Management:** The Returns entity is related to both Products and Territories, which suggests that the system tracks product returns based on the specific product and the territory where the return originated. This information can be useful for analyzing return patterns, identifying issues, and improving customer service.



Explanation of the above ERD:

- **Sales Transaction:** The Sales entity represents individual sales transactions, capturing details such as order date, stock date, and order number. This entity serves as the central point for recording sales data.

- **Product-Customer Relationship:** The relationship between Products and Customers entities, through the Sales entity, indicates that the system tracks which products are sold to which customers. This information can be valuable for analyzing customer purchasing behavior, identifying popular products, and targeted marketing efforts.

- **Sales Territory Tracking:** The relationship between Sales and Territories entities allows tracking sales transactions based on the territory where the sale occurred. This can be useful for analyzing sales performance across different regions, allocating resources, and identifying growth opportunities.

- **Customer Demographics:** The Customers entity includes demographic information such as name, birth date, marital status, gender, and annual income. This data can help segment customers, understand customer profiles, and tailor marketing and sales strategies accordingly.

Understanding the concepts of Joins:

1. **What is a JOIN?**

   In SQL, a join is used to combine rows from two or more tables based on a related column

between them. It allows you to retrieve data from multiple tables simultaneously, which is useful when the data you need is spread across different tables in your database.

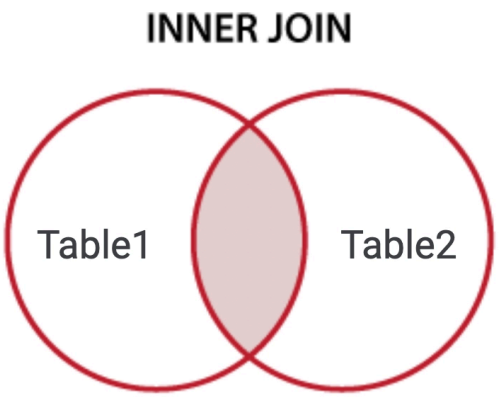
2. **Uses of JOINS:**

- **Combine Related Data:** Joins allow you to combine data from two or more tables that are related through a common column. This is useful for retrieving a complete set of information that is spread across multiple tables.
- **Retrieve Data from Multiple Tables:** Joins enable you to retrieve data from multiple tables in a single query. This reduces the need for multiple queries or manual data manipulation.
- **Perform Aggregations**: Joins can be used with aggregate functions like SUM, COUNT, AVG, etc., to perform calculations on related data from multiple tables. For example, you can calculate the total sales amount per customer by joining the orders and customers tables.
- **Filter Data Based on Related Values:** Joins can be used to filter data based on related values in another table. For example, you can retrieve all orders placed by customers from a specific city by joining the orders and customers tables and applying a filter on the city column.

3. **Different types of JOINS:**

a. **INNER JOINS:**

In SQL, an inner join is a type of join operation that combines rows from two or more tables based on a related column between them. The inner join returns only the rows that have matching values in the specified columns from both tables. In other words, it returns the intersection of the two tables based on the join condition.



Let's understand INNER JOINS using a simple analogy:

Imagine you have two different lists: one with names of students and their grades, and another with names of students and their attendance records. You want to combine these lists to get a complete picture of each student, including both their grades and

attendance.

Think of an inner join like an intersection of two sets. It only includes the students who are present in both lists. In our analogy, it would be like combining only the names that appear in both the grades list and the attendance list, showing both their grades and attendance records.

Basic syntax:

```
SELECT columns
FROM table1
INNER JOIN table2 ON table1.column_name = table2.column_name;
```

NOTE:

- *columns are the columns you want to retrieve from the tables.*
- *table1 and table2 are the tables you want to join.*
- *column_name is the column that is common between the two tables and used for the join condition.*

Example 1:

```
SELECT *
FROM Returns AS r
INNER JOIN territories AS t ON r. territoryKey=t. SalesTerritoryKey;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

ReturnDate	TerritoryKey	ProductKey	ReturnQuantity	SalesTerritoryKey	Region	Country	Continent
1/18/2015	9	312	1	9	Australia	Australia	Pacific
1/18/2015	10	310	1	10	United Kingdom	United Kingdom	Europe
1/21/2015	8	346	1	8	Germany	Germany	Europe
1/22/2015	4	311	1	4	Southwest	United States	North America
2/2/2015	6	312	1	6	Canada	Canada	North America
2/15/2015	1	312	1	1	Northwest	United States	North America
2/19/2015	9	311	1	9	Australia	Australia	Pacific
2/24/2015	8	314	1	8	Germany	Germany	Europe

Result 153

Note: The output might not contain all the rows from the returned table.

Example 2:

```
SELECT p. ProductName, ps. subcategoryName, pc. categoryName
FROM Products AS p
INNER JOIN Product_subcategories AS ps ON ps.
productsubcategorykey=p.productsubcategorykey
INNER JOIN product_categories AS pc ON pc. ProductCategoryKey= ps.
ProductCategoryKey;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

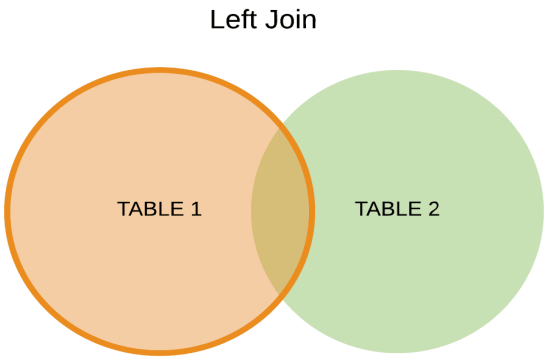
ProductName	subcategoryName	categoryName
Sport-100 Helmet, Red	Helmets	Accessories
Sport-100 Helmet, Black	Helmets	Accessories
Mountain Bike Socks, M	Socks	Clothing
Mountain Bike Socks, L	Socks	Clothing
Sport-100 Helmet, Blue	Helmets	Accessories
AWC Logo Cap	Caps	Clothing
Long-Sleeve Logo Jersey, S	Jerseys	Clothing
Long-Sleeve Logo Jersey, M	Jerseys	Clothing

Result 154

Note: The output might not contain all the rows from the returned table.

b. LEFT JOINS:

A LEFT JOIN (or LEFT OUTER JOIN) in SQL is a type of join operation that returns all rows from the left table (the table specified before the JOIN keyword), along with the matching rows from the right table (the table specified after the JOIN keyword). If there are no matching rows in the right table, the result will contain NULL values for the columns from the right table.



Let's understand LEFT JOINS using the same analogy that we have used above:

A left join includes all the students from the first list (the left list), regardless of whether they are present in the second list. If a student is present in the first list but not in the second, the fields from the second list will be shown as empty (NULL). In our analogy, it would be like combining all the names from the grades list and adding their attendance records if they exist, or showing NULL for attendance if the student is not in the attendance list.

Basic Syntax:

```
SELECT column1, column2, ...
FROM left_table
LEFT JOIN right_table ON left_table.column = right_table.column;
```

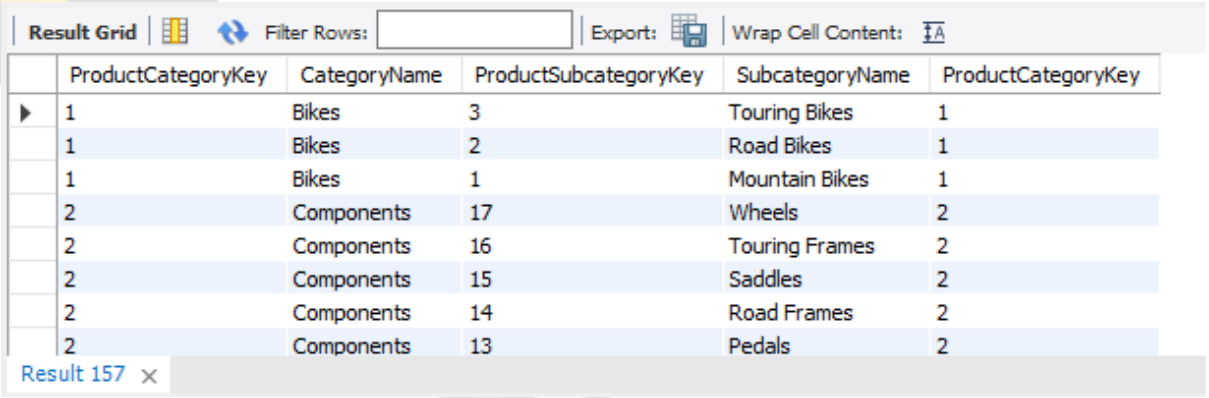
NOTE:

In a LEFT JOIN, all rows from the left table (table1) are returned, along with matching rows from the right table (table2). If there is no match, NULL values are returned for columns from the right table.

Example 1:

```
SELECT *
FROM product_categories AS pc
LEFT JOIN product_subcategories AS ps ON pc. ProductCategoryKey=ps.
ProductCategoryKey;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductCategoryKey	CategoryName	ProductSubcategoryKey	SubcategoryName	ProductCategoryKey
1	Bikes	3	Touring Bikes	1
1	Bikes	2	Road Bikes	1
1	Bikes	1	Mountain Bikes	1
2	Components	17	Wheels	2
2	Components	16	Touring Frames	2
2	Components	15	Saddles	2
2	Components	14	Road Frames	2
2	Components	13	Pedals	2

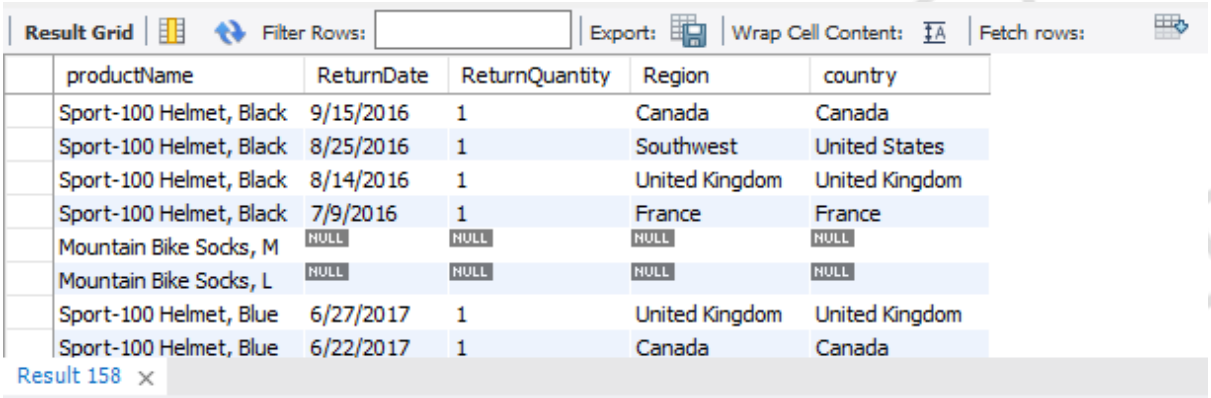
Result 157

Note: The output might not contain all the rows from the returned table.

Example 1:

```
SELECT p.productName, r.ReturnDate, r.ReturnQuantity, t. Region,
t.country
FROM products AS p
LEFT JOIN returns AS r ON r.productKey = p.productKey
LEFT JOIN territories AS t ON t.SalesTerritoryKey = r.TerritoryKey;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

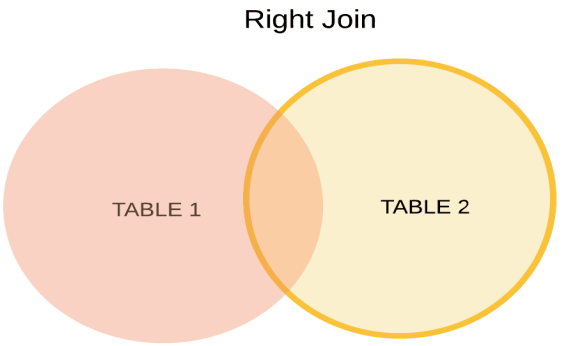
productName	ReturnDate	ReturnQuantity	Region	country
Sport-100 Helmet, Black	9/15/2016	1	Canada	Canada
Sport-100 Helmet, Black	8/25/2016	1	Southwest	United States
Sport-100 Helmet, Black	8/14/2016	1	United Kingdom	United Kingdom
Sport-100 Helmet, Black	7/9/2016	1	France	France
Mountain Bike Socks, M	NULL	NULL	NULL	NULL
Mountain Bike Socks, L	NULL	NULL	NULL	NULL
Sport-100 Helmet, Blue	6/27/2017	1	United Kingdom	United Kingdom
Sport-100 Helmet, Blue	6/22/2017	1	Canada	Canada

Result 158

Note: The output might not contain all the rows from the returned table. The output mentioned above shows NULL values which means there was no match from the right tables i.e. returns table and territories table.

c. **RIGHT JOINS:**

A RIGHT JOIN (or RIGHT OUTER JOIN) in SQL is a type of outer join operation that returns all rows from the right table (the table specified after the JOIN keyword), along with the matching rows from the left table (the table specified before the JOIN keyword). If there are no matching rows in the left table, the result will contain NULL values for the columns from the left table.



Let's understand RIGHT JOINS using the same analogy that we have used above:

A right join is similar to a left join, but it includes all the students from the second list (the right list), regardless of whether they are present in the first list. If a student is present in the second list but not in the first, the fields from the first list will be shown as empty (NULL). In our analogy, it would be like combining all the names from the attendance list and adding their grades if they exist, or showing NULL for grades if the student is not in the grades list.

Basic syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2 ON table1.column_name = table2.column_name;
```

Note:
In a right join, all rows from the right table (table2) are returned, along with matching rows from the left table (table1). If there is no match, NULL values are returned for columns from the left table.

Example 1:

```
SELECT p.productName, r.ReturnDate, r.ReturnQuantity, t. Region,
t.country
FROM products AS p
RIGHT JOIN returns AS r ON r.productKey = p.productKey
RIGHT JOIN territories AS t ON t.SalesTerritoryKey = r.TerritoryKey;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

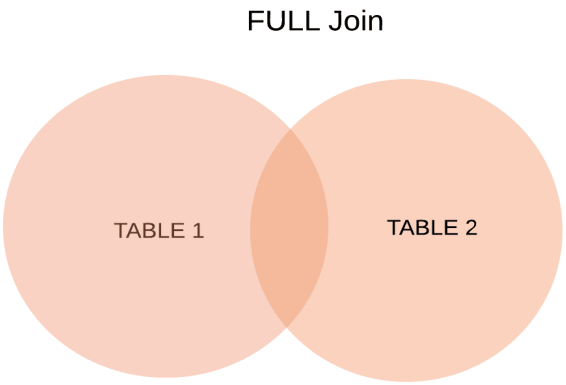
productName	ReturnDate	ReturnQuantity	Region	country
Road-150 Red, 62	5/13/2015	1	Northwest	United States
Road-150 Red, 56	4/27/2015	1	Northwest	United States
Road-150 Red, 48	2/15/2015	1	Northwest	United States
NULL	NULL	NULL	Northeast	United States
NULL	NULL	NULL	Central	United States
Mountain-400-W Silver, 38	6/29/2017	1	Southwest	United States
Mountain-200 Silver, 46	6/29/2017	1	Southwest	United States
Mountain Tire Tube	6/28/2017	2	Southwest	United States

Result 160

Note: The output might not contain all the rows from the returned table. The output mentioned above shows NULL values which means there was no match from the left tables i.e. returns table and products table.

d. FULL JOINS:

A FULL JOIN (or FULL OUTER JOIN) in SQL is a type of join operation that returns all rows from both tables being joined, regardless of whether there are matching values in the joined columns or not. It combines the results of a LEFT JOIN and a RIGHT JOIN, including all rows from both tables and filling in NULL values for any unmatched rows.



Let's understand FULL JOINS using the same analogy that we have used above:

A full join includes all the students from both lists. If a student is present in only one list, the fields from the other list will be shown as empty (NULL). In our analogy, it would be like combining all the names from both the grades and attendance lists, showing both their grades and attendance records if they exist, or showing NULL for grades or attendance if the student is not in one of the lists.

IMPORTANT:
Unfortunately, MySQL does not directly support the FULL JOIN or FULL OUTER JOIN syntax. However, you can achieve the same result by using a combination of LEFT JOIN and RIGHT JOIN with the UNION or UNION ALL operators.

This is where we are going to learn the concepts of the UNION function which helps us understand how UNION functions fill the gaps between FULL JOIN and UNION.

How UNION helps us to perform FULL JOIN in MySQL:

In MySQL, the UNION operator is used in conjunction with LEFT JOIN and RIGHT JOIN to simulate the behavior of a FULL OUTER JOIN, which is not directly supported in MySQL. Here's how the UNION operator helps us perform a FULL JOIN in MySQL:

- **Combining the Results of LEFT JOIN and RIGHT JOIN:** The FULL JOIN operation is designed to return all rows from both tables, regardless of whether there are matching rows or not. To achieve this in MySQL, we need to combine the results of a LEFT JOIN and a RIGHT JOIN.

- **LEFT JOIN:** The LEFT JOIN part of the query retrieves all rows from the left table (table1) and the matching rows from the right table (table2). If there are no matches in the right table, the columns from the right table will have NULL values.

- **RIGHT JOIN:** The RIGHT JOIN part of the query retrieves all rows from the right table (table 2) along with the matching rows from the left table (table 1). If there are no matches in the left table, the columns from the left table will have NULL values.

- **UNION vs. UNION ALL**: If you want to include all rows, including duplicates, you can use the UNION ALL operator instead of UNION. UNION ALL combines the results of the LEFT JOIN and RIGHT JOIN without removing duplicates.

Syntax of UNION:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column
UNION
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column
```

NOTE:

- ***SELECT columns: Specify the columns you want to include in the result set from both tables.***
- ***FROM table1: Specify the first table to be joined.***
- ***LEFT JOIN table2: Perform a LEFT JOIN with the second table.***
- ***ON table1.column = table2.column: Specify the join condition, typically matching a column from the first table with a column from the second table.***
- ***UNION: The UNION operator combines the results of the LEFT JOIN and the subsequent query without duplicates.***
- ***SELECT columns: Repeat the column selection from the first part of the query.***

- *FROM table1: Repeat the first table name.*
- *RIGHT JOIN table2: Perform a RIGHT JOIN with the second table.*
- *ON table1.column = table2.column: Specify the same join condition as in the LEFT JOIN.*

Example:

```
SELECT t.*, r.*
FROM territories AS t
LEFT JOIN returns AS r ON t.salesterritoryKey = r.territorykey
UNION ALL
SELECT t.*, r.*
FROM territories AS t
RIGHT JOIN returns AS r ON t.salesterritoryKey = r.territorykey
WHERE t.salesterritoryKey IS NULL;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SalesTerritoryKey	Region	Country	Continent	ReturnDate	TerritoryKey	ProductKey	ReturnQuantity
1	Northwest	United States	North America	6/28/2017	1	587	1
1	Northwest	United States	North America	6/28/2017	1	480	1
1	Northwest	United States	North America	6/28/2017	1	214	1
1	Northwest	United States	North America	6/26/2017	1	472	1
1	Northwest	United States	North America	6/24/2017	1	487	1
1	Northwest	United States	North America	6/24/2017	1	485	1
1	Northwest	United States	North America	6/24/2017	1	480	1
1	Northwest	United States	North America	6/24/2017	1	479	1

Result 161

Note: The output might not contain all the rows from the returned table.