



Lesson - 304.3

Data Definition Language and Data Integrity



Data Definition Language and Data Integrity Constraints



Learning Objectives:

By the end of this lesson, learners will be able to:

- Describe Data Definition Language (DDL).
- Demonstrate how to create a database in SQL.
- Demonstrate how to create tables in a database.
- Describe Data Integrity and Data Integrity Constraints.

Table of contents

- Overview of Data Definition Language
- Naming Conventions
- Create Database in SQL
- Create Table in Database
- Data Persistence
- What is Data Integrity ?
- What is Data Integrity Constraints?
- 1 - Domain Integrity
 - Domain Integrity - SQL Data Types
 - Domain Integrity - NOT NULL and DEFAULT Constraints
- 2 - Entity Integrity
 - Entity Integrity - Primary Keys
 - Entity Integrity - Primary Keys for group of columns
 - Auto-Generated - Primary Keys
 - Entity Integrity - UNIQUE Constraints
- 3 - Referential Integrity
 - Syntax: Foreign Key Constraints
- 4 - User-Defined Constraints
- Alter Table Statement
 - Rename Table Name
- Drop Table Statement
- A Universally Unique Identifier

Introduction to Database Integrity and Constraints

- ❑ **Scenario:** When authorized users make changes to the database, integrity constraints ensure that the data remains consistent. When designing the database schema, integrity constraints are added. It defines constraints in SQL DDL commands such as 'CreateTable' and 'AlterTable.'
- ❑ Integrity Constraints are the protocols that a table's data columns must follow. These are used to restrict the types of information that can be entered into a table. This means that the data in the database is accurate and reliable. You may apply integrity Constraints at the column or table level. The table-level Integrity constraints apply to the entire table, while the column level constraints are only applied to one column."
- ❑ We will explore in detail about Database integrity and constraints in upcoming slides, we will utilize **classicmodels** database for demonstrations and examples. You must have **classicmodels** database setup.

Data Definition Language

Data Definition Language (DDL) helps you to define the database structure and schema (tables, table fields, table constraints, and data integrity).

The DDL statements are **CREATE**, **ALTER**, and **DROP**.

Naming Conventions

Before creating a **database** and **tables**, consider best practices around table names:

- Use real words (non-cryptic): ~~xdsffbus~~
- The name should indicate the purpose, example: trainingFeedback
- Avoid dated names: ~~db2012~~
- Avoid special characters: ~~db@fms~~
- Avoid numbers: ~~db007~~

Create Database in SQL

To create a new database in SQL, you can use the **CREATE DATABASE** statement with the following *syntax*:

```
CREATE DATABASE [IF NOT EXISTS]
database_name
[CHARACTER SET charset_name]
[COLLATE collation_name]
```

[IF NOT EXISTS]

[CHARACTER SET charset_name]
[COLLATE collation_name]

- All of the above clauses are optional.
- If you skip the **CHARACTER SET** and **COLLATE** clauses, MySQL will set the default **CHARACTER SET**, and **COLLATE** for the new database.

Example:

Create database IF NOT EXISTS demoDatabase;

A RDBMS creates a new tablespace named **demoDatabase**. Some vendors instead support the **schema** keyword as shown below:

Create schema IF NOT EXISTS demoDatabase;

Create a Table in Database

The **CREATE TABLE** statement allows us to create a new table in a database.

Table definitions define the **table name** and **fields** within the table, and the **constraints/relationships** for the fields. Below is the syntax and an example.

Syntax

```
CREATE TABLE [IF NOT EXISTS]
table_name(
    column_1_definition,
    column_2_definition,
    ...,
    table_constraints
) ENGINE=storage_engine;
```

[IF NOT EXISTS]
ENGINE=storage_engine;

Both clauses above are *optional*.

Example

```
CREATE TABLE vehicles (
    vehicleId INT,
    year INT NOT NULL,
    make VARCHAR(100) NOT NULL
);
```

Defines a table named “**vehicles**,” which holds three fields.

[To learn more about Storage Engine, visit the Wiki document.](#)



Data Persistence and Longevity

Databases are intended to accrue information over long periods of time. In many cases, the data becomes increasingly valuable as it contains hidden trends and associations, which are not evident in the short-term.

Designing a database with well-chosen constraints on data relationships and values is the best means for maximizing this value.



Data Integrity

- ❑ **Data integrity** refers to the **accuracy**, **maintenance** and **consistency** of data stored in a database, and is a critical aspect to the **design**, **implementation**, and **usage** of any system that stores, processes, or retrieves data. Data with “integrity” is said to have a complete structure, (i.e., all characteristics defining the data are correct).
- ❑ To be most useful, data needs to meet the basic criteria. For example:
 - ❑ What use is a “**primary key**” if a table permits duplicate keys?
 - ❑ What does it mean if an *Orders* table references a *Customers* table through a foreign key, but some of those customers are missing?
 - ❑ What good is an *Employee* table if names are null?
 - ❑ Is *abcd+xyz.3#* a valid entry for a customer email address?
 - ❑ Do we want to allow “alien” in a column specifying gender?

Databases give us powerful tools to enforce data integrity.

Data Integrity Constraints

Enforce Data Integrity by Database Constraints

- Data integrity constraints refers to the **rules** and **policies** applied to maintain the quality of data.
- Data integrity constraints often include checks and corrections for invalid data based on a fixed schema or a predefined set of rules.
- **Constraints** can be classified into two types - *column-level* and *table-level*.
- There are two **main reasons** why using database constraints is a preferred way of enforcing data integrity.
 - **First**, constraints are inherent to the database engine, and as so, uses less system resources to perform their dedicated tasks. We resort to external user-defined integrity enforcement only if constraints are not sufficient to do the job properly.
 - **Second**, database constraints are always checked by the database engine before insert, update, or delete operations. Invalid operations are cancelled before the operation is undertaken; therefore, they are more reliable and robust for enforcing data integrity.

Data Integrity Constraints (continued)

There are four categories of data integrity constraints enforced by a database:

<u>1) Domain Integrity</u> SQL data types NOT NULL constraints Defaults	<u>2) Entity Integrity</u> Primary keys Unique constraints
<u>3) Referential Integrity</u> Foreign key constraints	<u>4) User-Defined Integrity:</u> Check constraints Triggers

constraint_expression:

```

| PRIMARY KEY [index_type] (index_col_name, ...) [index_option] ...
| FOREIGN KEY [index_name] (index_col_name, ...)
  REFERENCES tbl_name (index_col_name, ...)
  [ON DELETE reference_option]
  [ON UPDATE reference_option]
| UNIQUE [INDEX|KEY] [index_name]
  [index_type] (index_col_name, ...) [index_option] ...
| CHECK (check_constraints)

```

1 - Domain Integrity

- ❑ Domain Constraints are user-defined columns that help the user to enter the value according to the data type. A data type represents the type of data that can be stored and processed.
- ❑ SQL Data types comes under **Domain Integrity** category.

You can find all data types in below link:

<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>



Domain Integrity - SQL Data Types

SQL Data Types are a part of the language where vendors have diverged. There are, however, a few core types (or type aliases) supported by all vendors:

Data Type	SQL	Details
Numeric, integral	int(size)	size indicates maximum # digits
floating point, high precision	decimal(size, d)	size is the maximum # digits; d is # decimal places
Character	char(size) nchar(size)	Fixed-length character. Unused slots are padded. 'n' is unicode.
Character	varchar(size) nvarchar(size)	Variable-length character. 'n' is unicode.
Numeric, floating-point	double(size,d)	size is the maximum # digits; d is # decimal places



Domain Integrity - SQL Data Types (continued)

Data Type	SQL	Details
Text		A small non-binary string
Binary (images)	BLOB	Binary large object
Date/time	date, datetime, timestamp, year	timestamp is high precision
null	null	<ul style="list-style-type: none">•Multiple meanings:<ul style="list-style-type: none">•Unknown data.•Inapplicable data.•No value supplied.•Value undefined.
ENUM	enum	An enumeration: Each column value may be assigned one enumeration member.

To learn more about **SQL data types**, visit the [Wiki document](#).

Domain Integrity Example: SQL Data Types

In this example, we will create tables named `payments_info`, and we will specify column names with **Data Types**.

```
CREATE TABLE payments_info (  
    customerNumber int,  
    checkNumber varchar(50),  
    paymentDate date,  
    amount decimal(10,2),  
    is_completed BOOLEAN,  
    descriptions TEXT,  
    created_at TIMESTAMP,  
    price DECIMAL(10,2),  
    quantity SMALLINT, -- Allow negative  
    priority ENUM('Low', 'Medium', 'High')  
);
```


Domain Integrity: NOT NULL and DEFAULT Constraints

- ❖ The **NOT NULL** constraint prevents the column from having a NULL value; it specifies that a NULL value is not allowed.
- ❖ When designing tables, it is essential to consider where **NULL** values are appropriate. Most often, this is a common-sense decision. For example:
 - ❑ Can the first line of an address be NULL?
 - ❑ Can a customer's surname be NULL?
 - ❑ Can an employee's date of birth be NULL?
 - ❑ Can feedback provided by a customer be NULL?

Domain Integrity: NOT NULL and DEFAULT Constraints

- ❖ **Default** constraint specifies a default value for a column if no value is supplied when adding a record to a table.
 - *Note: If no **DEFAULT** is defined, a column defaults to NULL unless there is a **NOT NULL** constraint; in which case, a value must be provided in the **INSERT** statement.*
- ❖ DEFAULT values can use the return value of **built-in** and **user-defined** functions.

Domain Integrity Example: NOT NULL and DEFAULT Constraints

In the this example, we will demonstrate how to use the **NOT NULL** constraint and the **DEFAULT** constraint.

```
create table Salesfeedback(  
    customerFeedback varchar(2048) DEFAULT NULL,  
    createDate datetime NOT NULL DEFAULT current_timestamp,  
    customerReference varchar(15) DEFAULT 'OnlineSales',  
    username varchar(50) NOT NULL  
);
```

current_timestamp is built-in function in SQL.

2 - Entity Integrity

Entity Integrity ensures that there are no duplicate records within the table and that the field that identifies each record within the table is unique and never **null**.

Entity Integrity requires that each row of a table, representing a unique “entity,” can be uniquely identified. It is enforced to use **primary key** constraints and **unique** constraints.

Entity Integrity - Primary Keys

- ❏ **Primary key** constraints are defined in the **CREATE TABLE** statement, using either **column-level** or **table-level**.
- ❏ When the **primary key** constraint is defined for a table, the database will ensure that it is both present (not null) and unique.
- ❏ One table can only have one **primary key**, which may consist of a single column or multiple columns. If the **primary key** contains multiple columns, it is called a **composite primary key**, and the combination of the multiple columns must contain distinct values.

Remember: It is NOT required for each table to have a **primary key** constraint.

Example - Primary Key

Example: **field-level**

```
create table department_one(  
    id int primary key,  
    name varchar(32) NOT NULL  
);
```

Example: **table-level**

```
create table department_two(  
    id int,  
    name varchar(32) NOT NULL,  
    constraint pk_dept PRIMARY KEY(id)  
);
```

The *constraint* keyword + *identifier* are *optional*.
The *identifier* must be unique in the database.

Entity Integrity - Primary Keys - Groups of Columns

We can specify a **group of columns** as the **primary key** by using the **table-level**.

Syntax:

```
PRIMARY KEY (col1,col2,...)
```

Example:

```
CREATE TABLE Student (  
    std_id INT,  
    roll_id INT,  
    fullname VARCHAR(255),  
    is_completed BOOLEAN,  
    PRIMARY KEY (std_id , roll_id)  
);
```

Auto-Generated - Primary Keys

SQL provides mechanisms for generating primary keys. The “**AUTO_INCREMENT**” attribute can be used to generate a unique identity for new rows when you insert a new record to the table. The **AUTO_INCREMENT** indicates that the value of the column is incremented automatically, by one, whenever a new row is inserted into the table.

Example

```
CREATE TABLE checklists (  
    todo_id INT AUTO_INCREMENT,  
    task_id INT,  
    todo VARCHAR(255),  
    is_completed BOOLEAN,  
    PRIMARY KEY (todo_id , task_id)  
);
```


Universal Unique Identifier

A Universal Unique Identifier (UUID) is a generated number that is globally unique even if it is generated by two independent programs on different computers.

- The probability that a **UUID** is not unique is close enough to zero to be negligible.
- A **UUID** is generated from a timestamp (temporal difference) and computer node ID (spatial difference).
- A **UUID** value is a 128-bit number, and can be used to create a primary key in a distributed environment.
- To generate **UUID** values, use the `UUID()` function: **UUID()**

Example: `SELECT UUID() AS UUID_Value`

[For more information about UUID, visit the Wiki document](#)

Example - SQL Universal Unique Identifier

Let's take a look at an example of using **UUID** as the primary key. The following query creates a new table named **account**.

```
CREATE TABLE account (
  id varchar(130) PRIMARY KEY,
  name VARCHAR(255) );
```

To insert **UUID** values into the id column, we can use the **UUID()** function as follows:

```
INSERT INTO account(id, name)
VALUES(UUID(),'John Doe'),
      (UUID(),'Will Smith'),
      (UUID(),'Mary Jane');
```

Let's query data from the account table.

```
SELECT id, name FROM account;
```

Result

Result Grid		Filter Rows:	Edit:
	id	name	
▶	f38ed4f4-2e0c-11ed-b053-14b31f0e5a69	John Doe	
	f38ed8bf-2e0c-11ed-b053-14b31f0e5a69	Will Smith	
	f38ed9ab-2e0c-11ed-b053-14b31f0e5a69	Mary Jane	
*	NULL	NULL	

Entity Integrity - UNIQUE Constraints

- ❖ A field created with the **UNIQUE** constraint resembles a primary key – all values for that field must be unique – **no** duplicates are allowed.
- ❖ Like primary keys, a **UNIQUE** constraint can apply to compound fields, which means that each combination of those fields must be unique. Unlike primary keys, NULL is allowed, and you can have many **UNIQUE** constraints per table.
- ❖ Sometimes, the data in a column must be unique, even though the column does not act as **PRIMARY KEY** of the table. For example, the **CategoryName** column is unique in the categories table, but **CategoryName** is not a primary key of the table. In this case, we create a UNIQUE constraint, which determines that the values in a column or columns must be unique.

Example - UNIQUE Constraints

```
CREATE TABLE productcategories (  
    productId INT AUTO_INCREMENT,  
    productCode varchar(255),  
    productname varchar(50) UNIQUE,  
    CategoryName varchar(15) UNIQUE,  
    PRIMARY KEY (productId, productCode)  
);
```

A good business practice: In the example above, all of the product names and category names must be unique.

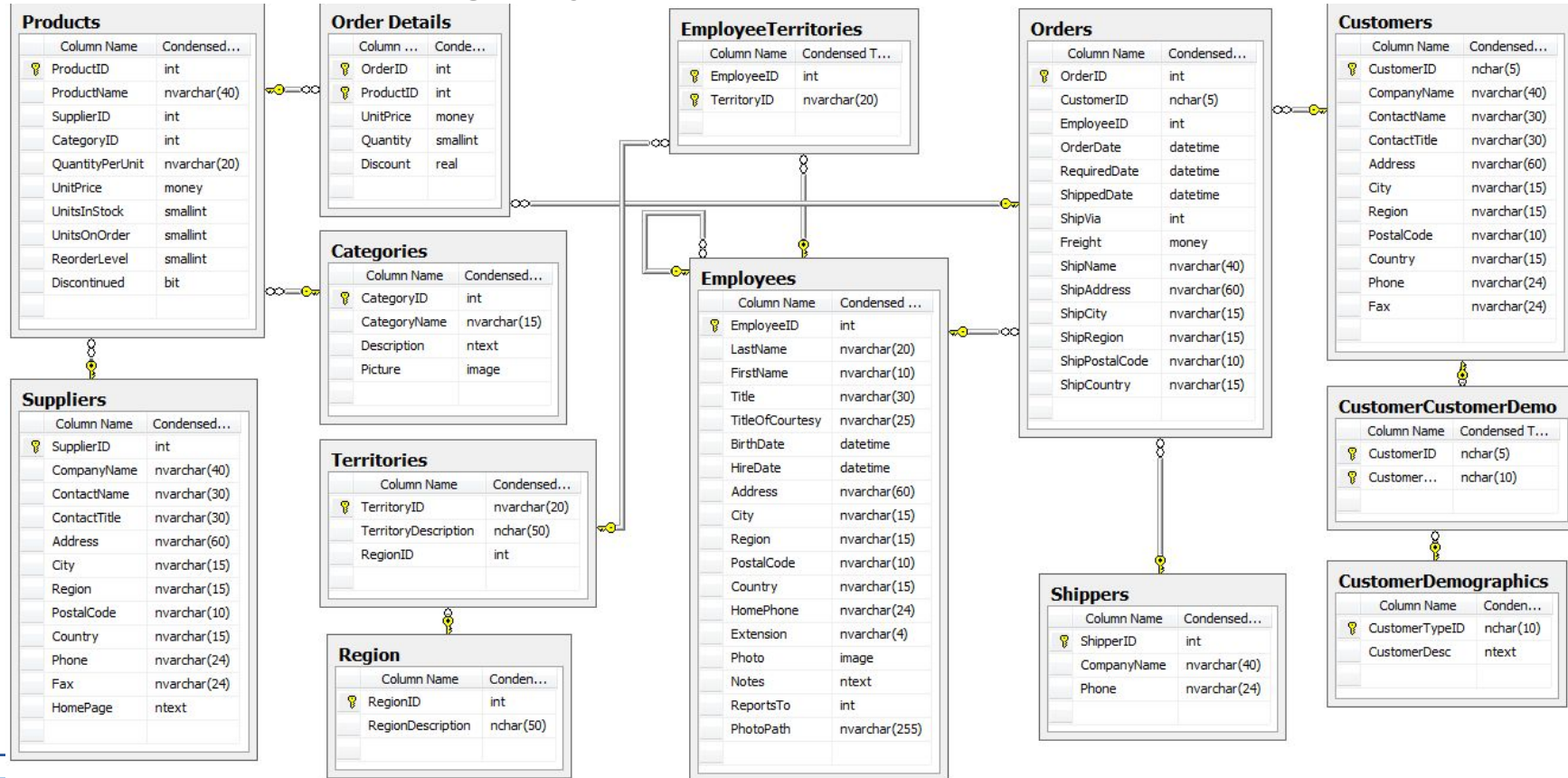
3 - Referential Integrity

- ❖ Referential Integrity is about enforcing the relationships defined between tables. This is done with “**Foreign Key**” constraints.
- ❖ The table with the foreign key can be called a **child table**, **referencing table**, or **foreign key table**. The table with the primary key can be called **parent table**, **referenced table**, or **primary key table**.
- ❖ **Foreign Key** constraints are defined at the table level.

NOT NULL is not required. Sometimes, it makes sense to allow a NULL **Foreign Key**.

Referential Integrity - Foreign Keys

Foreign Keys define relationships between tables.



Syntax: Foreign Key Constraints

Foreign Keys are created with a **CREATE TABLE** or **ALTER TABLE** statement. A table can have multiple **foreign keys** that refer to the primary keys of different parent tables. The definition must follow this **syntax**:

```
Create table tablename
Column_Name Datatype
[CONSTRAINT [identifier] FOREIGN KEY [Column_Name]
REFERENCES tbl_name (index_col_name)
[ON DELETE reference_option]
[ON UPDATE reference_option]
```

reference_option: **RESTRICT** | **CASCADE** | **SET NULL** | **NO ACTION** | **SET DEFAULT**

The **reference_option** provides actions that SQL will take when values in the parent table are deleted (on delete) and/or updated (on update).

CASCADE	All records related to that key will be deleted from their respective table(s).	set null	All related key fields will be set to null. Records will not be deleted, but the relationship is gone.
SET DEFAULT	All related key fields will assume the default value defined for them, but will only be worked with a PBXT engine.	RESTRICT	A key violation exception will be raised.

Example 1: Foreign Key Constraints

With Automatic Delete and Update

Step 1: Create a table as shown below:

```
CREATE TABLE department (  
  id INT(10) PRIMARY key AUTO_INCREMENT NOT null,  
  dname VARCHAR(20) NOT NULL DEFAULT 'pending',  
  dcode VARCHAR(10) UNIQUE NOT NULL,  
  depManager VARCHAR(15) DEFAULT "No boss",  
  depphone INT(7) UNIQUE,  
  depCreateDate timestamp,  
  depstatus ENUM('Active', 'Deactive', 'onhold')  
);
```

The depID in the employee table is the foreign key that references the **id** column in the department table. For each row in the **employee** table, you can find a corresponding row in the **department** table.

Step 2: Create a table as shown below:

```
CREATE TABLE employee (  
  empId INT(10) PRIMARY KEY AUTO_INCREMENT ,  
  eName VARCHAR(10) NOT NULL,  
  etitle VARCHAR(15) NOT NULL DEFAULT  
  'employee',  
  eManagerID INT(10),  
  eSalary DOUBLE(6, 3),  
  depID INT(10),  
  CONSTRAINT depFK FOREIGN KEY(depID)  
  REFERENCES department(id)  
    ON DELETE CASCADE  
    ON UPDATE CASCADE  
);
```


Example 2: Foreign Key Constraints

Example One-to-One Relationship

Without Automatic Delete and Update

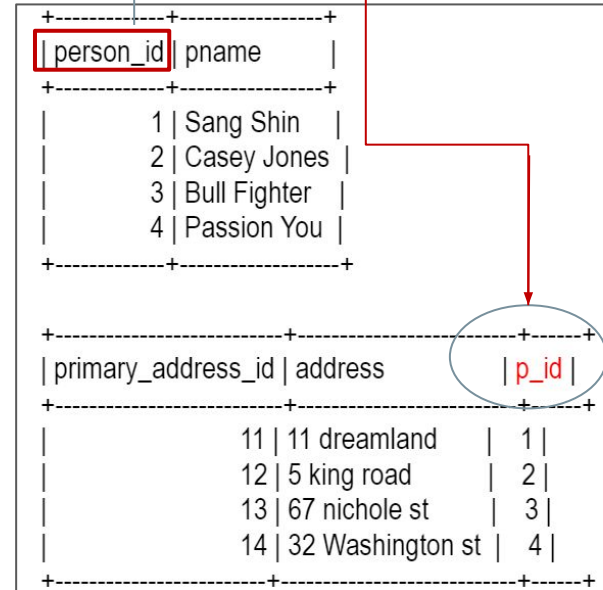
- A **person** has only one primary address.
- The **"person" table** has a 1-1 relationship with primary-address table.
- The **"primary_address" table** has a foreign key field referring to the primary key field of the **"person" table**.

```

/* Create "person" table */
CREATE TABLE person (
  person_id INT NOT NULL AUTO_INCREMENT,
  pname varchar(255) NOT NULL,
  PRIMARY KEY (person_id)
);

/* Create "primary_address" table with FOREIGN KEY */
CREATE TABLE primary_address (
  primary_address_id INT NOT NULL,
  address varchar(255) NOT NULL,
  p_id INT NOT NULL,
  PRIMARY KEY (primary_address_id),
  FOREIGN KEY (p_id) REFERENCES person (person_id)
);

```



person_id	pname
1	Sang Shin
2	Casey Jones
3	Bull Fighter
4	Passion You

primary_address_id	address	p_id
11	11 dreamland	1
12	5 king road	2
13	67 nichole st	3
14	32 Washington st	4

Example 3: Foreign Key Constraints

```
/* Create students table */
CREATE TABLE students (
  student_id INT NOT NULL AUTO_INCREMENT,
  sname varchar(255) NOT NULL,
  PRIMARY KEY (student_id)
) ;

/* Create course table */
CREATE TABLE course (
  course_id INT NOT NULL AUTO_INCREMENT,
  cname varchar(255) NOT NULL,
  PRIMARY KEY (course_id)
);

/* Create "student_course" join table with FOREIGN KEY to
both student and course tables. */
CREATE TABLE student_course (
  s_id INT NOT NULL,
  c_id INT NOT NULL,
  PRIMARY KEY (s_id, c_id),
  FOREIGN KEY (s_id) REFERENCES students (student_id),
  FOREIGN KEY (c_id) REFERENCES course (course_id) );
```

Example: Many-to-Many (n-m) Relationship

- A student takes many courses, and each course has many students.
- “**student**” and “**course**” have an **n-m** relationship.
- Therefore, we need a “**student_course**” **join table** (intersection table) called “**student-course**.”
 - The “**student_course**” **table** has foreign key fields to both “student” and “course” tables.
 - The “**student_course**” **table** primary key is typically a composite of the student and course primary keys.
 - The “**student_course**” **table** can contains other fields such as “**course registration date**.”

4 - User-Defined Integrity

We often want to constrain field values in a customized manner. The simplest way to accomplish this is with a **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 column, it will allow 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 sale(  
    saleId int PRIMARY KEY,  
    locationId int NOT NULL,  
    amount decimal check(amount >= 10.0)  
);
```

This constraint enforces a minimum sale of 10.00 currency units.

Example 1: CHECK Constraints

In this statement, we have two column **CHECK** constraints; one for the **cost** column and the other for the **price** column.

Step 1

```
CREATE TABLE parts (  
    part_no VARCHAR(18) PRIMARY KEY,  
    description VARCHAR(40),  
    cost DECIMAL(10,2 ) NOT NULL CHECK (cost >= 0),  
    price DECIMAL(10,2) NOT NULL CHECK (price >= 0)  
);
```

Step 2

```
INSERT INTO parts(part_no,  
description,cost,price)  
VALUES('A-001','Cooler',0,-100);
```

*The above insert query will give an Error because the value of the **price** column is negative. Expression ($price > 0$) evaluates to FALSE.*

Step 3

```
INSERT INTO parts(part_no,  
description,cost,price)  
VALUES  
('A-001','Cooler color is black',10,100);
```

The above insert query will execute successfully.

Example 2: CHECK Constraints

```
create table modelCar(  
    carName varchar(20) NOT NULL,  
    generation varchar(20) NOT NULL,  
    constraint chk_gene CHECK (generation in ('Antique', 'Classic', 'Modern'))  
);
```

In the above query, the **CHECK** Constraint ensures that the **genre** column will be limited to the three given values.

The following query will return **errors** due to the NON-Antique value for the generation column.

```
INSERT INTO `modelcar`  
(`carName`, `generation`)  
VALUES ('Honda', 'NON-Antique');
```

The below query will execute successfully.

```
INSERT INTO `modelcar`  
(`carName`, `generation`)  
VALUES ('Honda', 'Antique');
```

Alter Table Statement

The **Alter Table** Statement allow us to change/modify an existing table. We can:

- **Add** a new column.
- **Drop** an existing column.
- **Modify** an existing column.
- **Add** a constraint.
- **Drop** an existing constraint.

Alter Table - Examples

-- Add new column in customer table.

alter table customers add dob date default null;

-- Drop phone column from customers_new table

alter table customers_new drop phone;

-- Modify the data type of comments column in the orders table.

alter table orders modify comments varchar(2000);

-- Add multiple columns to the vehicles table.

alter table vehicles

ADD color VARCHAR(50),

ADD note VARCHAR(255);

Note: Refresh your tables after running these queries to see the results in MySQL Workbench.

Rename Table Name

MySQL offers two ways to rename tables.

The first one uses the **ALTER TABLE statement syntax**:

- ❑ **ALTER TABLE** old_table_name **RENAME** new_table_name;

The second way is to use **RENAME TABLE statement syntax**:

- ❑ **RENAME TABLE** old_table_name **TO** new_table_name;

The **RENAME TABLE** offers more flexibility. It allows renaming multiple tables in one statement. This can be useful when replacing a table with a new pre-populated version:

```
RENAME TABLE products TO products_old, products_new TO products;
```

The above statement is executed left to right, so there is no conflict naming **products_new** to **products** since the existing table has already been renamed to **products_old**. Furthermore, this is done automatically.



Example: Rename Table

In this example, we will create two tables, and then we will modify/rename the tables. Run the queries below in the MySQL Workbench.

Step 1

```
CREATE TABLE `animalinfo` (  
  `id` INT,  
  `name` CHAR(30) NOT NULL  
);
```

Step 2

```
CREATE TABLE `animalDescription` (  
  `age` INT,  
  `color` varChar(30),  
  `description` varchar(255)  
);
```

Step 3

```
Rename table animalinfo to animal_Details, animalDescription to animalinfo;
```

The above query will replace the ***animalinfo*** table name to ***animal_Details***, and then replace the ***animalDescription*** table name to ***animalinfo***. The above statement is executed left to right, so there is no naming conflict.



Example: Alter Statement

In this example, we will demonstrate how to add a new column and modify a column in an existing table.

Step 1: Create a new table named **EMP_DEMO** by using the query below:

```
CREATE TABLE EMP_DEMO(  
    EMP_ID INT AUTO_INCREMENT PRIMARY KEY,  
    FIRST_NAME VARCHAR(20) NOT NULL,  
    LAST_NAME VARCHAR(20),  
    BIRTH_DATE DATE DEFAULT '1900-01-01',  
    HIRE_DATE DATE DEFAULT (CURRENT_DATE()) );
```

Step 2: Add a new column to the **EMP_DEMO** table called **salary**. The **Alter Table Statement** is used to modify a table's structure as shown in the query below:

```
ALTER TABLE EMP_DEMO ADD COLUMN SALARY VARCHAR(40);
```

Example: Alter Statement

Step 3: Let's modify an existing column in the table.

```
ALTER TABLE EMP_DEMO MODIFY SALARY FLOAT;
```

The above query will change the data type of Salary column from VARCHAR to FLOAT.

Drop-Table Statement

The **Drop-Table** Statement removes the table definition and the data.

Syntax:

```
drop table tableName [RESTRICT | CASCADE ];
```

Note: The **RESTRICT** and **CASCADE** are *optional*, and are reserved for the future versions of MySQL.

Example: Drop Table

-- Drop vehicles table.

DROP TABLE vehicles;

-- Drop payments_new table.

DROP TABLE payments_new;

-- Drop multiple tables in single Drop Statement.

DROP TABLE animal_Details, animalinfo;

Practice Assignment

Complete the INSERT and DELETE exercise.

If you have any technical questions while performing the lab activity, ask your instructors for assistance.

Please take note that some questions in this Lab may be difficult for you because they cover advanced topics; however, we will cover such advanced topics in later presentations.

Knowledge Check

1. What are Referential Integrity constraints in MySQL?
2. What is a Primary Key constraint in MySQL ?
3. What is a Unique Key constraint in MySQL?
4. What qualities maximize the value of a database?
5. What is the difference between a Primary Key constraint and a Unique Key constraint?
6. What is a Composite Key?

Summary

Constraints can be specified when a table is created with the CREATE TABLE statement, or you can use the ALTER TABLE statement to create constraints even after the table is created.

Integrity constraints are used to ensure the accuracy and consistency of the data in a relational database. There are four categories of data integrity constraints enforced by a database: 1) domain, 2) entity, 3) referential, and 4) user-defined.

Some of the most commonly used constraints available in SQL include:

- NOT NULL Constraint – Ensures that a column cannot have a NULL value.
- DEFAULT Constraint – Provides a default value for a column when none is specified.
- UNIQUE Constraint – Ensures that all values in a column are different.
- PRIMARY Key – Uniquely identifies each row/record in a database table.
- FOREIGN Key – Uniquely identifies a row/record in any of the given database tables.
- CHECK Constraint – Ensures that all of the values in a column satisfy certain conditions.
- INDEX – Used to create and retrieve data from the database very quickly.

The SQL CREATE DATABASE statement is used to create a new SQL database, and the SQL CREATE TABLE statement is used to create a new table. SQL Data Types are attributes that specify the type of data of any object.

Questions?

