

Table Of Contents

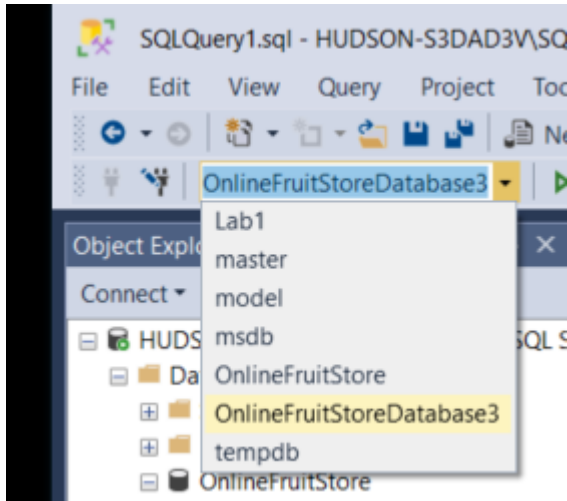
Sample Demonstration (SQL Server)	2
1. Unique & Not Null Constraints	2
1A. Detailed Explanation	3
1B. Another Example	4
i. Further Explanation	6
2. Primary Key Constraints	8
1A. Explanation of PRIMARY KEY	8
1B. Differences Between PRIMARY KEY and UNIQUE + NOT NULL	8
1C. Revised Example	9
1D. Explanation of the Table	9
1E. Summary	9
1F. MS SQL Server Key Icon	10
3. Demonstrations	11
4. Practice	13
4A. Redundancy in Data	13
4B. How to Eliminate Redundancy	14
4C. Identify The Primary Key & Foreign Key	15
4D. Store Data in Database	16
i. Format Breakdown (Foreign Key)	19
Exercise	22
Question 1:	22
Answer a.:	23
Answer b.:	23
Answer c.:	24
Question 2:	24
Answer:	24
Question 3:	25
Answer:	25
Question 4:	25
Answer:	25
Question 5:	25
Answer:	25

Sample Demonstration (SQL Server)

Main Resources: <https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver16>

1. Unique & Not Null Constraints

- **Database Name:** OnlineFruitStoreDatabase3
- **Table Name:** Fruit
- **Table Column & (Data Type):** FruitID (nvarchar 50), FruitName (nvarchar 50), Price (decimal 5,2), Quantity (integer), Supplier (nvarchar 50)

<p>-- To Create A Database</p> <pre>CREATE DATABASE OnlineFruitStoreDatabase3</pre>	<p>Make sure to execute the query in the right Database</p> 
<p>-- To Create A Table</p> <pre>CREATE TABLE Fruit(FruitID nvarchar(50) NOT NULL UNIQUE, FruitName nvarchar(50) NOT NULL, Price decimal(5,2) NOT NULL, Quantity integer NOT NULL, Supplier varchar(50) NOT NULL)</pre>	<pre>SELECT * FROM Fruit</pre>

1A. Detailed Explanation

- **CREATE TABLE:** This SQL command is used to create a new table in the database. The table will have the columns specified, each with its own data type and constraints.
- **FruitID nvarchar(50):**
 - **nvarchar(50)** means that this column will store variable-length string data up to 50 characters.
 - **NOT NULL** means that the column **must contain a value**; it **cannot be left empty**.
 - **UNIQUE** ensures that every value in this column is unique across all rows in the table. **No two rows can have the same FruitID.**
- **FruitName nvarchar(50) NOT NULL:**
 - This column will store the name of the fruit.
 - **NOT NULL** ensures that every row must have a value in the **FruitName** column.
- **Price decimal(5,2) NOT NULL:**
 - **decimal(5,2)** specifies a decimal number with up to 5 digits, with 2 digits to the right of the decimal point. For example, a value could be **123.45**.
 - **NOT NULL** ensures that a price is always provided.
- **Quantity integer NOT NULL:**
 - This column stores the quantity of the fruit.
 - **NOT NULL** ensures that the quantity must always be specified.
- **Supplier varchar(50) NOT NULL:**
 - **varchar(50)** means that this column will store a variable-length string up to 50 characters.
 - **NOT NULL** ensures that the supplier's name is always provided.

1B. Another Example

Let's say you want to create a table for storing information about books in a library. Each book must have a unique **BookID**, and no column should be left empty.

SQL

```
CREATE TABLE Books (  
  BookID nvarchar(20) NOT NULL UNIQUE,  
  Title nvarchar(100) NOT NULL,  
  Author nvarchar(50) NOT NULL,  
  PublicationYear integer NOT NULL,  
  ISBN nvarchar(13) NOT NULL UNIQUE,  
  CopiesAvailable integer NOT NULL  
);
```

In this example:

- **BookID nvarchar(20) NOT NULL UNIQUE:**
 - Stores a unique identifier for each book. For example, **BK001**.
 - **UNIQUE** ensures no two books can have the same **BookID**.
- **Title nvarchar(100) NOT NULL:**
 - Stores the title of the book, up to 100 characters.
- **Author nvarchar(50) NOT NULL:**
 - Stores the name of the author.
- **PublicationYear integer NOT NULL:**
 - Stores the year the book was published.
- **ISBN nvarchar(13) NOT NULL UNIQUE:**
 - Stores the ISBN number of the book, which is a unique identifier in the publishing industry.
 - **UNIQUE** ensures no two books can have the same ISBN.
- **CopiesAvailable integer NOT NULL:**
 - Stores the number of copies of the book available in the library.

Key Points to Remember:

- **NOT NULL:** Ensures that a column cannot have empty (null) values.
- **UNIQUE:** Ensures that all values in a column are different across rows.
- **Data Types:** Specify the kind of data that can be stored in each column (e.g., `nvarchar`, `decimal`, `integer`).

These constraints help maintain the integrity of your data by preventing duplicates and ensuring that important information is always present in each row.

i. Further Explanation

The examples provided primarily focus on the **UNIQUE** and **NOT NULL** constraints, which are essential for maintaining data integrity in a database. Here's a summary of these constraints:

1. **UNIQUE** Constraint

- **Purpose:** Ensures that all the values in a column are unique across all rows in the table. No two rows can have the same value for that column.
- **Usage:**
 - It is commonly used for columns that should have distinct values, like identifiers (e.g., **FruitID**, **BookID**, **ISBN**).
 - You can apply the **UNIQUE** constraint to one or more columns.
 - If a **UNIQUE** constraint is applied to multiple columns (a composite unique constraint), the combination of the values across these columns must be unique.

2. **NOT NULL** Constraint

- **Purpose:** Ensures that a column cannot have empty (null) values. Every row in the table must contain a value for the columns with this constraint.
- **Usage:**
 - Typically used for essential data that must be present for each entry, like names, prices, or quantities.
 - It prevents the insertion of rows with missing information in those columns.

Example: Combining **UNIQUE** and **NOT NULL**

- Here's a practical example to clarify how these constraints work together:
- **Table Definition:** **Employees**

SQL

```
CREATE TABLE Employees (  
    EmployeeID int NOT NULL UNIQUE,  
    FirstName nvarchar(50) NOT NULL,  
    LastName nvarchar(50) NOT NULL,  
    Email nvarchar(100) NOT NULL UNIQUE,  
    HireDate date NOT NULL  
);
```

Breakdown:

- **EmployeeID int NOT NULL UNIQUE:**
 - **NOT NULL:** Every employee must have an **EmployeeID**, and it cannot be left empty.
 - **UNIQUE:** Each **EmployeeID** must be distinct. No two employees can have the same **EmployeeID**.
- **Email nvarchar(100) NOT NULL UNIQUE:**
 - **NOT NULL:** Every employee must have an email address.
 - **UNIQUE:** Each email address must be unique to ensure no two employees share the same email.
- **FirstName, LastName, HireDate:**
 - These columns are also **NOT NULL**, meaning that every employee must have a first name, last name, and hire date.

Real-World Scenario:

Imagine a scenario where you're managing a database for a company. The **EmployeeID** and **Email** are crucial identifiers for each employee:

- **UNIQUE** ensures that each employee is uniquely identifiable by their **EmployeeID** and that no two employees accidentally share the same email address.
- **NOT NULL** ensures that you don't have incomplete records—every employee must have the necessary information (like **FirstName**, **LastName**, and **HireDate**) to be a valid entry in the database.

These constraints together help in creating a robust and reliable database schema, where data accuracy and consistency are maintained.

2. Primary Key Constraints

SQL

```
CREATE TABLE Fruit(  
    FruitID nvarchar(50) PRIMARY KEY,  
    FruitName nvarchar(50),  
    Price decimal(5,2),  
    Quantity integer,  
    Supplier varchar(50)  
);
```

1A. Explanation of PRIMARY KEY:

- **PRIMARY KEY:**
 - The **PRIMARY KEY** constraint uniquely identifies each record in a table.
 - A column defined as the **PRIMARY KEY** must be unique and cannot contain **NULL** values. This means every entry in the **FruitID** column must be unique, and every record must have a **FruitID**.
 - Each table can have only one **PRIMARY KEY**, but that key can consist of one or more columns (a composite key).

1B. Differences Between PRIMARY KEY and UNIQUE + NOT NULL:

- **PRIMARY KEY:**
 - Automatically implies both **UNIQUE** and **NOT NULL** constraints.
 - It's used to uniquely identify each row in a table.
 - There can be only one **PRIMARY KEY** per table, but it can span multiple columns if needed.
- **UNIQUE + NOT NULL:**
 - You can have multiple **UNIQUE** constraints in a table.
 - While **UNIQUE** ensures uniqueness, it doesn't automatically prevent **NULL** values unless explicitly combined with **NOT NULL**.
 - A **UNIQUE** column can be **NULL** if **NOT NULL** is not specified.

1C. Revised Example:

- Using **PRIMARY KEY** instead of **UNIQUE** and **NOT NULL** simplifies your table definition because **PRIMARY KEY** automatically ensures both uniqueness and non-nullability.
- Here's what your table definition looks like with a **PRIMARY KEY**:

SQL

```
CREATE TABLE Fruit (  
    FruitID nvarchar(50) PRIMARY KEY,  
    FruitName nvarchar(50),  
    Price decimal(5,2),  
    Quantity integer,  
    Supplier varchar(50)  
);
```

1D. Explanation of the Table:



- **FruitID nvarchar(50) PRIMARY KEY:**
 - This column is the unique identifier for each fruit. Every fruit in the table will have a unique **FruitID**, and it cannot be **NULL**.
- **FruitName nvarchar(50):**
 - Stores the name of the fruit. It is not required to be unique or non-null unless specified otherwise.
- **Price decimal(5,2):**
 - Stores the price of the fruit with up to 5 digits, including 2 decimal places.
- **Quantity integer:**
 - Stores the quantity of the fruit in stock.
- **Supplier varchar(50):**
 - Stores the name of the supplier providing the fruit.

1E. Summary:

Using a **PRIMARY KEY** for **FruitID** is a clear and efficient way to ensure that each row in your table is uniquely identifiable. The other columns (**FruitName**, **Price**, **Quantity**, **Supplier**) are not constrained by **NOT NULL** or **UNIQUE**, so they can contain duplicate values or **NULL** unless you add additional constraints to them.

1F. MS SQL Server Key Icon

- If you go to the **DESIGN** view of that table after executing this command, the primary key should be visible like this:

	Column Name	Data Type	Allow Nulls
	emp_id	int	<input type="checkbox"/>
	e_name	varchar(100)	<input type="checkbox"/>
	e_salary	decimal(8, 2)	<input checked="" type="checkbox"/>
	e_age	tinyint	<input checked="" type="checkbox"/>
	e_join_date	datetime	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

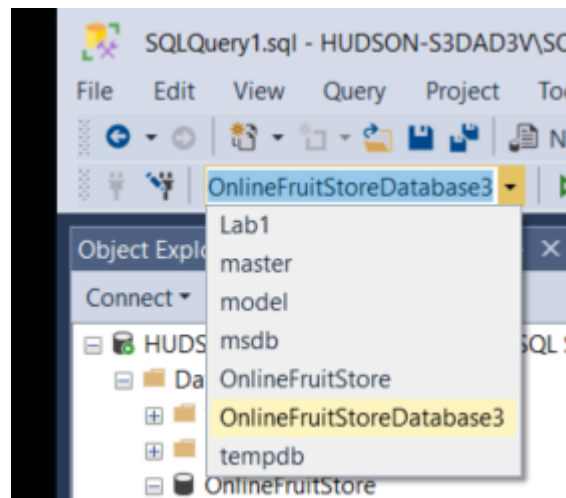
- The **yellow key** symbol on emp_id and e_name fields shows that these are the primary keys.

3. Demonstrations

-- To Create A Database

```
CREATE DATABASE OnlineFruitStoreDatabase3
```

Make sure to execute the query in the right Database



-- To Create A Table

```
CREATE TABLE Fruit(  
FruitID nvarchar(50) PRIMARY KEY,  
FruitName nvarchar(50),  
Price decimal(5,2),  
Quantity integer,  
Supplier varchar(50)  
)
```

View From: Table Design

	Column Name	Data Type	Allow Nulls
	FruitID	nvarchar(50)	<input type="checkbox"/>
	FruitName	nvarchar(50)	<input checked="" type="checkbox"/>
	Price	decimal(5, 2)	<input checked="" type="checkbox"/>
	Quantity	int	<input checked="" type="checkbox"/>
	Supplier	varchar(50)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

-- Insert Data

```
INSERT INTO Fruit  
(FruitID, FruitName, Price, Quantity, Supplier)  
VALUES  
('F01', 'Apple', 1.50, 50, 'ABC'),  
('F02', 'Orange', 1.50, 50, 'HJK'),  
('F03', 'Banana', 4.50, 60, 'ABC'),  
('F04', 'Kiwi', 4.00, 30, 'XYZ')
```

-- Display The Table

-- Double Check the value insert is that correct?

```
SELECT * FROM Fruit
```

-- Insert Data NULL to Try Primary Constraints

```
INSERT INTO Fruit  
(FruitID, FruitName, Price, Quantity, Supplier)  
VALUES  
(NULL, 'Apple', 6.50, 100, 'PQR')
```

Message Display:

*Cannot insert the value NULL into column 'FruitID',
table 'OnlineFruitStoreDatabase3.dbo.Fruit';
column does not allow nulls. INSERT fails.
The statement has been terminated.*

Result After The Query:

FruitID	FruitName	Price	Quantity	Supplier
F01	Apple	1.50	50	ABC
F02	Orange	1.50	50	HJK
F03	Banana	4.50	60	ABC
F04	Kiwi	4.00	30	XYZ

4. Practice

Fruit Table

FruitID	FruitName	Price	Quantity	Supplier
F01	Apple	1.50	50	ABC
F02	Orange	1.50	50	HJK
F03	Banana	4.50	60	ABC
F04	Kiwi	4.00	30	XYZ

Order Table

OrderID	CustomerName	FruitName	Price	Quantity
001	Jane	Apple	1.50	10
002	William	Apple	1.50	15
003	Ben	Orange	4.50	3
004	Ali	Kiwi	4.00	5

4A. Redundancy in Data

There is redundancy between the **Fruit** and **Customer** tables, particularly in the **Price** and **FruitName** fields. Here's why:

- **Price:**
 - The **Price** of the fruit is listed both in the **Fruit** table and the **Customer** table. Since the price of a fruit doesn't usually change per customer, storing it separately on the **Customer** table is redundant.
- **FruitName:**
 - The **FruitName** is stored in both tables. This creates redundancy, especially if the name of the fruit changes, as it would need to be updated in multiple places.

4B. How to Eliminate Redundancy:

- **Normalize the Data:**

- Instead of storing **Price** and **FruitName** in the **Customer** table, you can store only the **FruitID** in the **Customer** table and then join this with the **Fruit** table whenever you need to retrieve the price or name.
- This way, the **Customer** table would look like this:

Revised Customer Table

OrderID	CustomerName	FruitID	Quantity
001	Jane	F01	10
002	William	F01	15
003	Ben	F02	3
004	Ali	F04	5

- By using **FruitID**, you eliminate the redundancy of storing the **Price** and **FruitName** multiple times.

4C. Identify The Primary Key & Foreign Key

To identify the primary and foreign keys in the **Fruit** and **Customer** tables, you can follow these steps:

1. Primary Key:

- A primary key is a unique identifier for each record in a table. It must contain unique values and cannot contain **NULL** values.
- **Fruit Table:**
 - **Primary Key:** **FruitID**
 - Each fruit is uniquely identified by its **FruitID**.
- **Customer Table:**
 - **Primary Key:** **OrderID**
 - Each customer order is uniquely identified by its **OrderID**.

2. Foreign Key:

- A foreign key is a field (or fields) in one table that uniquely identifies a row of another table. The foreign key establishes a relationship between the two tables.
- **Customer Table:**
 - **Foreign Key:** **FruitID**
 - The **FruitID** in the **Customer** table would be the foreign key that links to the **FruitID** primary key in the **Fruit** table.
 - This creates a relationship between the fruit ordered by the customer and the details of that fruit on the **Fruit** table.

3. Summary:

- **Fruit Table:**
 - **Primary Key:** **FruitID**
- **Customer Table:**
 - **Primary Key:** **OrderID**
 - **Foreign Key:** **FruitID** (linked to **FruitID** in the **Fruit** table)
- This structure ensures that each order can reference a specific fruit and its details (like price, quantity available, and supplier) without redundancy.

4D. Store Data in Database

- **Database Name:** OnlineFruitStoreDatabase3

<div>-- To Create A Database</div> <div>CREATE DATABASE OnlineFruitStoreDatabase3</div>	<div>-- To Create a Fruit Table</div> <div>CREATE TABLE Fruit(FruitID nvarchar(50) PRIMARY KEY, FruitName nvarchar(50), Price decimal(5,2), Quantity integer, Supplier varchar(50))</div>																														
<div>-- Insert Fruit Table Data</div> <div>INSERT INTO Fruit (FruitID, FruitName, Price, Quantity, Supplier) VALUES ('F01', 'Apple', 1.50, 50, 'ABC'), ('F02', 'Orange', 1.50, 50, 'HJK'), ('F03', 'Banana', 4.50, 60, 'ABC'), ('F04', 'Kiwi', 4.00, 30, 'XYZ')</div>	<div>-- To Create a Order Table</div> <div>CREATE TABLE OrderTable(OrderID nvarchar(50) PRIMARY KEY, CustomerName nvarchar(50), Quantity integer, FruitID nvarchar(50) FOREIGN KEY REFERENCES Fruit(FruitID))</div>																														
<div>-- Insert Order Table Data</div> <div>INSERT INTO OrderTable (OrderID, CustomerName, Quantity, FruitID) VALUES ('001', 'Jane', 10, 'F01'), ('002', 'William', 15, 'F01'), ('003', 'Ben', 3, 'F02'), ('004', 'Ali', 5, 'F04')</div>	<div>SELECT * FROM Fruit</div> <table><tr><th></th><th>FruitID</th><th>FruitName</th><th>Price</th><th>Quantity</th><th>Supplier</th></tr><tr><td>1</td><td>F01</td><td>Apple</td><td>1.50</td><td>50</td><td>ABC</td></tr><tr><td>2</td><td>F02</td><td>Orange</td><td>1.50</td><td>50</td><td>HJK</td></tr><tr><td>3</td><td>F03</td><td>Banana</td><td>4.50</td><td>60</td><td>ABC</td></tr><tr><td>4</td><td>F04</td><td>Kiwi</td><td>4.00</td><td>30</td><td>XYZ</td></tr></table>		FruitID	FruitName	Price	Quantity	Supplier	1	F01	Apple	1.50	50	ABC	2	F02	Orange	1.50	50	HJK	3	F03	Banana	4.50	60	ABC	4	F04	Kiwi	4.00	30	XYZ
	FruitID	FruitName	Price	Quantity	Supplier																										
1	F01	Apple	1.50	50	ABC																										
2	F02	Orange	1.50	50	HJK																										
3	F03	Banana	4.50	60	ABC																										
4	F04	Kiwi	4.00	30	XYZ																										

SELECT * FROM OrderTable

	OrderID	CustomerName	Quantity	FruitID
1	001	Jane	10	F01
2	002	William	15	F01
3	003	Ben	3	F02
4	004	Ali	5	F04

-- Selects only those rows where the Price column is less than or equal to 1.50.

SELECT * FROM Fruit WHERE Price <= 1.50;

	FruitID	FruitName	Price	Quantity	Supplier
1	F01	Apple	1.50	50	ABC
2	F02	Orange	1.50	50	HJK

-- Filters the rows to include only those where the CustomerName column exactly matches 'Jane'.

-- Note that this comparison is case-sensitive, so it will not match names like 'jane' or 'JANE' unless your database collation settings specify otherwise.

SELECT * FROM OrderTable WHERE
CustomerName = 'Jane'

	OrderID	CustomerName	Quantity	FruitID
1	001	Jane	10	F01

-- Deleting by Customer Name

-- Purpose: This statement deletes all rows where the CustomerName is 'Ali'.

-- Impact: If there are multiple orders placed by a customer named 'Ali', all those orders will be deleted. The OrderID or any other columns in those rows are not considered for this deletion.

	OrderID	CustomerName	Quantity	FruitID
1	001	Jane	10	F01
2	002	William	15	F01
3	003	Ben	3	F02
4	004	Ali	5	F04
5	007	Ali	5	F04
6	008	Ali	5	F04

DELETE FROM OrderTable WHERE CustomerName
= 'Ali';

	OrderID	CustomerName	Quantity	FruitID
1	001	Jane	10	F01
2	002	William	15	F01
3	003	Ben	3	F02

-- Deleting by OrderID

-- Purpose: This statement deletes the row where the OrderID is '004'

-- Impact: This operation targets a specific order based on its unique OrderID. Only the row with OrderID '004' will be deleted. If no such OrderID exists, no rows will be deleted.

	OrderID	CustomerName	Quantity	FruitID
1	001	Jane	10	F01
2	002	William	15	F01
3	003	Ben	3	F02
4	004	Ali	5	F04
5	007	Ali	5	F04
6	008	Ali	5	F04

DELETE FROM OrderTable WHERE OrderID = '004';

	OrderID	CustomerName	Quantity	FruitID
1	001	Jane	10	F01
2	002	William	15	F01
3	003	Ben	3	F02
4	007	Ali	5	F04
5	008	Ali	5	F04

-- Foreign Key can accept NULL Value but Primary Key cannot

INSERT INTO OrderTable

(OrderID, CustomerName, Quantity, FruitID)

VALUES

('010','Alex',7,NULL)

	OrderID	CustomerName	Quantity	FruitID
1	001	Jane	10	F01
2	002	William	15	F01
3	003	Ben	3	F02
4	007	Ali	5	F04
5	008	Ali	5	F04
6	010	Alex	7	NULL

i. Format Breakdown (Foreign Key)

Syntax:

```
sql
```

```
FOREIGN KEY (ColumnName) REFERENCES ReferencedTable(ReferencedColumn)
```

Components:

- **FOREIGN KEY (ColumnName):**
 - This specifies the column(s) in the current table that will act as a foreign key.
 - In your example: **FruitID** is the column in the **OrderTable** that will reference the **Fruit** table.
- **REFERENCES ReferencedTable(ReferencedColumn):**
 - **ReferencedTable** is the name of the table that contains the column you want to reference.
 - **ReferencedColumn** is the column in the **ReferencedTable** that the foreign key column refers to.
 - In your example: **Fruit** is the referenced table, and **FruitID** is the column in the **Fruit** table that **FruitID** in the **OrderTable** will refer to.

In This Example:

- **FruitID** in the **OrderTable** is declared as a foreign key.
- This means that any value inserted into the **FruitID** column of **OrderTable** must already exist in the **FruitID** column of the **Fruit** table.
- If you try to insert an **OrderTable** row with a **FruitID** that doesn't exist in the **Fruit** table, the database will return an error, thus maintaining referential integrity.

Purpose and Benefits:

- **Data Integrity:** Ensures that relationships between tables are consistent.
- **Prevents Orphaned Records:** Prevents records in the **OrderTable** from referring to non-existent fruits.
- **Data Consistency:** Helps maintain consistency across related tables.

By using foreign keys, you enforce rules that help keep your data accurate and meaningful across related tables.

Uniqueness of Foreign Keys

- **Foreign Key Uniqueness:** A foreign key itself does not need to be unique. The foreign key constraint is used to enforce referential integrity between tables, meaning that values in the foreign key column(s) must match values in the primary key column(s) of the referenced table. However, the foreign key column(s) **can have duplicate values** if the referenced table's primary key column(s) allows it.
- **Example:** In the **OrderTable**, if **FruitID** is a foreign key referencing the **Fruit** table's **FruitID**, multiple rows in **OrderTable** can have the same **FruitID** if the same fruit is ordered multiple times.

```
-- Fruit table
CREATE TABLE Fruit (
    FruitID nvarchar(50) PRIMARY KEY,
    FruitName nvarchar(50)
);

-- OrderTable with a foreign key
CREATE TABLE OrderTable (
    OrderID nvarchar(50) PRIMARY KEY,
    CustomerName nvarchar(50),
    Quantity integer,
    FruitID nvarchar(50),
    FOREIGN KEY (FruitID) REFERENCES Fruit(FruitID)
);
```

- In this case, **FruitID** in **OrderTable** can have multiple rows with the same value, provided that each value exists in the **Fruit** table.

NULL Values in Foreign Keys

- **NULL Values:** Foreign key columns can accept **NULL** values unless otherwise constrained by additional rules or constraints. A **NULL** value in a foreign key column means that the row does not reference any row in the referenced table, which is permissible unless the foreign key column is explicitly defined as **NOT NULL**.
- **Example:** If **FruitID** in **OrderTable** is allowed to be **NULL**, this means an order might not specify a fruit (or the fruit is not known at the time of insertion).

```
-- OrderTable with nullable foreign key
CREATE TABLE OrderTable (
  OrderID nvarchar(50) PRIMARY KEY,
  CustomerName nvarchar(50),
  Quantity integer,
  FruitID nvarchar(50) NULL,
  FOREIGN KEY (FruitID) REFERENCES Fruit(FruitID)
);
```

- Here, **FruitID** can be **NULL**, meaning some orders might not have an associated fruit.

Summary

- **Foreign Key Uniqueness:** Foreign keys do not need to be unique in the referencing table.
- **NULL Values:** Foreign key columns can accept **NULL** values unless specifically constrained by a **NOT NULL constraint**.

These features allow for flexible relationships and data modeling while maintaining referential integrity.

Exercise

Question 1:

- **Database Name:** Lab2
- **Table Name:** Publisher
- **PublisherID:** is a primary key in Publisher Table

PublisherID	Name	Address
P01	Pearson	Bukit Jalil
P02	Deitel	Puchong
P03	Rainbow	Subang
P04	MacHill	Kuala Lumpur

Attributes	Data Type
PublisherID	nvarchar(50)
Name	nvarchar(50)
Address	nvarchar(50)

- **Table Name:** Book
- **Note:** BookID is a primary key in Book Table
- **Note:** PublisherID is a foreign key in Book Table

BookID	Name	Author	Price	PublishedDate	PublisherID
B01	Maths	J.Wenton	50.60	10 Jan 2016	P01
B02	Science	S.Hanson	100.00	12 Feb 2016	P01
B03	English	K.Vince	89.30	9 March 2016	P02
B04	Biology	K.Vince	150.80	24 April 2016	P03
B05	Computing	J.Denzin	NULL	NULL	NULL

Attributes	Data Type
BookID	nvarchar(50)
Name	nvarchar(50)
Author	nvarchar(50)
Price	decimal(10,2)
PublishedDate	date
PublisherID	nvarchar(50)

- a. Using MS SQL Server, create a new database Lab2
- b. Write a query to create the tables given above
- c. Write a query to add each row of data to the tables

Answer a.:

```
CREATE DATABASE Lab2
```

Answer b.:

<pre>CREATE TABLE Publisher(PublisherID nvarchar(50) PRIMARY KEY, Name nvarchar(50), Address nvarchar(50))</pre>	<pre>CREATE TABLE Book(BookID nvarchar(50) PRIMARY KEY, Name nvarchar(50), Author nvarchar(50), Price decimal(10,2), PublishedDate date, PublisherID nvarchar(50) FOREIGN KEY REFERENCES Publisher(PublisherID),)</pre>
--	---

Answer c.:

<pre>INSERT INTO Publisher (PublisherID, Name, Address) VALUES ('P01', 'Pearson', 'Bukit Jalil'), ('P02', 'Deitel', 'Puchong'), ('P03', 'Rainbow', 'Subang'), ('P04', 'MacHill', 'Kuala Lumpur')</pre>	<pre>INSERT INTO Book (BookID, Name, Author, Price, PublishedDate, PublisherID) VALUES ('B01', 'Maths', 'J.Wenton', 50.60, '10 Jan 2016', 'P01'), ('B02', 'Science', 'S.Hanson', 100.00, '12 Feb 2016', 'P01'), ('B03', 'English', 'K.Vince', 89.30, '9 March 2016', 'P02'), ('B04', 'Biology', 'K.Vince', 150.80, '24 April 2016', 'P03'), ('B05', 'Computing', 'J.Denzin', NULL, NULL, NULL)</pre>
--	--

Question 2:

- Try to insert a new row in Book Table where PublisherID does not exist in Publisher table. Can or cannot? Why?

Answer:

```
SQL> INSERT INTO Book
(BookID, Name, Author, Price, PublishedDate, PublisherID)
VALUES
('B09','Psychology','C.Maas',87.00,'15 May 2016','P05')
```

Cannot. The **PublisherID** column in the **Book** table is a foreign key that references the **PublisherID** in the **Publisher** table. SQL Server enforces referential integrity, meaning that you cannot insert a **PublisherID** in the **Book** table that doesn't exist in the **Publisher** table. If you try, the database will return an error.

For example, **PublisherID 'P05'** is being inserted into the **Book** table. The **PublisherID 'P05'** does not exist in the **Publisher** table. Since **PublisherID** is a foreign key in the **Book** table that references the **PublisherID** in the **Publisher** table, SQL Server enforces a rule called referential integrity. This rule ensures that every **PublisherID** in the **Book** table must exist in the **Publisher** table. **Result:** The database will return an error, preventing the insertion.

Question 3:

- Try to delete the row for Publisher named Deitel. Can or cannot? Why

Answer:

```
SQL> DELETE FROM Publisher WHERE Name = 'Deitel'
```

Cannot. The **Deitel** publisher has a **PublisherID** of 'P02', which is referenced in the **Book** table by the book 'English' (BookID 'B03'). Because **PublisherID** 'P02' is being used as a foreign key in the **Book** table, deleting it would create a situation where the **Book** table has a **PublisherID** that no longer exists in the **Publisher** table. This would violate referential integrity. The deletion will fail, and SQL Server will return an error.

Question 4:

- Try to delete the row for Publisher named MacHill. Can or cannot? Why

Answer:

```
SQL> DELETE FROM Publisher WHERE Name = 'MacHill'
```

Can. The **MacHill** publisher has a **PublisherID** of 'P04', but it is not referenced by any book in the **Book** table. Therefore, SQL Server will allow this operation. **However**, if you tried to delete a publisher that is referenced in the **Book** table (like **Deitel**), the deletion would be blocked unless you use a **CASCADE DELETE** or first delete the referencing rows in the **Book** table. The **PublisherID** 'P04' is **not referenced by any rows** in the **Book** table. Since there are no dependencies on this row, deleting it will not violate referential integrity.

Result: The deletion will succeed, and the row for 'MacHill' will be removed from the **Publisher** table.

Question 5:

- Try to delete any row for a book. Can or cannot? Why

Answer:

```
SQL> DELETE FROM Book WHERE Name = 'Biology'
```

Can. You can delete rows from the **Book** table as long as no other tables are referencing these rows with foreign keys. Since there are no dependent tables referencing the **Book**, you can delete any row without causing a **referential integrity** issue. SQL Server will allow this deletion without any issues.