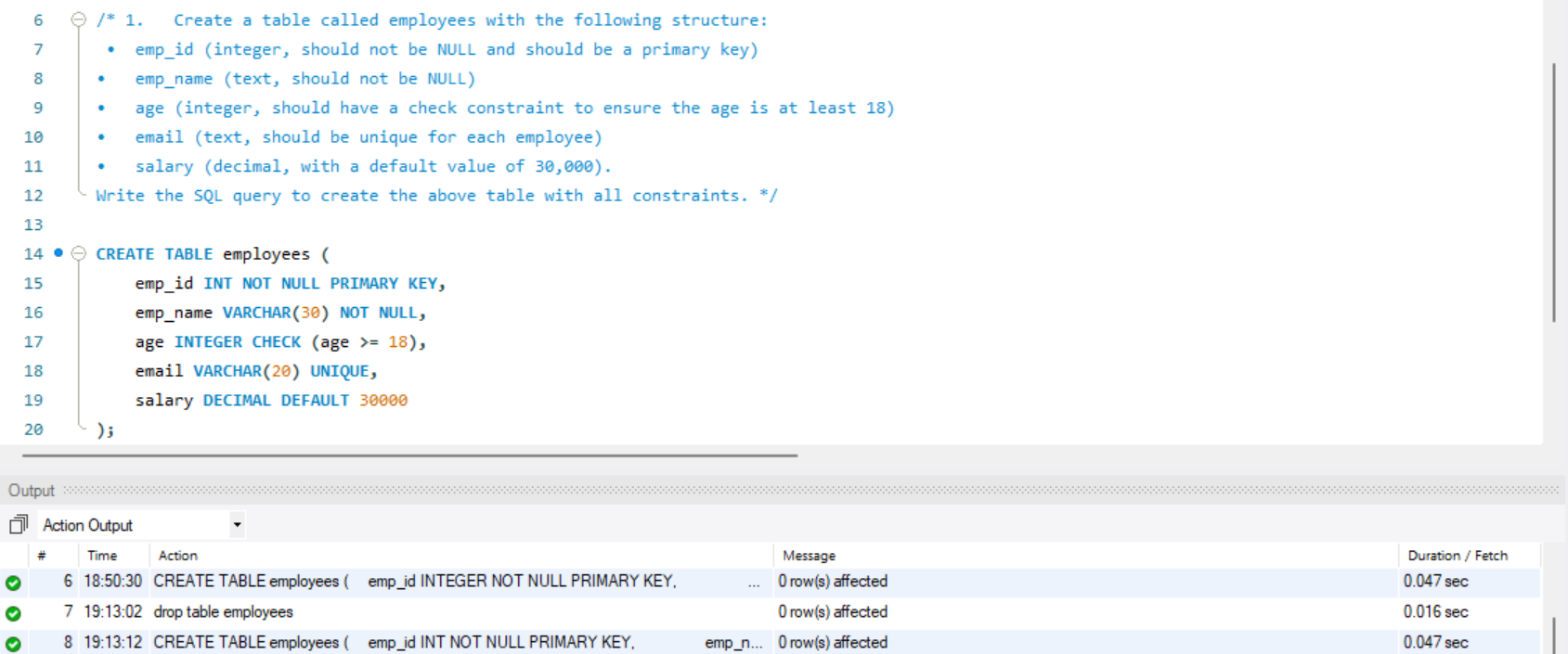
**SQL Assignment**

1. Create a table called employees with the following structure:

* emp\_id (integer, should not be NULL and should be a primary key)
* emp\_name (text, should not be NULL)
* age (integer, should have a check constraint to ensure the age is at least 18)
* email (text, should be unique for each employee)
* salary (decimal, with a default value of 30,000).

Write the SQL query to create the above table with all constraints.



**2. Explain the purpose of constraints and how they help maintain data integrity in a database. Provide examples of common types of constraints.**

**Ans. Purpose of Constraints**

Constraints are rules or conditions applied to database columns or tables to ensure the data's validity, consistency, and accuracy. They help maintain data integrity, which means that the data stored in the database is accurate, reliable, and conforms to the defined rules. Without constraints, inserting incorrect, inconsistent, or invalid data into a database would be easier.

**Constraints are essential for enforcing the following types of data integrity:**

* **Entity Integrity**: Ensures that each row in a table is uniquely identifiable (usually by a primary key).
* **Referential Integrity**: Ensures that relationships between tables are consistent (i.e., foreign keys point to valid rows in other tables).
* **Domain Integrity**: Ensures that values in a column are of the correct data type, within an acceptable range, and follow specific rules.
* **User-Defined Integrity**: Enforces custom business rules and logic, such as constraints on data formats or the relationship between different columns.

**Common Types of Constraints**

**PRIMARY KEY Constraint**

Ensures that each row in a table has a unique identifier. No two rows can have the same value for the primary key column(s), and the column(s) cannot contain NULL values.

**Example:**

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(30) NOT NULL

);

**FOREIGN KEY Constraint**

Ensures referential integrity by linking columns in one table to the primary key in another table. This guarantees that relationships between tables are valid.

**Example:**

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

emp\_id INT,

FOREIGN KEY (emp\_id) REFERENCES employees (emp\_id)

);

**UNIQUE Constraint**

Ensures that all values in a column (or a group of columns) are unique across the table. Unlike the primary key, a table can have multiple unique constraints.

**Example:**

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(30) NOT NULL,

email VARCHAR(20) UNIQUE

);

**NOT NULL Constraint**

Ensures that a column does not accept NULL values, meaning that every row must have a value for this column.

**Example:**

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(30) NOT NULL,

age INT

);

**CHECK Constraint**

Enforces a condition on the values that can be inserted into a column. This can be used to ensure that data falls within a specific range or meets certain criteria.

**Example:**

CREATE TABLE employees (

emp\_id INTEGER PRIMARY KEY,

emp\_name VARCHAR(30) NOT NULL,

age INTEGER CHECK (age >= 18)

);

**DEFAULT Constraint**

Specifies a default value for a column if no value is provided during insertion.

**Example:**

CREATE TABLE employees (

emp\_id INTEGER PRIMARY KEY,

emp\_name VARCHAR(30) NOT NULL,

salary DECIMAL DEFAULT 30000

);

**3. Why would you apply the NOT NULL constraint to a column? Can a primary key contain NULL values? Justify your answer.**

**Ans. NOT NULL** constraint ensures that a column always contains a valid value and is never NULL. This is important for data integrity, consistency, and accurate database operations.

**A primary key cannot contain NULL values because:**

1. It must uniquely identify each row in the table.
2. NULL is not considered a valid or comparable value for uniqueness or referential integrity.

**4. Explain the steps and SQL commands used to add or remove constraints on an existing table. Provide an example for both adding and removing a constraint.**

**Steps to Add or Remove Constraints**

**1. Adding a Constraint to an Existing Table**

To add a constraint to an existing table, you use the ALTER TABLE statement with the ADD CONSTRAINT clause. The syntax will depend on the type of constraint you're adding.

**SQL Syntax to Add a Constraint:**

ALTER TABLE table\_name

ADD CONSTRAINT constraint\_name constraint\_type (column\_name);

**Example 1: Adding a NOT NULL constraint**

If you have an existing table, employees, and you want to add a NOT NULL constraint to the email column:

ALTER TABLE employees

MODIFY email VARCHAR(255) NOT NULL;

**Example 2: Adding a PRIMARY KEY constraint**

If you forgot to set a column as the primary key when the table was created, you can do it afterward like this:

ALTER TABLE employees

ADD CONSTRAINT pk\_emp\_id PRIMARY KEY (emp\_id);

**Example 3: Adding a FOREIGN KEY constraint**

If you want to add a foreign key constraint to the orders table that references the emp\_id from the employees table:

ALTER TABLE orders

ADD CONSTRAINT fk\_emp\_id FOREIGN KEY (emp\_id) REFERENCES employees (emp\_id);

**Example 4: Adding a CHECK constraint**

You can add a CHECK constraint to ensure that an employee's age is at least 18:

ALTER TABLE employees

ADD CONSTRAINT chk\_age CHECK (age >= 18);

**2. Removing a Constraint from an Existing Table**

To remove a constraint, you use the ALTER TABLE statement with the DROP CONSTRAINT clause. You must specify the name of the constraint you want to remove.

**SQL Syntax to Remove a Constraint:**

ALTER TABLE table\_name

DROP CONSTRAINT constraint\_name;

**Example 1: Removing a PRIMARY KEY constraint**

If you no longer want a column to be the primary key, you can drop the primary key constraint:

ALTER TABLE employees

DROP CONSTRAINT pk\_emp\_id;

**Example 2: Removing a FOREIGN KEY constraint**

To remove a foreign key constraint (e.g., if the relationship between orders and employees changes), you can drop the constraint:

ALTER TABLE orders

DROP CONSTRAINT fk\_emp\_id;

**Example 3: Removing a CHECK constraint**

If you want to remove the CHECK constraint on the age column, you would do the following:

ALTER TABLE employees

DROP CONSTRAINT chk\_age;

**Example 4: Removing a UNIQUE constraint**

To remove a unique constraint (e.g., if you no longer need the email field to be unique), you would use:

ALTER TABLE employees

DROP CONSTRAINT unique\_email;

**5. Explain the consequences of attempting to insert, update, or delete data in a way that violates constraints. Provide an example of an error message that might occur when violating a constraint.**

Ans. When you attempt to insert, update, or delete data in a way that violates constraints in a database, the operation will **fail**, and the database will generate an **error message** to inform you about the violation. Constraints are designed to **enforce data integrity**, and any operation that contradicts the rules defined by these constraints is considered invalid.

1. **Insertion Failure (INSERT operation)**:

* When you try to insert data that violates a constraint, the database will **reject the operation** and **prevent the new row from being added** to the table. This ensures that the data stored in the database adheres to the rules set by the constraints.

1. **Update Failure (UPDATE operation)**:

* If you attempt to update existing data in a way that violates a constraint, the update will fail. For example, trying to set a column to NULL when it has a NOT NULL constraint, or trying to insert a duplicate value in a column with a UNIQUE constraint will trigger an error.

1. **Deletion Failure (DELETE operation)**:

* If you try to delete data that is referenced by a foreign key constraint (i.e., the data in one table is related to data in another table), the database will either prevent the deletion entirely, or it may require you to delete related records first, depending on how the foreign key constraint is configured (e.g., **ON DELETE CASCADE** or **ON DELETE RESTRICT**).

**Example Scenario for Violating Constraints**

**Scenario**: You have a table with a NOT NULL constraint on a column, and you try to insert a NULL value into that column.

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(30) NOT NULL,

email VARCHAR(20)

);

**Violation**: Attempting to insert a row with NULL in the emp\_name column (which cannot be NULL).

INSERT INTO employees (emp\_id, emp\_name, email)

VALUES (1, NULL, 'employee1@example.com');

**Error Message:**

**ERROR**: null value in column "emp\_name" violates not-null constraint

**DETAIL**: Failing row contains (1, null, employee1@example.com).

**Explanation**: Since emp\_name is defined with the NOT NULL constraint, the insertion fails because the column receives a NULL value.

**SQL Commands**

**1-Identify the primary keys and foreign keys in maven movies db. Discuss the differences.**

**1. Primary Keys (PK)**

A **Primary Key** is a unique identifier for each record in a table. Each record must have a unique primary key, which cannot contain NULL values. The primary key guarantees the uniqueness of rows within a table.

**2. Foreign Keys (FK)**

A **Foreign Key** is a column (or set of columns) in a table that links to the primary key of another table. The foreign key creates a relationship between the two tables and enforces referential integrity, ensuring that the data in one table corresponds to valid data in another.

In the context of a **Maven Movies** database, common primary keys and foreign keys would be:

**1. actor Table**

* **Primary Key**:
  + actor\_id — The unique identifier for each actor.
* **Foreign Keys**:
  + None.

**2. actor\_award Table**

* **Primary Key**:
  + actor\_award\_id — The unique identifier for each actor award entry.
* **Foreign Keys**:
  + actor\_id — This could reference actor.actor\_id (though it's not explicitly defined as a foreign key in your schema, it's implied by the column's name and context).

**3. address Table**

* **Primary Key**:
  + address\_id — The unique identifier for each address.
* **Foreign Keys**:
  + city\_id — This references city.city\_id via the constraint fk\_address\_city.

**4. advisor Table**

* **Primary Key**:
  + advisor\_id — The unique identifier for each advisor.
* **Foreign Keys**:
  + None.

**5. category Table**

* **Primary Key**:
  + category\_id — The unique identifier for each category.
* **Foreign Keys**:
  + None.
* **Differences Between Primary Keys and Foreign Keys**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Primary Key (PK)** | **Foreign Key (FK)** |
| **Definition** | Uniquely identifies each record in a table. | Establishes a link between two tables based on their keys. |
| **Uniqueness** | Must be unique for each row in the table. | Foreign keys do not require uniqueness in the referencing table. |
| **Null Values** | Cannot contain NULL values. | Can contain NULL values, representing an optional relationship. |
| **Reference Table** | Does not reference another table. | References the primary key of another table. |
| **Purpose** | Ensures each row is uniquely identifiable. | Maintains referential integrity by ensuring valid relationships between tables. |
| **Constraints** | Enforces the uniqueness of values in a column. | Enforces valid relationships between tables (i.e., the referenced key must exist). |
| **Examples** | customer\_id in the customer table, film\_id in the film table. | customer\_id in the rental table (FK referencing customer table), film\_id in the inventory table (FK referencing film table). |

**Normalisation & CTE**

**1. First Normal Form (1NF):**

**a. Identify a table in the Sakila database that violates 1NF. Explain how you would normalize it to achieve 1NF.**

**Understanding 1NF (First Normal Form):**

To be in **First Normal Form (1NF)**, a table must satisfy the following conditions:

1. **Atomicity**: Each column must contain atomic (indivisible) values. This means that each column should store only a single value for each row.
2. **Uniqueness**: Each row in the table must be unique. There should be no duplicate rows.
3. **No repeating groups**: A column should not contain multiple values or sets of values. If multiple values need to be stored, they should be separated into individual rows or new tables.

**Violating 1NF:**

In the **Sakila** database, an example of a table that violates 1NF is the **rental** table. This table stores rental records, but one column, inventory\_id, could potentially violate 1NF if it contains more than one inventory\_id per rental (i.e., if a single rental involved multiple inventory items or films).

However, looking at the structure of the rental table in the Sakila schema, it's not inherently violating 1NF (since each rental should correspond to one inventory item). Instead, if we examine relationships in more complex tables (such as film\_category or actor tables), violations of 1NF can occur if there are columns that store **multiple values** (like a list of categories or actors) in a single row.

For this example, let's focus on the film\_category table, which might have more obvious 1NF issues related to repeating groups.

**Example: film\_category table**

The **film\_category** table might look like this:

|  |  |
| --- | --- |
| **film\_id** | **category\_id** |
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 3 | 3 |

Here, there is no direct violation of 1NF because each row is atomic (each row has only one film\_id and one category\_id value). However, if the table allowed for multiple categories for a single film in the same row (i.e., storing a comma-separated list of categories), it would violate 1NF.

**Violation Example:**

Imagine a **film\_category** table like this:

|  |  |
| --- | --- |
| **film\_id** | **categories** |
| 1 | Action, Drama |
| 2 | Comedy, Drama |
| 3 | Drama |

Here, the categories column stores multiple values (Action, Drama, Comedy) in a single cell, which violates 1NF because the values are not atomic.

**Normalizing to Achieve 1NF:**

To **normalize** this table and bring it into 1NF, we would need to ensure that each column contains only **atomic values**. The solution would be to **split** the multiple values in the categories column into separate rows:

**Corrected film\_category table (1NF-compliant):**

|  |  |
| --- | --- |
| **film\_id** | **category\_id** |
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 2 | 3 |
| 3 | 2 |

In this normalized version:

* Each row has only one film\_id and one category\_id, ensuring atomic values.
* There are no repeating groups or multiple values in a single column.
* The relationship between films and categories is represented properly in a **many-to-many** relationship via a **junction table** (film\_category).

**2. Second Normal Form (2NF):**

**a. Choose a table in Sakila and describe how you would determine whether it is in 2NF. If it violates 2NF, explain the steps to normalize it.**

To determine if a table is in Second Normal Form (2NF), we must first ensure that the table is in First Normal Form (1NF) and then check for partial dependencies (i.e., dependencies where a non-prime attribute is dependent on part of a composite primary key).

If a table in the Sakila database is in 2NF, use the rental table as an example.

**Step 1:** **Review the rental Table Structure**

**Step 2:** **Check if the Table is in 1NF**

* 1NF requires that the table have atomic values (no repeating groups or arrays) and each record be unique.
* In the rental table:
  + Every field has a single value (atomic values).
  + There are no repeating groups or arrays.
  + The rental\_id is the primary key, ensuring uniqueness.

Thus, the rental table is in 1NF.

**Step 3: Check for 2NF**

To be in Second Normal Form (2NF), a table must:

1. Be in 1NF.
2. Have no partial dependencies (i.e., no non-prime attribute depends on part of the composite primary key).

**2.1: Check for Composite Primary Key**

* A composite primary key exists when the primary key is made up of two or more columns.
* In the rental table, the primary key is rental\_id, which is a single column and not a composite key.
  + Since there's no composite primary key, there are no partial dependencies to worry about in the first place.

**2.2: Identify Non-Prime Attributes**

* Non-prime attributes are those that are not part of the primary key. In the case of the rental table, the primary key is rental\_id, so the non-prime attributes are:
  + rental\_date
  + inventory\_id
  + customer\_id
  + return\_date
  + staff\_id
  + last\_update

**2.3: Check for Partial Dependencies**

* Since there is no composite primary key, there are no partial dependencies in the rental table.

Thus, the rental table satisfies 2NF because:

1. It is already in 1NF.
2. There are no partial dependencies, as there is no composite key.

**Step 4: If Violations of 2NF Were Found**

If the rental table did have a composite primary key, we would need to:

* Check if any non-prime attribute depends on only a part of the composite key.
  + If there were partial dependencies, we would need to decompose the table into separate tables to eliminate those dependencies.

For example, if the primary key were composed of inventory\_id and customer\_id, and if some non-prime attribute (like rental\_date) only depended on inventory\_id or customer\_id but not both, this would be a violation of 2NF. In such a case, we would decompose the rental table into two (or more) tables to ensure that all attributes depend on the full composite key.

For example, we could create two tables:

1. Rental: Store rental\_id, inventory\_id, customer\_id, staff\_id, last\_update.
2. Rental\_Details: Store rental\_id, rental\_date, return\_date.

This decomposition would ensure that all non-prime attributes depend on the full primary key, thus eliminating partial dependencies.

**3. Third Normal Form (3NF):**

**a. Identify a table in Sakila that violates 3NF. Describe the transitive dependencies present and outline the steps to normalize the table to 3NF.**

To identify a table in the **Sakila** database that violates **Third Normal Form (3NF)**, we need to look for **transitive dependencies**. These occur when a non-prime attribute depends on another non-prime attribute, which in turn depends on the primary key.

In **3NF**, a table is required to meet two conditions:

1. **It is in 2NF.**
2. **It has no transitive dependencies.** This means that a non-prime attribute (i.e., an attribute that is not part of the primary key) should not depend on another non-prime attribute.

**Identifying a Table that Violates 3NF**

One table in the **Sakila** database that might violate **3NF** is the **address** table.

**Step 1: Identify the Primary Key and Non-Prime Attributes**

* The primary key in the address table is address\_id.
* Non-prime attributes are:
  + address (varchar)
  + address2 (varchar)
  + district (varchar)
  + city\_id (foreign key)
  + postal\_code (varchar)
  + phone (varchar)
  + last\_update (timestamp)

**Step 2: Check for Transitive Dependencies**

In 3NF, there should be no **transitive dependencies**, which means that non-prime attributes must directly depend on the primary key and not on another non-prime attribute.

The potential **transitive dependency** arises here with the city\_id attribute:

* The city\_id references the city table, which contains the city name and potentially other details (like country\_id).
* This implies that the district, postal\_code, and possibly phone (or other address-related information) could be **dependent on the city** rather than just on address\_id.

For example:

* **city\_id → city\_name** (in the city table, which may be implicitly related to the address).
* **district, postal\_code, and phone** depend on city\_id, which indirectly depends on address\_id. This creates a transitive dependency, as the city details are indirectly dependent on the address\_id via the city\_id.

**Step 3: Transitive Dependency Example**

Consider the following transitive dependency:

* address\_id → city\_id → city\_name
* address\_id → district, but district may depend on city\_name or city\_id rather than address\_id.

Thus, district and postal\_code could be **transitively dependent** on the primary key (address\_id) via the city\_id. In this case, the address table would not be in **3NF** because we have transitive dependencies.

**Step 4: Steps to Normalize to 3NF**

To normalize the address table to **3NF**, we need to eliminate the transitive dependency. Specifically, we need to remove the indirect dependency between district, postal\_code, and city\_id.

**4.1: Move city\_id to a Separate Table**

The transitive dependencies are caused by the fact that attributes like district, postal\_code, etc., depend on city\_id. To resolve this, we can create a new table that handles the city-specific information.

**Step-by-Step Normalization Process:**

1. **Create a city table**:
   * We can move city\_id and city\_name into a new table. This separates out the city information, which is responsible for the transitive dependency.
2. **Modify the address table**:
   * Remove the city-related attributes (city\_id and possibly district, postal\_code) from the address table.
   * The address table should now only store attributes directly related to the address.

**4.2: Final Structure after Normalization**

1. **city table** stores:
   * city\_id
   * city\_name
   * country\_id (foreign key to the country table)
2. **address table** stores:
   * address\_id
   * address
   * address2
   * district
   * postal\_code
   * phone
   * city\_id (foreign key to city)

**4. Normalization Process:**

**a. Take a specific table in Sakila and guide through the process of normalizing it from the initial unnormalized form up to at least 2NF.**

For this example, let’s choose the **rental** table. We will assume it starts in an **unnormalized form (UNF)**, where data might be stored in a non-optimal way with redundant information.

**Step 1: Unnormalized Form (UNF)**

Assume that the rental table in its unnormalized form looks like this:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **rental\_id** | **rental\_date** | **customer\_id** | **customer\_name** | **address** | **city** | **rental\_duration** | **rental\_rate** |
| 1 | 2024-01-01 10:00:00 | 1 | John Doe | 123 Elm St. | New York | 5 | 3.99 |
| 2 | 2024-01-02 11:00:00 | 2 | Jane Smith | 456 Oak Ave | Los Angeles | 3 | 4.99 |
| 3 | 2024-01-03 12:00:00 | 1 | John Doe | 123 Elm St. | New York | 5 | 3.99 |

In this unnormalized form:

* The customer\_name, address, and city fields are repeated for each rental made by the same customer, resulting in **data redundancy**.
* This is a clear violation of **First Normal Form (1NF)**, which requires that each column contain atomic values (i.e., no repeating groups or multi-valued attributes).

**Step 2: First Normal Form (1NF)**

To bring the table into **1NF**, we need to:

1. Remove any repeating groups.
2. Ensure that all attributes contain atomic (indivisible) values.
3. Make sure each row is unique (no duplicates).

Here’s how we would transform the table into **1NF**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rental\_id** | **rental\_date** | **customer\_id** | **rental\_duration** | **rental\_rate** |
| 1 | 2024-01-01 10:00:00 | 1 | 5 | 3.99 |
| 2 | 2024-01-02 11:00:00 | 2 | 3 | 4.99 |
| 3 | 2024-01-03 12:00:00 | 1 | 5 | 3.99 |

Additionally, we will create separate tables for **customer** and **address** details to avoid redundancy.

We now have the following tables:

* **Rental table**: Stores rentals, but **customer-related information** is still redundant.
* **Customer table**: Stores customer details (no repetition across rentals).
* **Address table**: Stores customer addresses.

**1NF Table Breakdown**

**Rental Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rental\_id** | **rental\_date** | **customer\_id** | **rental\_duration** | **rental\_rate** |
| 1 | 2024-01-01 10:00:00 | 1 | 5 | 3.99 |
| 2 | 2024-01-02 11:00:00 | 2 | 3 | 4.99 |
| 3 | 2024-01-03 12:00:00 | 1 | 5 | 3.99 |

**Customer Table:**

|  |  |  |
| --- | --- | --- |
| **customer\_id** | **customer\_name** | **address\_id** |
| 1 | John Doe | 101 |
| 2 | Jane Smith | 102 |

**Address Table:**

|  |  |  |
| --- | --- | --- |
| **address\_id** | **address** | **city** |
| 101 | 123 Elm St. | New York |
| 102 | 456 Oak Ave | Los Angeles |

Now, the data is in **1NF** because each column contains atomic values and each row is unique. There is no redundant data for customer names or addresses repeated across multiple rentals.

**Step 3: Second Normal Form (2NF)**

To bring the data into **2NF**, we must ensure that:

1. The table is in **1NF**.
2. **No partial dependencies** exist. This means that non-prime attributes (i.e., attributes that are not part of the primary key) must depend on the **entire** primary key, not just a part of it.

In our **Rental table**, the **primary key** is rental\_id, and the **non-prime attributes** are rental\_date, rental\_duration, and rental\_rate. However, none of these attributes are dependent on any part of a composite primary key (since rental\_id is a single column key).

We need to check for the **dependency** between the non-prime attributes and ensure that they all depend on the full primary key.

In this case, there is no partial dependency, so the **Rental table** is already in **2NF**.

However, the **Customer table** might still contain a form of redundancy where the **address** is directly linked to the customer. Since the address\_id in the **Customer table** depends on the customer\_id, and address is independent of the rental\_id, it may be better to create a separate **Address table** and link it through address\_id in the **Customer** table, which we already did in the **1NF** step.

**Final Structure After Normalization to 2NF:**

**Rental Table (No change from 1NF):**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rental\_id** | **rental\_date** | **customer\_id** | **rental\_duration** | **rental\_rate** |
| 1 | 2024-01-01 10:00:00 | 1 | 5 | 3.99 |
| 2 | 2024-01-02 11:00:00 | 2 | 3 | 4.99 |
| 3 | 2024-01-03 12:00:00 | 1 | 5 | 3.99 |

**Customer Table (No change from 1NF):**

|  |  |  |
| --- | --- | --- |
| **customer\_id** | **customer\_name** | **address\_id** |
| 1 | John Doe | 101 |
| 2 | Jane Smith | 102 |

**Address Table (No change from 1NF):**

|  |  |  |
| --- | --- | --- |
| **address\_id** | **address** | **city** |
| 101 | 123 Elm St. | New York |
| 102 | 456 Oak Ave | Los Angeles |