

Database Management System

IT 261

Bachelor of Information Technology

Prepared by Dr. Khaled Eskaf

Lecture 9



Revision



MySQL Workbench

a product sales database. A product sales database typically consists of many tables, e.g., products, customers, suppliers, orders, payments, employees, among others. Let's call our database "**southwind**"

We shall begin with the first table called "products" with the following columns (having data types as indicated) and rows:

Database: southwind Table: products				
productID INT	productCode CHAR(3)	name VARCHAR(30)	quantity INT	price DECIMAL(10,2)
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49

You can create a new database using SQL command "CREATE DATABASE databaseName"; and delete a database using "DROP DATABASE databaseName". You could optionally apply condition "IF EXISTS" or "IF NOT EXISTS" to these commands. For example,

```
mysql> CREATE DATABASE southwind;  
Query OK, 1 row affected (0.03 sec)
```

```
mysql> DROP DATABASE southwind;  
Query OK, 0 rows affected (0.11 sec)
```

```
mysql> CREATE DATABASE IF NOT EXISTS southwind;  
Query OK, 1 row affected (0.01 sec)
```

```
mysql> DROP DATABASE IF EXISTS southwind;  
Query OK, 0 rows affected (0.00 sec)
```

IMPORTANT: Use SQL DROP (and DELETE) commands with extreme care, as the deleted entities are irrecoverable. THERE IS NO UNDO!!!

```
-- Remove the database "southwind", if it exists.  
-- Beware that DROP (and DELETE) actions are irreversible and not recoverable!  
mysql> DROP DATABASE IF EXISTS southwind;  
Query OK, 1 rows affected (0.31 sec)  
  
-- Create the database "southwind"  
mysql> CREATE DATABASE southwind;  
Query OK, 1 row affected (0.01 sec)
```

-- Create the table "products". Read "explanations" below for the column definitions

```
mysql> CREATE TABLE IF NOT EXISTS products (  
    productID  INT UNSIGNED NOT NULL AUTO_INCREMENT,  
    productCode CHAR(3)    NOT NULL DEFAULT "",  
    name       VARCHAR(30) NOT NULL DEFAULT "",  
    quantity   INT UNSIGNED NOT NULL DEFAULT 0,  
    price      DECIMAL(7,2) NOT NULL DEFAULT 99999.99,  
    PRIMARY KEY (productID)  
);
```

-- Show all the tables to confirm that the "products" table has been created

```
mysql> SHOW TABLES;
```

The types are:

- productID is INT UNSIGNED - non-negative integers.
- productCode is CHAR(3) - a fixed-length alphanumeric string of 3 characters.
- name is VARCHAR(30) - a variable-length string of up to 30 characters.
- We use fixed-length string for productCode, as we assume that the productCode contains exactly 3 characters. On the other hand, we use variable-length string for name, as its length varies - VARCHAR is more efficient than CHAR.
- quantity is also INT UNSIGNED (non-negative integers).
- price is DECIMAL(10,2) - a decimal number with 2 decimal places.
- DECIMAL is precise (represented as integer with a fix decimal point). On the other hand, FLOAT and DOUBLE (real numbers) are not precise and are approximated. DECIMAL type is recommended for currency.

-- Describe the fields (columns) of the "products" table

mysql> DESCRIBE products;

Field	Type	Null	Key	Default	Extra
productID	int(10) unsigned	NO	PRI	NULL	auto_increment
productCode	char(3)	NO			
name	varchar(30)	NO			
quantity	int(10) unsigned	NO		0	
price	decimal(7,2)	NO		99999.99	

Let's fill up our "products" table with rows. We set the productID of the first record to 1001, and use AUTO_INCREMENT for the rest of records by inserting a NULL, or with a missing column value. Take note that strings must be enclosed with a pair of single quotes (or double quotes).

-- Insert a row with all the column values

```
mysql> INSERT INTO products VALUES (1001, 'PEN', 'Pen Red', 5000, 1.23);
```

-- Insert multiple rows in one command

-- Inserting NULL to the auto_increment column results in max_value + 1

```
mysql> INSERT INTO products VALUES  
      (NULL, 'PEN', 'Pen Blue', 8000, 1.25),  
      (NULL, 'PEN', 'Pen Black', 2000, 1.25);
```

Query OK, 2 rows affected (0.03 sec)

Records: 2 Duplicates: 0 Warnings: 0

-- Insert value to selected columns

-- Missing value for the auto_increment column also results in max_value + 1

```
mysql> INSERT INTO products (productCode, name, quantity, price) VALUES
```

```
    ('PEC', 'Pencil 2B', 10000, 0.48),
```

```
    ('PEC', 'Pencil 2H', 8000, 0.49);
```

Query OK, 2 row affected (0.03 sec)

-- Missing columns get their default values

```
mysql> INSERT INTO products (productCode, name) VALUES ('PEC', 'Pencil HB');
```

Query OK, 1 row affected (0.04 sec)

-- 2nd column (productCode) is defined to be NOT NULL

```
mysql> INSERT INTO products values (NULL, NULL, NULL, NULL, NULL);
```

ERROR 1048 (23000): Column 'productCode' cannot be null

-- Query the table

```
mysql> SELECT * FROM products;
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49
1006	PEC	Pencil HB	0	9999999.99

6 rows in set (0.02 sec)

-- List all rows for the specified columns

```
mysql> SELECT name, price FROM products;
```

```
+-----+-----+
| name   | price |
+-----+-----+
| Pen Red   | 1.23 |
| Pen Blue  | 1.25 |
| Pen Black | 1.25 |
| Pencil 2B | 0.48 |
| Pencil 2H | 0.49 |
+-----+-----+
5 rows in set (0.00 sec)
```

-- List all rows of ALL the columns. The wildcard * denotes ALL columns

```
mysql> SELECT * FROM products;
```

```
+-----+-----+-----+-----+-----+
| productID | productCode | name       | quantity | price |
+-----+-----+-----+-----+-----+
| 1001      | PEN         | Pen Red   | 5000     | 1.23 |
| 1002      | PEN         | Pen Blue  | 8000     | 1.25 |
| 1003      | PEN         | Pen Black | 2000     | 1.25 |
| 1004      | PEC         | Pencil 2B | 10000    | 0.48 |
| 1005      | PEC         | Pencil 2H | 8000     | 0.49 |
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

For numbers (INT, DECIMAL, FLOAT), you could use comparison operators: '=' (equal to), '<>' or '!=' (not equal to), '>' (greater than), '<' (less than), '>=' (greater than or equal to), '<=' (less than or equal to), to compare two numbers. For example, price > 1.0, quantity <= 500.

```
mysql> SELECT name, price FROM products WHERE price < 1.0;
```

```
+-----+-----+
| name      | price |
+-----+-----+
| Pencil 2B | 0.48  |
| Pencil 2H | 0.49  |
+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql> SELECT name, quantity FROM products WHERE quantity <= 2000;
```

```
+-----+-----+
| name      | quantity |
+-----+-----+
| Pen Black | 2000     |
+-----+-----+
1 row in set (0.00 sec)
```

CAUTION: Do not compare FLOATs (real numbers) for equality ('=' or '<>'), as they are not precise. On the other hand, DECIMAL are precise.

For strings, you could also use '=', '<>', '>', '<', '>=', '<=' to compare two strings (e.g., `productCode = 'PEC'`). The ordering of string depends on the so-called collation chosen. For example,

```
SELECT name, price FROM products WHERE productCode = 'PEN';  
-- String values are quoted
```

```
+-----+-----+  
| name      | price |  
+-----+-----+  
| Pen Red   | 1.23  |  
| Pen Blue  | 1.25  |  
| Pen Black | 1.25  |  
+-----+-----+  
3 rows in set (0.00 sec)
```

For strings, in addition to full matching using operators like '=' and '<>', we can perform pattern matching using operator LIKE (or NOT LIKE) with wildcard characters. The wildcard '_' matches any single character; '%' matches any number of characters (including zero). For example,

- 'abc%' matches strings beginning with 'abc';
- '%xyz' matches strings ending with 'xyz';
- '%aaa%' matches strings containing 'aaa';
- '____' matches strings containing exactly three characters; and
- 'a_b%' matches strings beginning with 'a', followed by any single character, followed by 'b', followed by zero or more characters.

-- "name" begins with 'PENCIL'

```
mysql> SELECT name, price FROM products WHERE name LIKE 'PENCIL%';
```

name	price
Pencil 2B	0.48
Pencil 2H	0.49

-- "name" begins with 'P', followed by any two characters,

-- followed by space, followed by zero or more characters

```
mysql> SELECT name, price FROM products WHERE name LIKE 'P__ %';
```

name	price
Pen Red	1.23
Pen Blue	1.25
Pen Black	1.25

You can combine multiple conditions with boolean operators AND, OR. You can also invert a condition using operator NOT. For examples,

```
SELECT * FROM products WHERE quantity >= 5000 AND name LIKE 'Pen %';
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25

```
SELECT * FROM products WHERE quantity >= 5000 AND price < 1.24 AND name LIKE 'Pen %';
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.23

To check if the value is within a range, you could use BETWEEN ... AND ... operator. Again, this is easier and clearer than the equivalent AND-OR expression.

```
SELECT * FROM products  
WHERE (price BETWEEN 1.0 AND 2.0) AND (quantity BETWEEN 1000 AND 2000);
```

productID	productCode	name	quantity	price
1003	PEN	Pen Black	2000	1.25

NULL is a special value, which represent "no value", "missing value" or "unknown value". You can checking if a column contains NULL by IS NULL or IS NOT NULL. For example,

```
SELECT * FROM products WHERE productCode IS NULL;
```

Using comparison operator (such as = or <>) to check for NULL is a mistake - a very common mistake. For example,

```
SELECT * FROM products WHERE productCode = NULL;  
-- This is a common mistake. NULL cannot be compared.
```

-- Order the results by price in descending order

```
SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC;
```

productID	productCode	name	quantity	price
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1001	PEN	Pen Red	5000	1.23

-- Order by price in descending order, followed by quantity in ascending (default) order

```
mysql> SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC, quantity;
```

productID	productCode	name	quantity	price
1003	PEN	Pen Black	2000	1.25
1002	PEN	Pen Blue	8000	1.25
1001	PEN	Pen Red	5000	1.23

```
SELECT productID AS ID, productCode AS Code,  
       name AS Description, price AS `Unit Price` -- Define aliases to be used as display names  
FROM products  
ORDER BY ID; -- Use alias ID as reference
```

ID	Code	Description	Unit Price
1001	PEN	Pen Red	1.23
1002	PEN	Pen Blue	1.25
1003	PEN	Pen Black	1.25
1004	PEC	Pencil 2B	0.48
1005	PEC	Pencil 2H	0.49

```
SELECT CONCAT(productCode, ' - ', name) AS `Product Description`, price FROM products;
```

+-----+-----+	
Product Description	price
+-----+-----+	
PEN - Pen Red	1.23
PEN - Pen Blue	1.25
PEN - Pen Black	1.25
PEC - Pencil 2B	0.48
PEC - Pencil 2H	0.49
+-----+-----+	

DISTINCT

-- Without DISTINCT

SELECT price FROM products;

price
1.23
1.25
1.25
0.48
0.49

-- With DISTINCT on price

SELECT DISTINCT price AS `Distinct Price` FROM products;

Distinct Price
1.23
1.25
0.48
0.49

-- DISTINCT combination of price and name

SELECT DISTINCT price, name FROM products;

price	name
1.23	Pen Red
1.25	Pen Blue
1.25	Pen Black
0.48	Pencil 2B
0.49	Pencil 2H

-- Increase the price by 10% for all products

UPDATE products SET price = price * 1.1;

SELECT * FROM products;

productID	productCode	name	quantity	price
1001	PEN	Pen Red	5000	1.35
1002	PEN	Pen Blue	8000	1.38
1003	PEN	Pen Black	2000	1.38
1004	PEC	Pencil 2B	10000	0.53
1005	PEC	Pencil 2H	8000	0.54

-- Modify selected rows

mysql> UPDATE products SET quantity = quantity - 100 WHERE name = 'Pen Red';

mysql> SELECT * FROM products WHERE name = 'Pen Red';

productID	productCode	name	quantity	price
1001	PEN	Pen Red	4900	1.35

```
DELETE FROM products WHERE name LIKE 'Pencil%';
```

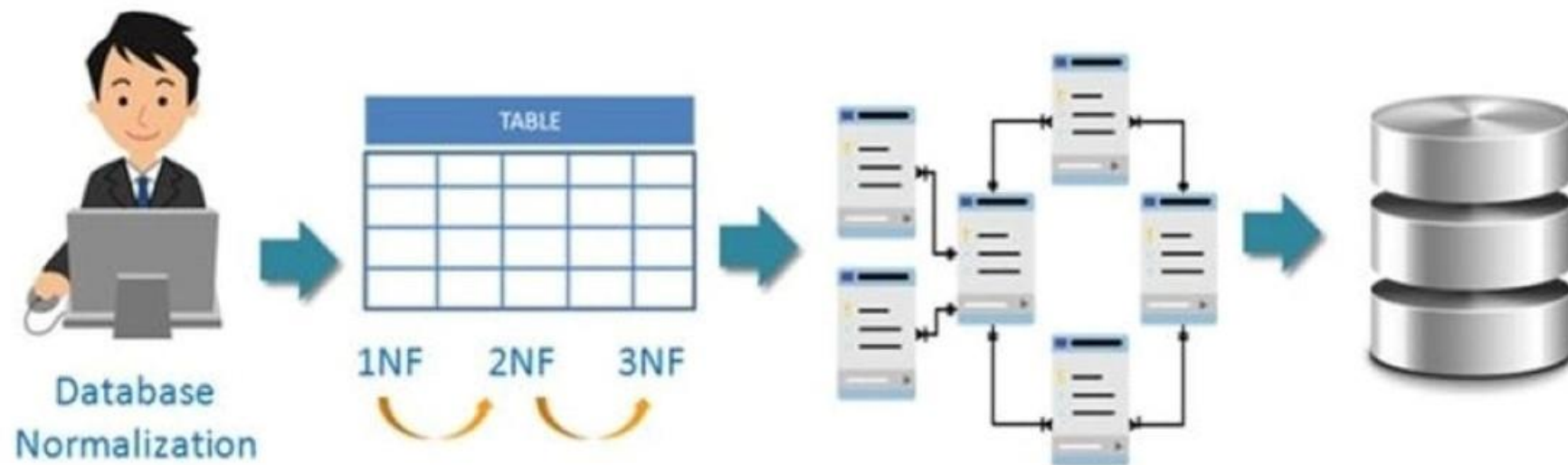
```
SELECT * FROM products;
```

productID	productCode	name	quantity	price
1001	PEN	Pen Red	4950	1.23
1002	PEN	Pen Blue	8000	1.38
1003	PEN	Pen Black	2000	1.38

Database normalization

CHAPTER 9

Database normalization — Normal forms - 1nf 2nf 3nf



In relational database design, we not only want to create a structure that stores all of the data, but we also want to do it in a way that minimize potential errors when we work with the data. The default language for accessing data from a relational database is SQL. In particular, SQL can be used to manipulate data in the following ways: insert new data, delete unwanted data, and update existing data. Similarly, in an un-normalized design, there are 3 problems that can occur when we work with the data:

INSERT ANOMALY: This refers to the situation when it is impossible to insert certain types of data into the database.

DELETE ANOMALY: The deletion of data leads to unintended loss of additional data, data that we had wished to preserve.

UPDATE ANOMALY: This refers to the situation where updating the value of a column leads to database inconsistencies (i.e., different rows on the table have different values).

To address the 3 problems above, we go through the process of normalization. When we go through the normalization process, we increase the number of tables in the database, while decreasing the amount of data stored in each table. There are several different levels of database normalization:

- 1st Normal Form (1NF)
- 2nd Normal Form (2NF)
- 3rd Normal Form (3NF)

Normalization is the branch of relational theory that provides design insights. It is the process of determining how much redundancy exists in a table.

The goals of normalization are to:

- Be able to characterize the **level of redundancy in a relational schema**
- Provide mechanisms for transforming schemas **in order to remove redundancy**

Normal Forms

All the tables in any database can be in one of the normal forms we will discuss next.

Ideally we only want minimal redundancy for PK to FK. Everything else should be derived from other tables.

There are three basic normal forms, which are:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)
- **Boyce-Codd normal form (BCNF)**

BCNF is rarely used.

Advantages of Normal Form

- **Reduced data redundancy:** Normalization helps to eliminate duplicate data in tables, reducing the amount of storage space needed and improving database efficiency.
- **Improved data consistency:** Normalization ensures that data is stored in a consistent and organized manner, reducing the risk of data inconsistencies and errors.
- **Simplified database design:** Normalization provides guidelines for organizing tables and data relationships, making it easier to design and maintain a database.
- **Improved query performance:** Normalized tables are typically easier to search and retrieve data from, resulting in faster query performance.
- **Easier database maintenance:** Normalization reduces the complexity of a database by breaking it down into smaller, more manageable tables, making it easier to add, modify, and delete data.

Important Points Regarding Normal Forms in DBMS:

- **First Normal Form (1NF):** This is the most basic level of normalization. In 1NF, each table cell should contain only a single value, and each column should have a unique name. The first normal form helps to eliminate duplicate data and simplify queries.
- **Second Normal Form (2NF):** 2NF eliminates redundant data by requiring that each non-key attribute be dependent on the primary key. This means that each column should be directly related to the primary key, and not to other columns.
- **Third Normal Form (3NF):** 3NF builds on 2NF by requiring that all non-key attributes are independent of each other. This means that each column should be directly related to the primary key, and not to any other columns in the same table.
- **Boyce-Codd Normal Form (BCNF):** BCNF is a stricter form of 3NF that ensures that each determinant in a table is a candidate key. In other words, BCNF ensures that each non-key attribute is dependent only on the candidate key.

	1NF	2NF	3NF
Decomposition of Relation	R	R₁₁ R₁₂	R₂₁ R₂₂ R₂₃
Conditions	Eliminate Repeating Groups	Eliminate Partial Functional Dependency	Eliminate Transitive Dependency

First Normal Form (1NF)

The first normal form (1NF), is the most basic form of data normalization. The **core outcome** of this rule ensures that there are **no repeating entries in a group**. This means:

- Every cell should only have **one** single value.
- Every record should be unique.

For a table to be in the first normal form, it must meet the following criteria:

- A single cell must not hold more than one value (atomic)
- There must be a primary key for identification
- No duplicated rows or columns
- Each column must have only one value for each row in the table

Suppose you have a table that stores information about customers and their orders. This table is not in 1NF because the “Orders” column contains multiple values separated by commas, violating the rule of atomic values:

CustomerID	CustomerName	Orders
1	John	Order1, Order2
2	Alice	Order3

To bring it into 1NF, you can split the “Orders” column into separate rows:

CustomerID	CustomerName	Order
1	John	Order1
1	John	Order2
2	Alice	Order3

How do we bring an unnormalized table into first normal form? Consider the following example:

TABLE_PRODUCT

Product ID	Color	Price
1	red, green	15.99
2	yellow	23.99
3	green	17.50
4	yellow, blue	9.99
5	red	29.99

This table is not in first normal form because the [Color] column can contain multiple values. For example, the first row includes values "red" and "green."

To bring this table to first normal form, we split the table into two tables and now we have the resulting tables:

TABLE_PRODUCT

Product ID	Color	Price
1	red, green	15.99
2	yellow	23.99
3	green	17.50
4	yellow, blue	9.99
5	red	29.99

**TABLE_PRODUCT_PRICE**

Product ID	Price
1	15.99
2	23.99
3	17.50
4	9.99
5	29.99

TABLE_PRODUCT_COLOR

Product ID	Color
1	red
1	green
2	yellow
3	green
4	yellow
4	blue
5	red

	1NF	2NF	3NF
Decomposition of Relation	R	R₁₁ R₁₂	R₂₁ R₂₂ R₂₃
Conditions	Eliminate Repeating Groups	Eliminate Partial Functional Dependency	Eliminate Transitive Dependency

Second Normal Form (2NF)

2NF is the second normal form that builds on the rules of the first normal form. Again, the goal is to ensure that there are no repeating entries in a dataset. Entries with this data normalization rule applied must:

- Adhere to all the 1NF requirements.
- Have one primary key applied.

The application of one primary key essentially means that a **separate table needs to be created** for subsets of data that can be placed in multiple rows. The data in each table can then be linked with foreign key labels (numbers in this case).

Second Normal Form (2NF)

A table is in 2NF if it meets the following criteria:

- It is in 1NF.
- All non-key attributes (columns) are fully functionally dependent on the primary key.

Example 1:

Consider a table that stores information about courses and the instructors who teach them:

CourseID	Instructor	CourseName
101	Smith	Biology
102	Johnson	Chemistry
103	Smith	Physics

In this table, the combination of “CourseID” and “Instructor” forms a composite primary key. However, the “CourseName” column depends only on “CourseID,” not on “Instructor.” To bring this table into 2NF, you can split it into two separate tables:

CourseID	CourseName
101	Biology
102	Chemistry
103	Physics

Table 1 (Courses)

CourseID	Instructor
101	Smith
102	Johnson
103	Smith

Table 2 (Instructors)

Consider the following example:

Example Not In Second Normal Form

This table has a composite primary key [Customer ID, Store ID]. The non-key attribute is [Purchase Location]. In this case, [Purchase Location] only depends on [Store ID], which is only part of the primary key. Therefore, this table does not satisfy second normal form.

TABLE_PURCHASE_DETAIL

Customer ID	Store ID	Purchase Location
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles
3	2	New York
4	3	San Francisco

To bring this table to second normal form, we break the table into two tables, and now we have the following:

TABLE_PURCHASE_DETAIL

Customer ID	Store ID	Purchase Location
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles
3	2	New York
4	3	San Francisco

**TABLE_PURCHASE**

Customer ID	Store ID
1	1
1	3
2	1
3	2
4	3

TABLE_STORE

Store ID	Purchase Location
1	Los Angeles
2	New York
3	San Francisco

What we have done is to remove the partial functional dependency that we initially had. Now, in the table [TABLE_STORE], the column [Purchase Location] is fully dependent on the primary key of that table, which is [Store ID].

	1NF	2NF	3NF
Decomposition of Relation	R	R₁₁ R₁₂	R₂₁ R₂₂ R₂₃
Conditions	Eliminate Repeating Groups	Eliminate Partial Functional Dependency	Eliminate Transitive Dependency

Third Normal Form (3NF)

A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency. 3NF is used to reduce the data duplication. It is also used to achieve the data integrity. If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds at least one of the following conditions for every non-trivial function dependency $X \rightarrow Y$.

- 1) X is a super key.
- 2) Y is a prime attribute, i.e., each element of Y is part of some candidate key.

Third Normal Form (3NF)

A table is in 3NF if it meets the following criteria:

- It is in 2NF.
- There is no transitive dependency between non-key attributes.

By transitive functional dependency, we mean we have the following relationships in the table: A is functionally dependent on B, and B is functionally dependent on C. In this case, C is transitively dependent on A via B.

Example:

Continuing with the “Courses” and “Instructors” tables from 2NF, suppose you add an “InstructorEmail” column to the “Instructors” table. If the email is unique to each instructor and not related to a course, you have a transitive dependency issue. To achieve 3NF, you would create a separate “InstructorInfo” table:

CourseID	CourseName
101	Biology
102	Chemistry
103	Physics

Table 1 (Courses): (Unchanged from 2NF)

CourseID	Instructor
101	Smith
102	Johnson
103	Smith

Table 2 (Instructors): (Without email)

Instructor	InstructorEmail
Smith	<u>smith@email.com</u>
Johnson	<u>johnson@email.com</u>

Table 3 (InstructorInfo): (Instructor and Email)

Example:

In the following table Roll_No is the candidate key which determines ExamType and ExamType determines the MaxMarks. So the following table holds transition property. That's why it is not in 3NF.

RollNo	Name	ExamType	MaxMarks
1	Ali	Viva	20
2	Ahmad	Theory	100
3	Akram	Practical	50
4	Nauman	Viva	20
5	Haider	Practical	50

Exam table

Transitive property can be removed by dividing the table into its parts.

RollNo	Name	ExamType	MaxMarks
1	Ali	Viva	20
2	Ahmad	Theory	100
3	Akram	Practical	50
4	Nauman	Viva	20
5	Haider	Practical	50

Exam table

Transitive Property Removed

JOIN				
RollNo	Name	ExamType	ExamType	MaxMarks
1	Ali	Viva	Viva	20
2	Ahmad	Theory	Theory	100
3	Akram	Practical	Practical	50
4	Nauman	Viva		
5	Haider	Practical		

Table: Exam

Table: Marks

3rd Normal Form Example 3:

Consider the following example:

TABLE_BOOK_DETAIL

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

In the table, [Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type]. Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.

To bring this table to third normal form, we split the table into two as follows:

TABLE_BOOK_DETAIL

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

**TABLE_BOOK**

Book ID	Genre ID	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

Now all non-key attributes are fully functional dependent only on the primary key. In [TABLE_BOOK], both [Genre ID] and [Price] are only dependent on [Book ID]. In [TABLE_GENRE], [Genre Type] is only dependent on [Genre ID].

Transitively Dependent

Dependent

Dependent

<u>UnitCode</u>	UnitName	CourseCode	CourseName
UG45783	Advance Database	COMP2009	Computing
UG45832	Network Systems	COMP2009	Computing
UG45734	Multi-User Operating Systems	COMP2009	Computing
UG45951	Project	BUS22009	Business & Computing

Transitively Dependent

Dependent

Dependent

<u>UnitCode</u>	UnitName	CourseCode	CourseName
UG45783	Advance Database	COMP2009	Computing
UG45832	Network Systems	COMP2009	Computing
UG45734	Multi-User Operating Systems	COMP2009	Computing
UG45951	Project	BUS22009	Business & Computing

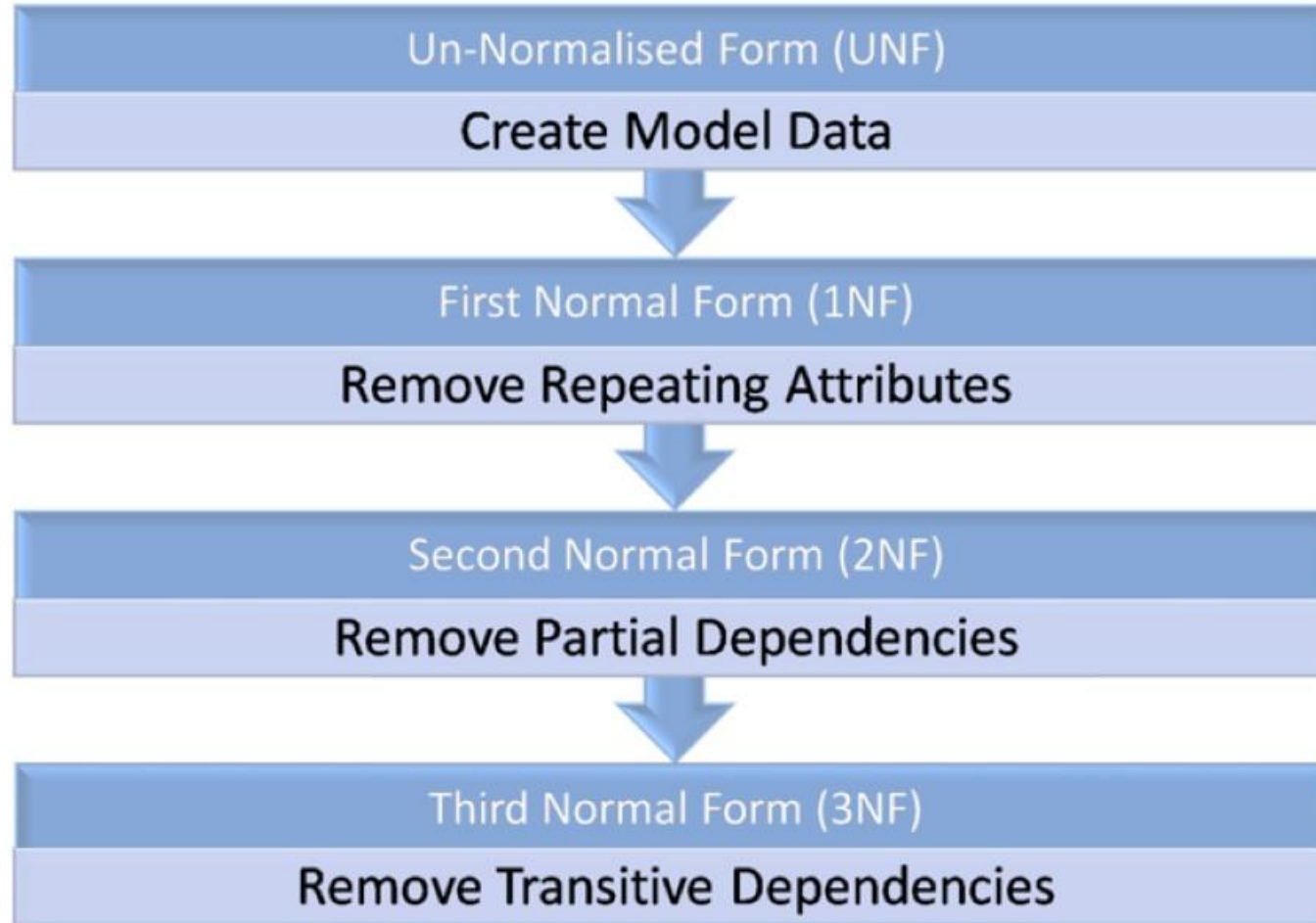
Foreign Key

Tables in Third Normal Form

Primary Key

<u>UnitCode</u>	UnitName	CourseCode*
UG45783	Advance Database	COMP2009
UG45832	Network Systems	COMP2009
UG45734	Multi-User Operating Systems	COMP2009
UG45951	Project	BUS22009

<u>CourseCode</u>	CourseName
COMP2009	Computing
BUS22009	Business & Computing



1.Un-Normalised Form (UNF) – Data Modelling

2.First Normal Form (1NF) – Repeating Groups

3.Second Normal Form (2NF) – Partial Dependencies

4.Third Normal Form (3NF) – Transitive Dependencies



Thank You