Database Normalization

Definition

Normalization is a process in **Database Management Systems (DBMS)** used to organize data in a way that **reduces redundancy** and **eliminates data anomalies**.

The primary goal is to **break down large tables into smaller ones**, ensuring **data integrity** and **efficient storage**.

Normal Forms

1 NF (First Normal Form)

2 NF (Second Normal Form)

3 NF (Third Normal Form)

4 NF (Fourth Normal Form)

5 NF (Fifth Normal Form)

First Normal Form (1 NF)

- Table shouldn’t implicitly use row order to convey information
- Mixed type values under single column is not allowed
- Each row in a table must be uniquely identifiable (Primary Key)
- Every cell in a row must contain indivisible (Atomic) Values. In other words, repeating groups are not allowed

1 NF - Rule 1 Violation

- **Table shouldn’t implicitly use row order to convey information**
- Consider that you want to store the names of cricketers in a table
- You must do it an arbitrary way without caring about row order
- Like this

cricketer
kohli
rohit
dhoni
sachin

1 NF - Rule 1 Violation

- Rows in your table must not indicate any order that you're thinking of
- For example trying to store the names in the descending order of **centuries scored** is **violating the 1 NF**

cricketer
kohli
rohit
dhoni
sachin

cricketer
sachin
kohli
rohit
dhoni

1 NF - Rule 1 Violation - Issues

- Trying to define an implicit row order **leads to inconsistency of data**
- Imagine you need to insert a new name **ponting** into the table
- You wouldn't be able to insert him in the right position as you are only **imagining the order**.

cricketer
kohli
rohit
dhoni
sachin

cricketer
sachin
kohli
rohit
dhoni

1 NF - Solution to Rule 1 Violation

- Instead of **imagining the order,** add an **actual column** called **centuries** to the table
- No implicit order needed, since we **store** the centuries in that column

cricketer
kohli
rohit
dhoni
sachin

cricketer	centuries
kohli	80
rohit	45
dhoni	18
sachin	100

1 NF – Advantages of solution to Rule 1 Violation

- Now you can use **centuries column to order cricketers**
- Not only that, **insertion of a new record doesn't need any order now**
- Just using **ORDER BY** on centuries column, you can sort the data

cricketer
kohli
rohit
dhoni
sachin

cricketer	centuries
kohli	80
rohit	45
dhoni	18
sachin	100
ponting	**71**

1 NF – Rule 2 Violation

- **No column should contain values of mixed data type**
- Luckily, **databases won't let us do it.**
- As shown below

cricketer	centuries
kohli	Somewhere between 75 and 85
rohit	45
dhoni	Eighteen
sachin	100

1 NF – Rule 3 Violation

- **Every row must be uniquely identifiable**
- The table on the left has duplicates entries of **kohli** with 80 and 41 centuries
- It leads to data inconsistency as one can't figure out which one is correct

cricketer	centuries
kohli	80
rohit	45
dhoni	18
sachin	100
kohli	41

1 NF – Solution to Rule 3 Violation

- Implement a **Primary Key**, a unique value that identifies each row individually
- Database design doesn't allow multiple entries of **Primary Key**

Primary Key

c_id	cricketer	centuries
1	kohli	80
2	rohit	45
3	dhoni	18
4	sachin	100
1	kohli	41

Database doesn't allow duplicate values

1 NF – Rule 4 Violation

- No **multi-valued attributes** are allowed
- **A cell must contain atomic (indivisible) values**
- Consider the **user** table on right
- If you want to maintain the mobile number of user and a user can have **more than one mobile number**
- Storing them in the way shown on right **violates the first normal form**

user table

u_id	u_name
1	alice
2	bob

user table

u_id	u_name	mobile
1	alice	9054xxxxxx, 7086xxxxxx
2	bob	8999xxxxxx, 6301xxxxxx, 8142xxxxxx

1 NF – Issues with Rule 4 Violation

- Hard to **get each mobile number separately**, as it needs complex string split operation
- **Adding another mobile number** requires string concatenation
- No straight way to **count mobile numbers available**
- **Deletion** of a mobile number is quite hard

user table

u_id	u_name	mobile
1	alice	9054xxxxxx, 7086xxxxxx
2	bob	8999xxxxxx, 6301xxxxxx, 8142xxxxxx

1 NF – Solution to Rule 4 violation

- Just store each mobile number in **a different column**

user table

u_id	u_name	mobile
1	alice	9054xxxxxx, 7086xxxxxx
2	bob	8999xxxxxx, 6301xxxxxx, 8142xxxxxx

user table

u_id	u_name	mobile1	mobile2	mobile3
1	alice	9054xxxxxx	7086xxxxxx	
2	bob	8999xxxxxx	6301xxxxxx	8142xxxxxx

1 NF – **Actual** Solution to Rule 4 violation

- Decompose the user table
- Create a new table to maintain **phone numbers of users separately**

u_id	u_name	mobile
1	alice	9054xxxxxx, 7086xxxxxx
2	bob	8999xxxxxx, 6301xxxxxx, 8142xxxxxx

user table

user table

u_id	u_name
1	alice
2	bob

user_mobiles

u_id	mobile
1	9054xxxxxx
1	7086xxxxxx
2	8999xxxxxx
2	6301xxxxxx
2	8142xxxxxx

1 NF – **Advantages** of the solution to Rule 4 violation

- No complex string operations required to
  - Insert
  - Update
  - Delete mobile numbers
- Easy to fetch (Just write a join) and count
- Flexible to expand
- For instance, if we want to maintain **status (Working / Not working)** of a particular phone number, we just need to **add a new column in user_mobiles table**

user table

u_id	u_name
1	alice
2	bob

user_mobiles

u_id	mobile
1	9054xxxxxx
1	7086xxxxxx
2	8999xxxxxx
2	6301xxxxxx
2	8142xxxxxx

1 NF – **Advantages** of the solution to Rule 4 violation

- No complex string operations required to
  - Insert
  - Update
  - Delete mobile numbers
- Easy to fetch (Just write a join) and count
- Flexible to expand
- For instance, if we want to maintain **status (Working / Not working)** of a particular phone number, we just need to **add a new column in user_mobiles table**

user table

u_id	u_name
1	alice
2	bob

ALTER command to add new column

user_mobiles

u_id	mobile	Staus
1	9054xxxxxx	Working
1	7086xxxxxx	Not Working
2	8999xxxxxx	Not Working
2	6301xxxxxx	Working
2	8142xxxxxx	Working

1 NF – Rule 4 Violation Another Example

- No **multi-valued attributes** are allowed
- **A cell must contain atomic (indivisible) values**
- Assume that you are maintaining the data of player in a arcade game
- Where **player can collect different items as the game progresses**
- Such as **copper coins, diamonds, swords, guns and bullets etc.**
- Think of games like Temple Run, Subway Surfers or even Super Mario

1 NF – Rule 4 Violation Another Example

- Let's say you decided to maintain the **player** table as shown below

player table

p_id	p_name	items
1	alice	5 copper coins,3 swords,4 guns
2	bob	6 copper coins,9 swords,2 guns
3	charlie	2 copper coins,6 swords,5 guns
4	Diana	4 copper coins,2 swords,7 guns

1 NF – Rule 4 Violation Another Example

- The below way of storing data violates **Rule 4** of 1 NF
- As the table contains repeating groups under items column

player table

p_id	p_name	items
1	alice	5 copper coins,3 swords,4 guns
2	bob	6 copper coins,9 swords,2 guns
3	charlie	2 copper coins,6 swords,5 guns
4	diana	4 copper coins,2 swords,7 guns

1 NF – Rule 4 Violation Issues

- Hard to extract each item from items, as it requires complex string split operation
- Extremely difficult to update a particular item when collected
- Takes a lot of work to decrement item count when a particular item is used by player
- No easy way to make analysis by comparing players
- For instance, it's hard to say which player is having more guns

player table

p_id	p_name	items
1	alice	5 copper coins,3 swords,4 guns
2	bob	6 copper coins,9 swords,2 guns
3	charlie	2 copper coins,6 swords,5 guns
4	diana	4 copper coins,2 swords,7 guns

1 NF – Solution for Rule 4 Violation

- Decompose the table
- Create a new table named **player_inventory**
- Store items data of each player in the **player_inventory**

player table

p_id	p_name
1	alice
2	bob
3	charlie
4	diana

p_id	p_name	items
1	alice	5 copper coins,3 swords,4 guns
2	bob	6 copper coins,9 swords,2 guns
3	charlie	2 copper coins,6 swords,5 guns
4	diana	4 copper coins,2 swords,7 guns

player table

player_inventory table

p_id	item	count
1	copper coin	5
1	sword	4
1	gun	3
2	copper coin	6
2	sword	9
2	gun	2
3	copper coin	2
3	sword	6
3	gun	5

First Normal Form (1 NF)

- Table shouldn’t implicitly use row order to convey information
- Mixed type values under single column is not allowed
- Each row in a table must be uniquely identifiable (Primary Key)
- Every cell in a row must contain indivisible (Atomic) Values. In other words, repeating groups are not allowed

Second Normal Form (2 NF)

Table should already be in **1 NF**

All **non-key attributes** are fully functionally dependent on **entire primary key**

In other words, **no partial dependencies are allowed.**

What is functional dependency?

A **functional dependency** is a relationship between two sets of attributes in a relational database table that describes how one set of attributes (the determinant) determines another set of attributes (the dependent)

If you know the value of the **determinant**, you can uniquely **determine the value of the dependent attribute(s)** for any row in the table

A formal definition for functional dependency

Formally, a functional dependency is written as X → Y, where

- X is the determinant (a set of one or more attributes).
- Y is the dependent (a set of one or more attributes).
- In simpler words, **for every value of X there is exactly one corresponding value of Y**.

Examples of Functional Dependencies – Q1

- Consider the **student** table on right
- If we take {gender} as dependent (**Y**), can we tell which of the other attributes (**X**) uniquely identifies (without ambiguity) gender?
- So, what is **{X}** for {gender}

student

s_id	name	age	gender
1	alice	18	F
2	bob	19	M
3	charile	21	M
4	diana	20	F

Examples of Functional Dependencies – Q1

- Is it **{name}**
- Can we say
  - **{name}** is determinant?
  - **{gender}** is dependent?
- So, our functional dependency becomes
  - {name} → {gender}?

student

s_id	name	age	gender
1	alice	18	F
2	bob	19	M
3	charile	21	M
4	diana	20	F

Examples of Functional Dependencies – Q1

- The answer is **NO**
- If we add another alice to the table who turns out to be a Male, {name} isn't going to determine {gender} uniquely anymore without ambiguity
- {name} → {gender}
- {alice} → {F}
- {alice} → {M}

ambiguity

student

s_id	name	age	gender
1	alice	18	F
2	bob	19	M
3	charile	21	M
4	diana	20	F
5	alice	24	M

Examples of Functional Dependencies – Q1

- **{age}** is also **NOT** the determinant for **{gender}**, since its completely possible have different gender values in same age group

student

s_id	name	age	gender
1	alice	18	F
2	bob	19	M
3	charile	21	M
4	diana	20	F
5	eric	18	M

Examples of Functional Dependencies – Q1

- The column that uniquely identifies the **{gender}** is **{s_id}**
- Because there won't be any ambiguity in terms of primary key
- {1} – {F}
- {2} – {M}
- {5} – {M}

student

s_id	name	age	gender
1	alice	18	F
2	bob	19	M
3	charile	21	M
4	diana	20	F
5	eric	18	M

2 NF – What are non-key attributes?

- Consider the **student** table on right
- In the given table **s_id** is the primary key
- **Functional Dependencies** in the table
  - {s_id} → {name}
  - {s_id} → {age}
  - {s_id} → {gender}

Primary Key

Non-key attributes

s_id	name	age	gender
1	alice	18	F
2	bob	19	M
3	charile	21	M
4	diana	20	F

student

2 NF – **Violation Scenario**

- Consider the **student** table on right
- And also the **courses** table on right
- Let's suppose we created a junction table **enrolments** that maps students to courses as follows

Composite Primary Key

s_id	c_id	e_date	price	grade	duration	gender
1	101	15-09-2023	6000	A	180	Female
3	103	21-02-2022	8000	C	200	Male
4	102	27-05-2020	7000	B	220	Female
1	103	06-04-2021	8000	B	200	Female
2	101	31-12-2024	6000	D	180	Male
2	102	15-03-2025	7000	A	220	Male
1	102	29-02-2024	7000	C	220	Female

Primary Key

students

s_id	s_name	email
1	alice	alice@example.com
2	bob	bob@example.com
3	charile	charlie@example.com
4	diana	diana@example.com

Primary Key

courses

c_id	c_name
101	C Programming
102	Java
103	Python