Database Designs

Section 5: Database Design Theory

5.1 Practical Tips on Database Design

A good database design is when a database can withstand the test of time. If the data structure does not change that much even when the data grows and expands into new scenarios/areas of information then this can signify a good good database design. This is a hard topic and so much can be said about this topic alone. Below are a few important principles of designing a good schema for a database.

Designing the tables of a database is called designing the schema of a database.The list below is not a comprehensive list but will help when setting up a database to withstand the test of time.

- Identify the problem you are trying to model/capture from the real worlds
- Have a good idea of the types of queries (i.e. information to drive business decisions) that you want to run on the database
- Break up the information into logical units where each logical unit is one table in your database
- Each logical component will probably connect with other logical components. Capture the relationships either via a constraint or other tables devoted to holding relationship specific data

Tables typically hold information about an entity or a relationship. Think of whether you need a way to uniquely identify each row in a table. If the answer to this question is yes (in most circumstances this will be true) you need a Primary Key.

- A Primary Key can never have a NULL value
- A Primary Key will be used to represent that row and other tables can reference that row by specifying the Primary Key value
- Do not use data that can change as a Primary Key value
- If there are no such values in the fields you want to store then you can generate a value which will be used as a unique identifier i.e. DBMS have auto-increment to generate unique identifiers for you
- Numeric values tend to be more efficient in representation and lookup and should be preferred for Primary Key values
- Having an identifier whose job is to be a key value is standard practice
- Remember, most DBMS will automatically create an index on the Primary Key for faster lookup and retrieval

Do columns have other constraints which they need to satisfy?
- Are NULL values allowed for any column? If the answer is no then the column should be specified as NOT NULL
- Should columns have a default value? If the answer is yes then the column should be setup with the DEFAULT keyword followed by what the default value for that column should be

- Ensure the column datatype are setup correctly e.g. don't use a string for storing date information (even though it is allowed) as this will limit the features applicable to date and time values for example comparing date difference between two date values
- Think about the number of characters/numbers a column will hold e.g. if a data is a single character don't use a CHAR(30) to store the string. Database occupy space and space that are not used up are a waste of resource and storage resource is expensive

5.2 Further Tips on Database Design (Table Relationships)

How many tables should we use to store information? To answer this question we should always think of the relationship between the different pieces of information being represented.

If the information is a one-to-one compulsory information e.g. students and their email address there is no need to store the information is separate tables. Every student has just one email address and therefore there is no need for the email address to be stored in a separate table from the rest of the students information.

When the relationship between the various pieces of information are one-to-one and the information is not optional (i.e. it is always specified) then we can put that information into the same table. Remember this as a rule of thumb. You may want to separate it for some reason but typically if the information has a one-to-one relationship there is no reason to separate the data.

StudentID	Name	Email
1	John Doe	j.doe@email.com
2	Martha Jones	m.jones@email.com

If the information is a one-to-one but one of the information is optional e.g. students and siblings information you may decide to hold these information in separate tables. A student may or may not have siblings and if the main students table included sibling information then there would be many empty cells. In this case the students table will hold only the mandatory information while the sibling information should be held in a separate table.

When the relationship between the various pieces of information are one-to-one but some information are optional (i.e. may or may not exist) then it may be best to separate the information keeping all the mandatory information in one table and the optional information in another. The main table will hold a Primary Key as an identifier for each row. The second optional data table would reference the main table's Primary Key value as a Foreign Key value, thereby linking both the table information together.

Optional data are represented by the lack of rows in the second table and not by empty cell value in a row. This is a much cleaner setup.

StudentID	Name	Email
1	John Doe	j.doe@email.com
2	Martha Jones	m.jones@email.com

StudentID	Sibling Name
1	Joanne Doe
1	Jonathon Doe
2	Katey Jones

If the information is a one-to-many or a many-to-one relationship e.g. employees and managers information then at a minimum we would need two tables to represent this information. One table would be used to store all information relating to employees and another to hold the relationship between an employee and his/her manager.

Notice in the example table below, the second table uses only the unique identifiers to specify the relationship. Once again, the Primary Key is used on the main table to represent the entire row details of the employee. The second table merely makes reference to these keys only.

This is not the only way to represent this information and there are different ways of representing this data but typically speaking a one-to-many or many-to-one relationship is best represented using two tables at a minimum and what the two tables contain can be designed differently.

EmployeeID	Name	Email
1	John Doe	j.doe@email.com
2	Martha Jones	m.jones@email.com
3	Katey Jones	k.jones@email.com

EmployeeID	Manager_EmployeeID
1	2
3	2

Finally if the information is a many-to-many relationship e.g. stores, products and revenue information then we would need 3 or more tables to represent the information. The revenue data is specified for every product sold in every store and therefore links the stores and products data in a many-to-many relationship.

In such a case we would need at a minimum 3 tables, the first table representing the first entity's information, the second table representing the second entity's informations and the third table linking the two entities tables in a many-to-many relationship.

StoreID	StoreLocation
1	Birmingham
2	Coventry

ProductID	ProductName
1	Bread
2	Milk
3	Nutella

StoreID	ProductID	Date	Revenue
1	1	30 November 2020	453.22
1	3	30 November 2020	225.55
2	2	30 November 2020	1860.98

It is best to keep the entities information in separate tables. If we were to add more details about the product e.g. product details, if we did not have a separate table for stores and products, we would end up adding more and more information to the single table making the table fatter even if the information has nothing to do with the store itself. Separate tables allows us to expand the information logically while still maintain tall and thin structures. The linking table itself will have a tall and fat structure.

The linking table would use Foreign Keys to link the the Primary Key values of the other entity tables which combines all three tables together in a many-to-many relationship. This provides a clean structure for specifying a many-to-many relationship.

Understanding the relationships of the information you are trying to represent is vital in setting up the tables and its column in the best way possible to stand the test of time.

5.3 Normal Forms in Database Design

Database Normalisation i.e. Normal Forms are basically rules that tell you how a good relational database should be designed. Database theory provides us the standard rules to test if a table is well designed. These rules are specified in the form of Normal Forms (and there are many available to us). Normal Forms are very inaccessible, but if you can understand them they are very good rules that are well thought through and can help improve the design of the database table. Below are some examples for Normal Forms translated to normal understandable english.

<u>1NF - “First Normal Form”:</u>
“A relation is in first Normal Form if the domain of each attribute contains only atomic (indivisible) values and the value of each attribute contains only a single value from that domain.”

This means that a table should not have any column with concatenated/compound values. If this is the case, the data should be split into multiple columns which are atomic/indivisible values. This will allow greater flexibility with the way you specify attributes of a particular row.

For example the table below would be best split into separate columns to make comparisons, ordering, grouping, etc. much easier to perform. The second table below follows the First Normal Form rule.

StudentID	Sudent_FirstName_LastName	Subject_GraduatingClass
1	John Doe	CS_2020
2	Martha Jones	ECE_2019

StudentID	FirstName	LastName	Subject	GraduatingClass
1	John	Doe	CS	2020
2	Martha	Jones	ECE	2019

<u>2NF - "Second Normal Form":</u>
"A table is in 2NF if it is in 1NF and no non-prime attribute is dependant on any proper subset of any candidate key of the table. (A non-prime attribute of a table is is an attribute that is not a part of any candidate key of the table)."

This means a table should have every non-key column fully defined by or dependant on a Primary Key.

Below is an example of a table that violates the 2NF rule. The Primary Key for the table is the combination of StoreID, ProductID and Date columns which provides the unique information of the revenue of a particular product sold by a particular store on a particular date. There are two additional non-key columns.

The Revenue column is clearly defined i.e. "identified" by the Primary Key(s). The Revenue column as a value makes absolute sense for that particular logical unit that the Primary Keys specifies (i.e. for a particular store, product and date the revenue was X amount).

The CategoryManager column is not clearly identified by the Primary Key(s) i.e. while the CategoryManager can be for a product or a product in a store; however, the date for the CategoryManager does not make any sense here.

StoreID	ProductID	Date	Revenue	CategoryManager
Birmingham	Bananas	30 November 2020	2245.60	John
Coventry	Bananas	30 November 2020	4455.00	Martha

In this scenario the CategoryManager violates the 2NF rule and so we would extract the CategoryManager from this table and create a separate products or products_store table where it would make sense for the data to be present in the new table.

<u>3NF - “Third Normal Form”:</u>
“A table is in 3NF if
(1) The Entity is in 2NF, and
(2) All the attributes in a table are determined only by the candidate keys of that table and not by any non-prime attributes.”

This means a table should have all non-key columns independent of each other and should not be related in any way.

In the below example table the StoreID is a unique identifier for the store and therefore is the Primary Key for this table. All other columns are non-key columns. If we were to combine all of the non-key columns data together we would notice they are closely related to each other and not completely independent of one another. In fact using only the storeAddress we could figure out the StoreCity and the StoreCountry. Therefore, the city and country follow from the store location and are not independent columns and should not be part of the below table. Instead the StoreCity and StoreCountry should move into a separate tabled Keyed by the Store Location. You should split up the tables so that the data follows 3NF as seen in the second table example below

StoreID	StoreAddress	StoreCity	StoreCountry
1	84 Boroughbridge Road	Birmingham	England
2	57 Whatlington Road	Coventry	England

StoreID	StoreAddress
1	84 Boroughbridge Road
2	57 Whatlington Road

StoreID	StoreCity	StoreCountry
1	Birmingham	England
2	Coventry	England

(Note: this is one possible way of splitting the store location data into multiple tables).

Theoretically, there are higher forms of database normalisations like Boyce-Codd Normal Form (a.k.a 3.5NF), 4NF, 5NF. However, satisfying unto 3NF is more than enough for most databases and 3NF rule is the more widely used normalisation form in production databases.

We will explore the other higher forms of database normalisation rules but it is not necessary to achieve these higher rules in every databases schemas. However, having some understanding will be beneficial.

<u>BCNF 3.5NF - “Boyce-Codd Normal Form”:</u>
“A table is considered Boyce-Codd Normal Form, if (1) it’s already in the 3NF and (2) for every functional dependency between A and B (A → B), A should be a super key.”

To understand this rule we need to first need to understand some really important terminologies. What is a Functional Dependancy and a Super Key?

A Functional Dependency (FD) is when an attribute or column of a table uniquely identifies another attribute(s) or column(s) of the same table. For example, an EmployeeID column uniquely identifies the other columns like Employee Name, Employee Salary, etc. in the Employee table.

A Super Key is a key or group of multiple keys that could uniquely identify a single row in a table. In general terms, we know such keys as Composite Keys.

The BCNF is known as 3.5NF because it is an updated/advanced version of 3NF with more strict rules.

The below table scenario will help demonstrate and understand the problem with 3NF and how BCNF comes in to help normalise the database from redundant data.

EmployeeID	FirstName	EmpCity	DeptName	DeptHead
1	John	New York	Accounts	Racheal
1	John	New York	Technology	Daniel
2	Martha	Berlin	Accounts	Samantha
3	Joanne	London	HR	Elizabeth
3	Joanne	London	Infrastructure	Harvey

In this example scenario, employees with EmployeeID of 1 and 3 work in two different departments. Each department has a department head. There can be multiple department heads for each department. For example the Accounts department, Racheal and Samantha are the two heads of departments.

The EmployeeID and DeptName are super keys, which implies that DeptName is a prime attribute. Based on these two columns, we can identify every single row uniquely.

Also, the DeptName depends on DeptHead column, which implies that DeptHead is a non-prime attribute. This criterion disqualifies the above table from being part of BCNF.

To solve this we will break the table into two different tables as shown below. Each table has a super key:

EmployeeID	FirstName	EmpCity	DeptNum
1	John	New York	D1
1	John	New York	D2

2	Martha	Berlin	D1
3	Joanne	London	D3
3	Joanne	London	D4

DepartmentID	DeptName	DeptHead
D1	Accounts	Racheal
D1	Accounts	Samantha
D2	Technology	Daniel
D3	HR	Elizabeth
D4	Infrastructure	Harvey

Alternatively, using three tables as shown below:

EmployeeID	FirstName	EmpCity
1	John	New York
1	John	New York
2	Martha	Berlin
3	Joanne	London
3	Joanne	London

DepartmentID	DeptName	DeptHead
D1	Accounts	Racheal
D1	Accounts	Samantha
D2	Technology	Daniel
D3	HR	Elizabeth
D4	Infrastructure	Harvey

EmployeeID	DeptNum
1	D1
1	D2
2	D1
3	D3
3	D4

Below is another simplified example table demonstrating the BCNF normalisation rule.

StudentName	CourseName	CourseTeacher
John	Accounts	Racheal
Martha	Accounts	Samantha
Joanne	Computer Science	Tom
Katey	Computer Science	Tom
Daniel	Accounts	Samantha

In the above table the StudentName and CourseName columns combined together form the Primary Key. This is because both these columns together can be used to uniquely find all other table columns, in this case there is only the CourseTeacher column. Furthermore, since a CourseTeacher teaches one subject, we can use the CourseTeacher column to find the CourseName as well.

The above table is in 1NF because all the columns are atomic. It also satisfies 2NF because there are no partial dependencies between the columns. Finally, the table also satisfies 3NF because there is no transitive dependencies. We can represent the dependancy as below:

{ StudentName, CourseName } => { CourseTeacher }

However, there is one more functional dependancy which we can see below:

{ CourseTeacher } => { CourseName }

While CourseName is a prime attribute, the CourseTeacher is a non-prime attribute and this is not allowed by BCNF i.e. the functional dependancy between the CourseName (B) on CourseTeacher (A) does not satisfy BCNF because CourseTeacher is not a Super Key.

To solve this problem, we will decompose the above table into two different tables as shown below. Each table now has a Super Key:

StudentName	CourseName
John	Accounts
Martha	Accounts
Joanne	Computer Science
Katey	Computer Science
Daniel	Accounts

CourseName	CourseTeacher
Accounts	Racheal

CourseName	CourseTeacher
Accounts	Samantha
Computer Science	Tom

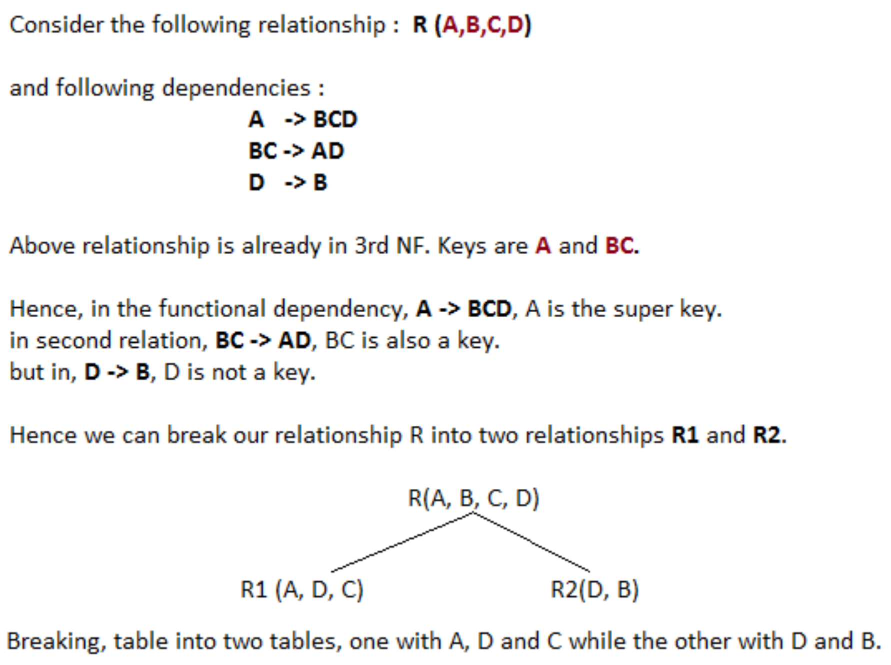
Notice that in the second table we only need to record the CourseName and CourseTeacher combination once and remove any redundant duplicate combinations. Notice now that both tables have a Super Key where A determines B (A being the left column on the table).

Important Note: we could alternatively represent the table like the below table:

StudentName	CourseTeacherID
John	1
Martha	2
Joanne	3
Katey	3
Daniel	2

CourseTeacherID	CourseTeacher	CourseName
1	Racheal	Accounts
2	Samantha	Accounts
3	Tom	Computer Science

Below is an image explaining BCNF in terms of relations:



Consider the following relationship : **R (A,B,C,D)**

and following dependencies :

A -> BCD
BC -> AD
D -> B

Above relationship is already in 3rd NF. Keys are **A** and **BC.**

Hence, in the functional dependency, **A -> BCD**, A is the super key.
in second relation, **BC -> AD**, BC is also a key.
but in, **D -> B**, D is not a key.

Hence we can break our relationship R into two relationships **R1** and **R2.**

$$\begin{tikzcd} & R(A, B, C, D) \arrow[dl, no head] \arrow[dr, no head] & \\ R1\ (A, D, C) & & R2(D, B) \end{tikzcd}$$

Breaking, table into two tables, one with A, D and C while the other with D and B.

This demonstrate the higher level rule of BCNF compared the 3NF and why it is referred to as 3.5NF. BCNF which was developed by Raymond F. Boyce and Edgar F. Codd builds on top of Edgar F. Codd first three normalisation rules.

<u>4NF - “Forth Normal Form”:</u>
“A table is in 4NF if
(1) The Entity is in BCNF, and
(2) The table does not have any multi-valued dependancy.”

This means a table should not have any multi-valued dependancy. What is a multi-valued dependancy? A table is said to have multi-valued dependency, if the following conditions are true:

1. For a dependency A → B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency
2. A table should have at-least 3 columns for it to have a multi-valued dependency
3. And, for a relation of `R(A,B,C)`, if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.

If all these conditions are true for any relation (table), it is said to have multi-valued dependency. Below is an example table demonstrating the 4NF normalisation rule:

StudentID	Course	Hobbies
1	Science	Football
1	Mathematics	Cricket
2	English Language	Football
2	English Literature	Cricket

In the above example, we have a student with a StudentID of 1 who has opted for two courses Science and Mathematics and has two hobbies of Football and Cricket. This will give rise to two more records for StudentID 1 in a cartesian join as seen in the below:

1	Science	Football
1	Mathematics	Cricket
1	Science	Cricket
1	Mathematics	Football

Furthermore, there is no relationship between the columns Course and Hobbies i.e. they are both independent of each other. Thus, there is multi-value dependency, which leads to un-necessary repetition of data and other anomalies as well.

To satisfy the 4NF rule we can decompose the table into two separate tables as seen in the example below:

StudentID	Course
1	Science
1	Mathematics
2	English Language
2	English Literature

StudentID	Hobbies
1	Football
1	Cricket
2	Football
2	Cricket

A table can also have functional dependency along with multi-valued dependency. In that case, the functionally dependent columns are moved in a separate table and the multi-valued dependent columns are moved to separate tables. This can be seen in the below example:

StudentID	Address	Course	Hobbies
1	84 Boroughbridge Road	Science	Football
1	84 Boroughbridge Road	Mathematics	Cricket
2	57 Whatlington Road	English Language	Football
2	57 Whatlington Road	English Literature	Cricket

There is a functional dependancy between the StudentID → Address columns and a multi-valued dependancy between the StudentId → Course and StudentID → Hobbies columns. This table would therefore be decomposed into three tables as shown below:

StudentID	Course
1	Science
1	Mathematics
2	English Language
2	English Literature

StudentID	Hobbies
1	Football
1	Cricket
2	Football

2	Cricket

StudentID	Hobbies
1	84 Boroughbridge Road
2	57 Whatlington Road

Multi-valued dependancy occurs due to bad database design. If you design your database carefully, you can easily avoid these issues.

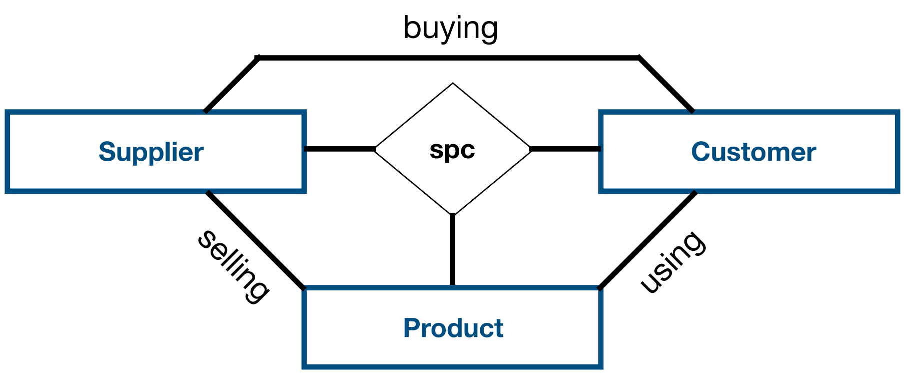
<u>5NF - “Fifth Normal Form”:</u>
“A table is in 5NF if
(1) The Entity is in 4NF, and
(2) The table does not have any join dependancy.”

5NF is sometime referred to as PJNF (Project Join Normal Form). This means a table should not have any Join Dependancy. What is a Join Dependancy? A table is said to have Join Dependancy dependency, whereby the table can be divided into smaller relations such that if we combine the smaller relations we would get back the original table.

Below is an example to help demonstrate 5NF:

Supplier	Product	Customer
ACME	72X SW	Ford
ACME	GEAR L	GM
ROBUSTO	E SWITCH	Ford
ROBUSTO	ODB II	Mercedes
ALWAT	72X SW	GM
ALWAT	ODB II	Mercedes
ALWAT	GEAR L	Mercedes

The Entity Relationship for the above table can be shown as below:

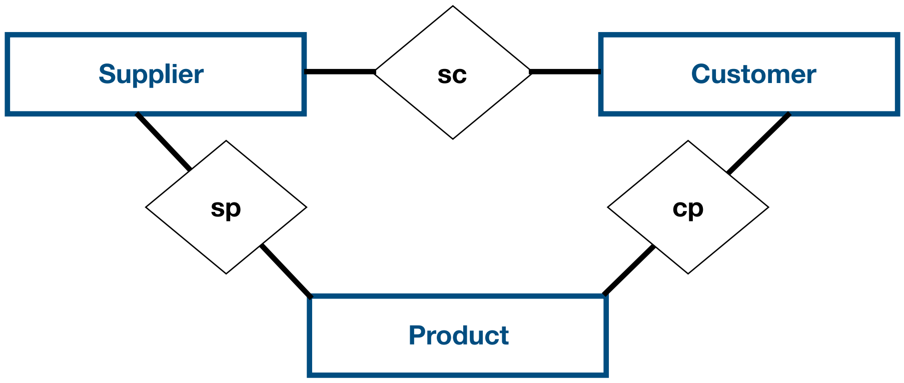


This is clearly a ternary relationship as it involves more than two entities.

The binary 1-to-1 relation between each entities are:

- A given supplier can supply a given product to either one or more customers (i.e. one product supplied to many customers)
- A given customer buying from a supplier can have one or more products (out of the products that a given supplier can supply)
- A given product used by a customer can be supplied by either one or more suppliers (i.e. multiple suppliers supply the same product)

If we broke the table down into the exclusive binary relationships between attributes the ER diagram would look the below:



The decomposed table representation would look like the below:

SUPP_CUST_TABLE	
Supplier	**Customer**
ACME	Ford
ACME	GM
ROBUSTO	Ford
ROBUSTO	Mercedes
ALWAT	GM
ALWAT	Mercedes
ALWAT	Mercedes

SUPP_PROD_TABLE	
Supplier	**Product**
ACME	72X SW
ACME	GEAR L
ROBUSTO	E SWITCH
ROBUSTO	ODB II

ALWAT	72X SW
ALWAT	ODB II
ALWAT	GEAR L

CUST_PROD_TABLE	
Customer	**Product**
Ford	72X SW
GM	GEAR L
Ford	E SWITCH
Mercedes	ODB II
GM	72X SW
Mercedes	ODB II
Mercedes	GEAR L

Using the first record we can break the three table down to three parts of:
{ ACME sells 72X SW }
{ Ford uses 72X SW }
{ ACME supplies to Ford }

However, the issue with the above is that we cannot imply ACME sells 72X SW to Ford even though this statement is true as per the original single table. Therefore, decomposing into the three binary tables results in a loss of information.

5NF Join Dependancy rule states that if a table is decomposed into smaller tables and that leads to either some loss of information or some additional information being created, then we should not decompose the table because that will lead to incorrect information.

So the takeaway from 5NF is that if we focused purely on the data (and ignored the business logic) we have to see if decomposing the table into smaller relations which when combined may lead to extra row of data or less row of data i.e. creation or information loss then it is best to stick to the original table as per 5NF.
However, if decomposing down the table does not lead to loss of information and using the decomposed relations and can still verify all the facts about the data then we should break down the table into smaller relations.

To conclude a table can be considered in Fifth Normal Form only if it satisfies the conditions for Fourth Normal Form and can be broken down into multiple tables without loss of any data.

5NF database normalisation rule is generally not implemented in real life database design, but you should know what it it.

END