DATABASE MANAGEMENT SYSTEMS
SCHEMA REFINEMENT - INTRODUCTION

Schema refinement is the process of improving a database schema to ensure it meets the desired criteria of correctness, efficiency, and maintainability. This involves restructuring the schema to eliminate redundancy, ensure data integrity, and optimize performance.

It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.

The Schema Refinement refers to refine the schema by using some techniques.

Importance of Schema Refinement:

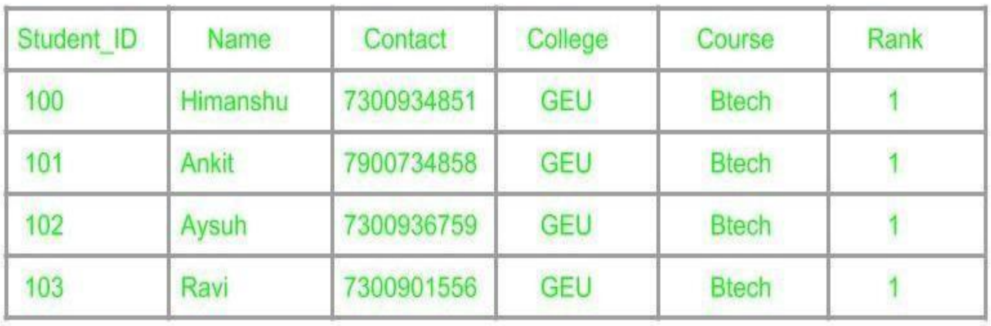
1. **Eliminate Redundancy:**
   - Reduces duplicated data, which saves storage and avoids anomalies.
2. **Ensure Data Integrity:**
   - Maintains accuracy and consistency of data through well-defined relationships and constraints.
3. **Optimize Performance:**
   - Enhances query performance by organizing data in an efficient manner.
4. **Simplify Maintenance:**
   - Eases database management tasks like updates, insertions, and deletions by ensuring a clean, logical structure.

Process of Schema Refinement:

1. **Analysis**:
   Assess the current schema for inefficiencies and anomalies.
2. **Normalization**:
   Apply normalization principles to structure the schema into appropriate normal forms.
3. **Decomposition**:
   Break down larger tables into smaller, related tables to eliminate redundancy.
4. **Validation**:
   Ensure that decompositions are lossless and preserve dependencies.
5. **Optimization:**
   Fine-tune the schema for performance by indexing, partitioning, and other techniques.

Redundancy:

Redundancy means having multiple copies of same data in the database. This problem arises when a database is not normalized. Suppose a table of student details attributes are: student Id, student name, college name, college rank, course opted.

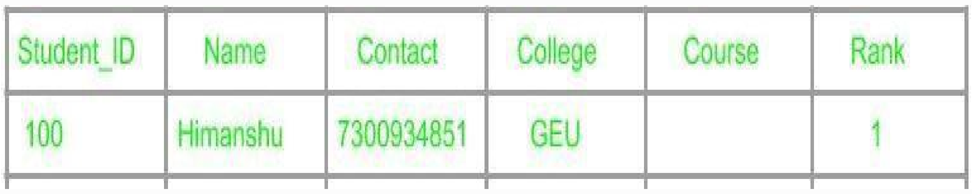


Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU	Btech	1
101	Ankit	7900734858	GEU	Btech	1
102	Aysuh	7300936759	GEU	Btech	1
103	Ravi	7300901556	GEU	Btech	1

<u>The Problem of redundancy in Database</u>

As it can be observed that values of attribute college name, college rank, course is being repeated which can lead to problems. Problems caused due to redundancy are: Insertion anomaly, Deletion anomaly, and Updation anomaly.

1. **Insertion Anomaly –**
   If a student detail has to be inserted whose course is not being decided yet then insertion will not be possible till the time course is decided for student.

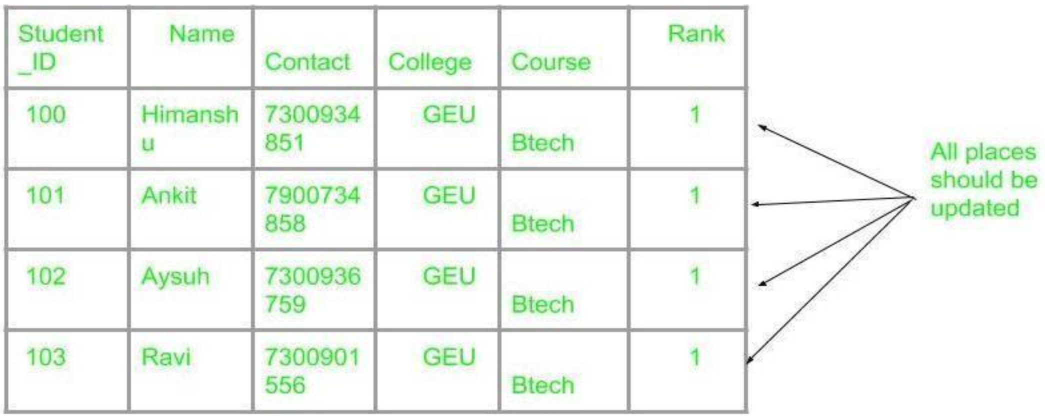


Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU		1

This problem happens when the insertion of a data record is not possible without adding some additional unrelated data to the record.

2. **Deletion Anomaly –**
   If the details of students in this table is deleted then the details of college will also get deleted which should not occur by common sense.
   This anomaly happens when deletion of a data record results in losing some unrelated information that was stored as part of the record that was deleted from a table.

3. **Updation Anomaly –**
   Suppose if the rank of the college changes then changes will have to be all over the database which will be time-consuming and computationally costly.



Student_ID	Name	Contact	College	Course	Rank
100	Himanshu	7300934851	GEU	Btech	1
101	Ankit	7900734858	GEU	Btech	1
102	Aysuh	7300936759	GEU	Btech	1
103	Ravi	7300901556	GEU	Btech	1

All places should be updated

If updation do not occur at all places then database will be in inconsistent state.

Decomposition:

Decomposition is the process of breaking down in parts or elements.
It replaces a relation with a collection of smaller relations.
It breaks the table into multiple tables in a database.
It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations.
If there is no proper decomposition of the relation, then it may lead to problems like loss of information.

Types of Decomposition:

1. Lossless Decomposition
2. Lossy Decomposition

Lossless Decomposition:

Ensures that the original table can be reconstructed from the decomposed tables without any loss of information.
"The decomposition of relation R into R1 and R2 is **lossless** when the join of R1 and R2 yield the same relation as in R."
A relational table is decomposed (or factored) into two or more smaller tables, in such a way that the designer can capture the precise content of the original tablPre by joining the decomposead parts.
This is called **lossless-join (or non-additive join) decomposition.**

Let us consider the following example:

<EmpInfo>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**	**Dept_ID**	**Dept_Name**
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables:

<EmpDetails>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	**Emp_ID**	**Dept_Name**
Dpt1	E001	Operations
Dpt2	E002	HR
Dpt3	E003	Finance

Now, Natural Join is applied on the above two tables −

The result will be −

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**	**Dept_ID**	**Dept_Name**
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Therefore, the above relation had lossless decomposition i.e. no loss of information.

Lossy Decomposition:
Some information might be lost when decomposing and rejoining the tables.
"The decomposition of relation R into R1 and R2 is lossy when the join of R1 and R2 does not yield the same relation as in R."
One of the disadvantages of decomposition into two or more relational schemes (or tables) is that some information is lost during retrieval of original relation or table.

Let us see an example −

<EmpInfo>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**	**Dept_ID**	**Dept_Name**
E001	Jacob	29	Alabama	Dpt1	Operations
E002	Henry	32	Alabama	Dpt2	HR
E003	Tom	22	Texas	Dpt3	Finance

Decompose the above table into two tables −

<EmpDetails>

Emp_ID	**Emp_Name**	**Emp_Age**	**Emp_Location**
E001	Jacob	29	Alabama
E002	Henry	32	Alabama
E003	Tom	22	Texas

<DeptDetails>

Dept_ID	**Dept_Name**
Dpt1	Operations
Dpt2	HR
Dpt3	Finance

Now, you won't be able to join the above tables, since **Emp_ID** isn't part of the **DeptDetails** relation.

Therefore, the above relation has lossy decomposition.

Problems related to Decomposition:

- Lossy Decomposition:
  - Can lead to loss of information and data integrity issues.
- Increased Complexity:
  - Managing multiple tables and their relationships can become complex.
- Performance Overhead:
  - More joins required in queries can impact performance.
- Dependency Preservation:
  - Ensuring all functional dependencies are preserved in decomposed tables can be challenging.

Properties of Decomposition:

1. Lossless Decomposition

2. Dependency Preservation

3. Lack of Data Redundancy

DATABASE MANAGEMENT SYSTEMS
FUNCTIONAL DEPENDENCIES

Functional Dependency (FD) is a constraint that determines the relation of one attribute to another attribute in a Database Management System (DBMS). Functional Dependency helps to maintain the quality of data in the database. It plays a vital role to find the difference between good and bad database design.

A functional dependency is denoted by an arrow "→". The functional dependency of X on Y is represented by X → Y. Let's understand Functional Dependency in DBMS with example

Example:

Employee number	**Employee Name**	**Salary**	**City**
1	Dana	50000	San Francisco
2	Francis	38000	London
3	Andrew	25000	Tokyo

In this example, if we know the value of Employee number, we can obtain Employee Name, city, salary, etc. By this, we can say that the city, Employee Name, and salary are functionally depended on Employee number.

Key terms

Here, are some key terms for Functional Dependency in Database:

Key Terms	**Description**
Axiom	Axioms is a set of inference rules used to infer all the functional dependencies on a relational database.
Decomposition	It is a rule that suggests if you have a table that appears to contain two entities which are determined by the same primary key then you should consider breaking them up into two different tables.
Dependent	It is displayed on the right side of the functional dependency diagram.

Determinant	It is displayed on the left side of the functional dependency Diagram.
Union	It suggests that if two tables are separate, and the PK is the same, you should consider putting them. together

Rules of Functional Dependencies

Below are the Three most important rules for Functional Dependency in Database:

- Reflexive rule –. If X is a set of attributes and Y is_subset_of X, then X holds a value of Y.
- Augmentation rule: When x -> y holds, and c is attribute set, then ac -> bc also holds. That is adding attributes which do not change the basic dependencies.
- Transitivity rule: This rule is very much similar to the transitive rule in algebra if x -> y holds and y -> z holds, then x -> z also holds. X -> y is called as functionally that determines y.

Types of Functional Dependencies in DBMS

There are mainly four types of Functional Dependency in DBMS. Following are the types of Functional Dependencies in DBMS:

- **Multivalued Dependency**
- **Trivial Functional Dependency**
- **Non-Trivial Functional Dependency**
- **Full Functional Dependency**
- **Partial Functional Dependency**
- **Transitive Dependency**

Multivalued Dependency in DBMS:

Multivalued dependency occurs in the situation where there are multiple independent multivalued attributes in a single table. A multivalued dependency is a complete constraint between two sets of attributes in a relation. It requires that certain tuples be present in a relation. Consider the following Multivalued Dependency Example to understand.

Example:

Car_model	**Maf_year**	**Color**
H001	2017	Metallic
H001	2017	Green
H005	2018	Metallic
H005	2018	Blue
H010	2015	Metallic
H033	2012	Gray

In this example, maf_year and color are independent of each other but dependent on car_model. In this example, these two columns are said to be multivalue dependent on car_model.

This dependence can be represented like this:

car_model -> maf_year

car_model-> colour

Trivial Functional Dependency in DBMS:

The Trivial dependency is a set of attributes which are called a trivial if the set of attributes are included in that attribute.

So, X -> Y is a trivial functional dependency if Y is a subset of X. Let's understand with a Trivial Functional Dependency Example.

For example:

Emp_id	**Emp_name**
AS555	Harry
AS811	George
AS999	Kevin

Consider this table of with two columns Emp_id and Emp_name.

{Emp_id, Emp_name} -> Emp_id is a trivial functional dependency as Emp_id is a subset of {Emp_id,Emp_name}.

Non Trivial Functional Dependency in DBMS:

Functional dependency which also known as a nontrivial dependency occurs when A->B holds true where B is not a subset of A. In a relationship, if attribute B is not a subset of attribute A, then it is considered as a non-trivial dependency.

Company	**CEO**	**Age**
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Apple	Tim Cook	57

Example:

(Company} -> {CEO} (if we know the Company, we knows the CEO name)

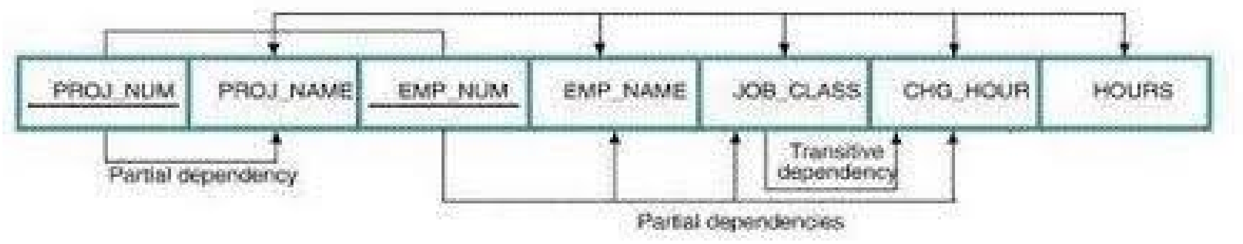
But CEO is not a subset of Company, and hence it's non-trivial functional dependency.

Full Functional Dependency:

A functional dependency X→Y is said to be a full functional dependency, if removal of any attribute A from X, the dependency does not hold any more. i.e. Y is fully functional dependent on X, if it is Functionally Dependent on X and not on any of the proper subset of X.

For example, {Emp_num,Proj_num} → Hour

Is a full functional dependency. Here, Hour is the working time by an employee in a project.



Partial Functional Dependency:

A functional dependency X →Y is said to be a partial functional dependency, if after removal of any attribute A from X, the dependency still holds. i.e. Y is dependent on a proper subset of X.

So X is partially dependent on X.

For example, as per the above relation,

If {Emp_num, Proj_num} → Emp_name but also Emp_num → Emp_name

then Emp_name is partially functionally dependent on{Empl_num,Proj_num}.

a non-key attribute is determined by a part, but not the whole, of a COMPOSITE primary key.

Transitive Dependency in DBMS:

A Transitive Dependency is a type of functional dependency which happens when t is indirectly formed by two functional dependencies. Let's understand with the following Transitive Dependency Example.

Example:

Company	**CEO**	**Age**
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Alibaba	Jack Ma	54

{Company} -> {CEO} (if we know the compay, we know its CEO's name)

{CEO } -> {Age} If we know the CEO, we know the Age

Therefore according to the rule of rule of transitive dependency:

{ Company} -> {Age} should hold, that makes sense because if we know the company name, we can know his age.

Note: You need to remember that transitive dependency can only occur in a relation of three or more attributes.

Advantages of Functional Dependency

- Functional Dependency avoids data redundancy. Therefore same data do not repeat at multiple locations in that database
- It helps you to maintain the quality of data in the database
- It helps you to defined meanings and constraints of databases
- It helps you to identify bad designs
- It helps you to find the facts regarding the database design

DATABASE MANAGEMENT SYSTEMS
BASIC NORMAL FORMS

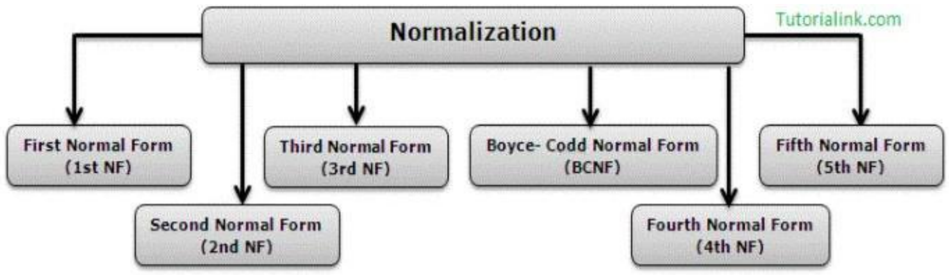
Normalization

Normalization is a method of organizing the data in the database which helps you to avoid data redundancy, insertion, update & deletion anomaly. It is a process of analyzing the relation schemas based on their different functional dependencies and primary key.

Normalization is inherent to relational database theory. It may have the effect of duplicating the same data within the database which may result in the creation of additional tables.

Objectives of Normalization:

1. Eliminate Redundant Data:
   Reduces the duplication of data to save space and ensure consistency.
2. Ensure Data Dependencies Make Sense:
   Organizes data so that dependencies are logical, ensuring data integrity.
3. Simplify Data Management:
   Makes databases easier to maintain by structuring data efficiently.


Normalization
Tutorialink.com
First Normal Form (1st NF)
Second Normal Form (2nd NF)
Third Normal Form (3rd NF)
Boyce- Codd Normal Form (BCNF)
Fourth Normal Form (4th NF)
Fifth Normal Form (5th NF)

Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF, if it is in 1NF and all non-key attributes are fully functional dependent on the primary key. (Removes partial dependency)
3NF	A relation will be in 3NF, if it is in 2NF and no transition dependency exists.
BCNF	A relation will be BCNF, if it is in 3NF and **every functional dependency X → Y, X should be the super key of the table.(i.e** every FD, LHS is super key.)
4NF	A relation will be in 4NF, if it is in Boyce Codd's normal form and has no multi-valued dependency.
5NF	A relation is in 5NF, If it is in 4NF and does not contain any join dependency, joining should be lossless.

First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 8123450987

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says "each attribute of a table must have atomic (single) values", the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	Subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}
Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:
teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal form (3NF)

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Super keys: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}…so on
Candidate Keys: {emp_id}
Non-prime attributes: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

employee table:

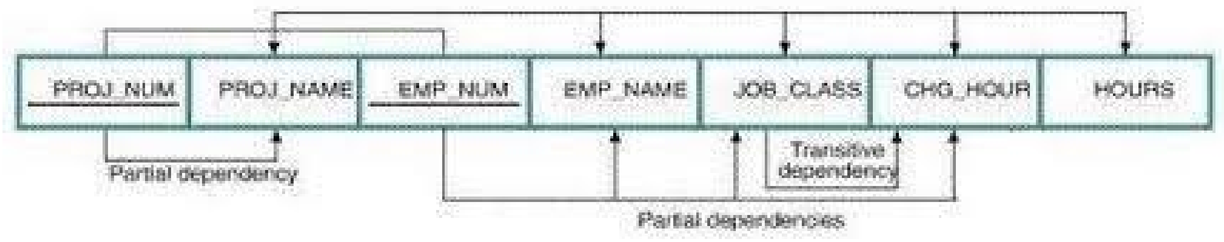
emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Lora	282007
1101	Lilly	292008
1201	Steve	222999

employee_zip table:

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

Normalization – Example

Consider a Relation as given below. Apply the different normal form as applicable



First Normal Form (2NF)

A table is in **1NF** if:

- All columns contain only atomic (indivisible) values.
- Each column contains values of a single type.
- Each column contains only a single value per row.

The given table already satisfies 1NF.

Second Normal Form (2NF)
A table is in 2NF if:

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

The primary key for this table can be a composite key consisting of **PROJ_NUM** and **EMP_NUM**.

However, there are partial dependencies:

PROJ_NUM → PROJ_NAME (Partial dependency)
EMP_NUM → EMP_NAME, **JOB_CLASS** (Partial dependencies)

To move to 2NF, we need to remove partial dependencies:
Create separate tables to eliminate partial dependencies:

Project Table:
PROJ_NUM (Primary Key)
PROJ_NAME

DATABASE MANAGEMENT SYSTEMS
ADVANCED NORMAL FORMS

Boyce Codd normal form (BCNF)

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every functional dependency X->Y, X should be the super key of the table.

Example: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	Stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

Functional dependencies in the table above:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}

Candidate key: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:

emp_nationality table:

emp_id	emp_nationality
1001	Austrian
1002	American

emp_dept table:

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
Stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

emp_dept_mapping table:

emp_id	emp_dept
1001	Production and planning
1001	Stores
1002	design and technical support
1002	Purchasing department

Functional dependencies:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}

Candidate keys:
For first table: emp_id
For second table: emp_dept
For third table: {emp_id, emp_dept}

This is now in BCNF as in both the functional dependencies left side part is a key.

Fourth normal form (4NF)

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
- For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency

EXAMPLE

STUDENT

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.
In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.
So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

STUDENT_HOBBY

STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

Fifth normal form (5NF)

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
- 5NF is also known as Project-join normal form (PJ/NF).

Example

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

P1

SEMESTER	SUBJECT
Semester 1	Computer
Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	LECTURER
Computer	Anshika
Computer	John
Math	John
Math	Akash
Chemistry	Praveen

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

.

DATABASE MANAGEMENT SYSTEMS
ATTRIBUTE CLOSURE

What is Attribute Closure? Attribute closure is a fundamental concept in relational database theory used to determine all possible attributes that can be functionally determined by a given set of attributes in a relation schema, based on a set of functional dependencies.

Formal Definition: The *closure of an attribute set X*, denoted X^+, is the set of all attributes that can be functionally determined by X, given a set of functional dependencies F.

Steps to Find the Attribute Closure

Let's go over the procedure in detail:

1. **Initialization**:
   - Begin by adding the attribute X (whose closure we are calculating) to the closure set.
   - Example: If we are calculating the closure of A, start with A+={A}
2. **Apply Functional Dependencies**:
   - Look at each functional dependency Y→Z in the set F:
     - If all attributes of Y are already in X+, then add all attributes of Z to X+.
   - Repeat this process for each functional dependency until no new attributes can be added.
3. **Termination**:
   - The process stops when no more attributes can be added to X+.

Example :

Consider a relation schema R={A,B,C,D,E} and a set of functional dependencies F.

- A→B
- B→C
- CD→E
- E→A

Calculate the Closure of A (A+):

1. **Initialize**: A+={A}
2. **Apply A→B**: Since A is in A+, add B to A+. Now, A+= {A, B}.
3. **Apply B→C**: Since B is in A+, add C to A+. Now, A+= {A, B, C}.
4. **Apply CD→E**: We need both C and D in A+ to apply this dependency. D is not in A+, so this dependency cannot be applied yet.
5. **Apply E→A**: E is not in A+, so this dependency cannot be applied.
6. **Stop**: No further dependencies can be used, so A+ = {A,B,C}.

Purpose of Attribute Closure

1. **Schema Design**:
   - Helps in normalizing database schemas by understanding which attributes are dependent on others. This is crucial for ensuring minimal redundancy and avoiding anomalies in database operations.
2. **Dependency Preservation**:
   - While decomposing relations to achieve higher normal forms, it's essential to ensure that all original functional dependencies are preserved. Calculating closures can help verify this.
3. **Key Identification**:
   - Attribute closure is used to determine candidate keys of a relation. If the closure of an attribute set equals all attributes of the relation, that set is a candidate key.
4. **Testing for Super keys**:
   - We use closure to test if a given set of attributes is a superkey. If the closure of an attribute set includes all attributes of the relation, it is a superkey.

Benefits of Calculating Attribute Closure

1. **Efficient Schema Design**:
   - Attribute closure simplifies the process of schema normalization by identifying the minimal sets of attributes needed to enforce dependencies.
2. **Redundancy Reduction**:
   - By understanding dependencies, database designers can reduce redundancy and avoid update, insert, and delete anomalies.
3. **Dependency Analysis**:
   - It helps in analysing which attributes are dependent on others, aiding in making decisions about splitting or merging tables for better performance.
4. **Candidate Key Identification**:
   - It aids in identifying all possible candidate keys, which are crucial for determining the primary key of a relation.

Practical Applications

1. **Normalization**: Attribute closure is extensively used in the normalization process to ensure that a database schema is free from undesirable characteristics like redundancy and inconsistency.

2. **Query Optimization**: Understanding functional dependencies through attribute closure can lead to more efficient query execution plans by the database optimizer.

3. **Data Integrity**: Ensures that the schema design maintains data integrity by enforcing functional dependencies.

Example : Finding candidate key using attribute closure

R={A,B,C,D,E}.   The FD'S are AB->CD,   C->A,   D->A

To determine the candidate keys, prime attributes, and non-prime attributes for the relation R={A,B,C,D,E} with the given functional dependencies (FDs):

Functional Dependencies (FDs):

1. AB→CD
2. C→A
3. D→A

Step 1: Determine the Candidate Keys

To find the candidate keys, we need to determine which combinations of attributes can uniquely identify all attributes in the relation RRR.

Closure of Individual Attributes:

- **Closure of A**:
  - A+={A} (Only A is determined, so A is not a candidate key)
- **Closure of B**:
  - B+={B} (Only B is determined, so B is not a candidate key)
- **Closure of C**:
  - Using C→A, C+={C,A} (Only C and A are determined, so C is not a candidate key)
- **Closure of D**:
  - Using D→A, D+={D,A} (Only D and A are determined, so D is not a candidate key)
- **Closure of E**:
  - E+={E} (Only E is determined, so E is not a candidate key)

Closure of Attribute Combinations:

- **Closure of AB**:
  - Using AB→ CD, (AB)+={A,B,C,D}
  - Since AB+ does not include E, AB is not a candidate key.
- **Closure of CE**:
  - CE+={C,E} (Initially)
  - Using C→A, CE+={C,E,A}
  - Since CE+ does not include B,D,CE is not a candidate key.
- **Closure of BD**:
  - BD+={B,D} (Initially)
  - Using D→A, BD+={B,D,A}
  - Since BD+ does not include C,E, BD is not a candidate key.
- **Closure of BE**:
  - BE+={B,E} (Initially)
  - Since BE+ does not include A,C,D, BE is not a candidate key.
- **Closure of CDE**:
  - Using C→A, CDE+={C,D,E,A}
  - Since CDE+ does not include B, CDE is not a candidate key.
- **Closure of BDB**:
  - BDE+={B,D,E} (Initially)
  - Using D→A, BDE+={B,D,E,A}
  - Since BDE+ does not include C, BDE is not a candidate key.
- **Closure of BCE**:
  - Using C→A, BCE+={B,C,E,A}
  - Since BCE+ does not include DBCE is not a candidate key.

- **Closure of BDCE**:
  - Using BDCE+={B,D,A,E}
  - Now BDCE+= {A,B,D,C,E} will be a candidate key.

The two combinations that will not identify the other attributes are: BD and BCE. Therefore, the only candidate key possible is BDCE.

Step 2: Determine Prime and Non-Prime Attributes

- **Prime Attributes**: Attributes that are part of any candidate key.
  - In this case, the prime attributes are B,D,C,E.
- **Non-Prime Attributes**: Attributes that are not part of any candidate key.
  - In this case, the non-prime attribute is A.

Result

- Candidate Keys: BDCE
- Prime Attributes: B,D,C,E
- Non-Prime Attributes: A