RAMAIAH Institute of Technology

M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of CSE (AIML) and CSE (Cyber Security)

Course Name: Database Management Systems
Course Code: CI32
Credits: 3:0:1
UNIT 4

Reference:

Abraham Silberschatz, Henry F Korth, S. Sudarshan
Database System Concepts (7th Edition) →***Prescribed textbook as per curriculum***
Abraham Silberschatz, Henry F Korth, S. Sudarshan
Database System Concepts (3rd Edition)
Abraham Silberschatz, Henry F Korth, S. Sudarshan
Database System Concepts (6th Edition)

RAMAIAH
Institute of Technology

Unit IV Syllabus

- Database Design : Informal Guidelines for Relation Schemas
- Functional Dependencies
- Inference Rules
- Equivalence and Minimal Cover
- Normal Forms based on Primary keys
- First Normal Form
- General Definitions of 2nd and 3rd Normal Forms
- Boyce Codd Normal Form
- Properties of Relational Decomposition
- Algorithms for relational database schema design

RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. Informal Design Guidelines for Relational Databases
   1.1 Semantics of the Relation Attributes
   1.2 Redundant Information in Tuples and Update Anomalies
   1.3 Null Values in Tuples
   1.4 Spurious Tuples

RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. Informal Design Guidelines for Relational Databases

1.1 Semantics of the Relation Attributes

GUIDELINE 1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).

- Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation
- Only foreign keys should be used to refer to other entities
- Entity and relationship attributes should be kept apart as much as possible.

- **Conclusion** - Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.

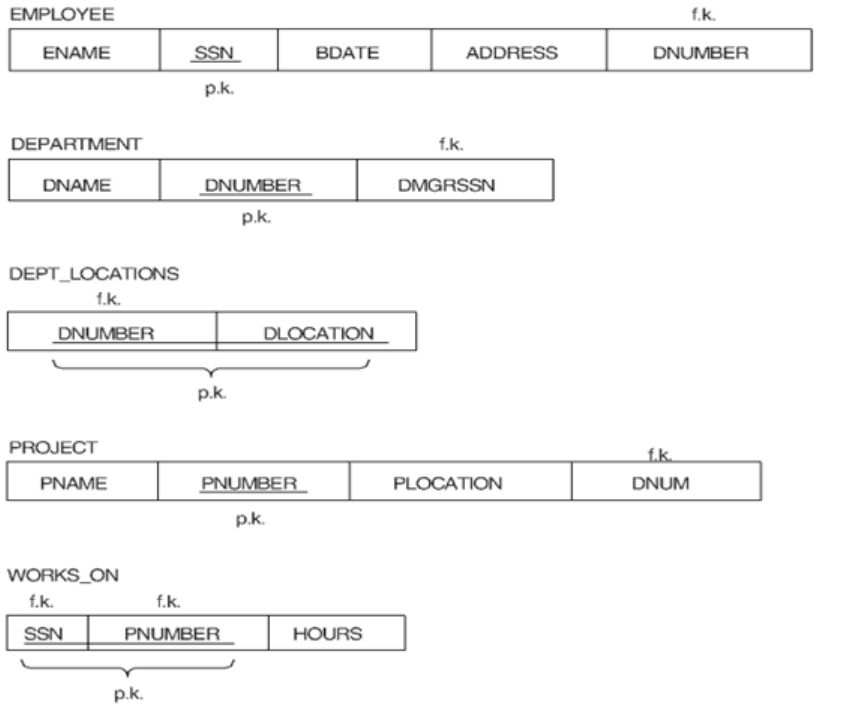
DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. Informal Design Guidelines for Relational Databases

1.1 Semantics of the Relation Attributes

- A simplified COMPANY relational database schema

Simplified version of the COMPANY relational database schema.



RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. Informal Design Guidelines for Relational Databases

1.2 Redundant Information in Tuples and Update Anomalies

- Mixing attributes of multiple entities may cause problems
- Information is stored redundantly wasting storage
- Problems with update anomalies
  - Insertion anomalies
  - Deletion anomalies
  - Modification anomalies

RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. **Informal Design Guidelines for Relational Databases**

1.2 Redundant Information in Tuples and Update Anomalies

EXAMPLE OF AN UPDATE ANOMALY

Consider the relation: **EMP_PROJ (Emp#, Proj#, Ename, Pname, No_hours)**

Update Anomaly: Changing the name of project number P1 from "Billing" to "Customer-Accounting" may cause this update to be made for all 100 employees working on project P1.

Insert Anomaly: Cannot insert a project unless an employee is assigned to the project. In other words, cannot insert an employee unless he/she is assigned to a project.

Delete Anomaly: When a project is deleted, it will result in deleting all the employees who work on that project. Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

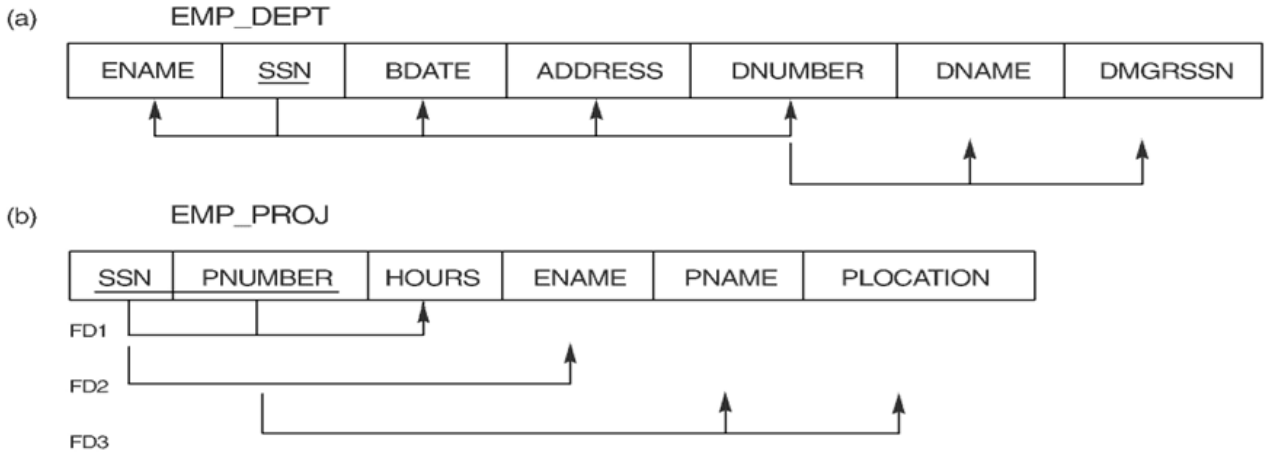
DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. **Informal Design Guidelines for Relational Databases**

1.2 Redundant Information in Tuples and Update Anomalies

Two relation schemas suffering from update anomalies

Two relation schemas and their functional dependencies. Both suffer from update anomalies. (a) The EMP_DEPT relation schema. (b) The EMP_PROJ relation schema.



DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. Informal Design Guidelines for Relational Databases

1.2 Redundant Information in Tuples and Update Anomalies

Figure 14.4 Example relations for the schemas in Figure 14.3 that result from applying NATURAL JOIN to the relations in Figure 14.2. These may be stored as base relations for performance reasons.

EMP_DEPT

ENAME	SSN	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5	Research	333445555
Wong,Franklin T.	333445555	1955-12-08	638 Voss,Houston,TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring,TX	4	Administration	987654321
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4	Administration	987654321
Narayan,Ramesh K.	666884444	1962-09-15	975 FireOak,Humble,TX	5	Research	333445555
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5	Research	333445555
Jabbar,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4	Administration	987654321
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1	Headquarters	888665555

EMP_PROJ

SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION
123456789	1	32.5	Smith,John B.	ProductX	Bellaire
123456789	2	7.5	Smith,John B.	ProductY	Sugarland
666884444	3	40.0	Narayan,Ramesh K.	ProductZ	Houston
453453453	1	20.0	English,Joyce A.	ProductX	Bellaire
453453453	2	20.0	English,Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong,Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong,Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong,Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong,Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya,Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya,Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar,Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar,Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace,Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace,Jennifer S.	Reorganization	Houston
888665555	20	null	Borg,James E.	Reorganization	Houston

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. Informal Design Guidelines for Relational Databases

1.2 Redundant Information in Tuples and Update Anomalies

GUIDELINE 2: Design a schema that does not suffer from the insertion, deletion and update anomalies. If there are any present, then note them so that applications can be made to take them into account

1.3 Null Values in Tuples

GUIDELINE 3: Relations should be designed such that their tuples will have as few NULL values as possible. Attributes that are NULL frequently could be placed in separate relations (with the primary key)

Reasons for nulls:

- attribute not applicable or invalid
- attribute value unknown (may exist)
- value known to exist, but unavailable

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. Informal Design Guidelines for Relational Databases

1.4 Spurious Tuples

- Bad designs for a relational database may result in erroneous results for certain JOIN operations.
- The "lossless join" property is used to guarantee meaningful results for join operations.

GUIDELINE 4: The relations should be designed to satisfy the lossless join condition. No spurious tuples should be generated by doing a natural-join of any relations.
There are two important properties of decompositions:

- non-additive or losslessness of the corresponding join
- preservation of the functional dependencies.

Note that
Property (a) is extremely important and cannot be sacrificed.
Property (b) is less stringent and may be sacrificed.

RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

- Thus, to conclude we can derive at the following:
- Problem Statement: Designing a schema for a relational database.
- Goal / Target: To generate a set of relation schemas that allows us
  - to store information without unnecessary redundancy.
  - Allows us to retrieve information easily.
- This goal is achieved by designing the schemas in a **NORMAL FORM** (a series of guidelines that help ensure a database's design is efficient, organized, and free from data anomalies).

- How do we know that a relational schema is in one of the desirable normal forms?
  1. We compare the relational schema design for our mini world with the real-world enterprise.
  2. We can look at the well-designed ER diagram that we had created as part of the conceptual modelling for our database.

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

- Now here, there is one more way to validate the relational schema designed for our miniworld in a more formal approach.
- This is done by the introduction of a concept known as **FUNCTIONAL DEPENDENCIES**.
- ***Assessing the desirability of a collection of relation schemas***
- Consider the relation schema for the University database that we had seen in previous chapters.

classroom(<u>*building*</u>, <u>*room_number*</u>, *capacity*)
department(<u>*dept_name*</u>, *building*, *budget*)
course(<u>*course_id*</u>, *title*, *dept_name*, *credits*)
instructor(<u>*ID*</u>, *name*, *dept_name*, *salary*)
section(<u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>, *building*, *room_number*, *time_slot_id*)
teaches(<u>*ID*</u>, <u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>)
student(<u>*ID*</u>, *name*, *dept_name*, *tot_cred*)
takes(<u>*ID*</u>, <u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>, *grade*)
advisor(<u>*s_ID*</u>, *i_ID*)
time_slot(<u>*time_slot_id*</u>, <u>*day*</u>, <u>*start_time*</u>, *end_time*)
prereq(<u>*course_id*</u>, <u>*prereq_id*</u>)

Schema for the university database.

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

1. Design Alternative : Larger Schemas

- Suppose that instead of having the schemas instructor and department, we have the schema:

inst_dept (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

- This represents the result of a natural join on the relations corresponding to instructor and department.
- This is the instance of the *inst_dept* relation

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

The *inst_dept* table.

RAMAIAH
Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

- Notice that we have to repeat the department information ("building" and "budget") once for each instructor in the department.
- For example, the information about the Comp. Sci. department (Taylor, 100000) is included in the tuples of instructors Katz, Srinivasan, and Brandt.
- **Data Inconsistency :** In our original design using instructor and department, we stored the amount of each budget exactly once. This suggests that using ***inst_dept*** is a bad idea since it stores the budget amounts redundantly and runs the risk that some user might update the budget amount in one tuple but not all, and thus create inconsistency.
- **Updation becomes tedious**: Another problem with the ***inst_dept*** schema is that suppose we are creating a new department in the university. In the alternative design above, we cannot directly represent the information concerning a department (dept name, building, budget) unless that department has at least one instructor at the university. This is because tuples in the ***inst_dept*** table require values for ID, name, and salary. This means that we cannot record information about the newly created department until the first instructor is hired for the new department. So, in the old design (slide 4), the relation department can handle this scenario. As per the revised design, the only solution is to create a tuple with a NULL value for building and budget.

RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

2. Design Alternative : Smaller Schemas

- **DECOMPOSITION**
- For example, we consider the schema of ***inst_dept***.
- The question here is
  - when and how we can decide ***on splitting the relation*** into the schemas ***instructor*** and ***department***?
- The end goal here is
  - To avoid the repetition-of-information problem in the inst_dept relation
- How can splitting the relation or decomposition/decomposing of the two schemas(instructor and department) be done.
- As a general rule of thumb,

A SCHEMA THAT EXHIBITS REPETITION OF INFORMATION MAY HAVE TO BE DECOMPOSED INTO SEVERAL SMALLER SCHEMAS.

RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

2. Design Alternative : Smaller Schemas

- **DECOMPOSITION**
- When does decomposition of schemas not helpful?
  - Consider the case in which all schemas consist of only one attribute – no meaningful relationships can be formed after decomposition.
  - Consider another case in which we wish to decompose the *employee* schema below into the following schemas:

employee (ID, *name*, *street*, *city*, *salary*)

↓

employee1 (ID, *name*)
employee2 (*name*, *street*, *city*, *salary*)

RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

2. Design Alternative : Smaller Schemas

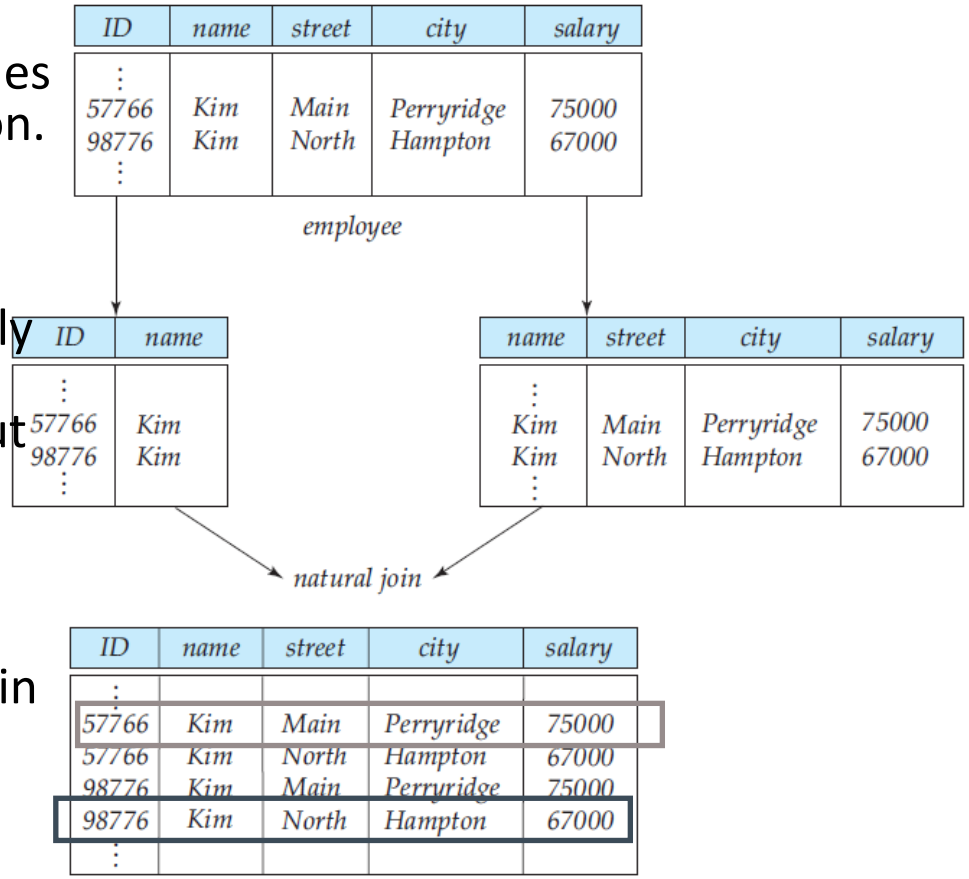
- **DECOMPOSITION**
  - When will this decomposition be a flaw or challenge?
  1. When the enterprise or real-world has 2 employees with the same name. (This scenario is not unlikely in the real-world)
  - Now here, as per the schema, each person has a unique employee-id and this ID can serve as a primary key.
  - Example, let's assume two employees, both named Kim, work at the university and have the following tuples in the relation on schema employee in the original design:

(57766, Kim, Main, Perryridge, 75000)
(98776, Kim, North, Hampton, 67000)

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

2. Design Alternative : Smaller Schemas

- **DECOMPOSITION**
  - The first level and second level of tables shows tuples using the schemas resulting from the decomposition.
  - The last table shows the result ,if we attempted to regenerate the original tuples using a natural join.
  - Here we see that the two original tuples appear in the result along with two new tuples that incorrectly mix data values of the two employees named Kim.
  - This indicates that though there are more tuples but we are having less information that makes sense.
  - Thus, the decomposition is unable to represent certain important facts about the university employees. Such decompositions that result in loss of valuable information or which causes ambiguity in the data is referred to as a LOSSY DECOMPOSITION and those that are not causing so are known as LOSSLESS DECOMPOSITIONS.



DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

2. Design Alternative : Smaller Schemas

- **LOSSLESS DECOMPOSITION AND LOSSY DECOMPOSITION – Formal Definition**
  - Let R be a relation schema.
  - Let R1 and R2 form a decomposition of R ie., R1 and R2 are a set of attributes, R=R1UR2.
  - Given these, we say that the decomposition is lossless decomposition if there is no loss of information by replacing R with the two relation schemas R1 and R2.
  - Example :

    select * from (select R1 from r)
    natural join
    (select R2 from r)

    So in this case, we say that **the decomposition is lossless** if every instance of R ie., r(R) is a combination of r1(R1) and r2(R2)
  - In relational algrebra this can be denoted as

    πR1(r) <join-operator symbol> πR2(r) = r

  - A **decomposition is lossy** when we compute the natural join of the projection results, we get a proper superset of the original relation.
  - In relational algrebra this can be denoted as
  - R <subset or contained operator> πR1(r) <join-operator symbol> πR2(r)

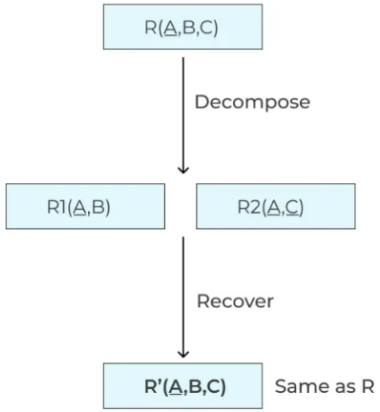
RAMAIAH Institute of Technology

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

2. Design Alternative : Smaller Schemas

- **LOSSLESS DECOMPOSITION - Concept**

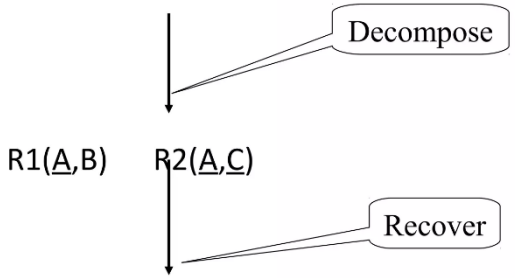
Lossless Join decomposition



Lossless Decomposition

A decomposition is *lossless* if we can recover:

R(A,B,C)



R'(A,B,C) should be the same as R(A,B,C)

Must ensure R' = R

DATABASE DESIGN: Informal Design Guidelines for Relation Schemas

2. Design Alternative : Smaller Schemas

- **LOSSLESS DECOMPOSITION AND LOSSY DECOMPOSITION - Examples**

Lossless Decomposition

- Sometimes the same set of data is reproduced:

Name	Price	Category
Word	100	WP
Oracle	1000	DB
Access	100	DB

Name	Price
Word	100
Oracle	1000
Access	100

Name	Category
Word	WP
Oracle	DB
Access	DB

- (Word, 100) + (Word, WP) → (Word, 100, WP)
- (Oracle, 1000) + (Oracle, DB) → (Oracle, 1000, DB)
- (Access, 100) + (Access, DB) → (Access, 100, DB)

Lossy Decomposition

- Sometimes it's not:

Name	Price	Category
Word	100	WP
Oracle	1000	DB
Access	100	DB

What's wrong?

Category	Name
WP	Word
DB	Oracle
DB	Access

Category	Price
WP	100
DB	1000
DB	100

- (Word, WP) + (100, WP) = (Word, 100, WP)
- (Oracle, DB) + (1000, DB) = (Oracle, 1000, DB)
- (Oracle, DB) + (100, DB) = (Oracle, **100**, DB)
- (Access, DB) + (1000, DB) = (Access, **1000**, DB)
- (Access, DB) + (100, DB) = (Access, 100, DB)

NORMALIZATION THEORY

NORMALIZATION THEORY

- **Goal** - A general methodology for deriving a set of schemas each of which is in "good form"; that is, does not suffer from the repetition-of-information problem.
- The method for designing a relational database is to use a process commonly known as **normalization**.
- **Approach for normalization** –
  1. Decide if a given relation schema is in "good form."
     1. How? - There are a number of different forms (called normal forms) – 1NF-First Normal Form,2NF-Second Normal Form,3NF-Third Normal Form, and Boyce Codd Normal Form
  2. If a given relation schema is not in "good form," then we
     1. decompose it into a number of smaller relation schemas,
     2. each of the relation schemas should be in an appropriate normal form.
  3. Check and ensure that the decomposition is a lossless decomposition.

FUNCTIONAL DEPENDENCIES

- **FUNCTIONAL DEPENDENCIES**
  - A database models a set of entities and relationships in the real world.
  - There are usually a variety of constraints (rules) on the data in the real world.
  - Example – In a university database,
    - Students and instructors are uniquely identified by their ID.
    - Each student and instructor has only one name.
    - Each instructor and student is (primarily) associated with only one department
    - Each instructor and student is (primarily) associated with only one department
    - Each department has only one value for its budget, and only one associated building.
  - What is LEGAL INSTANCE OF THE RELATION?
    An instance of a relation that satisfies all real-world constraints.
  - What is LEGAL INSTANCE OF THE DATABASE?
    All the relation instances are legal instances.

FUNCTIONAL DEPENDENCIES

- **NOTATIONAL CONVENTIONS** – For a relational database design,

Terms	Notations
Set of Attributes	Greek letters (Eg., α)
Relation Schema	Upper case roman letters (Eg., I, V, X, L,)
Refers to a relation r with the schema R	r(R) - refers both to the relation and its schema
A set of attributes is a superkey	K Eg., K is a superkey for R
Lowercase names for relations. But in algorithms or formal definitions we use single letters like ‘r’	Eg., instructors, students
Instance of ‘r’ - a particular value at any given time	Use the relation name ‘r’

FUNCTIONAL DEPENDENCIES

- **KEYS AND FUNCTIONAL DEPENDENCIES**
- The most commonly used types of real-world constraints can be represented formally as keys or as functional dependencies.
  - **Super keys** – set of one or more attributes that, taken collectively, allows us to identify uniquely a tuple in the relation
  - **Candidate keys** – a minimal super key, meaning it's a super key that doesn't include any redundant attributes
  - **Primary Keys** - uniquely identify and index each row within a single table.
  - **Functional Dependency (FD)**
    - ✓ A functional dependency is a constraint that describes a relationship between two sets of attributes in a relation (or table).
    - ✓ A set of attributes X functionally determines a set of attributes Y if the value of X determines a unique value for Y
    - ✓ X -> Y holds if, whenever two tuples have the same value for the attributes in X, they must have the same value for Y

RAMAIAH Institute of Technology

FUNCTIONAL DEPENDENCIES

- **KEYS AND FUNCTIONAL DEPENDENCIES**
  - **Functional Dependency (FD)**
    For any two tuples t1 and t2 in any relation instance r(R): If t1[X]=t2[X], then t1[Y]=t2[Y]
    X -> Y in R specifies a constraint on all relation instances r(R)
    Written as X -> Y; can be displayed graphically on a relation schema as ⟶
  - **Examples of Functional Dependencies**
    1. social security number determines employee name SSN -> ENAME
    2. project number determines project name and location PNUMBER -> {PNAME, PLOCATION}
    3. Employee ssn and project number determine the hours per week that the employee works on the project {SSN, PNUMBER} -> HOURS

Suppose consider the example of the schema that we considered earlier

inst_dept (ID, name, salary, dept_name, building, budget)

Here the functional dependency **dept_name → budget** holds because for each department (identified by dept_name) there is a unique budget amount.

FUNCTIONAL DEPENDENCIES

- **KEYS AND FUNCTIONAL DEPENDENCIES**

  the pair of attributes (ID, dept name) forms a superkey for in_dep and we can write it as

$$ID, dept\_name \rightarrow name, salary, building, budget$$

Usage of Functional Dependencies

1. To test instances of relations to see whether they satisfy a given set F of functional Dependencies.
2. To specify constraints on the set of legal relations. When a constraint is specified for relations on schema r(R), then it should be such that the constraint satisfies a set F of functional dependencies, then we can say that F **holds** on r(R).

FUNCTIONAL DEPENDENCIES

- **KEYS AND FUNCTIONAL DEPENDENCIES**
  - Consider the instance of relation r to see which functional dependencies are satisfied.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Sample instance of relation r.

1. Here, **A → C is satisfied**.
   - **Reason:** There are two tuples that have an A value of a1. These tuples have the same C value—namely, c1.
   - Similarly, the two tuples with an A value of a2 have the
   - same C value, c2.

2. Here, **C → A is not satisfied**.
   - **Reason:** Consider the tuples t1 = (a2, b3, c2, d3) and t2 =(a3, b3, c2, d4)
   - These two tuples have the same C values, c2, but they have different A values, a2 and a3, respectively.
   - Thus, we have found a pair of tuples t1 and t2 such that t1[C] = t2[C], but t1[A] ≠ t2[A].

Thus in this case the functional dependency FD is said to be **non-trivial**.

- However, certain functional dependencies are said to be **trivial** because they are satisfied by all relations.

FUNCTIONAL DEPENDENCIES

- **KEYS AND FUNCTIONAL DEPENDENCIES**
  - Consider the instance of relation *classroom* to see which functional dependencies are satisfied.

1. Here, **room_number → capacity is satisfied**.
2. In the real world, two classrooms in different buildings can have the same room number but with different room capacity. Thus, it is possible, at some time, to have an instance of the classroom relation in which **room number → capacity is not satisfied.**

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 7.5 An instance of the *classroom* relation.

FUNCTIONAL DEPENDENCIES

- **KEYS AND FUNCTIONAL DEPENDENCIES**
  - Hence,
    Given a schema r(A, B, C), if functional dependencies A → B and B → C hold on r,
    we can infer the functional dependency A → C must also hold on r.
  - **Closure:** the set of all functional dependencies that can be inferred given the set F. notation F^+ to denote the closure of the set F. F+ contains all of the functional dependencies in F.
  - **LOSSLESS DECOMPOSITION AND FUNCTIONAL DEPENDENCIES**
  - *Purpose:* To show that certain **decompositions are lossless** we can use functional dependencies.
  - Consider R = a relation schema
    R1, R2 = decompositions of R
    F = Given set of FD in R
  - R1 and R2 form a lossless decomposition of R if and only if atleast one of the functional dependencies is in F^+
    - $R_1 \cap R_2 \rightarrow R_1$
    - $R_1 \cap R_2 \rightarrow R_2$

FUNCTIONAL DEPENDENCIES

- **LOSSLESS DECOMPOSITION AND FUNCTIONAL DEPENDENCIES**
- If R1∩R2 forms a superkey for either R1 or R2, the decomposition of R is a lossless decomposition.
- Example consider the student table

student_id	student_name	course_id	course_name	instructor_name
1	Alice	101	Math 101	Dr. Smith
2	Bob	102	History 101	Prof. Johnson

Functional Dependencies (F):

- `student_id -> student_name` (A student ID uniquely determines the student's name)
- `course_id -> course_name, instructor_name` (A course ID uniquely determines the course name and instructor)

FUNCTIONAL DEPENDENCIES

- **LOSSLESS DECOMPOSITION AND FUNCTIONAL DEPENDENCIES**

We decompose the original table `Student` into two sub-relations: **R1** and **R2**.

R1: Contains the attributes `student_id`, `student_name`, and `course_id`

student_id	student_name	course_id
1	Alice	101
2	Bob	102

R2: Contains the attributes `course_id`, `course_name`, and `instructor_name`

course_id	course_name	instructor_name
101	Math 101	Dr. Smith
102	History 101	Prof. Johnson

FUNCTIONAL DEPENDENCIES

- **LOSSLESS DECOMPOSITION AND FUNCTIONAL DEPENDENCIES**

Reconstructed Table (Result of the Join):

student_id	student_name	course_name	instructor_name
1	Alice	Math 101	Dr. Smith
2	Bob	History 101	Prof. Johnson

- This decomposition is lossless because:
  - R1 and R2 share the common attribute course_id.
  - The functional dependency course_id -> course_name, instructor_name ensures that the data in R2 can be fully reconstructed when we join it with R1.
  - The original data can be perfectly reconstructed from R1 and R2, ensuring no information is lost.

FUNCTIONAL DEPENDENCIES

- **LOSSLESS DECOMPOSITION AND FUNCTIONAL DEPENDENCIES**

After decomposition, how to ensure that the contents are consistent with the original schema?

- Suppose we decompose a relation schema r(R) into r1(R1) and r2(R2), where R1 ∩ R2 → R1.
- Then the following SQL constraints must be imposed on the decomposed schema to ensure their contents are consistent with the original schema.
  1. R1 ∩ R2 is the primary key of r1.
     This constraint enforces the functional dependency.
  2. R1 ∩ R2 is a foreign key from r2 referencing r1.
     This constraint ensures that each tuple in r2 has a matching tuple in r1, without which it would not appear in the natural join of r1 and r2.

RAMAIAH
Institute of Technology

FUNCTIONAL DEPENDENCIES

1.1 Closure Set of Functional Dependencies
1.1.1 Inference Rules
1.1.1 Closure of Attribute Sets
1.2 Equivalence and Minimal Cover – Canonical Cover

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**
- Given a set F of functional dependencies on a schema, we can prove that certain other functional dependencies also hold on the schema.
- Such additional functional dependencies are "logically implied" by F.
- When testing for normal forms, it is not sufficient to consider the given set of functional dependencies; rather, we **need to consider all functional dependencies that hold on the schema**.
- Formally,
  - Given relation schema r(R),

    a functional dependency f on R is logically implied by a set of functional dependencies F on R,

    if every instance of a relation r(R) that satisfies F also satisfies f .

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**
- Consider the scenario below:
- **Given:**
  - **relation schema r(A, B, C, G, H, I) and**
  - **the set of functional dependencies:**

    A → B

    A → C

    CG → H

    CG → I

    B → H
  - **To prove that there is an implied functional dependency:**

    A → H (Meaning: we need to prove that whenever a relation instance satisfies our given set of functional dependencies, A→H must also be satisfied by that relation Instance**)**

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**
- **Proof:**
  - ✓ Suppose that t1 and t2 are tuples such that
    t1[A] = t2[A]
  - ✓ Now since, we are given that A → B, it follows from the definition of functional dependency that:
    t1[B] = t2[B]
  - ✓ Then, since we are given that B → H, it follows from the definition of functional dependency that:
    t1[H] = t2[H]
- **Conclusion:**
  - Hence, whenever t1 and t2 are tuples such that t1[A] = t2[A],
  - Thus the definition of A → H is proved.

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**
- ***Challenges*** *in computing the closure set F^+ for a set of Functional Dependency F*
  1. Given F be a set of functional dependencies, the **closure** of F, denoted by **F+**, is the set of all functional dependencies logically implied by F.
  2. Given F we can compute F^+ directly from the formal definition of functional dependency.
  3. Now imagine a scenario where F is large, the computation process for F^+ would be lengthy and difficult.
- **AXIOMS**
  - Rules of Inference or Axioms provide a simpler technique for reasoning about functional dependencies.
  - Here the following notations are used:
    - Greek letters (α, β, γ,…) for sets of attributes
    - uppercase Roman letters from the beginning of the alphabet for individual attributes
    - αβ denotes denote α ∪ β

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**
  - The Rules of Inference or Axioms can be used to find logically implied functional dependencies.
  - By applying these rules repeatedly, we can find all of F+, given F.
  - This collection of rules is called **ARMSTRONG'S AXIOMS.**
  - The collection of rules in the Armstrong's axioms are:
    1. **Reflexivity rule** - If α is a set of attributes and β ⊆ α, then α → β holds.
    2. **Augmentation rule** - α → β holds and γ is a set of attributes, then γα → γβ holds.
    3. **Transitivity rule** - If α → β holds and β → γ holds, then α → γ holds.

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**
  - **Reflexivity rule** - If α is a set of attributes and β ⊆ α, then α → β holds.

Example, consider the EMPLOYEE table

EmpID	EmpName	Department	Salary
1	Alice	HR	50000
2	Bob	IT	60000
3	Charlie	IT	55000

So when the rule of reflexivity is applied onto this table,

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**

Reflexivity Rule Applied to This Table

- Let $\alpha = \{EmpID, EmpName, Department, Salary\}$ (the set of all attributes in the table).
- Let $\beta = \{EmpID, EmpName\}$ (a subset of the attributes).

According to the Reflexivity Rule, since $\beta \subseteq \alpha$, we have:

$$\{EmpID, EmpName, Department, Salary\} \rightarrow \{EmpID, EmpName\}$$

This means that **if you know the values of** `EmpID`, `EmpName`, `Department`, **and** `Salary` **for a record**, you automatically know the values of `EmpID` and `EmpName`. This is trivially true because `EmpID` and `EmpName` are part of the full set of attributes in the relation.

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**

  Reflexivity rule conclusion:

  1. The Reflexivity Rule states that for any set of attributes α and any subset $\beta \subseteq \alpha$, the functional dependency $\alpha \rightarrow \beta$ always holds.
  2. In a relational table, this means that knowing all attributes in α allows you to determine any subset β, because a set of attributes always determines its own subsets.

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**
  - **Augmentation rule** - α → β holds and γ is a set of attributes, then γα → γβ holds.
  - Example, consider the EMPLOYEE table

EmpID	EmpName	Department	Salary
1	Alice	HR	50000
2	Bob	IT	60000
3	Charlie	IT	55000

So when the rule of augmentation is applied onto this table,

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**

Now, let's assume the following **functional dependency**:

- $EmpID \rightarrow EmpName$: If we know an employee's `EmpID`, we can determine their `EmpName`.

Augmentation Rule Example:

- We know that $EmpID \rightarrow EmpName$, i.e., knowing `EmpID` will determine `EmpName`.
- Let's say $\gamma = \{Department\}$, and we want to apply the **Augmentation Rule**.

According to the **Augmentation Rule**:

$$\text{If } EmpID \rightarrow EmpName, \text{ then } Department, EmpID \rightarrow Department, EmpName$$

This means that if you know `Department` and `EmpID`, you can also determine both `Department` and `EmpName` (where `EmpID` determines `EmpName`). By adding `Department` to both sides of the dependency, we maintain the same functional relationship.

So, if $\gamma = \{Department\}$, we get:

$$Department, EmpID \rightarrow Department, EmpName$$

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**

  Augmentation rule conclusion:

  1. The Augmentation Rule says that if α→β is a valid functional dependency, then adding the same set of attributes γ to both sides of the dependency will still result in a valid functional dependency. Specifically, if α→β, then γα→γβ will also hold.
  2. In a relational table, if we know α determines β, then adding any extra information (attributes) to both α and β still maintains the functional dependency relationship.

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**
  - **Transitivity rule -** If α → β holds and β → γ holds, then α → γ holds.
  - Example, consider the EMPLOYEE table

EmpID	EmpName	Department	ManagerID
1	Alice	HR	101
2	Bob	IT	102
3	Charlie	IT	102
4	Dave	HR	101

So when the rule of transitivity is applied onto this table,

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**

Now, let's assume the following functional dependencies:

- $EmpID \rightarrow EmpName$: If we know `EmpID`, we can determine `EmpName`.
- $EmpName \rightarrow Department$: If we know `EmpName`, we can determine `Department`.

Applying the Transitivity Rule

Now, let's apply the **Transitivity Rule**. We have two functional dependencies:

1. $EmpID \rightarrow EmpName$: Knowing the `EmpID` determines the `EmpName`.
2. $EmpName \rightarrow Department$: Knowing the `EmpName` determines the `Department`.

According to the **Transitivity Rule**, if $EmpID \rightarrow EmpName$ and $EmpName \rightarrow Department$, then $EmpID \rightarrow Department$ must also hold. This means knowing the `EmpID` directly gives us the `Department`, because of the transitive relationship through `EmpName`.

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**

  Transitivity rule conclusion:

  1. The Transitivity Rule states that if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ holds.
  2. This rule allows us to derive new functional dependencies by combining existing ones.
  3. In relational database design, the Transitivity Rule helps in simplifying the functional dependencies and reasoning about normalization.

- **Properties of Armstrong's axiom:**

  1. Armstrong's axioms are **sound**, because they do not generate any incorrect functional dependencies.
  2. Armstrong's axioms are **complete**, because or a given set F of functional dependencies, they allow us to generate all F^+

FUNCTIONAL DEPENDENCIES

- Despite the Armstrong's rule being sound and complete, it is tiresome to use them directly for the computation of F+. To simplify this, we have additional 3 rules,

4. Union rule – If α→β holds and α→ γ holds, then α →βγ holds.

5. Decompostion rule – If α→βγ holds and α→β holds and α →γ holds.

6. Pseudotransitivity rule – If α→β holds and γβ →δ holds, then αγ → δ holds.

FUNCTIONAL DEPENDENCIES

- Despite the Armstrong's rule being sound and complete, it is tiresome to use them directly for the computation of F+. To simplify this, we have additional 3 rules,

  4. Union rule – If α→β holds and α→ γ holds, then α →βγ holds.

  i.e., α →β U γ holds.

Example, consider the EMPLOYEE table

EmpID	EmpName	Department	ManagerID
1	Alice	HR	101
2	Bob	IT	102
3	Charlie	IT	102
4	Dave	HR	101

FUNCTIONAL DEPENDENCIES

Let's assume the following **functional dependencies**:

- $EmpID \rightarrow EmpName$: Knowing the `EmpID` determines the `EmpName`.
- $EmpID \rightarrow Department$: Knowing the `EmpID` also determines the `Department`.

Applying the Union Rule

We have two functional dependencies:

1. $EmpID \rightarrow EmpName$: Knowing `EmpID` determines `EmpName`.
2. $EmpID \rightarrow Department$: Knowing `EmpID` determines `Department`.

According to the **Union Rule**, if $EmpID$ determines both $EmpName$ and $Department$, then:

$$EmpID \rightarrow EmpName, Department$$

This means that knowing `EmpID` will determine both `EmpName` and `Department` together (the union of `EmpName` and `Department`).

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**

  Union rule conclusion:

  1. The Union Rule states that if $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta \cup \gamma$ holds.
  2. In simple terms, if a set of attributes α determines two other sets of attributes β and γ, then α will determine the union of β and γ as well.
  3. The Union Rule is useful for deriving new functional dependencies in a database schema when you have multiple dependencies that involve the same set of attributes.

FUNCTIONAL DEPENDENCIES

- Despite the Armstrong's rule being sound and complete, it is tiresome to use them directly for the computation of F+. To simplify this, we have additional 3 rules,

  4. Decomposition rule – If α→βγ holds and α→β holds and α →γ holds.

  if a set of attributes α determines the union of two sets β and γ, then α will also determine β and γ individually.

  Example, consider the EMPLOYEE table

EmpID	EmpName	Department	ManagerID
1	Alice	HR	101
2	Bob	IT	102
3	Charlie	IT	102
4	Dave	HR	101

FUNCTIONAL DEPENDENCIES

Let's assume the following **functional dependency**:

- $EmpID \rightarrow EmpName, Department$: Knowing the `EmpID` determines both `EmpName` and `Department` together (the union of `EmpName` and `Department`).

Applying the Decomposition Rule

We know that:

- $EmpID \rightarrow EmpName, Department$: Knowing `EmpID` gives us both `EmpName` and `Department` (the union of these attributes).

According to the **Decomposition Rule**, we can **decompose** this dependency into two simpler dependencies:

- $EmpID \rightarrow EmpName$: Knowing `EmpID` gives us `EmpName`.
- $EmpID \rightarrow Department$: Knowing `EmpID` gives us `Department`.

FUNCTIONAL DEPENDENCIES

- **Closure Set of Functional Dependencies**

  Decomposition rule conclusion:

  - The Decomposition Rule states that if $\alpha \rightarrow \beta \cup \gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ hold.
  - This rule allows you to break down a complex functional dependency (involving a union of attributes) into simpler functional dependencies.
  - In relational database design, this rule helps to simplify and normalize the functional dependencies, making the schema easier to analyze and maintain.

FUNCTIONAL DEPENDENCIES

- Despite the Armstrong's rule being sound and complete, it is tiresome to use them directly for the computation of F+. To simplify this, we have additional 3 rules,

  4. Pseudotransitivity rule – If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.
- In simpler terms, Pseudotransitivity says that if:
  - α determines β,
  - $\beta \cup \gamma$ determines δ,
  - Then, $\alpha \cup \gamma$ determines δ.
  - This rule helps us to extend the functional dependencies from one set of attributes to a larger set, essentially "adding" new attributes to a functional dependency and still maintaining the determination relationship.

FUNCTIONAL DEPENDENCIES

EmpID	EmpName	Department	ManagerID
1	Alice	HR	101
2	Bob	IT	102
3	Charlie	IT	102
4	Dave	HR	101

Let's assume the following **functional dependencies**:

1. $EmpID \rightarrow EmpName$: Knowing `EmpID` determines `EmpName`.
2. $EmpName \cup Department \rightarrow ManagerID$: Knowing both `EmpName` and `Department` determines `ManagerID`.

FUNCTIONAL DEPENDENCIES

Applying the Pseudotransitivity Rule

According to the **Pseudotransitivity Rule**:

- $EmpID \rightarrow EmpName$ tells us that `EmpID` determines `EmpName`.
- $EmpName \cup Department \rightarrow ManagerID$ tells us that the union of `EmpName` and `Department` determines `ManagerID`.

From this, we can apply the **Pseudotransitivity Rule**:

Since $EmpID$ determines $EmpName$, and $EmpName \cup Department$ determines `ManagerID`, we can say that:

$$EmpID \cup Department \rightarrow ManagerID$$

This means that knowing both `EmpID` and `Department` will give us `ManagerID`. This is consistent with the rule, as we extended $EmpID$ with $Department$ to determine $ManagerID$.

FUNCTIONAL DEPENDENCIES

- Let us apply our rules to the example of schema R = (A, B, C, G, H, I) and the set F of functional dependencies {A → B, A → C, CG → H, CG → I , B → H}.
- Several members of F+ here:
  - A → H. Since A → B and B → H hold, we apply the ***transitivity rule***.
  - CG → HI. Since CG → H and CG → I, the ***union rule*** implies that CG → HI.
  - AG → I. Since A → C and CG → I, the ***pseudo-transitivity*** rule implies that AG → I holds.
  - Another way of finding that AG → I holds is as follows: We use the augmentation rule on A → C to infer AG → CG.
  - Applying the transitivity rule to this dependency and CG → I, we infer AG → I.

FUNCTIONAL DEPENDENCIES

- **<u>Closure of Attribute Sets – FDs w.r.to SUPER KEYS</u>**
- An attribute B is functionally determined by α if α → B.
- How to test whether a set α is a super key ?
- **Approach I**:
  - To compute F+, take all functional dependencies with α as the left-hand side(i.e., α^+) and take the union of the right-hand sides of all such dependencies
  - Example – Consider an Employee table and Functional Dependencies (F) defined as
    1. EmpID → EmpName: EmpID determines EmpName
    2. EmpID→Department: EmpID determines Department.
    3. EmpID→ManagerID: EmpID determines ManagerID.
  - Now, let's say we want to compute the closure of α={EmpID}, which $\{EmpID\}^+$,

FUNCTIONAL DEPENDENCIES

- **Closure of Attribute Sets**

Steps to Compute F^+ Based on $\alpha = \{EmpID\}$:

1. **Step 1: Identify all functional dependencies with $EmpID$ on the left-hand side.**
   - From the functional dependencies, we can see that $EmpID$ appears on the left-hand side of the first three dependencies:
     1. $EmpID \rightarrow EmpName$
     2. $EmpID \rightarrow Department$
     3. $EmpID \rightarrow ManagerID$
2. **Step 2: Take the union of the right-hand sides of these dependencies.**
   - From the first functional dependency, we get `EmpName`.
   - From the second functional dependency, we get `Department`.
   - From the third functional dependency, we get `ManagerID`.

   So, the union of the right-hand sides will be:

$$\{EmpName, Department, ManagerID\}$$

FUNCTIONAL DEPENDENCIES

- **Closure of Attribute Sets**

3. **Step 3: Add these attributes to α^+.**
   - Initially, $\alpha^+ = \{EmpID\}$.
   - After adding the union of the right-hand sides, we have:

$$\{EmpID, EmpName, Department, ManagerID\}$$

4. **Step 4: Repeat if necessary** (if any new dependencies can be applied).
   - In this case, α^+ already contains all the attributes of the table, so no further steps are needed.

- So, after performing all the 4 steps we get $\{EmpID\}^+ = \{EmpID, EmpName, Department, ManagerID\}$
- This means that knowing EmpID determines all other attributes (EmpName, Department, and ManagerID). Therefore, EmpID is a **superkey** in this table because $\{EmpID\}^+$ contains all the attributes in the relation.

FUNCTIONAL DEPENDENCIES

- **Closure of Attribute Sets**
- *Algorithm for computing the set of attributes functionally determined by α:*
- Let α be a set of attributes.
- The set of all attributes functionally determined by α, under a set F of functional dependencies, the closure of α under F is denoted as α^+

```
result := α;
repeat
    for each functional dependency β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ;
        end
until (result does not change)
```

Figure 7.8 An algorithm to compute α^+, the closure of α under F.

- ***Inputs:***
  - A set F of functional dependencies and the set α of attributes.
  - The output is stored in the variable result

FUNCTIONAL DEPENDENCIES

- **Closure of Attribute Sets**
- *Attribute Closure Algorithm for computing the set of attributes functionally determined by α:*

- **Step 1:** Initially, the result is set to α. This is because the closure of α will start with α itself, and we will gradually add more attributes to result based on the functional dependencies.
- **Step 2:** This loop will continue until no new attributes can be added to result.
- **Step 3:** Iterates through every functional dependency in the set F.
- **Step 4:** For each functional dependency β→γ, check if the attributes in β are already present in the result. If β is a subset of result, this means we can apply the functional dependency and add the attributes in γ to result.

FUNCTIONAL DEPENDENCIES

- **Closure of Attribute Sets**

- *Uses of the Attribute Closure Algorithm :*
- To test if α is a superkey, we compute α+ and check if α+ contains all attributes in R.
- We can check if a functional dependency α → β holds (or, in other words, is in F+), by checking if β ⊆ α+. That is, we compute α+ by using attribute closure, and then check if it contains β.
- It gives us an alternative way to compute F+

FUNCTIONAL DEPENDENCIES

- **Canonical Cover**
- **Purpose:** Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies.
- Meaning that all the functional dependencies in F are satisfied in the new database state. system must roll back the update if it violates any functional dependencies in the set F.
- **Benefit**: Reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set
- **EXTRANEOUS ATTRIBUTES:**
- An attribute of a functional dependency is said to be extraneous if we can remove it without changing the closure of the set of functional dependencies.

FUNCTIONAL DEPENDENCIES

- **Equivalence of sets of FDs**
- Two sets of FDs F and G are equivalent if:

  - every FD in F can be inferred from G, and

  - every FD in G can be inferred from F
- Hence, F and G are equivalent if F + =G +
- **Definition:** F covers G if every FD in G can be inferred from F (i.e., if G + subset-of F +)
  - F and G are equivalent if F covers G and G covers F
  - Note: Please check problems on finding if two sets of FDs are equivalent or not, given 2 sets of FDs.

FUNCTIONAL DEPENDENCIES

- **Canonical Cover**
- **EXTRANEOUS ATTRIBUTES:**
- The formal definition of extraneous attributes is as follows:
  - Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
- Removal from the left side: Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- Removal from the right side: Attribute A is extraneous in β if $A \in \beta$and the set of functional dependencies
  $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

FUNCTIONAL DEPENDENCIES

- **Canonical Cover**
- **EXTRANEOUS ATTRIBUTES:**

1. Extraneous Attribute on the Left Side (Removal from the Left Side)

Formal Definition:

Consider a set F of functional dependencies, and a functional dependency $\alpha \rightarrow \beta$ in F.

- Attribute A is **extraneous in** α if:

$$A \in \alpha \quad \text{and} \quad F \text{ logically implies } (F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}.$$

This means that if we can remove A from α and still have the same set of functional dependencies logically implied by F, then A is extraneous in α.

FUNCTIONAL DEPENDENCIES

- **Canonical Cover**
- **EXTRANEOUS ATTRIBUTES:**

Example:

Consider the following set of functional dependencies for a relation:

- $F = \{\{A, B\} \rightarrow C, \{A, C\} \rightarrow D, \{A, B, D\} \rightarrow E\}$

Now, let's consider the functional dependency $\{A, B\} \rightarrow C$, and check if attribute A is extraneous in the left-hand side of this dependency.

1. Remove A from the left-hand side of $\{A, B\} \rightarrow C$, which gives us $\{B\} \rightarrow C$.
2. Now, check if $F - \{\{A, B\} \rightarrow C\}$ (i.e., the set of dependencies excluding $\{A, B\} \rightarrow C$) logically implies $\{B\} \rightarrow C$.
    - The remaining dependencies are:
        - $\{A, C\} \rightarrow D$
        - $\{A, B, D\} \rightarrow E$

FUNCTIONAL DEPENDENCIES

- **Canonical Cover**
- **EXTRANEOUS ATTRIBUTES:**

3. **Can we derive $\{B\} \rightarrow C$ from these remaining dependencies?**
    - We cannot derive $\{B\} \rightarrow C$ from the remaining dependencies directly because no dependency implies $\{B\}$ determining C without A. Thus, A **is not extraneous** in $\{A, B\} \rightarrow C$.

However, if the dependencies were different, say $\{A, B\} \rightarrow C$ and $\{B\} \rightarrow C$ already existed, A would be extraneous because we could derive $\{B\} \rightarrow C$ from the other dependencies.

FUNCTIONAL DEPENDENCIES

- **Canonical Cover**
- **EXTRANEOUS ATTRIBUTES:**

2. Extraneous Attribute on the Right Side (Removal from the Right Side)

Formal Definition:

Consider a set F of functional dependencies, and a functional dependency $\alpha \rightarrow \beta$ in F.

- Attribute A is **extraneous in** β if:

$$A \in \beta \quad \text{and} \quad (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\} \text{ logically implies } F.$$

This means that if we can remove A from β and still have the same set of functional dependencies logically implied by F, then A is extraneous in β.

FUNCTIONAL DEPENDENCIES

- **Canonical Cover**
- **EXTRANEOUS ATTRIBUTES:**

Example:

Let's continue with the previous set of functional dependencies:

- $F = \{\{A, B\} \rightarrow C, \{A, C\} \rightarrow D, \{A, B, D\} \rightarrow E\}$

Now, let's consider the functional dependency $\{A, B\} \rightarrow C$, and check if attribute C is extraneous in the right-hand side.

1. Remove C from the right-hand side of $\{A, B\} \rightarrow C$, which gives us $\{A, B\} \rightarrow$ (empty set).
2. Now, check if $F - \{\{A, B\} \rightarrow C\}$ logically implies the empty set.
    - The remaining dependencies are:
        - $\{A, C\} \rightarrow D$
        - $\{A, B, D\} \rightarrow E$

FUNCTIONAL DEPENDENCIES

- **Canonical Cover**
- **EXTRANEOUS ATTRIBUTES:**

3. **Can we derive that $\{A, B\} \rightarrow$ (empty set) from these dependencies?**
    - No, we cannot derive an empty set from the remaining dependencies; the dependency $\{A, B\} \rightarrow C$ is essential.

In this case, C **is not extraneous** in $\{A, B\} \rightarrow C$.

- Removing extraneous attributes helps simplify functional dependencies and is essential for database normalization, such as when computing **candidate keys** or simplifying **normal forms**.

FUNCTIONAL DEPENDENCIES

- **Canonical Cover -** construct a simplified set of functional dependencies **equivalent** to a given set of functional dependencies.
- A canonical cover Fc for F is a set of dependencies such that F logically implies all dependencies in Fc, and Fc logically implies all dependencies in F.
- Fc must have the following properties:
  1. No functional dependency in Fc contains an extraneous attribute.
  2. Each left side of a functional dependency in Fc is unique. That is, there are no two dependencies $\alpha 1 \rightarrow \beta 1$ and $\alpha 2 \rightarrow \beta 2$ in Fc such that $\alpha 1 = \alpha 2$.

$F_c = F$
repeat
  Use the union rule to replace any dependencies in F_c of the form $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.
  Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous attribute either in α or in β.
    /* Note: the test for extraneous attributes is done using F_c, not F */
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in F_c.
until (F_c does not change)

Figure 7.9 Computing canonical cover.

FUNCTIONAL DEPENDENCIES

- **Example**

  Given Functional Dependencies:

  We are given the following set **F** of functional dependencies on schema (**A, B, C**):

  1. A → BC
  2. B → C
  3. A → B
  4. AB → C

- The steps of computing the canonical cover (Fc) for the given set of functional dependencies F will be shown below:

FUNCTIONAL DEPENDENCIES

- **Example**

Step 1: Combine Functional Dependencies with the Same Left-Hand Side (LHS)

We start by examining if there are any dependencies that have the same set of attributes on the left-hand side (LHS). In this case, **A → BC** and **A → B** both have **A** as the LHS.

- **A → BC** and **A → B** can be combined into a single functional dependency **A → BC** because:
  - **A → BC** already implies **A → B**, and combining them does not add any new information. Therefore, we retain just **A → BC**.

Now, the updated set of functional dependencies is:

1. **A → BC**
2. **B → C**
3. **AB → C**

FUNCTIONAL DEPENDENCIES

- **Example**

Step 2: Remove Extraneous Attributes

Next, we look for any extraneous attributes in the functional dependencies.

- **AB → C**: We check if **A** or **B** is extraneous in this dependency.
  - First, we check if **B** is extraneous. If we remove **B**, we need to see if **A → C** can still be derived from the remaining set of functional dependencies.
  - From **A → BC** and **B → C**, we can derive **A → C** (since **A → BC** already implies **A → C**). Therefore, **B** is extraneous in **AB → C**, and we can simplify it to **A → C**.

Now, the set of functional dependencies is:

1. **A → BC**
2. **B → C**
3. **A → C**

FUNCTIONAL DEPENDENCIES

- **Example**

Step 3: Remove Further Extraneous Attributes

- **A → BC**: Next, we check if **C** is extraneous in **A → BC**.
  - We want to see if **A → B** and **B → C** together can imply **A → BC**. Since **A → B** and **B → C** together imply **A → BC** (because knowing **A → B** and **B → C** allows us to infer **A → BC**), **C** is extraneous.
  - Therefore, we can reduce **A → BC** to **A → B**.

Now the set of functional dependencies is:

1. **A → B**
2. **B → C**

Final Canonical Cover (Fc)

The canonical cover is:

- **A → B**
- **B → C**

FUNCTIONAL DEPENDENCIES

- **Dependency Preservation:**
- This is an additional property to ensure a good relational design.
- Let F be a set of functional dependencies on a schema R.
- Let R1, R2,…, Rn be a decomposition of R
- The restriction of F to Ri is the set Fi of all functional dependencies in F+ that include only attributes of Ri.
- Since all functional dependencies in a restriction

involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.

NORMALIZATION CONCEPTS

- NORMAL FORMS BASED ON PRIMARY KEYS
  - Normalization of Relations
  - Practical Use of Normal Forms
  - Definitions of Keys and Attributes Participating in Keys
  - First Normal Form
  - Second Normal Form
  - Third Normal Form

- **Normalization:** The process of decomposing unsatisfactory "bad" relations ***by breaking up their attributes into smaller relations***
- **Normal Form (NF):** Condition ***using keys and FDs of a relation*** to ***certify whether a relation schema is in a particular normal form***.
- How keys and FDs are used to determine the type of normal form (NF)?

NORMALIZATION CONCEPTS

- ***How keys and FDs are used to determine the type of normal form (NF)?***

Keys	Functional dependant	Multi-valued Dependencies	Type of NF
Based on Keys of a relation schema	Based on FD of a relation schema	-	2NF,3NF,BCNF
Based on Keys of a relation schema	-	Based on Multivalued dependencies	4NF

NORMALIZATION CONCEPTS

- **Normalization** is carried out in practice so that the resulting designs are of high quality and meet the desirable properties.
- The practical utility of these normal forms becomes questionable when the constraints on which they are based are ***hard to understand*** or to ***detect***.
- The database designers ***need not*** normalize to the highest possible normal form. (usually up to 3NF, BCNF or 4NF)
- **Denormalization:** the process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

NORMALIZATION CONCEPTS

- **ATOMIC DOMAINS AND FIRST NORMAL (1NF) FORMS:**
  - **Purpose:** The E-R model allows entity sets and relationship sets to have attributes that have some degree of substructure. Specifically, it allows multivalued attributes such as ***phone_number;*** and composite attributes (such as an attribute ***address*** with component attributes ***street, city, and state***
  - So, when we design tables, each of these type of attributes gets translated into the following from an E-R diagram.
    - Composite Attributes - each component be an attribute in its own right.
    - Multi-valued Attributes - create one tuple for each item in a multivalued set.
  - Hence in a relational model, a domain is **atomic** if elements of the domain are considered to be indivisible units.
  - A relation schema R **is in first normal form (1NF)** if the **domains of all attributes of R are atomic**.

NORMALIZATION CONCEPTS

- **ATOMIC DOMAINS AND FIRST NORMAL FORMS:**
  - **Examples –**
    - A ***set of names*** is an example of a non-atomic value.
    - **Integers** are assumed to be atomic.
  - So 1NF disallows composite attributes, multivalued attributes, and nested relations; attributes whose values for an individual tuple are non-atomic.

NORMALIZATION CONCEPTS

Figure 14.8 Normalization into 1NF. (a) Relation schema that is not in 1NF. (b) Example relation instance. (c) 1NF relation with redundancy.

(a)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS

(b)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

RAMAIAH Institute of Techno

NOR

Figure 14.9 Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a "nested relation" PROJS. (b) Example extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposing EMP_PROJ into 1NF relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

(a) **EMP_PROJ**

SSN	ENAME	PROJS	
		PNUMBER	HOURS

(b) **EMP_PROJ**

SSN	ENAME	PNUMBER	HOURS
123456789	Smith,John B.	1	32.5
		2	7.5
666884444	Narayan,Ramesh K.	3	40.0
453453453	English,Joyce A.	1	20.0
		2	20.0
333445555	Wong,Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya,Alicia J.	30	30.0
		10	10.0
987987987	Jabbar,Ahmad V.	10	35.0
		30	5.0
987654321	Wallace,Jennifer S.	30	20.0
		20	15.0
888665555	Borg,James E.	20	null

(c) **EMP_PROJ1**

SSN	ENAME

EMP_PROJ2

SSN	PNUMBER	HOURS

NORMALIZATION CONCEPTS

- **SECOND NORMAL (2NF) FORM:**
  - **Purpose:** 2NF (Second Normal Form) is a level of database normalization designed to reduce redundancy and ensure that the data is logically stored in a way that avoids certain types of data anomalies, specifically partial dependency anomalies.
  - A relation (or table) is in 2NF if it satisfies the following two conditions:
    1. **It is in 1NF (First Normal Form)**: This means that the table has only atomic (indivisible) values, each column contains only one value per record, and all entries in a column are of the same type.
    2. **No partial dependency:** There should be no partial dependency in the table. A partial dependency occurs when a non-prime attribute (i.e., an attribute that is not part of a candidate key) depends on only part of a composite primary key (a primary key made up of multiple attributes), rather than on the whole primary key.

NORMALIZATION CONCEPTS

- **SECOND NORMAL (2NF) FORM:**
  - Uses the concepts of FDs, primary key.
  - **Definitions:**
    - **Prime attribute** - attribute that is member of the primary key K. i.e., an attribute is considered a **prime attribute** if it is part of the **primary key** (i.e., any attribute that is included in the set of attributes that form the primary key).
    - **Full functional dependency** - Y→Z means that all attributes in Y (the determinant) are necessary to determine Z. If we remove any attribute from Y, the functional dependency no longer holds. In other words, there should be no partial dependencies where a subset of the determinant can still uniquely determine the dependent attributes.

NORMALIZATION CONCEPTS

- **SECOND NORMAL (2NF) FORM:**

Example 1: Relation in 1NF but Not in 2NF

Consider the following relation:

Student_Course (Student_ID, Course_ID, Instructor_Name, Instructor_Phone)

- Primary Key: $\{Student_ID, Course_ID\}$

Example 1: Breaking into 2NF

Original Table: Student_Course

Student_ID	Course_ID	Instructor_Name	Instructor_Phone
S001	C001	Dr. Smith	123-456-7890
S002	C001	Dr. Smith	123-456-7890
S003	C002	Prof. Johnson	987-654-3210
S001	C003	Dr. Lee	555-123-4567

NORMALIZATION CONCEPTS

- **SECOND NORMAL (2NF) FORM:**

  Primary Key: {Student_ID, Course_ID}

  In this relation, we can observe that `Instructor_Name` and `Instructor_Phone` depend only on `Course_ID`, not on `Student_ID`. This means there is a partial dependency, and we need to split the table to bring it into 2NF.

NORMALIZATION (

- **SECOND NORMAL (2NF) FOR**

Step 1: Split the Relation into Two

Relation 1: Course_Instructor

Course_ID	Instructor_Name	Instructor_Phone
C001	Dr. Smith	123-456-7890
C002	Prof. Johnson	987-654-3210
C003	Dr. Lee	555-123-4567

Primary Key: {Course_ID}

This table captures the instructor information, where `Instructor_Name` and `Instructor_Phone` depend only on the primary key `Course_ID`.

Relation 2: Student_Course

Student_ID	Course_ID
S001	C001
S002	C001
S003	C002
S001	C003

Primary Key: {Student_ID, Course_ID}

This table captures the enrollment information for students, with a composite primary key `{Student_ID, Course_ID}`.

NORMALIZATION CONCEPTS

- **SECOND NORMAL (2NF) FORM:**

Example 2: Breaking into 2NF

Original Table: Employee_Project

Employee_ID	Project_ID	Project_Manager	Hours_Worked
E001	P001	Alice	40
E002	P001	Alice	35
E003	P002	Bob	30
E001	P003	Charlie	25

Primary Key: {Employee_ID, Project_ID}

In this relation, we see that `Project_Manager` depends only on `Project_ID`, not on the full composite key. This is another example of a **partial dependency**, so we need to decompose the table into 2NF.

RAMAIAH Institute of Technology

NORMA

- **SECOND N**

Step 1: Split the Relation into Two

Relation 1: Project_Manager_Info

Project_ID	Project_Manager
P001	Alice
P002	Bob
P003	Charlie

Primary Key: {Project_ID}

This table captures the project manager information, where `Project_Manager` depends only on the `Project_ID`.

Relation 2: Employee_Project_Hours

Employee_ID	Project_ID	Hours_Worked
E001	P001	40
E002	P001	35
E003	P002	30
E001	P003	25

Primary Key: {Employee_ID, Project_ID}

This table captures the work hours for employ on specific projects, with a composite primary key `{Employee_ID, Project_ID}`.

NORMALIZATION CONCEPTS

- **SECOND NORMAL (2NF) FORM:**

Summary of the Tables

1. Before Normalization:

Student_Course

Student_ID	Course_ID	Instructor_Name	Instructor_Phone
S001	C001	Dr. Smith	123-456-7890
S002	C001	Dr. Smith	123-456-7890
S003	C002	Prof. Johnson	987-654-3210
S001	C003	Dr. Lee	555-123-4567

After Normalization (to 2NF):

Student_Course

Student_ID	Course_ID
S001	C001
S002	C001
S003	C002
S001	C003

Course_Instructor

Course_ID	Instructor_Name	Instructor_Phone
C001	Dr. Smith	123-456-7890
C002	Prof. Johnson	987-654-3210
C003	Dr. Lee	555-123-4567

NORMALIZATION CONCEPTS

- **SECOND NORMAL (2NF) FORM:**

Before Normalization:

Employee_Project

Employee_ID	Project_ID	Project_Manager	Hours_Worked
E001	P001	Alice	40
E002	P001	Alice	35
E003	P002	Bob	30
E001	P003	Charlie	25

After Normalization (to 2NF):

Employee_Project_Hours

Employee_ID	Project_ID	Hours_Worked
E001	P001	40
E002	P001	35
E003	P002	30
E001	P003	25

Project_Manager_Info

Project_ID	Project_Manager
P001	Alice
P002	Bob
P003	Charlie

NORMALIZATION CONCEPTS

- **Concluding SECOND NORMAL (2NF) FORM:**
  - A relation schema R is in second normal form (2NF) if every non-prime attribute A in R is **fully functionally dependent** on the primary key.
  - R can be decomposed into 2NF relations via the process of 2NF normalization.

NORMALIZATION CONCEPTS

- **THIRD NORMAL (3NF) FORM:**

  Purpose: The primary motivation for going from 2NF (Second Normal Form) to 3NF (Third Normal Form) is **to eliminate transitive dependencies** and further reduce redundancy in the database.

  - A relation schema R is in third normal form (3NF) if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key
  - R can be decomposed into 3NF relations via the process of 3NF normalization
  - **Definition:**

    A relation schema R is in third normal form with respect to a set F of functional dependencies if, for all functional dependencies in F+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

    1. $\alpha \rightarrow \beta$ is a trivial functional dependency.
    2. α is a superkey for R.
    3. Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

NORMALIZATION CONCEPTS

- **THIRD NORMAL (3NF) FORM:**

  Transitive Dependency: A transitive dependency occurs when a non-prime attribute depends on another non-prime attribute via a third attribute (i.e., A→B and B→C imply A→C).

Why Eliminate Transitive Dependencies?

Transitive dependencies create unnecessary redundancy and make the database structure more complex. When a relation is not in 3NF, it can lead to **update anomalies**, **insert anomalies**, and **delete anomalies**. Let's explore these issues in more detail.

1. Redundancy and Update Anomalies

If a non-prime attribute depends on another non-prime attribute, and that non-prime attribute is repeated across many tuples, it can lead to redundancy. For example:

Example:

Consider the relation (after 2NF normalization):

Employee (Employee_ID, Department_ID, Department_Name, Employee_Name)

- **Primary Key:** {Employee_ID}

NORMALIZATION CONCEPTS

- **THIRD NORMAL (3NF) FORM:**

1. Redundancy and Update Anomalies

If a non-prime attribute depends on another non-prime attribute, and that non-prime attribute is repeated across many tuples, it can lead to redundancy. For example:

Example:

Consider the relation (after 2NF normalization):

Employee (Employee_ID, Department_ID, Department_Name, Employee_Name)

- **Primary Key:** {Employee_ID}

- FDs - Employee_ID → {Department_ID, Department_Name, Employee_Name}
  Department_ID → Department_Name
- Problem : If we need to update the name of a department, we would need to update it in every row that references that department. If we forget to update one or more rows, the database will contain inconsistent data.

NORMALIZATION CONCEPTS

- **THIRD NORMAL (3NF) FORM:**

2. Insert and Delete Anomalies

Transitive dependencies can also lead to **insert anomalies** and **delete anomalies**.

Insert Anomaly:

- If we want to insert a new department, we would need to insert its name as well, even if no employees are associated with it yet. This can create unnecessary data entries and can lead to the introduction of null values or redundant information.

Delete Anomaly:

- If we delete the last employee from a department, we might lose information about the department, such as its name, even though the department itself still exists in the organization.

NORMALIZATION CONCEPTS

- **THIRD NORMAL (3NF) FORM:**

  Example of Normalizing from 2NF to 3NF:

  Let's start with an example relation that is in 2NF but not in 3NF:

  Employee (Employee_ID, Department_ID, Department_Name, Employee_Name)

  - Primary Key: {Employee_ID}

- **Functional Dependencies:** Employee_ID → {Department_ID, Department_Name, Employee_Name}
  Department_ID → Department_Name

  - So there is a transitive dependency i.e., Employee_ID → Department_Name
  - This violates 3NF.

NORMALIZATION CONCEPTS

- **THIRD NORMAL (3NF) FORM:**

Step 1: Convert to 3NF

To bring this into **3NF**, we need to eliminate the transitive dependency. We can split the relation into two:

Relation 1: Employee

Employee_ID	Department_ID	Employee_Name
E001	D001	Alice
E002	D001	Bob
E003	D002	Charlie

Primary Key: {Employee_ID}

Relation 2: Department

Department_ID	Department_Name
D001	HR
D002	IT

Primary Key: {Department_ID}

Now, the relation is in **3NF**, because:

- Every non-prime attribute is **directly dependent** on the primary key, and not through another non-prime attribute.
- There are **no transitive dependencies**.

NORMALIZATION CONCEPTS

BOYCE CODD NORMAL (BCNF) FORM:

- BCNF is a stricter version of 3NF.
- **Definition:** A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is a trivial functional dependency (i.e., $\beta \subseteq \alpha$).
  - α is a superkey for schema R.
- **How to test if a relation is in BCNF?**
  - In other words, A relation is in BCNF if it satisfies the following conditions:
    - It is in 3NF.
    - For every non-trivial functional dependency X→Y, X is a superkey (i.e., X must be a superkey, or it must uniquely identify a row in the table).
- A **trivial functional dependency** is a type of functional dependency where the right-hand side (RHS) of the dependency is always contained in the left-hand side (LHS) (or) a functional dependency is trivial if the set of attributes on the right-hand side is a subset of the set of attributes on the left-hand side.
- A functional dependency X→Y is trivial if:
  - Y is a subset of X. This means that all the attributes in Y are already part of the set X, so the dependency is obvious or self-evident or implied.

NORMALIZATION CONCEPTS

- **BOYCE CODD NORMAL (BCNF) FORM:**

Step 1: Initial Relation (Not in BCNF)

Consider the following table representing **Student-Course** relationships:

Relation: Student_Course

Student_ID	Course_ID	Instructor_Name	Instructor_Phone
S001	C001	Dr. Smith	123-456-7890
S002	C001	Dr. Smith	123-456-7890
S003	C002	Prof. Johnson	987-654-3210
S001	C003	Dr. Lee	555-666-7777

- FDs: {Student_ID, Course_ID} → {Instructor_Name, Instructor_Phone}
  {Course_ID} → {Instructor_Name, Instructor_Phone}
- Composite Primary key: {Student_ID,Course_ID} as each student is enrolled in multiple courses, and each combination of Student_ID and Course_ID is unique.

NORMALIZATION CONCEPTS

BOYCE CODD NORMAL (BCNF) FORM:

Now, **why is the above relation told not to be in BCNF?**

- A relation is in BCNF if, for every non-trivial functional dependency X→Y, X is a superkey (i.e., it uniquely identifies a tuple in the relation).
- In the current relation, we have a functional dependency:
  {Course_ID} → {Instructor_Name, Instructor_Phone}
- Here, {Course_ID} is **not a superkey** because it does not uniquely identify a row in the relation (Multiple students can be enrolled in the same course, so {Course_ID} does not determine a unique tuple by itself).
- Since, {Course_ID} is not a superkey, this violates the BCNF condition.

NORMALIZATION CONCEPTS

BOYCE CODD NORMAL (BCNF) FORM:

- **Decompose the Relation to Achieve BCNF:**

To convert this relation into BCNF, we need to remove the dependency where a non-superkey (in this case, {Course_ID} determines other attributes. We can do this by decomposing the relation.

1. Relation 1: Course_Instructor where the FD will be Course_ID → {Instructor_Name, Instructor_Phone}

Course_Instructor (Course_ID, Instructor_Name, Instructor_Phone)

Course_ID	Instructor_Name	Instructor_Phone
C001	Dr. Smith	123-456-7890
C002	Prof. Johnson	987-654-3210
C003	Dr. Lee	555-666-7777

- Primary Key: $\{Course_ID\}$

Hence in this relation (Relation 1), **Course_ID** is the **SuperKey** and so this table **satisfies BCNF.**

NORMALIZATION CONCEPTS

BOYCE CODD NORMAL (BCNF) FORM:

- **Decompose the Relation to Achieve BCNF:**
  2. Relation 2: Student_Course where the FD will be Course_ID → {Instructor_Name, Instructor_Phone}

Student_Course (Student_ID, Course_ID)

Student_ID	Course_ID
S001	C001
S002	C001
S003	C002
S001	C003

- Primary Key: $\{Student_ID, Course_ID\}$
- There are no functional dependencies other than the one implied by the primary key, so this relation is trivially in BCNF.

There are no non-trivial functional dependencies that violate the BCNF condition. The primary key {Student_ID, Course_ID} uniquely determines the tuple, so this relation is also in BCNF.

NORMALIZATION CONCEPTS

BOYCE CODD NORMAL (BCNF) FORM:

- **Verifying BCNF Compliance:**

Relation 1: Course_Instructor where the FD will be Course_ID → {Instructor_Name, Instructor_Phone} holds, and {Course_ID} is the superkey. Hence, the relation is in BCNF.

Relation 2: Student_Course
There are no non-trivial functional dependencies that violate the BCNF condition. The primary key {Student_ID, Course_ID} uniquely determines the tuple, so this relation is also in BCNF.

PROPERTIES OF RELATIONAL DECOMPOSITIONS

1. Relation Decomposition and Insufficiency of Normal Forms :

- **Universal Relation Schema:** a relation schema R={A1, A2, …, An} that includes all the attributes of the database.
- **Universal Relation Assumption:** Every attribute name is unique.
- **Decomposition:** The process of decomposing the universal relation schema R into a set of relation schemas D = {R1,R2, …, Rm} that will become the relational database schema by using the functional dependencies.
- **Attribute preservation condition:** Each attribute in R will appear in at least one relation schema Ri in the decomposition so that no attributes are “lost”.
- Another goal of decomposition is to have each individual relation Ri in the decomposition D be in BCNF or 3NF.
- Additional properties of decomposition are it is needed to prevent from generating spurious tuples.

PROPERTIES OF RELATIONAL DECOMPOSITIONS

2. Dependency Preservation Property of a Decomposition :

- **Definition:** Given a set of dependencies *F* on *R*, the **projection** of *F* on R_i, denoted by $p_{Ri}(F)$ where R_i is a subset of *R*, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i.
- Hence, the projection of F on each relation schema Ri in the decomposition D is the set of functional dependencies in F+, the closure of F, such that all their left- and right-hand-side attributes are in Ri.
- **Dependency Preservation Property:**
- A decomposition D = {R1, R2, ..., Rm} of R is dependency-preserving with respect to F if the union of the projections of F on each Ri in D is equivalent to F; that is, $((R1(F)) \cup \ldots \cup (Rm(F)))^+ = F^+$
- Let's see this property with an example.

PROPERTIES OF RELATIONAL DECOMPOSITIONS

2. Dependency Preservation Property of a Decomposition :

Example 1: Decomposition of Relation $R(A, B, C)$

Let's take a relation $R(A, B, C)$ and assume the following functional dependencies:

- $F = \{A \rightarrow B, B \rightarrow C\}$

We decompose R into two sub-relations:

- $R_1(A, B)$
- $R_2(B, C)$

Now, we check if this decomposition is dependency-preserving:

- $\pi_{R_1}(F) = \{A \rightarrow B\}$ (since only A and B appear in R_1).
- $\pi_{R_2}(F) = \{B \rightarrow C\}$ (since only B and C appear in R_2).

The union of these projections is:

$$\pi_{R_1}(F) \cup \pi_{R_2}(F) = \{A \rightarrow B, B \rightarrow C\}$$

PROPERTIES OF RELATIONAL DECOMPOSITIONS

2. Dependency Preservation Property of a Decomposition :

Now, let's check if these functional dependencies imply the closure of F, i.e., $F^+ = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$. The closure of $\{A \rightarrow B, B \rightarrow C\}$ does indeed include $A \rightarrow C$.

So, this decomposition is **dependency-preserving** because the functional dependencies implied by the projections are the same as those implied by the original set F.

- **Claim 1:** It is always possible to find a dependency-preserving decomposition D with respect to F such that each relation Ri in D is in 3NF.

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

Definition:

- **Lossless Join Property**: a decomposition D = {R1, R2, ..., Rm} of R has the lossless (nonadditive) join property with respect to the set of dependencies F on R if, for every relation state r of R that satisfies F, the following holds, where * is the natural join of all the relations in D:

$$* (\pi_{R1}(r), ..., \pi_{Rm}(r)) = r$$

- Note: The word loss in lossless refers to *loss of information*, not to loss of tuples. In fact, for "loss of information" a better term is **"addition of spurious information"**
- **Algorithm: Testing for Lossless Join Property**
- **Input**: A universal relation R, a decomposition D = {R1, R2, ..., Rm} of R, and a set F of functional dependencies.

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**

1. Create an initial matrix S with one row i for each relation Ri in D, and one column j for each attribute Aj in R.
2. Set $S(i,j):=b_{ij}$ for all matrix entries. (* each b_{ij} is a distinct symbol associated with indices (i,j) *).
3. For each row i representing relation schema Ri

{for each column j representing attribute Aj

{if (relation Ri includes attribute Aj) then set S(i,j):= aj;};};

(* each aj is a distinct symbol associated with index (j) *)

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**

4. Repeat the following loop until a complete loop execution results in no changes to S
   {for each functional dependency X → Y in F
   {for all rows in S which have the same symbols in the columns corresponding to attributes in X
   {make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: if any of the rows has an "a" symbol for the column, set the other rows to that same "a" symbol in the column. If no "a" symbol exists for the attribute in any of the rows, choose one of the "b" symbols that appear in one of the rows for the attribute and set the other rows to that same "b" symbol in the column ;};};};
5. If a row is made up entirely of "*a*" symbols, then the decomposition has the lossless join property; otherwise, it does not.

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**
- **Example:**

Let's work through the algorithm with a concrete example.

Given:

- Original Relation $R(A, B, C, D)$.
- Decomposition $D = \{R_1(A, B), R_2(B, C), R_3(C, D)\}$.
- Functional Dependencies:
  - $F = \{A \rightarrow B, B \rightarrow C\}$.

Step 1: Initialize the Matrix S

We have 3 relations and 4 attributes, so our matrix S will be a 3x4 matrix:

$$S = \begin{matrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \end{matrix}$$

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**
- **Example:**

Step 2: Update Matrix Based on Attribute Inclusion

For each relation, update the matrix:

- For $R_1(A, B)$:
  - A is in R_1, so set $S(1,1) = a_1$.
  - B is in R_1, so set $S(1,2) = a_2$.
  - C and D are not in R_1, so leave $S(1,3)$ and $S(1,4)$ as b_{13} and b_{14}.

The updated row for R_1 is:

a_1 & a_2 & b_{13} & b_{14}

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**
- **Example:**
  - For $R_2(B, C)$:
    - B is in R_2, so set $S(2, 2) = a_2$.
    - C is in R_2, so set $S(2, 3) = a_3$.
    - A and D are not in R_2, so leave $S(2, 1)$ and $S(2, 4)$ as b_{21} and b_{24}.

    The updated row for R_2 is:

    b_{21} & a_2 & a_3 & b_{24}
  - For $R_3(C, D)$:
    - C is in R_3, so set $S(3, 3) = a_3$.
    - D is in R_3, so set $S(3, 4) = a_4$.
    - A and B are not in R_3, so leave $S(3, 1)$ and $S(3, 2)$ as b_{31} and b_{32}.

    The updated row for R_3 is:

    b_{31} & b_{32} & a_3 & a_4

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**
- **Example:**

After this step, the matrix S is:

$$\begin{array}{cccc} a_1 & a_2 & b_{13} & b_{14} \\ b_{21} & a_2 & a_3 & b_{24} \\ b_{31} & b_{32} & a_3 & a_4 \end{array}$$

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**
- **Example:**

Step 3: Process Functional Dependencies

Now, process the functional dependencies:

- For $A \rightarrow B$:
  - Find rows where A has the same symbol. In this case, $S(1,1) = a_1$ is the symbol for A.
  - Since $A \rightarrow B$, all rows with a_1 in the A-column should have the same symbol in the B-column.
  - Row 1 already has a_2 in the B-column.
  - For Row 2, $S(2,2)$ is already a_2, so no change is needed.

After this step, the matrix remains unchanged.

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**
- **Example:**

  - For $B \rightarrow C$:
    - Find rows where B has the same symbol. In this case, $S(1,2) = a_2$ and $S(2,2) = a_2$.
    - Since $B \rightarrow C$, all rows with a_2 in the B-column should have the same symbol in the C-column.
    - Row 1 has b_{13} in the C-column, and Row 2 has a_3 in the C-column.
    - To make them the same, we update b_{13} in Row 1 to a_3.

  The updated matrix is:

$$\begin{matrix} a_1 & a_2 & a_3 & b_{14} \\ b_{21} & a_2 & a_3 & b_{24} \\ b_{31} & b_{32} & a_3 & a_4 \end{matrix}$$

PROPERTIES OF RELATIONAL DECOMPOSITIONS

3. Lossless (Non-additive) Join Property of a Decomposition:

- **Algorithm: Testing for Lossless Join Property**
- **Example:**

Step 4: Repeat Until No Changes

Repeat the process until no changes occur. In this case, after processing the functional dependencies, the matrix stabilizes.

Step 5: Lossless Join Check

If any row in the matrix consists entirely of a-symbols, the decomposition has the lossless join property.

In our case, none of the rows consist entirely of a-symbols (since there are still b-symbols in some columns), so **this decomposition does not satisfy the lossless join property**.

PROPERTIES OF RELATIONAL DECOMPOSITIONS

4. **Testing Binary Decompositions for Lossless Join Property:**

- **Binary Decomposition** : Decomposition of a relation R into two relations.
- **Property LJ1 (lossless join test for binary decompositions):**
  A decomposition D = {R1, R2} of R has the lossless join property wrt. a set of functional dependencies F on R if and only if either
  The f.d. ((R1 ∩ R2) → (R1- R2)) is in F+, or
  The f.d. ((R1 ∩ R2) → (R2 - R1)) is in F+

5. **Successive Lossless Join Decomposition**:
   - **Claim 2 (Preservation of non-additivity in successive decompositions):**
     If a decomposition D = {R1, R2, ..., Rm} of R has the lossless (non-additive) join property with respect to a set of functional dependencies F on R, and if a decomposition Di = {Q1, Q2, ..., Qk} of Ri has the lossless (non-additive) join property with respect to the projection of F on Ri, then the decomposition D2 = {R1, R2, ..., Ri-1, Q1, Q2, ..., Qk, Ri+1, ..., Rm} of R has the non-additive join property with respect to F.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis Algorithm:**
- **Goal -** Relational Synthesis into 3NF with Dependency Preservation.
- **Input -** A universal relation R and a set of functional dependencies F on the attributes of R.
- **Algorithm Steps –**
  1. Find a minimal cover G for F.
  2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes {X ∪ {A1} ∪ {A2} ... ∪ {Ak}}, where X → A1, X → A2, ..., X → Ak are the only dependencies in G with X as left-hand-side (X is the key of this relation)
  3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.
- **Claim –** Every relation schema created by this algorithm is in 3NF.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis Algorithm:**

  Example :

  Given:

  - A relation `R` with attributes: `R(A, B, C, D)`
  - A set of functional dependencies `F` :
    - `A → B`
    - `B → C`
    - `A → D`

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis Algorithm:**

Example :

Step 1: Find a minimal cover G for F.

A **minimal cover** is a simplified version of the set of functional dependencies such that:

- The right-hand side of each dependency has only one attribute.
- Each dependency is essential (i.e., no redundant dependencies).

Given Functional Dependencies:

- `A → B`
- `B → C`
- `A → D`

We can see that:

- `A → B` and `A → D` can be kept as they are.
- `B → C` is already minimal since there is only one attribute on the right-hand side.

So, **G = {A → B, B → C, A → D}**.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis Algorithm:**

Example :

Step 2: Create Relations Based on the Left-Hand-Sides of Dependencies.

Now we create relations based on the left-hand sides of each functional dependency.

For `A → B`:

- We create a relation schema for the set of attributes `{A, B}` because `A` is the left-hand side.

  Relation R1 = (A, B).

For `B → C`:

- We create a relation schema for the set of attributes `{B, C}` because `B` is the left-hand side.

  Relation R2 = (B, C).

For `A → D`:

- We create a relation schema for the set of attributes `{A, D}` because `A` is the left-hand side.

  Relation R3 = (A, D).

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis Algorithm:**

  Example :

Step 3: Place Any Remaining Attributes in a Single Relation.

We now check which attributes have been placed in one of the above relations, and if any attributes are missing, we put them in a new relation.

The attributes used so far are `A`, `B`, `C`, and `D`. There are no remaining attributes left, so there's no need for an additional relation.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis Algorithm:**

Example :

Step 4: Verify that All Relations are in 3NF.

To confirm that each relation is in **3NF**, we check:

- If a relation is in **2NF**: It is in 2NF if there are no partial dependencies (dependencies where a non-prime attribute is dependent on only part of the primary key).
- If there are no **transitive dependencies**: A non-prime attribute should not depend on another non-prime attribute.

Checking R1 = (A, B):

- Functional dependency: `A → B`.
- `A` is the primary key, and `B` is fully dependent on it. There are no transitive dependencies.
- R1 is in 3NF.

Checking R2 = (B, C):

- Functional dependency: `B → C`.
- `B` is the primary key, and `C` is fully dependent on it. No transitive dependencies.
- R2 is in 3NF.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis Algorithm:**

  Example :

Checking R3 = (A, D):

- Functional dependency: `A → D`.
- `A` is the primary key, and `D` is fully dependent on it. No transitive dependencies.
- R3 is in 3NF.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis Algorithm:**

  Example :

Result:

The final decomposition of the relation `R(A, B, C, D)` based on the functional dependencies `A → B`, `B → C`, and `A → D` is:

- R1(A, B)
- R2(B, C)
- R3(A, D)

Each of these relations is in **3NF** and preserves the functional dependencies.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Decomposition into BCNF with Lossless (non-additive) join property:**
  - **Goal –** To achieve Boyce Codd Normal Form and preserve the lossless join property in relations
  - **Input -** A universal relation R and a set of functional dependencies F on the attributes of R.
  - **Algorithm Steps –**
    1. Set D := {R};
    2. While there is a relation schema Q in D that is not in BCNF

       do {

       choose a relation schema Q in D that is not in BCNF;

       find a functional dependency X → Y in Q that violates BCNF;

       replace Q in D by two relation schemas (Q - Y) and (X ∪ Y);

       };
  - **Assumption:** No null values are allowed for the join attributes.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Decomposition into BCNF with Lossless (non-additive) join property:**
  - **Example –**

Given:

We have a relation `R(A, B, C)` with functional dependencies:

- `A → B`
- `B → C`

We need to decompose this relation into BCNF while ensuring that the decomposition has the **Lossless Join Property**.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Decomposition into BCNF with Lossless (non-additive) join property:**
  - **Example –**

Step 1: Check if the relation is in BCNF.

- The functional dependency `A → B` means that `A` determines `B`, but `A` is not a superkey (because `A` doesn't uniquely identify the relation—`A` alone isn't sufficient to identify both `B` and `C`).
- The functional dependency `B → C` means that `B` determines `C`, but `B` also isn't a superkey.

Thus, **the relation is not in BCNF.**

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Decomposition into BCNF with Lossless (non-additive) join property:**
  - **Example –**

Step 2: Decompose into BCNF.

We need to decompose the relation into smaller relations that are in BCNF.

Decompose based on the functional dependency `A → B`:

We create a relation for `A` and `B`:

- R1(A, B) with the functional dependency `A → B`.

We create a second relation for `B` and `C`:

- R2(B, C) with the functional dependency `B → C`.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Decomposition into BCNF with Lossless (non-additive) join property:**
  - **Example –**

Step 3: Check for Lossless Join Property.

To check if the decomposition is **lossless**, we need to ensure that the original relation `R(A, B, C)` can be reconstructed by joining `R1` and `R2` without losing any information.

- In the decomposition, both `R1(A, B)` and `R2(B, C)` share the attribute `B`.
- We can use `B` as the common attribute to join these two relations.

The **Lossless Join Property** holds in this case because we can join `R1` and `R2` on `B`, and the result will give us the original relation `R(A, B, C)` with no loss of information.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Decomposition into BCNF with Lossless (non-additive) join property:**
  - **Example –**

Step 4: Verify that the decomposed relations are in BCNF.

- **R1(A, B)**: The functional dependency `A → B` holds, and `A` is a **superkey** in `R1`. Thus, **R1 is in BCNF**.
- **R2(B, C)**: The functional dependency `B → C` holds, and `B` is a **superkey** in `R2`. Thus, **R2 is in BCNF**.

Result:

The final decomposition is:

- **R1(A, B)** (in BCNF)
- **R2(B, C)** (in BCNF)

This decomposition is in **BCNF** and also has the **Lossless Join Property**, meaning we can reconstruct the original relation `R(A, B, C)` by joining `R1` and `R2`.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Relational Synthesis into 3NF with Dependency Preservation and Lossless (Non-Additive) Join Property**
- **Goal –** To achieve 3NF ; preserve the functional dependencies; and maintain the lossless join property
- **Input –** A universal relation R and a set of functional dependencies F on the attributes of R.
- **Algorithm Steps -**
  1. Find a minimal cover G for F
  2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes {X ∪ {A1} ∪ {A2} ... ∪ {Ak}}, where X → A1, X → A2, ..., X –>Ak are the only dependencies in G with X as left-hand-side (X is the key of this relation).
  3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that form **a key of R**.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Finding a Key K for R Given a set F of Functional Dependencies**
- **Goal –** to identify a candidate key (or keys) for the relation. A candidate key is a minimal set of attributes that can uniquely identify a tuple in the relation, considering the functional dependencies provided.
- **Input –** A universal relation R and a set of functional dependencies F on the attributes of R.
- **Algorithm Steps –**

1. Set K := R.
2. For each attribute A in K {

compute (K - A)+ with respect to F;

If (K - A)+ contains all the attributes in R,

then set K := K - {A}; }

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Finding a Key K for R Given a set F of Functional Dependencies**
- **Example –**

Given:

- Relation $R(A, B, C, D)$
- Functional Dependencies:
  - $A \rightarrow B$
  - $B \rightarrow C$
  - $A \rightarrow D$

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Finding a Key K for R Given a set F of Functional Dependencies**
- **Example –**

Step 1: Find the Closure of Attribute Sets.

We compute the closure of different sets of attributes to see which ones determine all the attributes in the relation R.

- Closure of A:
  - $A^+ = \{A\}$ (start with A)
  - From $A \rightarrow B$, we get B.
  - From $B \rightarrow C$, we get C.
  - From $A \rightarrow D$, we get D.
  - So, $A^+ = \{A, B, C, D\}$. This closure includes all attributes in R.

Since the closure of A includes all attributes in the relation R, A is a **candidate key**.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Finding a Key K for R Given a set F of Functional Dependencies**
- **Example –**

Step 2: Check for Minimality.

- The set A is already minimal (it consists of just one attribute), and removing any attribute would result in a closure that doesn't determine all the attributes. Therefore, A is a **candidate key**.

Step 3: Verify if there are other Candidate Keys.

- Let's check if any other combination of attributes could be a candidate key.
  - For B, C, we have $B^+ = \{B\}$ and from $B \rightarrow C$, we get C, but we don't have A or D, so this is not a candidate key.
  - Similarly, other combinations like B, D or C, D won't be able to determine all attributes in the relation.

Thus, A is the **only candidate key** in this case.

Step 4: Primary Key (if needed).

Since A is the only candidate key, it becomes the **primary key** for the relation R.

ALGORITHMS FOR RELATIONAL DATABASE SCHEMA DESIGN

- **Finding a Key K for R Given a set F of Functional Dependencies**
- **Conclusion –**
  - Compute the closure of different attribute sets to check which sets can determine all attributes in the relation.
  - Identify the minimal set of attributes that determine all attributes — these are the candidate keys.
  - If necessary, choose one candidate key as the primary key.
- The goal is to find a minimal set of attributes (a candidate key) that can uniquely identify a tuple in the relation, ensuring the integrity and efficiency of the database schema.