SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

18CSC209J
Database Management Systems and Cloud Integration Services DBMSCIS

for

IV Sem

B.Tech (CSE - Cloud Computing)

Department of NWC

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

18CSC209J
Database Management Systems and Cloud Integration Services DBMSCIS

Unit IV

Unit IV

- Relational Algebra
  - Fundamental operators and syntax
  - Relational algebra queries, Tuple relational calculus
  - Pitfalls in Relational database, Decomposing bad schema
- Functional dependency
  - definition, trivial and non-trivial FD
  - closure of FD set, closure of attributes, Irreducible set of FD
- Normalization
  - 1NF, 2NF, 3NF
  - Decomposition using FD - dependency preservation
  - BCNF
  - Multi valued dependency, 4NF
  - Join dependency and 5NF

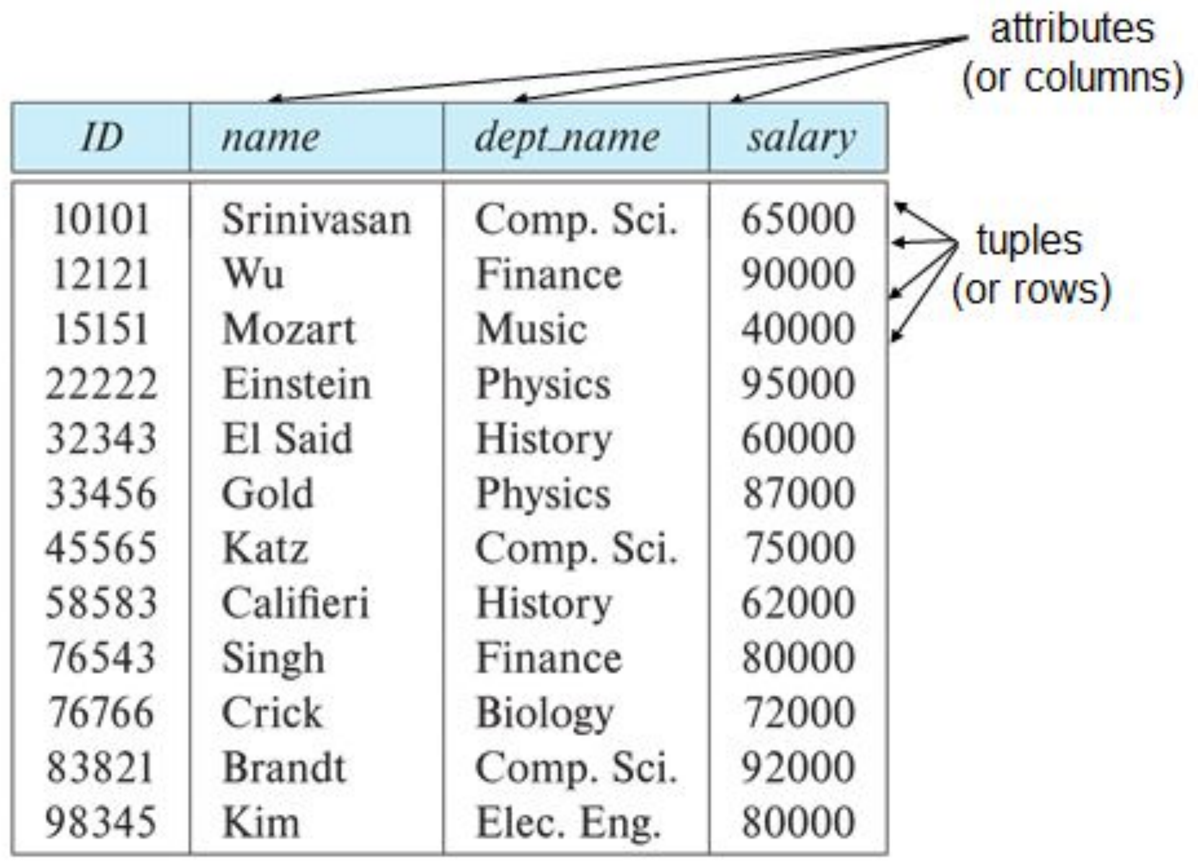
Relational Algebra

INTRODUCTION

- Procedural versus non-procedural, or declarative
- “Pure” languages:
  - **Relational algebra**
  - Tuple relational calculus
  - Domain relational calculus
- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
  - select: σ
  - project: ∏
  - union: ∪
  - set difference: –
  - Cartesian product: x
  - rename: ρ

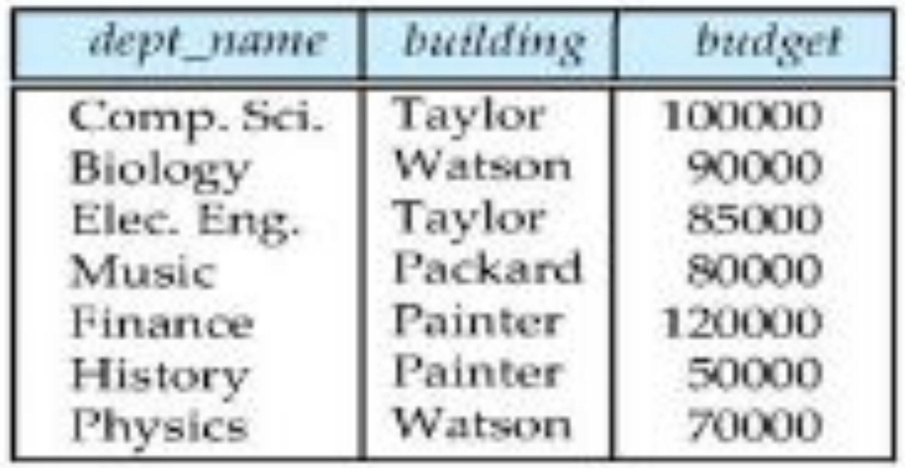
RELATIONAL ALGEBRA

Instructor Table


attributes
(or columns)
tuples
(or rows)

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

Department Table

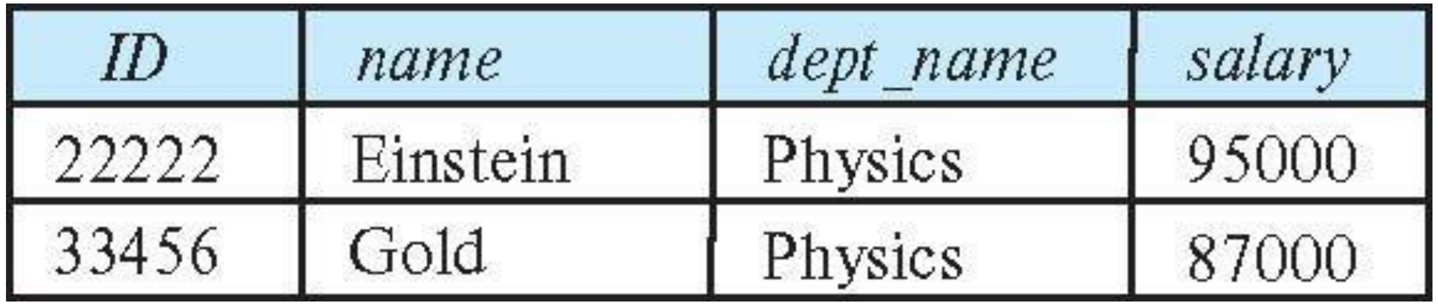


dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

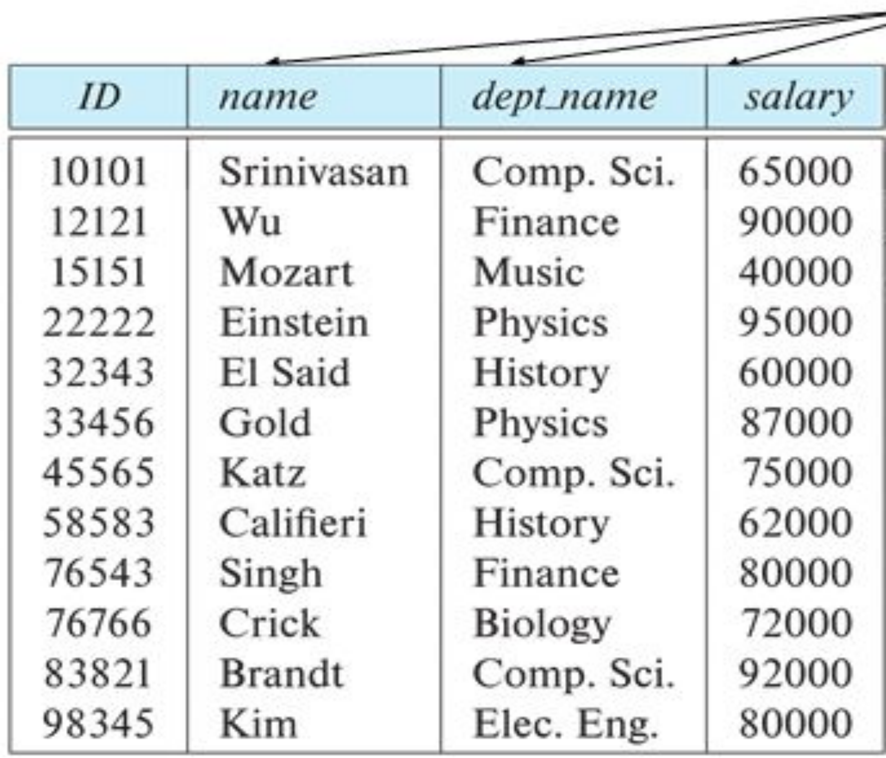
SELECT

- The **select** operation selects tuples that satisfy a given predicate.
- Notation: $\sigma_{p}(r)$
- *p* is called the **selection predicate**
- Example: select those tuples of the *instructor* relation where the instructor is in the "Physics" department.
  - Query

$$\sigma_{dept\_name=\text{“Physics”}}(instructor)$$



ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000



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

SELECT

- We allow comparisons using

  $=, \neq, >, \geq. <. \leq$ in the selection predicate.

- We can combine several predicates into a larger predicate by using the connectives:

  $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)

- Example: Find the instructors in Physics with a salary greater $90,000, we write:

$$\sigma_{dept\_name=\text{“Physics”} \wedge salary > 90{,}000}(instructor)$$

- The select predicate may include comparisons between two attributes.
  - Example, find all departments whose name is the same as their building name:
  - $\sigma_{dept\_name=building}(department)$

PROJECT

- A unary operation that returns its argument relation, with certain attributes left out.
- Notation:

$$\prod_{A1,A2,A3\ \ldots.Ak} (r)$$

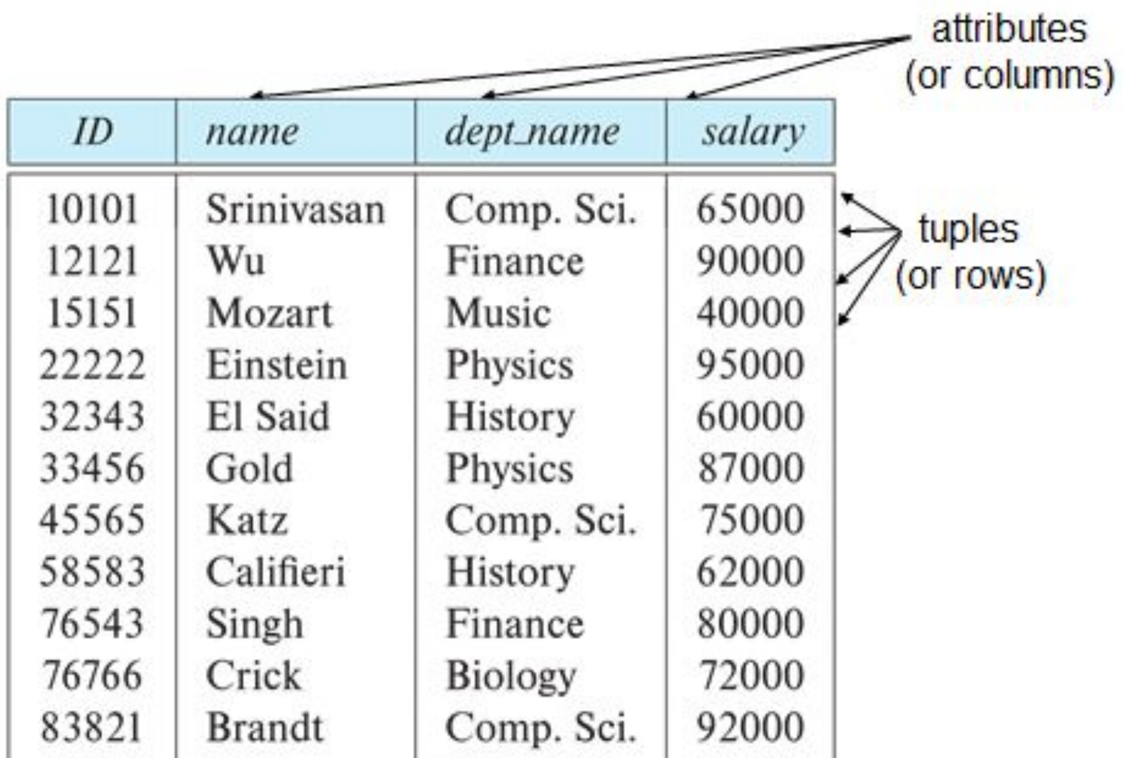
where $A_1, A_2, \ldots, A_k$ are attribute names and r is a relation name.

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets

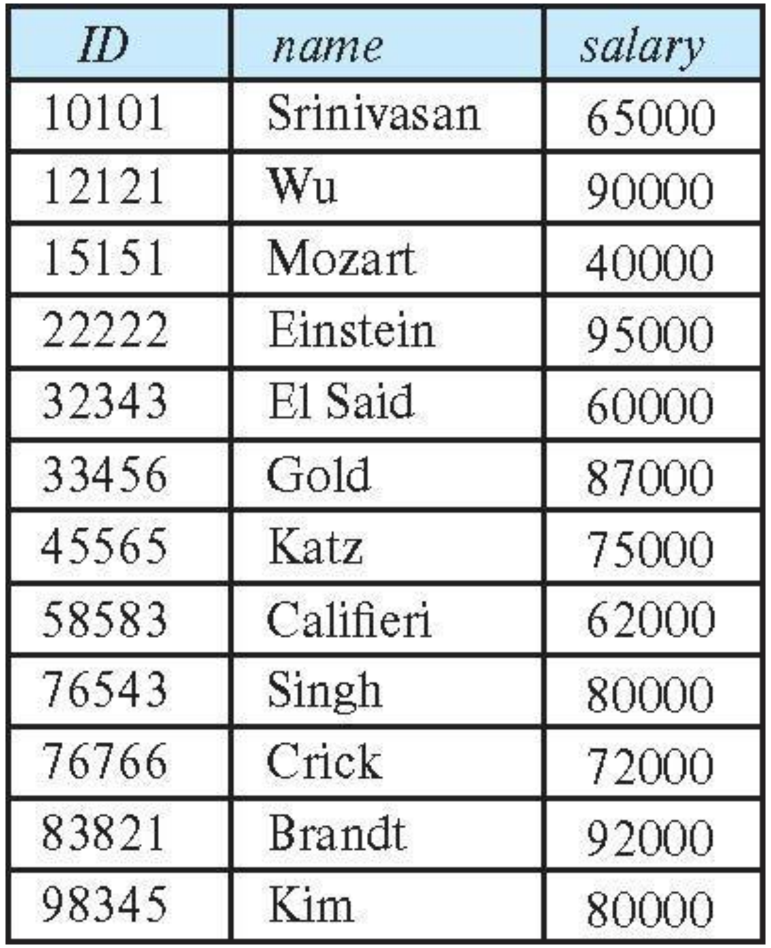
PROJECT

- Eliminate the *dept_name* attribute of *instructor*
- Query*:*

$$\Pi_{ID,\ name,\ salary}\ (instructor)$$


attributes
(or columns)
tuples
(or rows)

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000



ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

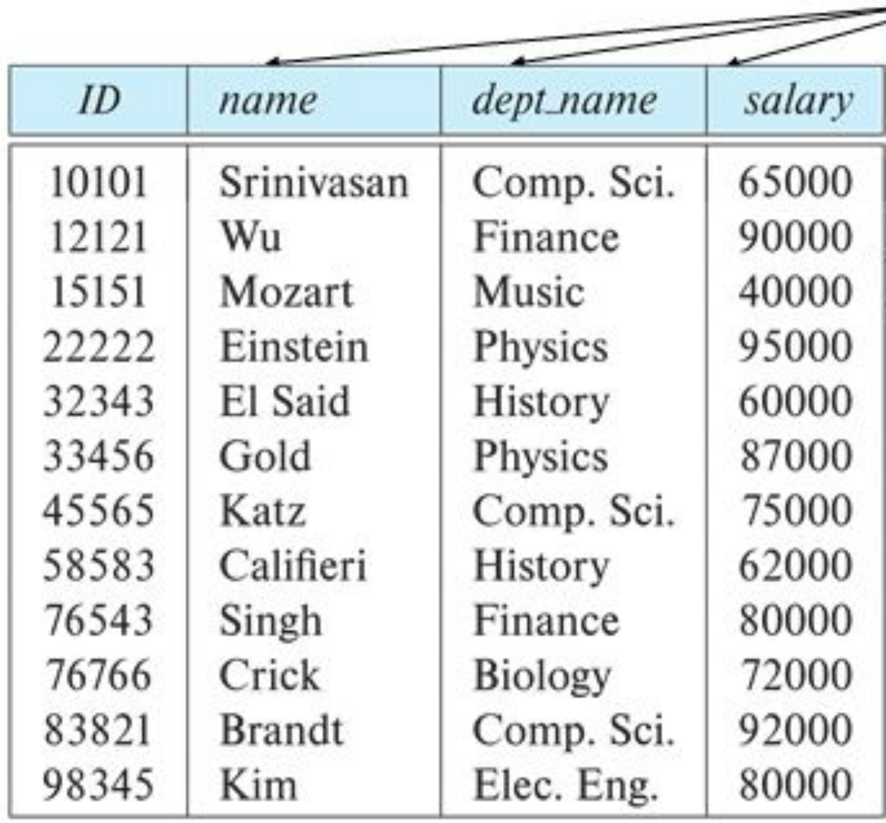
PROJECT

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

- Find the names of all instructors in the Physics department.

```
Select name
from instructor
where dept_name = ‘Physics’ ;
```

- $\Pi_{name}(\sigma_{dept\_name = \text{“Physics”}}(instructor))$



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

CARTESIAN PRODUCT

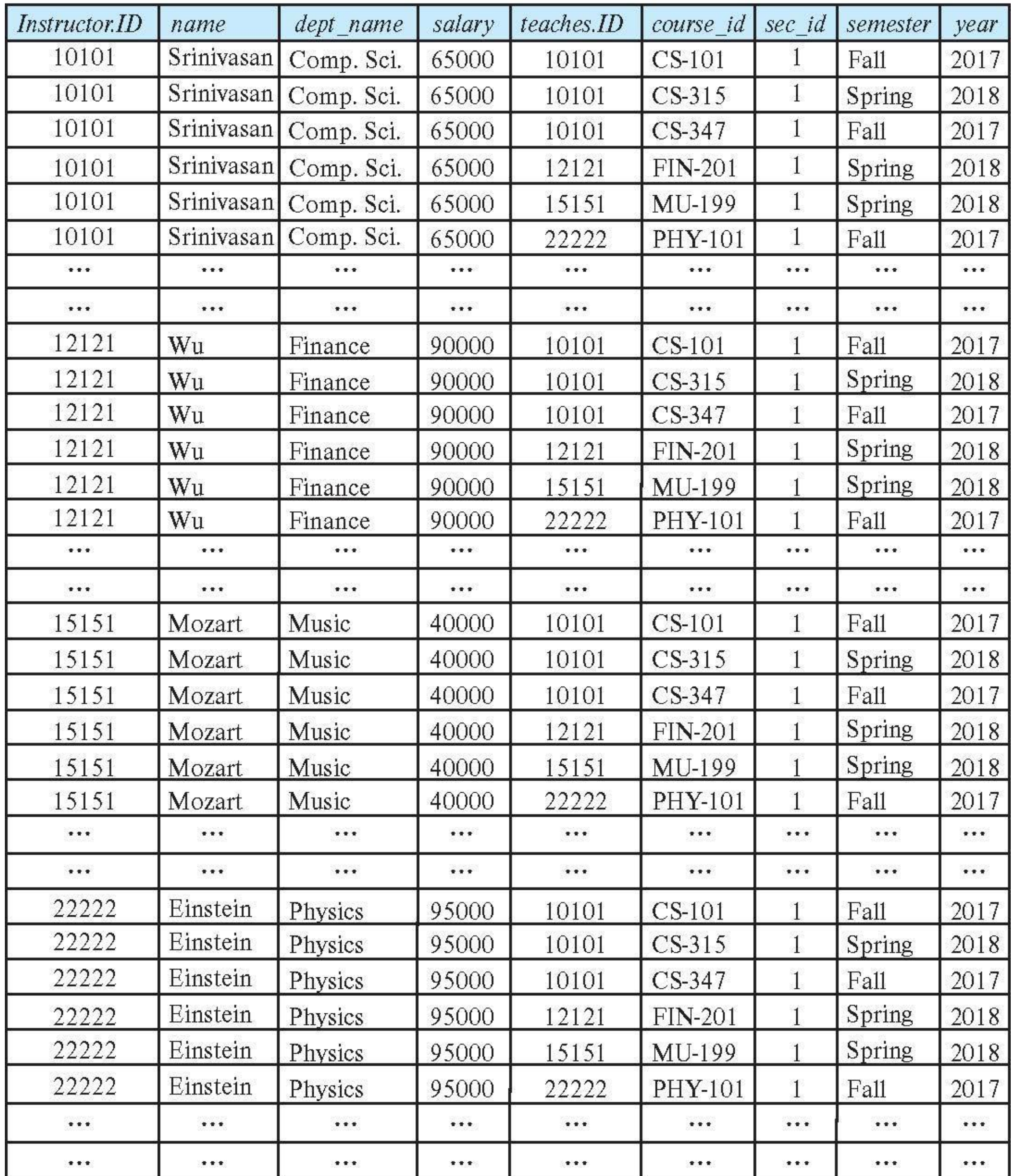
- The Cartesian-product operation (denoted by X) allows us to combine information from any two relations.
- Example: the Cartesian product of the relations *instructor* and *teaches* is written as:

  instructor X *teaches*

- Since the instructor *ID* appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
  - *instructor.ID*
  - *teaches.ID*

CARTESIAN RESULT

instructor X teaches



Instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

CARTESIAN PRODUCT

- The Cartesian-Product

  instructor X *teaches*

  associates every tuple of instructor with every tuple of teaches.

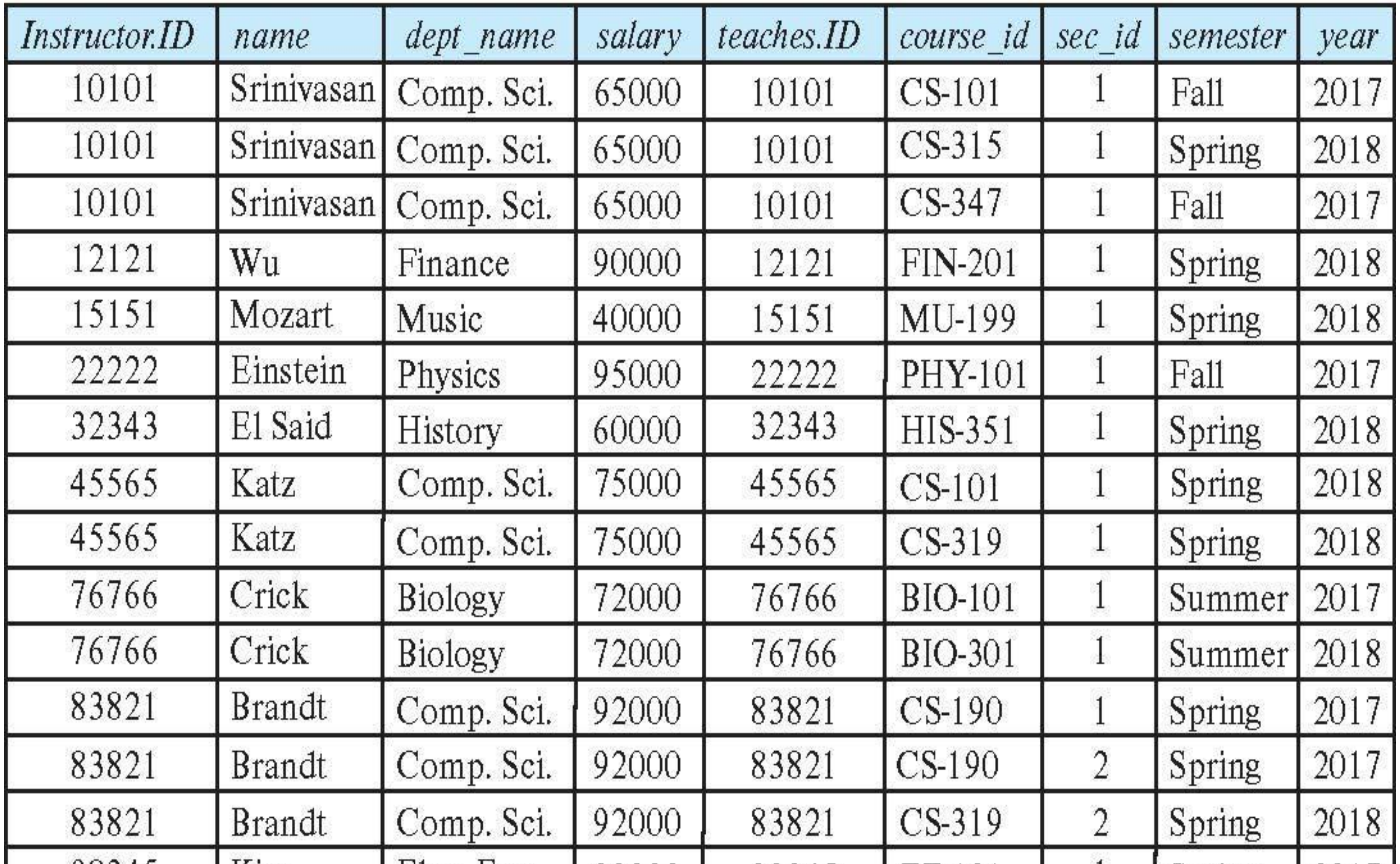
  - Most of the resulting rows have information about instructors who did NOT teach a particular course.

- To get only those tuples of "*instructor* X *teaches* " that pertain to instructors and the courses that they taught, we write:

$$\sigma_{instructor.id = teaches.id}\ (instructor\ \ x\ teaches\ ))$$

- The table corresponding to:

$$\sigma_{\textit{instructor.id} = \textit{teaches.id}}(\textit{instructor} \times \textit{teaches})$$

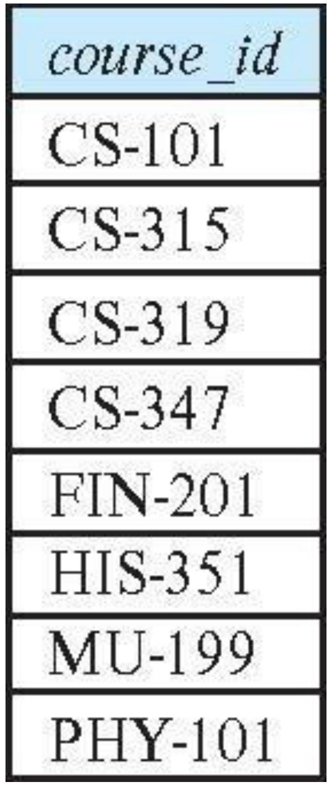


Instructor.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018

UNION

- The union operation allows us to **combine two relations**
- Notation: $r \cup s$
- For $r \cup s$ to be valid.
  1. *r, s* must have the *same* **arity** (same number of attributes)
  2. The attribute domains must be **compatible** (example: 2nd column of r deals with the same type of values as does the 2nd column of s)
- To find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$$\Pi_{course\_id}\,(\sigma_{semester=\text{“Fall”}\ \land\ year=2017}\,(section))\ \cup$$
$$\Pi_{course\_id}\,(\sigma_{semester=\text{“Spring”}\ \land\ year=2018}\,(section))$$



course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

INTERSECTION

- The set-intersection operation allows us to **find tuples that are in both the input relations.**
- Notation: ***r*** $\cap$ ***s***
- Assume:
    - ***r, s* have the *same arity***
    - **attributes of *r* and *s* are compatible**
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

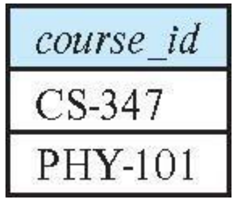
$$\Pi_{course\_id} (\sigma_{semester=\text{“Fall”} \ \land \ year=2017} (section)) \cap$$
$$\Pi_{course\_id} (\sigma_{semester=\text{“Spring”} \ \land \ year=2018} (section))$$

course_id
CS-101

SET DIFFERENCE

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation ***r* – *s***
- Set differences must be taken between **compatible** relations.
  - *r* and *s* must have the same arity
  - attribute domains of *r* and *s* must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{course\_id}\,(\sigma_{\,semester=\text{“Fall”}\ \wedge\ year=2017}\,(section))\ -$$
$$\Pi_{course\_id}\,(\sigma_{\,semester=\text{“Spring”}\ \wedge\ year=2018}\,(section))$$



course_id
CS-347
PHY-101

ASSIGNMENT

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by ← and works like assignment in a programming language.
- Find all instructor in the "Physics" and Music department.

$$Physics \leftarrow \sigma_{dept\_name=\text{“Physics”}}(instructor)$$
$$Music \leftarrow \sigma_{dept\_name=\text{“Music”}}(instructor)$$
$$Physics \cup Music$$

RENAME

- The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, ρ , is provided for that purpose
- The expression:

$$\rho_x (E)$$

returns the result of expression *E* under the name *x*

- Another form of the rename operation:

$$\rho_{x(A1,A2,\,..\,An)} (E)$$

INTRODUCTION

- Procedural versus non-procedural, or declarative
- “Pure” languages:
  - Relational algebra
  - **Tuple relational calculus**
  - Domain relational calculus

TUPLE RELATIONAL CALCULUS

- A nonprocedural query language, where each query is of the form

  $\{t \mid P(t)\}$
- It is the set of all tuples t such that predicate P is true for t
- t is a tuple variable
- t [A] denotes the value of tuple t on attribute A
- $t \in r$ denotes that tuple t is in relation r
- P is a formula similar to that of the predicate calculus

Symbols

- Set of attributes and constants
- Set of comparison operators: (e.g., <, ≤, =, ≠, >, ≥)
- Set of connectives: and (∧), or (v), not (¬)
- Implication (⇒): x ⇒ y, if x if true, then y is true

  $x \Rightarrow y \equiv \neg x \vee y$
- Set of quantifiers:
  - $\exists\ t \in r\,(Q\,(t\,))$ ≡ ”there exists” a tuple in *t* in relation *r* such that predicate *Q* (*t*) is true
  - $\forall\, t \in r\,(Q\,(t\,))$ ≡ *Q* is true “for all” tuples *t* in relation *r*

Examples

- Find the *ID, name, dept_name, salary* for instructors whose salary is greater than $80,000

  $\{t \mid t \in instructor \land t[salary] > 80000\}$

- Only the ID attribute value

  $\{t \mid \exists\, s \in \text{instructor}\ (t[ID] = s[ID] \land s[salary] > 80000)\}$

Examples

- Find the names of all instructors whose department is in the Watson building

$$\{t \mid \exists s \in \mathit{instructor}\ (t[\mathit{name}] = s[\mathit{name}]$$
$$\wedge\ \exists u \in \mathit{department}\ (u[\mathit{dept\_name}] = s[\mathit{dept\_name}]\text{ “}$$
$$\wedge\ u[\mathit{building}] = \text{“Watson”}\ ))\}$$

- Find the set of all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or both

$$\{t \mid \exists s \in \mathit{section}\ (t[\mathit{course\_id}] = s[\mathit{course\_id}] \wedge$$
$$s[\mathit{semester}] = \text{“Fall”} \wedge s[\text{year}] = 2009$$
$$\vee\ \exists u \in \mathit{section}\ (t[\mathit{course\_id}] = u[\mathit{course\_id}] \wedge$$
$$u[\mathit{semester}] = \text{“Spring”} \wedge u[\text{year}] = 2010\ )\}$$

Pitfalls in Relational Database

Introduction

- Relational databases are widely used in modern software applications for storing and managing data.
- However, designing a good schema that represents the data accurately and efficiently is a challenging task.
- In this presentation, we will explore the pitfalls in relational database design, and discuss how to decompose a bad schema and understand functional dependencies to improve it.

Pitfalls in Relational Database Design

Redundancy:

Storing the same data in multiple places can lead to inconsistencies and inefficiencies.

Inconsistency:

Data inconsistencies can occur when different parts of the database hold different versions of the same data.

Inefficiency:

Poor database design can result in slower query performance and increased storage requirements.

Complexity:

A complex database schema can be difficult to understand and maintain, leading to errors and inefficiencies.

Decomposing a Bad Schema

What is a bad schema?

- A bad schema is a database schema that suffers from common design problems that can lead to issues such as data redundancy, inconsistency, and inefficiency.
- Some common examples of bad schema design include tables with duplicate data, tables with many null values, and tables with inappropriate data types.
- A bad schema can make it difficult to manage data, slow down queries, and cause data integrity issues.
- Therefore, it is essential to design a good schema that accurately represents the data, is efficient, and minimizes redundancy and inconsistencies.

Decomposing a Bad Schema

Common problems in a bad schema:

- Redundancy
- Inconsistency
- Inefficiency

Decomposition process:

- Identify functional dependencies
- Normalize the schema using normalization rules
- Check for anomalies
- Evaluate trade-offs

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Functional Dependency

Functional Dependency

- Functional dependency (FD) is a constraint that specifies the relationship between two attributes in a database table.
- It states that **one attribute** (the dependent attribute) is **functionally dependent on another attribute** (the determinant attribute)
- In other words, the value of the dependent attribute is uniquely determined by the value of the determinant attribute.
- For example, in a database table that stores information about employees, the attribute "employee name" is functionally dependent on the attribute "employee ID", since each employee has a unique ID and a unique name.

Decomposition using Functional Dependency

- Decomposition is the process of splitting a single database table into multiple tables to improve its design and eliminate redundancy.

- In the context of functional dependencies, decomposition involves splitting a table into smaller tables that satisfy certain functional dependencies.

Dependency Preservation

- Dependency preservation is the property that ensures that a decomposition of a table into smaller tables preserves the functional dependencies that existed in the original table.

- In other words, if a functional dependency existed between two attributes in the original table, it should also exist in the smaller tables after decomposition.

Example

- Suppose we have a database table called "employees" with the following attributes:

- employees (employee_id, employee_name, department, salary)

Let's say that the following functional dependencies hold in this table:

employee_id -> employee_name, department, salary

department -> salary

Example

- ✔ In other words, the employee ID uniquely determines the employee name, department, and salary, and the department uniquely determines the salary.
- ✔ However, this table is not in a good design, because there is redundancy.
- ✔ For example, if an employee changes departments, we would need to update multiple rows in the table to reflect this change.

Example

- ✔ To eliminate this redundancy, we can decompose the table into two smaller tables, as follows:

- employees1(employee_id, employee_name, department)

- employees2(department, salary)

Example

- ✔ In this new design, each table has fewer attributes and there is no redundancy.
- ✔ However, we need to ensure that the functional dependencies still hold in the smaller tables.
- ✔ We can check this by computing the closure of the functional dependencies in each table:
- ✔ employees1 = {employee_id -> employee_name, department, salary}
- ✔ employees2 = {department -> salary}

Example

- ✔ The closure of each table includes the functional dependencies that hold in that table.
- ✔ We can see that the functional dependencies that held in the original table are preserved in the decomposition, and so the decomposition satisfies dependency preservation.

Understanding Functional Dependencies

Definition:

- Functional dependencies (FDs) are a fundamental concept in **database design that describes the relationship between attributes or columns in a table.**
- In a relational database, an **FD is a constraint that specifies that the value of one or more attributes uniquely determines the value of another attribute.**
- More specifically, let R be a relation schema with attributes A and B, where A and B are subsets of the set of attributes of R.
- We say that A functionally determines B, denoted $A \rightarrow B$, if for any two tuples t1 and t2 in R that agree on all attributes in A, they must also agree on all attributes in B.

Example:

- Suppose we have a relation called "students" with attributes "student_id," "first_name," and "last_name."
- If we know a student's "student_id," we can uniquely determine their "first_name" and "last_name."
- Thus, we can say that "student_id" → ("first_name", "last_name") is a functional dependency in the "students" relation.
- Functional dependencies are essential in database design because they help to ensure data integrity and prevent data anomalies.
- They also aid in the normalization process by identifying redundant data and helping to eliminate it from the schema.

Trivial vs Non Trivial FD

- The difference between trivial and non-trivial functional dependencies (FDs) lies in their **implications for the data in a database.**
- A **trivial FD** is a functional dependency where the dependent attribute is already determined by the determinant attribute(s) and does not provide any new information.
- In other words, a trivial FD always holds true and is not useful for data analysis or normalization.
- For example, in a table with attributes A, B, and C, the FD A → A is trivial because it simply means that the value of attribute A determines itself, which is already a given.
- Another example of a trivial FD is A, B → A, which states that knowing the values of A and B allows us to determine the value of A, which is again trivial.

Non Trivial FD

- In contrast, a non-trivial FD is a functional dependency where the dependent attribute is not determined by the determinant attribute(s) alone, and thus provides new information about the data.
- Non-trivial FDs are useful for analyzing and optimizing a database schema.
- For example, in a table with attributes "employee_id," "employee_name," and "employee_department," the FD "employee_id" → "employee_name" is non-trivial because knowing the "employee_id" determines the "employee_name," and this information is not already given.
- Similarly, in the same table,
- the FD "employee_id" → "employee_department" is also non-trivial.

Rules for determining FDs:

- Armstrong's Axioms
- Closure of Attributes

Benefits of understanding FDs:

- Helps to identify redundant data
- Simplifies schema design
- Improves data consistency

Armstrong’s axioms:

Armstrong's Axioms are a set of rules used to derive all the functional dependencies (FDs) that hold in a given database relation.

There are three axioms in Armstrong's Axioms:

1. Reflexivity: If A is a set of attributes in a relation R, then A → A holds true. This means that every attribute set is functionally dependent on itself.
2. Augmentation: If A → B is a functional dependency in relation R, then for any set of attributes C, A C → B C holds true. This means that if we can determine B from A, we can also determine B from any superset of A.
3. Transitivity: If A → B and B → C are functional dependencies in relation R, then A → C holds true. This means that if we can determine B from A and C from B, we can also determine C from A.

Using these axioms, we can derive all the other functional dependencies that hold in a given relation.

For example, given the FDs $A \rightarrow B$ and $B \rightarrow C$, we can use the transitivity axiom to derive the FD $A \rightarrow C$.

This process is called closure computation and is an essential step in database normalization.

Closure of an attribute:

The closure of attributes is a concept in database design that refers to the complete set of attributes that can be functionally determined by a given set of attributes in a relation.

The closure of attributes is important in database normalization because it helps to identify all the functional dependencies that hold in a relation and to eliminate redundancy and anomalies.

Formally, given a relation R and a set of attributes X in R, the closure of X, denoted by X+, is the set of all attributes in R that are functionally dependent on X. More specifically, X+ contains all the attributes that can be determined by X through a combination of the three axioms in Armstrong's Axioms: reflexivity, augmentation, and transitivity.

For example, consider a relation with attributes A, B, C, and D and the following functional dependencies:

- A → B
- B → C
- C → D
- To find the closure of the attribute set {A}, the axioms are used to derive all the other attributes that are functionally dependent on {A}.

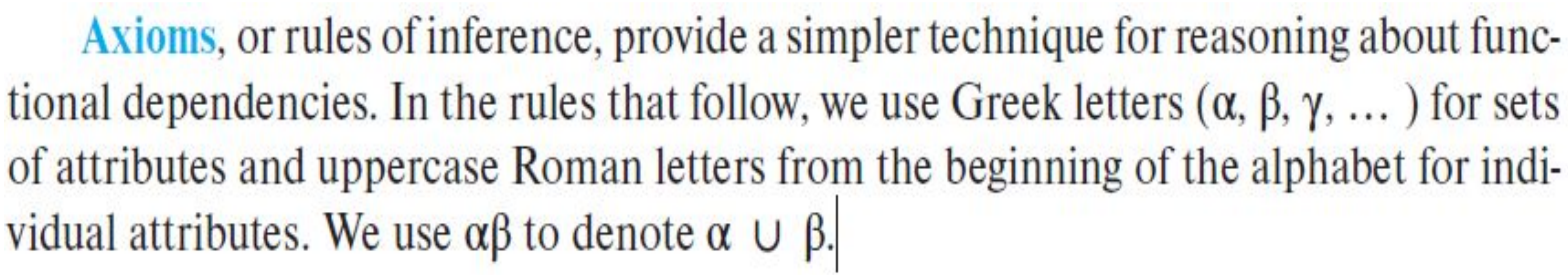
Using reflexivity, we know that A → A holds true, so we can augment {A} to get {A, B}. Then, using transitivity, we know that {A, B} → C holds true, so we can augment {A, B} to get {A, B, C}.

Finally, using transitivity again, we know that {A, B, C} → D holds true, so we can augment {A, B, C} to get the closure of {A}, which is {A, B, C, D}.

In this example, we have identified all the functional dependencies that hold in the relation and eliminated any redundancy or anomalies by normalizing the relation.

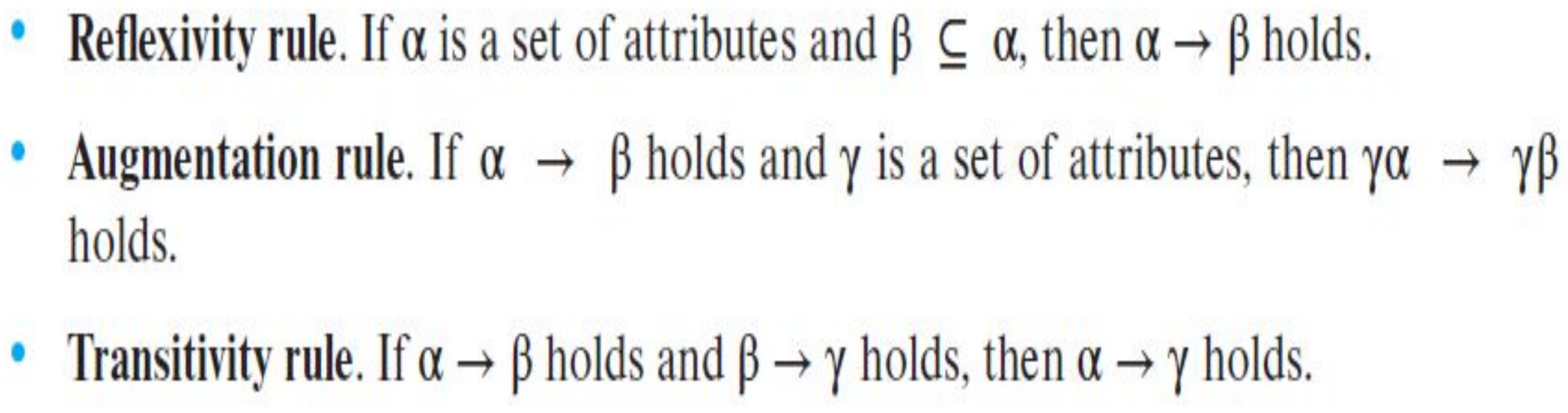
Closure of FD set

- Given a set F of functional dependencies on a schema, we can prove that certain other functional dependencies also hold on the schema. We say that such functional dependencies are "logically implied" by F.
- Let 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. Given F, we can compute F+ directly from the formal definition of functional dependency.



Axioms, or rules of inference, provide a simpler technique for reasoning about functional dependencies. In the rules that follow, we use Greek letters ($\alpha, \beta, \gamma, \ldots$) for sets of attributes and uppercase Roman letters from the beginning of the alphabet for individual attributes. We use $\alpha\beta$ to denote $\alpha \cup \beta$.

- We can use the following three rules 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 in honor of the person who first proposed it.



- **Reflexivity rule.** If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule.** If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule.** If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

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

ADDITIONAL RULES

- **Union rule**. If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule**. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- **Pseudotransitivity rule**. If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

PROCEDURE TO COMPUTE F+

```
$F+ = F$
apply the reflexivity rule /* Generates all trivial dependencies */
repeat
for each functional dependency $f$ in $F+$
apply the augmentation rule on $f$
add the resulting functional dependencies to $F+$
for each pair of functional dependencies $f1$ and $f2$ in $F+$
if $f1$ and $f2$ can be combined using transitivity
add the resulting functional dependency to $F+$
until $F+$ does not change any further
```

EXAMPLE

For relation $R = (A, B, C, G, H, I)$ and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. Find out the closure of functional dependency.

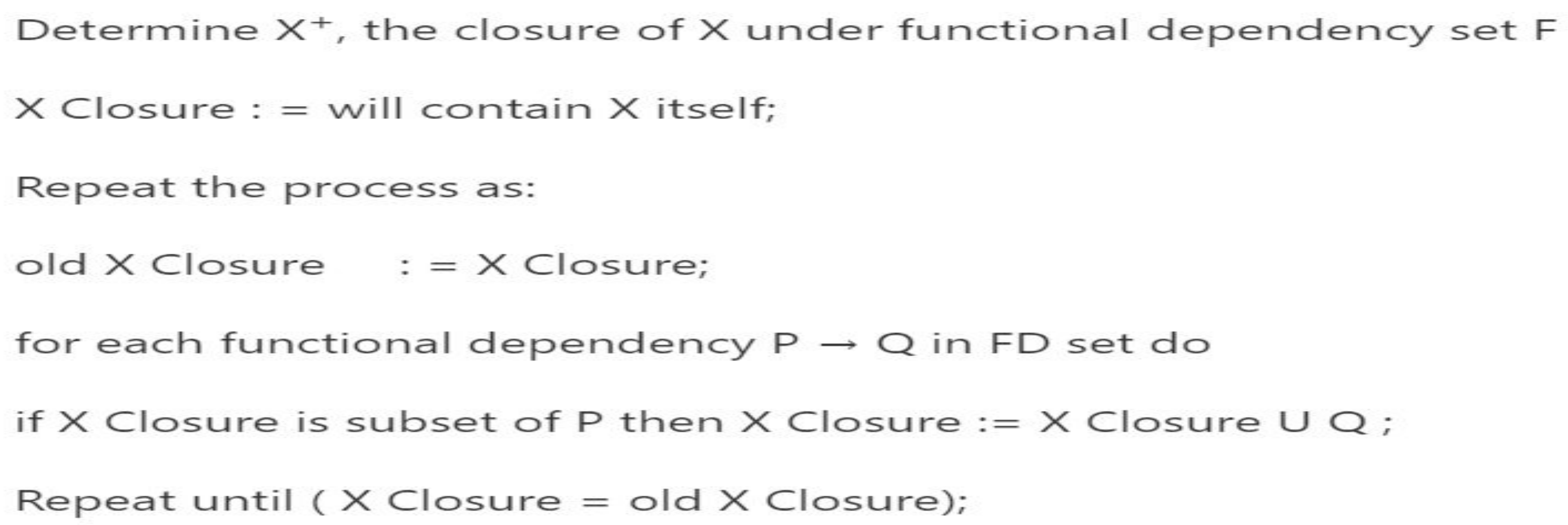
- $A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule.
- $CG \rightarrow HI$. Since $CG \rightarrow H$ and $CG \rightarrow I$, the union rule implies that $CG \rightarrow HI$.
- $AG \rightarrow I$. Since $A \rightarrow C$ and $CG \rightarrow I$, the pseudo transitivity rule implies that $AG \rightarrow I$ holds.

- Another way of finding that $AG \rightarrow I$ holds is as follows: We use the augmentation rule on $A \rightarrow C$ to infer $AG \rightarrow CG$. Applying the transitivity rule to this dependency and $CG \rightarrow I$, we infer $AG \rightarrow I$.

Closure of attributes

- Closure of an Attribute can be defined as a set of attributes that can be functionally determined from it.
- Closure of a set of attributes X concerning F is the set X+ of all attributes that are functionally determined by X

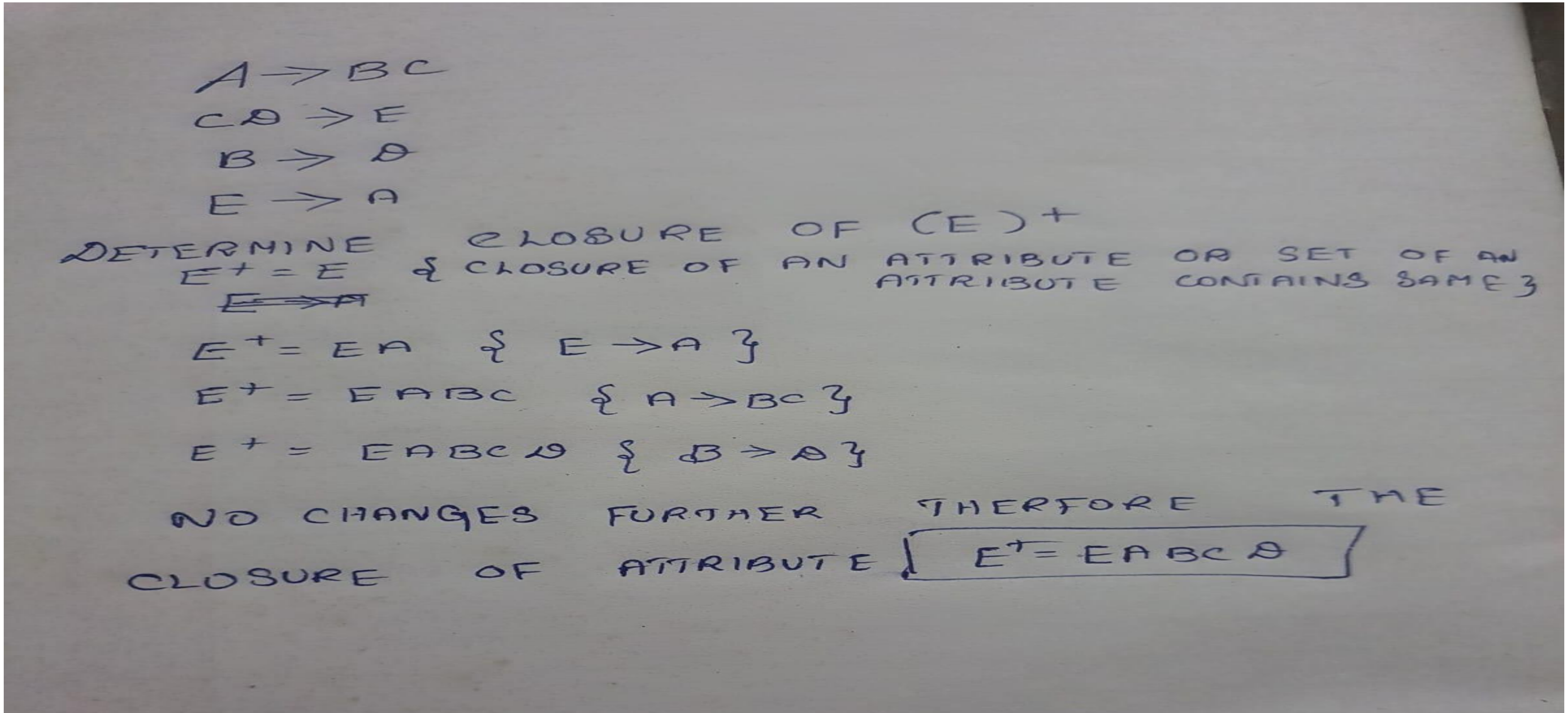
PSEUDOCODE TO FIND THE CLOSURE OF ATTRIBUTE



```
Determine $X^+$, the closure of X under functional dependency set F
X Closure : = will contain X itself;
Repeat the process as:
old X Closure      : = X Closure;
for each functional dependency P → Q in FD set do
if X Closure is subset of P then X Closure := X Closure U Q ;
Repeat until ( X Closure = old X Closure);
```

EXAMPLE

- Given Relational schema R(A,B,C,D,E) with the given FDs find the closure of an attribute ‘E’.



$A \rightarrow BC$

$CD \rightarrow E$

$B \rightarrow D$

$E \rightarrow A$

DETERMINE CLOSURE OF $(E)^+$

$E^+ = E$ { CLOSURE OF AN ATTRIBUTE OR SET OF AN ATTRIBUTE E CONTAINS SAME }

~~$E \rightarrow A$~~

$E^+ = EA$ { $E \rightarrow A$ }

$E^+ = EABC$ { $A \rightarrow BC$ }

$E^+ = EABCD$ { $B \rightarrow D$ }

NO CHANGES FURTHER THERFORE THE CLOSURE OF ATTRIBUTE $\boxed{E^+ = EABCD}$

Irreducible a set of functional dependencies (or)Canonical Cover

- A canonical cover or irreducible a set of functional dependencies FD is a simplified set of FD that has a similar closure as the original set FD.

Extraneous attributes

- An attribute of an FD is said to be extraneous if we can remove it without changing the closure of the set of FD.

PROCEDURE TO COMPUTE CANONICAL COVER

```
$Fc = F$
repeat
Use the union rule to replace any dependencies in $Fc$ of the form
$\alpha1 \rightarrow \beta1$ and $\alpha1 \rightarrow \beta2$ with $\alpha1 \rightarrow \beta1\ \beta2$.
Find a functional dependency $\alpha \rightarrow \beta$ in $Fc$ with an extraneous
attribute either in $\alpha$ or in $\beta$.
/* Note: the test for extraneous attributes is done using $Fc$, not $F$ */
If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in $Fc$.
until ($Fc$ does not change)
```

- 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*. Furthermore, *Fc* must have the following properties:
- No functional dependency in *Fc* contains an extraneous attribute.
- Each left side of a functional dependency in *Fc* is unique. That is, there are no two dependencies $\alpha1 \rightarrow \beta1$ and $\alpha2 \rightarrow \beta2$ in *Fc* such that $\alpha1 = \alpha2$

Normalization

Normalization

- ✔ Database Normalization is the technique of organizing data into more than one table in the database.
- ✔ It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristic like insertion, updation and deletion anomalies.
- ✔ It is a multi-step process that puts data into tabular form, removing duplicated data from the relation tables.

Two Main Reasons

- ✔ It eliminate redundancy data
- ✔ It ensure data dependency makes sense.

Problem without Normalization

- ✔ Without normalization it becomes difficult to handle and update database without facing data loss.

- ✔ Insertion, updation and deletion anomalies are very frequent if the database is not normalized.

Anomalies

- ✔ Insertion Anomaly: Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.

- ✔ Deletion Anomaly: The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.

- ✔ Updation Anomaly: The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

S_id	S_name	S_address	Subject
401	Adam	Noida	Biology
402	Alex	Panipat	Maths
403	Stuart	Jammu	Maths
404	Adam	Noida	Physics

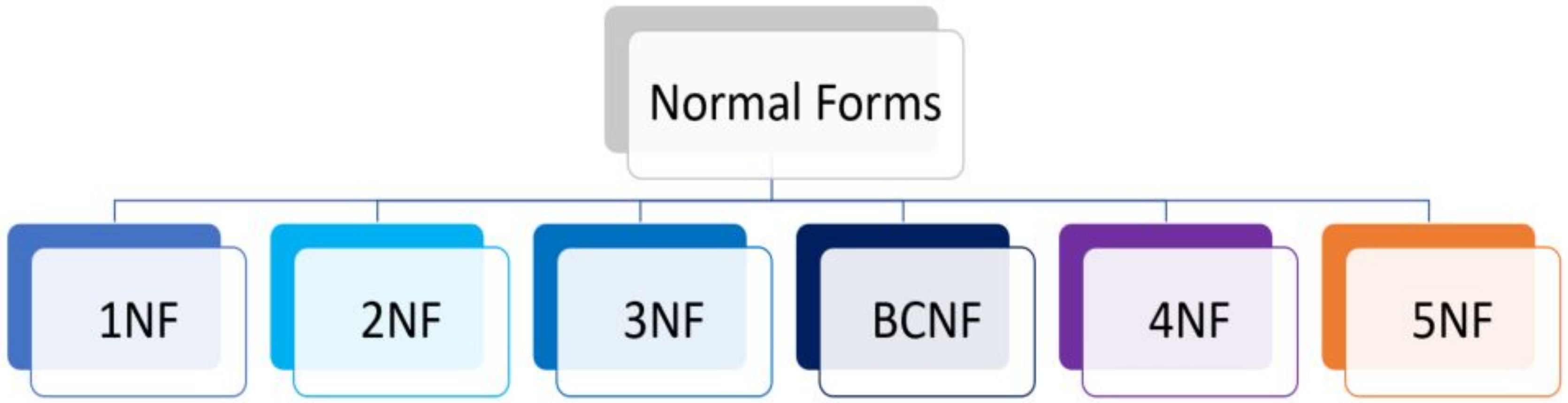
- ✔ Updation Anomaly: To update address of student to occurs twice (or) more than twice in a table we have to update address column in all the rows else the data will come inconsistent.

- ✔ Insertion Anomaly: Suppose for a student new admission, we have s_id, s_name, s_address of a student but the student has not opted for any subject then we have to insert null for the subject which leads to insertion anomalies.

- ✔ Deletion Anomaly: If s_id = 401 has only one subject and temporarily drop it when we delete that row entire student record will be deleted along with it.

NORMAL FORMS

- ✔ The term Normalization comes from the concept of normal form which describe how to organize the data in the database.
- ✔ Normal Forms were developed around the concept of table based on relational database.



Types

- ✔ 1NF (First Normal Form)
- ✔ 2NF (Second Normal Form)
- ✔ 3NF (Third Normal Form)
- ✔ BCNF (Boyce-Codd Normal Form)
- ✔ 4NF (Fourth Normal Form)
- ✔ 5NF (Fifth Normal Form)

UN-NORMALIZED FORM

- ✔ If a table contains one or more repeating groups it is called Un-Normalized form.

Course	Content
Programming	C, Java, Python
Web	HTML, ASP, PHP

FIRST NORMAL FORM (1NF)

- ✔ A relation is said to be in First Normal Form
  - ✔ If and only if all the attributes of relation are atomic (Single valued) in nature.
  - ✔ Must not contain any multi valued (or) composite attributes.

Course	Content
Programming	C, Java, Python
Web	HTML, ASP, PHP

- ✔ This relation consists of multi-values. Hence it is not in first normal form.
- ✔ The multi-values are eliminated and are written separately for each name (i.e.) multi-values are written atomic.
- ✔ The table look like, after normalizing,

Course	Content
Programming	C, Java, Python
Web	HTML, ASP, PHP

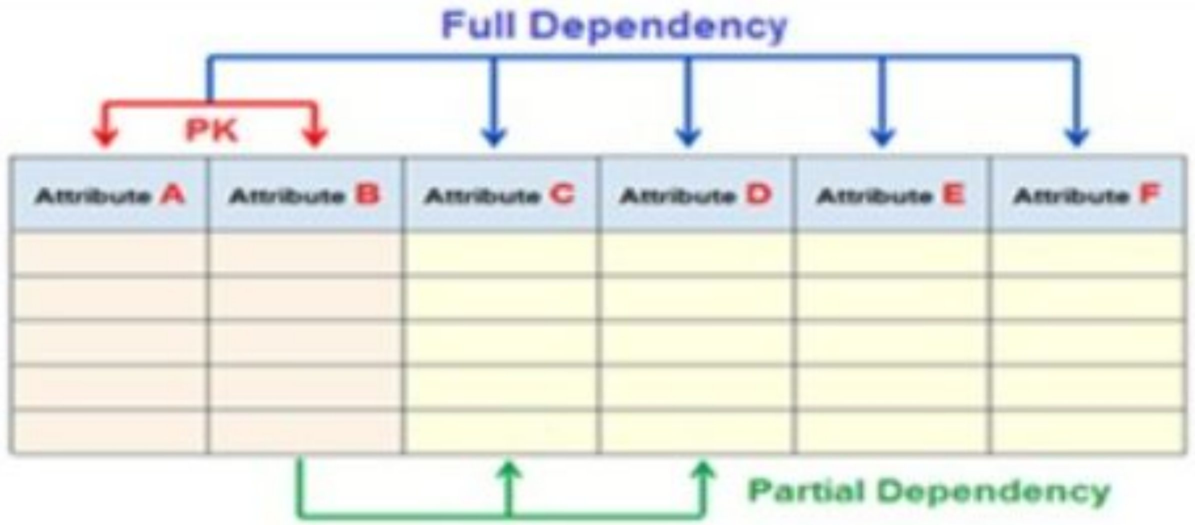
$\rightarrow$

Course	Content
Programming	C
Programming	Java
Programming	Python
Web	HTML
Web	ASP
Web	PHP

- ✔ This 1NF allows data redundancy and there will be many column with same data in multiple rows but each row as a whole will be unique.
- ✔ It is used in most small to medium application.

SECOND NORMAL FORM (2NF)

- ✔ A relation is said to be in Second Normal Form,
  - ✔ If and only if it is already in First Normal Form.
  - ✔ There exist no Partial Dependency.
- ✔ Partial Dependency:
  - ✔ It means that a non-prime attribute is functionally dependent on part of a prime attribute.
- ✔ Functional Dependency:
  - ✔ All the non key attributes are fully functional depends on key attribute.



AB ⟶ Primary Key (Composite Key)

Attributes A & B ⟶ Primary Key Attributes

Attributes CDEF ⟶ Non Key Attributes

AB ⟶ C D E F - Full Dependency

B ⟶ C D - Partial Dependency

To eliminate Partial Dependency

- ✔ Create a new relation for each primary key attributed that is a determinant in partial dependency.
- ✔ Move non-key attributes that are depend on primary key attributes from old relation to new relation.

Emp_id	E_name	Month	Sales	B_id	B_name
E1	AA	Jan	1000	B1	SBI
E1	AA	Feb	1200	B1	SBI
E2	BB	Jan	2200	B2	UBI
E2	BB	Feb	2500	B2	UBI
E3	CC	Jan	1700	B1	SBI
E3	CC	Feb	1800	B1	SBI

- ✔ In the example, Emp_id is a primary key.
- ✔ All the non-prime attributes depends on the Emp_id.

- ✔ But B_name depends on B_id which is not in 2NF.
- ✔ So this table is in 1NF but not in 2NF.

- ✔ After removing the position into another relation store the lesser amount of data in two relations without the loss of data.

Example Table

Emp_id	E_name	Month	Sales	B_id	B_name
E1	AA	Jan	1000	B1	SBI
E1	AA	Feb	1200	B1	SBI
E2	BB	Jan	2200	B2	UBI
E2	BB	Feb	2500	B2	UBI
E3	CC	Jan	1700	B1	SBI
E3	CC	Feb	1800	B1	SBI

Table 1

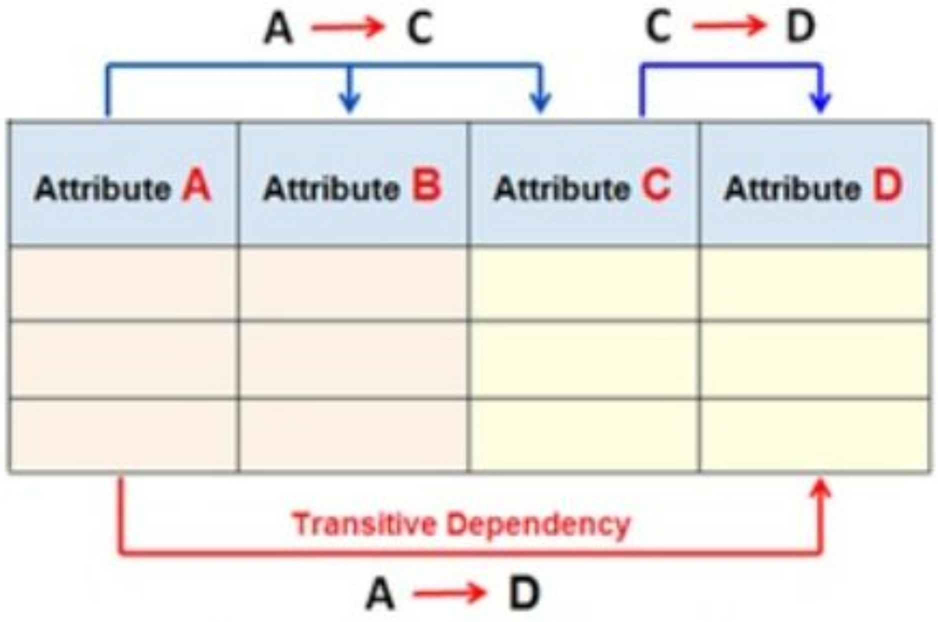
Emp_id	E_name	Month	Sales	B_id
E1	AA	Jan	1000	B1
E1	AA	Feb	1200	B1
E2	BB	Jan	2200	B2
E2	BB	Feb	2500	B2
E3	CC	Jan	1700	B1
E3	CC	Feb	1800	B1

Table 2

B_id	B_name
B1	SBI
B2	UBI

THIRD NORMAL FORM (3NF)

- ✔ A relation is said to be in Third Normal Form if and only if,
  - ✔ It is already in Second Normal Form.
  - ✔ There exist no Transitive Dependency.



Transitive Dependency

A->B (B depends on A)

B->C

A->C

✔ The derived dependency is called transitive dependency when such dependency becomes improbable (i.e.) non-prime attribute depends on another non-prime attribute.

Eg:

Rollno->marks

Marks->grade

Rollno->grade

To eliminate Transitive Dependency

- ✔ For each non-key attribute that is determinant in relation, create a new relation that attribute becomes a primary key in new relation.

- ✔ Move all of the attribute that are functionally dependent on attribute from old relation into new relation.

- ✔ Leave the attribute (which serve as primary key in new relation) in old relation to serve as a foreign key.

S_id	S_name	City	Pincode
1	Shiva	Chennai	15
2	Peter	Pondy	17
3	Meera	Villupuram	20
4	Rohan	Delhi	01
5	Shakthi	Bangalore	18

Example Table

S_id	S_name	City	Pincode
1	Shiva	Chennai	15
2	Peter	Pondy	17
3	Meera	Villupuram	20
4	Rohan	Delhi	01
5	Shakthi	Bangalore	18

Table 1

S_id	S_name	Pincode
1	Shiva	15
2	Peter	17
3	Meera	20
4	Rohan	01
5	Shakthi	18

Table 2

Pincode	City
15	Chennai
17	Pondy
20	Villupuram
01	Delhi
18	Bangalore

Fourth Normal Form (4NF)

Functional dependencies can help us detect poor E-R design

multivalued dependency arises from one of the following sources:

- A many-to-many relationship set.
- A multivalued attribute of an entity set.

A relation schema (R) is in fourth normal form(4NF)with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies in D of the form A→→B, where A⊆R and B ⊆R, at least one of the following holds:

- **A→→B is a trivial multivalued dependency.**
- **A is a super key for R.**

4NF

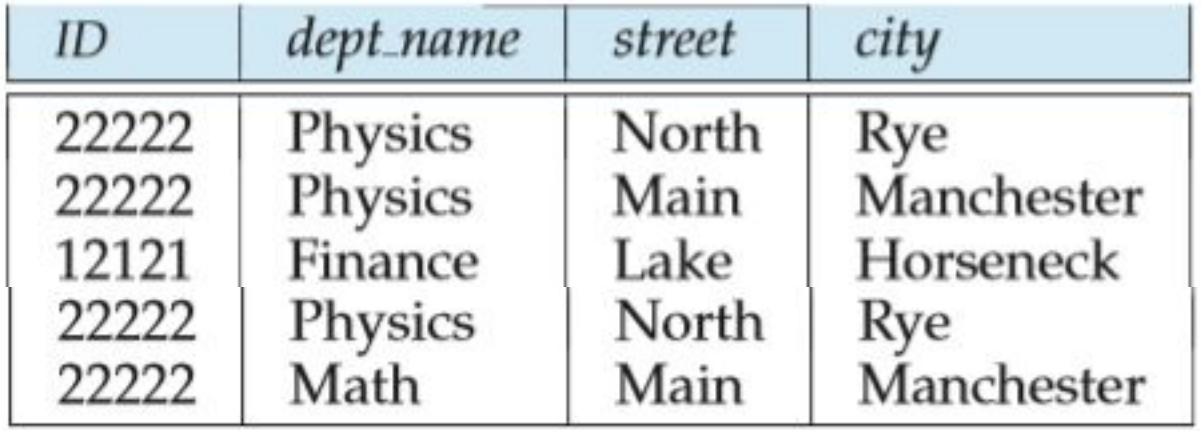
A table is said to be in 4NF if the following conditions are met,

- The table is in Boyce-Codd Normal Form (BCNF).
- The table is not any having an independent multi-valued dependency.

What is a Multivalued Dependency?

- A full constraint between two sets of attributes in a relation.
- Multivalued dependency would occur whenever two separate attributes in a given table happen to be independent of each other. And yet, both of these depend on another third attribute. The multivalued dependency contains at least two of the attributes dependent on the third attribute.

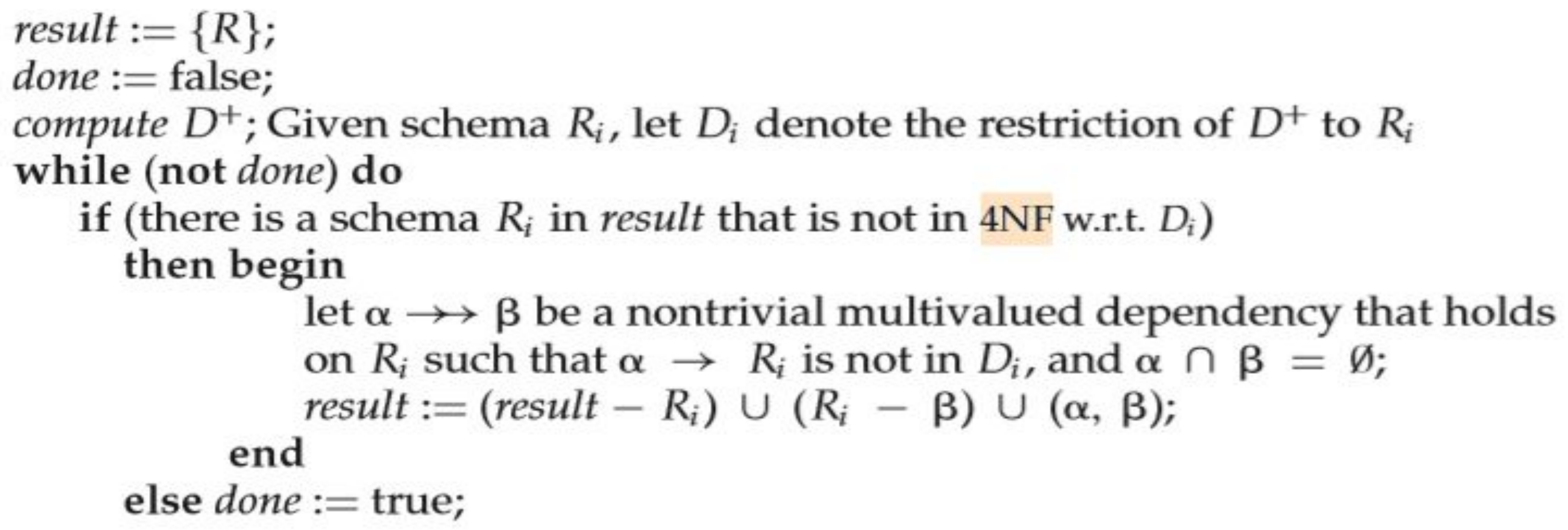
ID →→ street, city



ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Physics	Main	Manchester
12121	Finance	Lake	Horseneck
22222	Physics	North	Rye
22222	Math	Main	Manchester

ID→→dept name is a non-trivial multivalued dependency and ID is not a superkey for the schema.

4NF DECOMPOSITION ALGORITHM

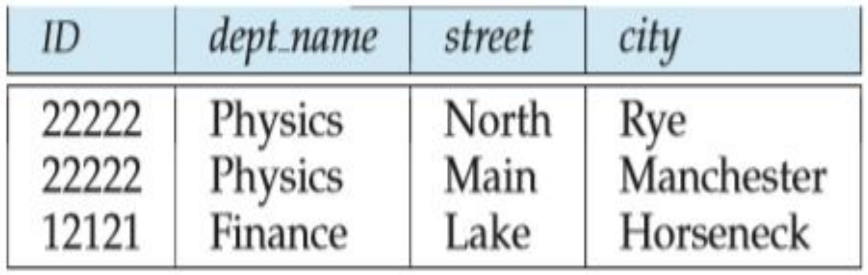


```
$result := \{R\};$
$done :=$ false;
compute $D^+$; Given schema $R_i$, let $D_i$ denote the restriction of $D^+$ to $R_i$
while (not $done$) do
    if (there is a schema $R_i$ in $result$ that is not in 4NF w.r.t. $D_i$)
      then begin
                let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that holds
                on $R_i$ such that $\alpha \rightarrow R_i$ is not in $D_i$, and $\alpha \cap \beta = \emptyset$;
                $result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$;
           end
      else $done :=$ true;
```

Figure 8.16 4NF decomposition algorithm.

Lossless-join Decomposition

A decomposition of R into R1 and R2 is a lossless join if at the following dependencies is in D+



ID	dept_name	street	city
22222	Physics	North	Rye
22222	Physics	Main	Manchester
12121	Finance	Lake	Horseneck

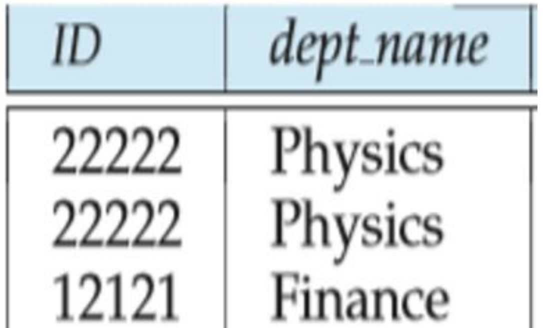
$$R_1 \cap R_2 \twoheadrightarrow R_1$$
$$R_1 \cap R_2 \twoheadrightarrow R_2$$

And lossless if R1 U R2=D+

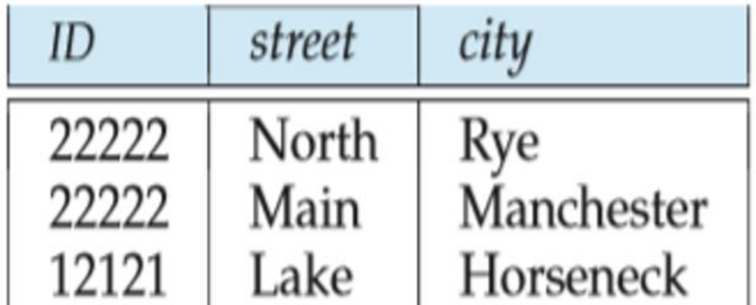
ID→→dept_name is a non-trivial multivalued dependency, and ID is not a superkey for the schema.

Following the algorithm,we replace it by two schemas:

r1 (ID, dept name) r2 (ID, street, city)



ID	dept_name
22222	Physics
22222	Physics
12121	Finance



ID	street	city
22222	North	Rye
22222	Main	Manchester
12121	Lake	Horseneck

Join dependency and 5 NF

5NF is also called project join normal form(PJNF)

It is based on **lossless** join decomposition.

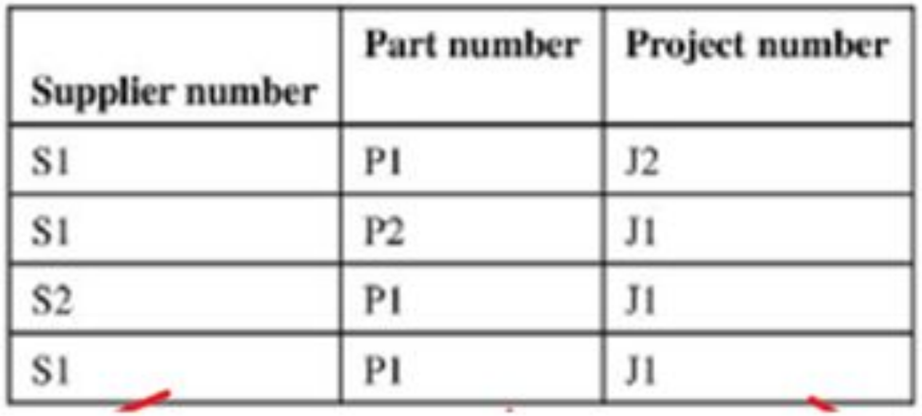
A relation is said to be in 5NF

- if and only if it satisfies 4NF and no join dependency exists.
- A relation is said to have join dependency **if it can be recreated by joining multiple sub relations** and each of these sub-relations has a subset of the attributes of the original relation.

Condition for join dependency

- Consider a relation R(P,Q,S) .
- where R1 and R2 are the decompositions such that
- R1 (P, Q) and R2 (Q, S)
- Then R1 and R2 are a lossless decomposition of R if R can be obtained by joining R1 and R2.

$$\begin{tikzcd}
 & R \arrow[dl] \arrow[dr] & \\
R1 \arrow[dr] & & R2\ (\text{Decompose}) \arrow[dl] \\
 & R &
\end{tikzcd}$$



Supplier number	Part number	Project number
S1	P1	J2
S1	P2	J1
S2	P1	J1
S1	P1	J1

Supplier number	Part number
S1	P1
S2	P2
S2	P1

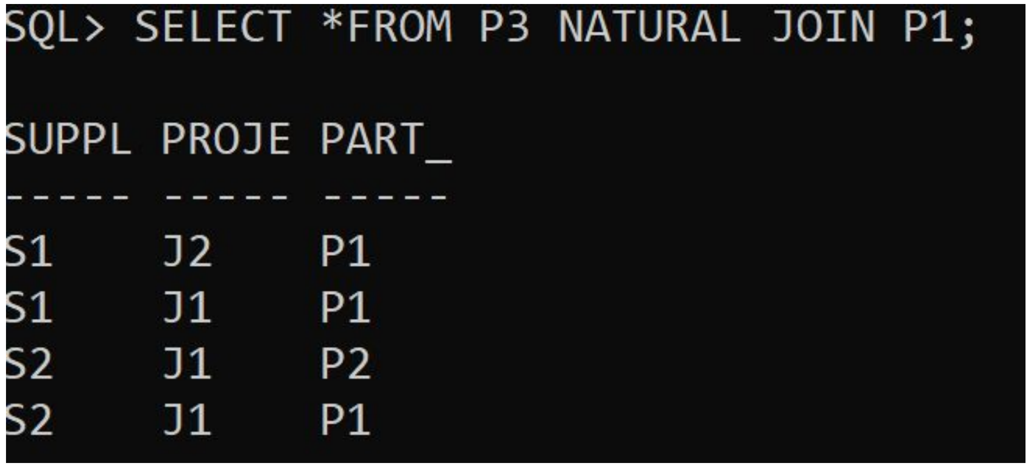
P1

Part number	Project number
P1	J2
P2	J1
P1	J1

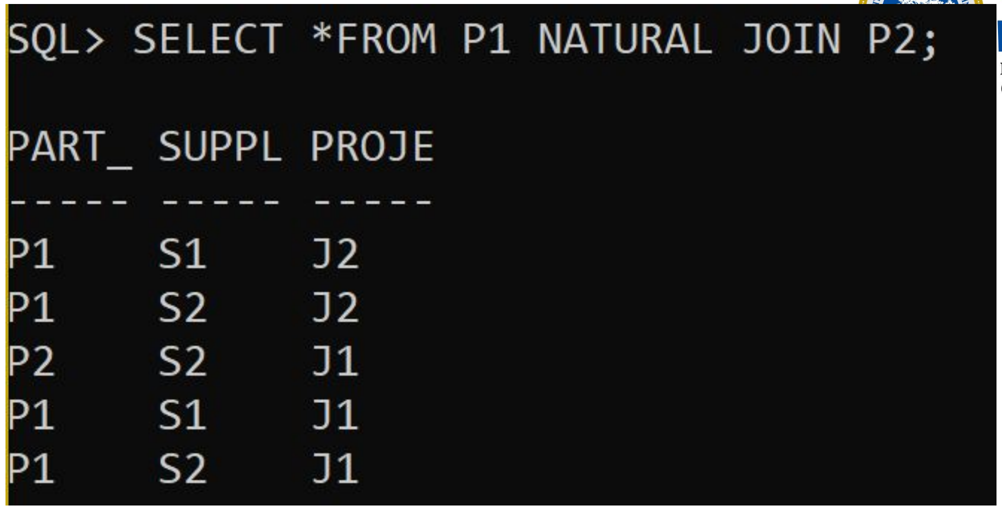
P2

Project number	Supplier number
J2	S1
J1	S1
J1	S2

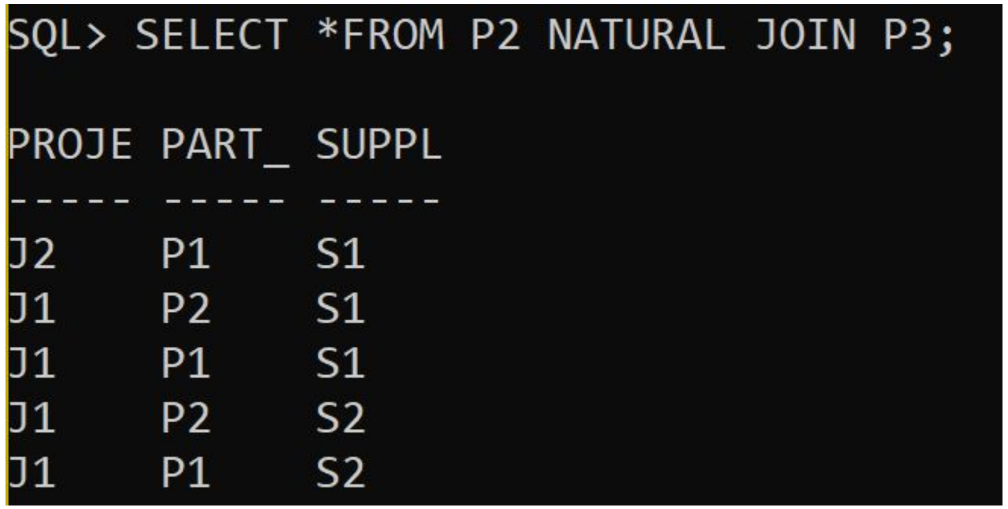
P3



```
SQL> SELECT *FROM P3 NATURAL JOIN P1;

SUPPL PROJE PART_
----- ----- -----
S1    J2    P1
S1    J1    P1
S2    J1    P2
S2    J1    P1
```

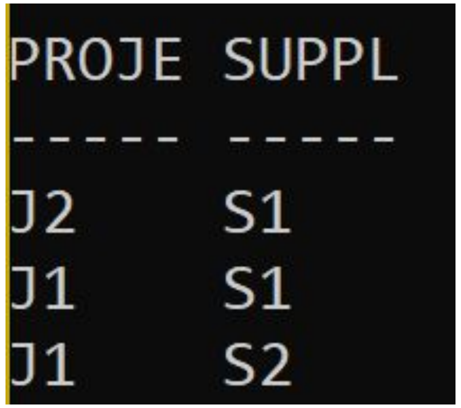


```
SQL> SELECT *FROM P1 NATURAL JOIN P2;

PART_ SUPPL PROJE
----- ----- -----
P1    S1    J2
P1    S2    J2
P2    S2    J1
P1    S1    J1
P1    S2    J1
```

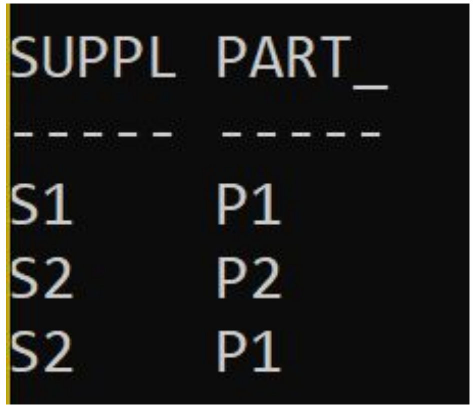


```
SQL> SELECT *FROM P2 NATURAL JOIN P3;

PROJE PART_ SUPPL
----- ----- -----
J2    P1    S1
J1    P2    S1
J1    P1    S1
J1    P2    S2
J1    P1    S2
```

SO THE TABLE TO SATISFY 5NF IS PARTITIONED INTO **P3 AND P1** since the natural join does not generate any additional rows



```
PROJE SUPPL
----- -----
J2    S1
J1    S1
J1    S2
```



```
SUPPL PART_
----- -----
S1    P1
S2    P2
S2    P1
```

Normal Form

1NF **A relation is in 1NF if it contains an atomic value.**

2NF **A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.**

3NF **A relation will be in 3NF if it is in 2NF and no transition dependency exists.**

BCNF **A stronger definition of 3NF & on the left-hand side of the functional dependency there is a candidate key**

4NF **A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency.**

5NF **A relation is in 5NF. If it is in 4NF and does not contain any join dependency, joining should be lossless.**

THANK YOU