4 Unit 4

Syllabus →

15
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
Lab10: PL/SQL Procedures on sample exercise. ** Frame and execute the appropriate Join Queries for the project*
closure of FD set , closure of attributes
irreducible set of FD
Normalization – 1Nf, 2NF, 3NF,
Decomposition using FD- dependency preservation,
Lab 11: PL/SQL Functions

** Frame and execute the appropriate Set Operators & Views for the project*
BCNF
Multi- valued dependency,
4NF
Join dependency and 5NF
Lab 12: PL/SQL Cursors ** Frame and execute the appropriate PL/SQL Conditional and Iterative Statements for the project*

Question Pattern →

20 MCQ
Two 5 marks(out of 3)
Two 10 marks(out of 3)

Important Topic →

Unit 4 -
4M
Dependencies
Relational Algebra

12M
Normalisation - 1NF, 2NF, 3NF and BCNF

Mcqs →

Q.No.	Questions	Course Outcome	Competence BT Level
1	**A _____expression forms a new relation after applying a number of algebraic operators to an existing set of relations** A. relational expression B. **relational algebra** C. relational calculus D. relational query	CO5	BT1
2	**_____ is used to change the values of some attributes in existing tuples.** A. **Update** B. Drop C. Truncate D. Select	CO5	BT1
3	**Which can be violated if a key value in the new tuple t already exists in another tuple in the relation r(R).** A. Domain constraints B. **Key constraints** C. Integrity constraints D. Rule constraints	CO5	BT1
4	**The relational algebra is a _____ Query language.** A. Structured B. Logical C. **Procedural** D. Relational	CO5	BT1
5	**The relational calculus is considered to be _____.** A. **a nonprocedural language** B. a procedural language C. a structured language D. a unstructured language	CO5	BT1
6	**Which of the following can be violated by delete operation.** A. primary key B. **referential integrity** C. alternate key D. super key	CO5	BT1
7	**Relationship can be created between** A. two tables only B. one table only C. **two or more tables** D. none of the above	CO5	BT1
8	**The value of the atom which evaluates either condition is TRUE or FALSE for particular combination of tuples is classified as** A. Intersection Value B. Union Value C. Deny Value D. **Truth Value**	CO5	BT1

9	**_____can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation.** A. Super Key B. **Referential integrity** C. Primary Key D. Candidate Key	CO5	BT1
10	**The relational calculus is important for two reasons. Which of the following is true?** A. **It has a firm basis in mathematical logic and the SQL for RDBMSs has some of its foundations in the tuple relational calculus.** B. It is in relational algebra and the values of some attributes are in existing tuples. C. It is a tuple relational calculus expression and It satisfy Functional dependency. D. None of the above	CO5	BT1
11	**A functional dependency X → Y is _____if removal of any attribute A from X means that the dependency does not hold any more.** A. a partial functional dependency B. a multivalued functional dependency C. **a full functional dependency** D. a transitive dependency	CO5	BT2
12	**_____ can be violated if an attribute value is given that does not appear in the corresponding domain.** A. **Domain constraints** B. Referential integrity constraints C. Key constraints D. Check constraints	CO5	BT1
13	**A tuple relational calculus expression may generate a/an** A. Finite Relation B. **Infinite Relation** C. Invalid Relation D. Composite Relation	CO5	BT1

14	**Which of the following statements about normal forms is FALSE?** A. BCNF is stricter than 3NF B. Lossless,dependency-preserving decomposition into 3NF is always possible. C. **Lossless,dependemcy-preserving decomposition into BNF is always possible.** D. Any relation with two attributes is in BCNF.	CO5	BT1
15	**Third normal form is based on the concept of_____.** A. **transitive dependency** B. partial dependency C. multivalued dependency D. full functional dependency	CO5	BT1
16	**The only attribute values permitted by 1NF are _____values.** A. Divisible B. **Single atomic** C. Multiple D. Numeric	CO5	BT1
17	**A relational query language L is considered relationally complete if we can express in L any query that can be expressed in_____.** A. Relational Algebra B. Structured Language C. **Relational calculus** D. Logical Language	CO5	BT1
18	**Anomalies are avoided by splitting the offending relation into multiple relations,is also known as_____.** A. Accupressure B. **Decomposition** C. Precomposition D. Both decomposition & precomposition E.	CO5	BT1

19	**Consider the relation(ABCDEF)** **FDs:** **A FC** **C D** **B E** **Find the 3NF relations:** A. ACDF,BE,AB B. ACDF,BE,AB,CD C. **CD,ACF,BE,AB** D. ACF,CDF,AB,BE	CO5	BT3
20	**The tuple relational calculus is based on specifying a number of _____.** A. String Variables B. Column Variables. C. Relation Variables D. **Tuple Variables.**	CO5	BT1
21	**Consider the relation (ABCDEF)** **A FC** **C D** **B E** **Find the 2NF relations** A. ACDF,AE B. **ACDF,BE,AB** C. BE,AB D. ACD,BE,AB	CO5	BT3
22	**The relation X(ABCDEF) with functional dependency set F={AB CD,C CA,B E,D B,E F}. The number of candidate keys of a relation R is _____.** A. 3 B. **4** C. 2 D. 5	CO5	BT3
23	**First normal form disallows _____.** A. indivisible values B. divisible values C. single atomic values D. **multivalued attributes**	CO5	BT1

24	**In a functional dependency X --> Y, if Y is functionally dependent on X, but not on X's proper subsets, then we would call the functional dependency as** A. **Full Functional Dependency** B. Partial Functional Dependency C. Multivalued Functional Dependency D. None of the above	CO5	BT2
25	**A functional dependency X → Y is a _____if some attribute A ε X can be removed from X and the dependency still holds.** A. **Partial Dependency** B. Multivalued Dependency C. Transitive Dependency D. Full Functional Dependency.	CO5	BT2
26	**An Multi Valued Dependency X →→ Y in R is called a _____ if (a) Y is a subset of X, or (b) X ∪ Y = R.** A. Total Multi Valued Dependency B. **Trivial Multi Valued Dependency** C. Non-trivial Multi Valued Dependency D. Partial Multi Valued Dependency	CO5	BT2
27	**Consider a Relation R(ABCDE) with Functional Dependency** **A BCDE,BC ADE,D E** **The Decomposition of R in 3NF will be** A. R1(ABCE) and R2(DE) B. R1(ADE) and R2(BC) C. R1(ABDE) and R2(BDE) D. **R1(ABCD) and R2(DE)**	CO5	BT3
28	**A large number of commercial applications running against relational databases is called as _____** A. Data Control Language B. Structured Query Language C. **Online Transaction Processing** D. MongoDataBase	CO5	BT1
29	**A simple tuple relational calculus query is of the form:** A. {p | COND(t)} B. **{t | COND(t)}** C. {p | COND(p)} D. {p| P(t)}	CO5	BT1

30	**Which of the following is the result of bad database design?** A. Repetition of Information B. Inability to represent some information C. Inconsistent database state due to some transaction D. **All of the above**	CO5	BT1
31	**Third normal form (3NF) is based on the concept of _____.** A. Partial Dependency B. Multivalued Dependency C. **Transitive Dependency** D. Join Dependency	CO5	BT1
32	**Second normal form (2NF) is based on the concept of** A. Partial Dependency B. Multivalued Dependency C. Transitive Dependency D. **Full Functional Dependency.**	CO5	BT1
33	**An attribute is called _____,if it is not a member of any candidate key.** A. Prime B. **Non-prime** C. Composite D. Derived	CO5	BT1
34	**Consider F1 and F2 as two sets of functional dependencies. If every functional dependency in F2 can be inferred from the functional dependencies of F1 using inference rules, then F1 is _____ of F2.** A. **Cover Set** B. Closure Set C. Minimal Set D. None of the above	CO5	BT2
35	**Two special symbols called quantifiers can appear in formulas; they are :** A. universal quantifier (∀) and the existential quantifier (∃). B. universal quantifier (A) and the existential quantifier (E). C. conditional quantifier ($\forall_A$) and the generalized quantifier ($\exists_A$). D. None of the above.	CO5	BT2

36	**_____denoted by JD(R1, R2, ..., Rn), specified on relation schema R, specifies a constraint on the states r of R.** A. Partial Dependency B. **Join Dependency** C. Join Decomposition D. Join Database	CO5	BT2
37	**The normalization process, as first proposed by** A. Edgar F. Codd B. Peter Landin C. Mark Edward D. Sandford	CO5	BT1
38	**Assume a relation R(A, B, C, D) with set of functional dependencies F={C→D,C→A,B→C}. Use this setup to answer the following questions; Which of the following is the candidate keys of R?** A. C B. BC C. **B** D. Both (b) and (c)	CO5	BT3
39	**An attribute of relation schema R is called _____of R if it is a member of some candidate key of R.** A. a non-prime attribute B. **a prime attribute** C. Composite attribute D. Derived attribute	CO5	BT2
40	**Consider the relation schema R={E,F,G,H,I,J,K,L,M,N} and the set of functional dependencies EF G,F IJ,EH KL,K M,L N ON r. What is the key for R?** A. {E,F} B. **{E,F,H}** C. {E,F,H,K,L} D. {E}	CO5	BT3
41	**If X --> Y is a functional dependency and X and Y are sets of attributes, what is the relationship between X and Y?** A. One-to-Many B. Many-to-One C. **One-to-One** D. Many-to-Many	CO5	BT1

42	**A _____of a relation schema R = {A1, A2, ... , An} is a set of attributes S ⊆ R with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$.** A. **super key** B. foreign key C. candidate key D. alternate key	CO5	BT1
43	**For a functional dependency X --> Y, it is said to be _____ if Y is the subset or equal to X.** E. Total functional dependency F. **Trivial functional dependency** G. Non-trivial functional dependency H. Partial functional dependency	CO5	BT1
44	**A functional dependency set F={A B,BC E,ED A,EF G,E F} Find out the closure of (AC)** A. {A,B,C,D,E,F,G} B. {A,B,D,E,F} C. {A,B,C,E} D. **{A,B,C,E,F,G}**	CO5	BT3
45	**Let R(A,B,C,D) be a relation schema and F{A BC,AB D,B C} Be the set of functional dependencies define over R. Which of the following represents the closure of the attribute set{B}?** A. {A,C,D} B. **{B,C}** C. {A,B,C} D. {B}	CO5	BT3
46	**A table is in 2NF if it is in 1NF and if:** A. no column that is not a part of the primary key is dependent on only a portion of the alternate key. B. **no column that is not a part of the primary key is dependent on only a portion of the primary key.** C. no column that is not a part of the primary key is dependent on only a portion of the foreign key. D. none of these	CO5	BT2

47	**_____is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.** A. **Denormalization** B. Normalization C. Dependency D. Relational Algebra	CO5	BT1
48	**R(A,B,C,D) is a relation,which of the following does not have a lossless join dependency preserving BCNF decomposition.** A. **A B,B CD** B. A B,B C,C D C. AB C,C AD D. A BCD	CO5	BT3
49	**Which normal form is considered adequate for relational database design?** A. 2NF B. **3NF** C. 4 NF D. BCNF	CO5	BT1
50	**A functional dependency of the form x —> y is trival if _____.** A. **y ⊆ x** B. y ⊂ x C. x ⊂ y D. x ⊂y and y⊂ x	CO5	BT2

Endsem pyqs →

29. a.	Give the notation and example code in relation algebra for the following operations (i) Outer join (ii) Assignment (iii) Select operation	10	3	4	2
	(OR)				
b.	What is the pit fall in relational data base? How it can be rectified? Explain 1st normal form.	10	3	4	3

29. a.	Define functional dependency with respect to normal forms. How 2NF and 3NF can be resolved? Give an example scenario for the above situation, using 'STUDENT' relation. Note: column names can be generated with respect to student relation.	10	5	4	4.3.1
	(OR)				
b.i.	Give an example table structure for multi-valued dependency and define with required explanations.	5	5	4	4.3.4
ii.	Analyze the syntax for writing 'cursors' and list any two real time examples of cursors.	5	5	4	4.3.4

31. a.	Define tuple relational calculus and state the formal syntax of the expression form with examples.	12	4	5	2
	(OR)				
b.	Discuss the concepts of functional dependency that are required for a database to be in third normal form. Illustrate them with proper examples.	12	4	5	2

31.	(a) Exemplify the multi-value dependency in relation. Explain in detail 2NF, 3NF and the fourth normal form-4NF.	12	3	4
	(OR)			
	(b) Determine the closer of the following set of functional dependencies for a relational scheme R(A,B,C,D) and FDs {AB → C, C → D, D → A}. List out the candidate keys of R.Find the highest normal form of the relation.			

CT pyqs →

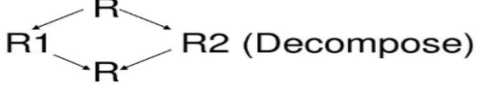
16	a. Write the following queries in relational algebra, using the University Schema: classroom(building, room number, capacity) department(dept name, building, budget) course(course id, title, dept name, credits) instructor(ID, name, dept name, salary) section(course id, sec id, semester, year, building, room number, time slot id) teaches(ID, course id, sec id, semester, year) student(ID, name, dept name, tot cred) takes(ID, course id, sec id, semester, year, grade) a. Find the titles of courses in the Comp. Sci. department that have 3 credits. $\Pi_{title}(\sigma_{dept\_name = \text{'Comp. Sci'} \wedge credits=3}(course))$ b. Find the highest salary of any instructor. $\mathcal{G}_{\mathbf{max}(salary)}(instructor)$ c. Find the enrolment of each section that was offered in Autumn 2009. $_{course\_id, section\_id}\mathcal{G}_{\mathbf{count}(*)\ \mathbf{as}\ enrollment}(\sigma_{year=2009 \wedge semester=\text{Autumn}}(takes))$ d. Find the names of all students who have taken at least one Comp. Sci. course. $\pi_{\text{name}}(\sigma_{\text{dept\_name='Comp. Sci.'}}(\text{student} \bowtie \text{takes} \bowtie \text{course}))$ **(or)** b. Suppose you are designing a database for a university that includes information about courses, students, and professors. Each course is taught by a	12	4	5

	single professor, and each professor can teach multiple courses. Additionally, each course can have multiple students enrolled in it, and each student can be enrolled in multiple courses. You have already designed tables for Courses, Students, and Professors, which are all in 3NF. However, you realize that there is a potential issue with redundancy in your database. Specifically, you notice that there could be multiple rows in the enrolment table for a single student if that student is enrolled in multiple courses. How would you modify your database to eliminate this redundancy and bring it up to 4NF? Discuss the concept of Multivalued dependency related to 4NF. **Ans:** To eliminate redundancy and bring the database up to 4NF, we would need to create a new table to represent the relationship between students and courses. The new table, called Enrollments, would store information about the enrolment of each student in each course and would have the following columns: StudentID: a foreign key that references the Students table CourseID: a foreign key that references the Courses table EnrollmentDate: the date on which the student enrolled in the course With this new table in place, we can eliminate the redundancy in the original enrollment table, as each student/course combination will only appear once in the Enrollments table. Additionally, we have eliminated the possibility of anomalies that could arise if we were to update, insert, or delete data in the original enrollment table. The new Enrollments table is in 4NF because it has no multi-valued dependencies and is free of redundant data. 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. 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			

<table>
<tr><td>11</td><td>What are some potential pitfalls we might encounter when designing a Relational database.
Redundancy, Inconsistency, Inefficiency, Complexity.</td><td>4</td><td>4</td><td>5</td></tr>
<tr><td>12</td><td>Consider the following dependencies,
FD={A->B,B->D, C->DE, CD-> AB}
Calculate the closure of attributes A^+, $(CD)^+$

A+ = {ABD}
(CD)+ = {CDABE}</td><td>4</td><td>3</td><td>5</td></tr>
<tr><td>13</td><td>Write a Relational Algebraic Expression for the following Queries.

<table>
<tr><th>Player Id</th><th>Team Id</th><th>Country</th><th>Age</th><th>Runs</th><th>Wickets</th></tr>
<tr><td>1001</td><td>101</td><td>India</td><td>25</td><td>10000</td><td>300</td></tr>
<tr><td>1004</td><td>101</td><td>India</td><td>28</td><td>20000</td><td>200</td></tr>
<tr><td>1006</td><td>101</td><td>India</td><td>22</td><td>15000</td><td>150</td></tr>
<tr><td>1014</td><td>105</td><td>Pakistan</td><td>21</td><td>3599</td><td>205</td></tr>
</table>
a. Find all tuples from player relation for which country is India.
$\sigma_{\text{"country" = "India"}}(\text{Player})$
b. Select all the tuples for which runs are greater than or equal to 15000.
$\sigma_{\text{"runs" >= "15000"}}(\text{Player})$
c. List all the countries in Player relation
$\pi_{\text{"country"}}(\text{Player})$
d. List all the team ids and countries in Player Relation
$\pi_{\text{Team Id, Country}}(\text{Player})$</td><td>4</td><td>3</td><td>5</td></tr>
</table>

11.	Consider the following relation. R (A, B, C, D). Assume the set of functional dependency F= {B ->C, D ->A}. R is decomposed in to R1(B, C) and R2(A, D) a. List down the Armstrong's axioms. (2 Marks) b. Find the candidate key(s) for R. (2 marks) Axiom of reflexivity, Axiom of augmentation, Axiom of transitivity Additional axioms. Union, composition, decomposition, pseudo transitivity. (BD)+ = BCDA. It includes all attributes of R, hence BD is the candidate key.	4	4	5
12.	Suppose you have a relation R representing a database of employees, with attributes Employee_ID, Name, Department, and Salary. Write the syntax of tuple relational calculus expression and a query to find all employees who work in the "Marketing" department and have a salary greater than $50,000. In tuple relational calculus, expressions are written in the form , {t | P(t)}, where t is a tuple variable and P(t) is a predicate that defines a condition that the tuples in the resulting relation must satisfy. We can write the tuple relational calculus expression for the given scenario as: {t | t ∈ R ∧ t.Department = "Marketing" ∧ t.Salary > 50000}	4	3	5
13.	Provide an example and brief about join dependency? 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. 	4	3	5

16.	a. Consider a small library system that tracks information about books, authors, and the borrowers who check out the books. Each book has a unique ISBN number, a title, a publication year, and a publisher. Each author has a unique author ID, a name, and a birth year. Each borrower has a unique borrower ID, a name, an address, and a phone number. Each book can be checked out by one or more borrowers, and each borrower can check out one or more books. For each instance of a book being checked out by a borrower, the system records the date the book was checked out and the date it was returned. Based on this scenario, can	12	4	5

	you provide a table design that represents the data in 3NF? Discuss the concept of Functional dependency involved in 3NF? A table design that represents the data in 3NF: Table 1: Books ISBN (primary key) Title Publication year Publisher Table 2: Authors Author ID (primary key) Name Birth year Table 3: Borrowers Borrower ID (primary key) Name Address Phone number Table 4: Book Borrowings Borrowing ID (primary key) ISBN (foreign key referencing Books table) Borrower ID (foreign key referencing Borrowers table) Check out date Return date In this design, we have split the data into separate tables based on their unique attributes and relationships. This table structure eliminates redundancy and ensures data integrity by avoiding data duplication and maintaining relationships between entities. **Transitive dependency : (Explanation)** 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 **(or)** b. Consider the relational database given below where primary keys are underlined. Give an expression in Relational Algebra to express the following Queries: employee (person_name, street, city) works (person_name, company_name, salary) company (company_name, city) manages (person_name, manager_name) i.)Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum. **select * from employee** **where person_name in** **(select person_name from Works** **where company_name='First Bank Corporation' and salary>10000**			

	select E.person_name, street, city **from Employee as E, Works as W** **where E.person_name=W.person_name and W.company_name='First Bank Corporation'** **and W.salary>10000** ii.)Find the names of all employees in this database who live in the same city as the company for which they work. **select E.person_name** **from Employee as E, Works as W, Company as C** **where E.person_name=W.person_name and E.city=C.city** **and W.company_name=C.company_name** iii.) Find names of all employees who earn more than every employee of Small Bank Corporation **select person_name from Works** **where salary > all** **(select salary from Works** **where company_name='Small Bank Corporation')** **select person_name from Works** **where salary>(select max(salary) from Works** **where company_name='Small Bank Corporation')** iv.)Find the employees of small Bank Corporation who earn maximum salary **select company_name** **from Works** **group by company_name** **having avg(salary)>(select avg(salary)** **from Works** **where company_name='First Bank Corporation')**			

Personal Notes →

Relational Algebera →

Relational algebra is a formal language for the relational model of data, which allows for the manipulation and retrieval of data stored in relational database management systems. Here's an overview of some basic and extended operations in relational algebra:

The relational algebra is a **procedural query language**.

- Consists of a **set of operations** that **take one or two relations as input** and **produce a new relation as their result.**
- The **fundamental operations** in the relational algebra are **select, project, union, set difference, Cartesian product, and rename.**
- In addition to the fundamental operations, **there are several other operations**—namely, **set intersection, natural join, and assignment.**

Fundamental Operations

The select, project, and rename operations are called unary operations, because they operate on one relation.

The other three operations operate on pairs of relations and are, **called binary operations.**

Select Operation →

The **Select** operation in relational algebra, denoted as σ (sigma), is used to filter rows in a database table based on a specific condition. This operation is fundamental in database querying as it allows users to specify criteria that determine which rows should be included in the result set.

Syntax

The select operation is generally written as:

$$\sigma_{\text{condition}}(\text{Relation})$$

Here, `condition` represents a predicate that each row of the relation must satisfy to be included in the resulting relation.

Examples

Consider the following table `Employees` as an example:

EmployeeID	Name	Age	Salary	DepartmentID
1	Alice	30	50000	1
2	Bob	22	48000	1
3	Carol	34	52000	2
4	David	29	45000	3

Example 1: Select employees older than 30

To find all employees whose age is greater than 30:

$$\sigma_{\text{Age}>30}(\text{Employees})$$

Result:

EmployeeID	Name	Age	Salary	DepartmentID
3	Carol	34	52000	2

Example 2: Select employees working in department 1

To find all employees who are part of department 1:

$$\sigma_{\text{DepartmentID}=1}(\text{Employees})$$

Result:

EmployeeID	Name	Age	Salary	DepartmentID
1	Alice	30	50000	1
2	Bob	22	48000	1

- To select those tuples of the instructor relation where the instructor is in the "Physics" department, we write:

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

The instructor relation

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

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

Result of σ_{dept_name} ="Physics" (instructor)

Find all instructors with salary greater than $90,000.

$\sigma_{salary>90000}(instructor)$

ID	Name	Dept_Name	Salary
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
83821	Brandt	Comp. Sci.	92000

The Select Operation

- In general, we allow comparisons using =, ≠, <, ≤, >, and ≥ in the selection predicate.
- We can combine several predicates into a larger predicate by using the connectives and (∧), or (∨), and not (¬).
- To find the instructors in Physics with a salary greater than $90,000

$$\sigma_{dept\_name = \text{“Physics”} \wedge salary > 90000}\ (instructor)$$

- To find all departments whose name is the same as their building name,

$$\sigma_{dept\_name = building}(department)$$

Project Operation →

The **Project** operation in relational algebra, denoted as π (pi), is used to select specific columns from a relation, effectively reducing the number of columns in the result but keeping all rows that contain the specified attributes. This operation helps in extracting only the necessary data fields from a larger set of data.

Syntax

The project operation is typically expressed as:

$\pi_{\text{attribute1, attribute2, ..., attributeN}}(\text{Relation})$

Here, `attribute1, attribute2, ..., attributeN` are the names of the columns you want to retain in the resulting relation.

Example

Let's consider the following table `Employees` as an example to demonstrate the projection operation:

Employees Table

EmployeeID	Name	Age	Salary	DepartmentID
1	Alice	30	50000	1
2	Bob	22	48000	1
3	Carol	34	52000	2
4	David	29	45000	3

Example 1: Project the Name and Salary Columns

To retrieve just the names and salaries of all employees:

$\pi_{\text{Name, Salary}}(\text{Employees})$

Result:

Name	Salary
Alice	50000
Bob	48000
Carol	52000
David	45000

Example 2: Project the DepartmentID Column

To get a list of department IDs from the `Employees` table:

$\pi_{\text{DepartmentID}}(\text{Employees})$

Result (assuming removal of duplicates):

DepartmentID
1
2
3

The Project Operation

- We list those attributes that we wish to appear in the result as a subscript to Π.
- The argument relation follows in parentheses.
- We write the query to produce such a list

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

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

Composition of Relational Operations

- The fact that the result of a relational operation is itself a relation is important.
- Consider the more complicated query "Find the name of all instructors in the Physics department."

$$\Pi_{name}\left(\sigma_{dept\_name = \text{"Physics"}}\left(instructor\right)\right)$$

- **Instead of giving the name of a relation as the argument of the projection operation,**
  – Give an expression that evaluates to a relation.

Union Operation →

The **Union** operation in relational algebra, denoted by ∪, is used to combine the rows of two relations into a single relation. The union operation can only be applied to two relations if they are union-compatible, meaning they have the same number of attributes and corresponding attributes share the same data types.

Syntax

The union operation is generally expressed as:
$R \cup S$
where R and S are relations.

Properties

- **Union Compatibility**: As mentioned, both relations R and S must have the same number and type of attributes for the operation to be valid.
- **Elimination of Duplicates**: Since relations in relational algebra are sets, the result of a union operation automatically removes any duplicate tuples.
- **Result Schema**: The schema of the result of a union operation is the same as the schemas of R and S.

Example

Let's assume we have two tables, `Employees1` and `Employees2`, each listing employees from different branches of a company but with the same structure.

Employees1 Table

EmployeeID	Name	Age
1	Alice	30
2	Bob	22

Employees2 Table

EmployeeID	Name	Age
3	Carol	34
4	David	29
1	Alice	30

Operation

To combine these two tables into one:
$\text{Employees1} \cup \text{Employees2}$

Resulting Table

The result will include all unique tuples from both `Employees1` and `Employees2`.

EmployeeID	Name	Age
1	Alice	30
2	Bob	22
3	Carol	34
4	David	29

The Union Operation

- Consider a query to find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both.
- The information is contained in the section relation as shown in figure.

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

The section relation

To find the set of all courses taught in the Fall 2009 semester

$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Fall"} \wedge year = 2009}\ (section))$$

To find the set of all courses taught in the Spring 2010 semester

$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Spring"} \wedge year = 2010}\ (section))$$

The Union Operation

- To answer the query, we need the union of these two sets; that is, we need all section IDs that appear in either or both of the two relations.
- We find these data by the binary operation union, denoted, as in set theory, by ∪.
- The expression needed is

$$\Pi_{course\_id}\ (\sigma_{semester = \text{"Fall"} \wedge year = 2009}\ (section)) \cup \Pi_{course\_id}\ (\sigma_{semester = \text{"Spring"} \wedge year = 2010}\ (section))$$

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

The Union Operation

- There are 8 tuples in the result, even though there are 3 distinct courses offered in the Fall 2009 semester and 6 distinct courses offered in the Spring 2010 semester.
- Since relations are sets, duplicate values such as CS-101, which is offered in both semesters, are replaced by a single occurrence.
- A union operation r ∪ s to be valid, we require that two conditions hold:

Set Difference operation →

The set difference operation is generally expressed as:

$R - S$

where R and S are relations.

Example

Let's consider two tables, `EmployeesNY` and `EmployeesCA`, representing employees from New York and California, respectively, with some employees potentially working in both states.

EmployeesNY Table

EmployeeID	Name
1	Alice
2	Bob
3	Carol

EmployeesCA Table

EmployeeID	Name
3	Carol
4	David
5	Eve

Operation

To find employees who are in the New York table but not in the California table:

$$\text{EmployeesNY} - \text{EmployeesCA}$$

Resulting Table

The result will include only those employees from `EmployeesNY` that do not appear in `EmployeesCA`.

EmployeeID	Name
1	Alice
2	Bob

Explanation

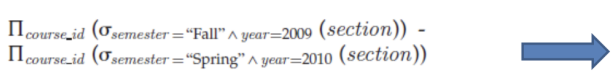
- Carol is present in both `EmployeesNY` and `EmployeesCA`, so she is excluded from the result.
- Alice and Bob are present only in `EmployeesNY`, so they remain in the result of the set difference.

The Set-Difference Operation

- denoted by –
- allows us to find tuples that are in one relation but are not in another.
- The expression r – s produces a relation containing those tuples in r but not in s.

find all the courses taught in the Fall 2009 semester but not in Spring 2010 semester

$$\Pi_{course\_id}\,(\sigma_{semester = \text{“Fall”} \wedge year = 2009}\,(section)) - \Pi_{course\_id}\,(\sigma_{semester = \text{“Spring”} \wedge year = 2010}\,(section))$$



course_id
CS-347
PHY-101

The Set-Difference Operation

- must ensure that set differences are taken between compatible relations.
- Therefore, for a set-difference operation r – s to be valid, we require that the relations r and s be of the same arity, and that the domains of the i^{th} attribute of r and the i^{th} attribute of s be the same, for all i.

Cartesian Product Operation →

- Total rows = R1rowsxR2rows
- Total columns = R1Columns + R2columns

Lec-47: Cross/Cartesian Product in Relational Algebra | Database Management System

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

▶ Database Management System(Complete Playlist):

https://www.youtube.com/watch?v=F6MBz_h3Uj8

The **Cartesian Product** operation in relational algebra, denoted by $\times$, combines all rows from one relation with all rows from another relation, resulting in a new relation. This operation is fundamental for constructing more complex join conditions that are not naturally supported by simpler join types.

Syntax

The Cartesian product is expressed as:
$R \times S$
where R and S are relations.

Example

Let's use the following two tables, `Customers` and `Products`, to illustrate the Cartesian Product.

Customers Table

CustomerID	Name
1	Alice
2	Bob

Products Table

ProductID	ProductName
A	Laptop
B	Phone

Operation

To combine these two tables using Cartesian Product:

$$\text{Customers} \times \text{Products}$$

Resulting Table

The resulting table will pair each customer with each product.

CustomerID	Name	ProductID	ProductName
1	Alice	A	Laptop
1	Alice	B	Phone
2	Bob	A	Laptop
2	Bob	B	Phone

Explanation

- Each row from the `Customers` table is combined with each row from the `Products` table, leading to 2 customers × 2 products = 4 combinations.
- This operation is useful for generating all possible combinations of tuples from two relations, which is essential in scenarios such as generating potential orders where each customer could potentially buy each product.

Usage

In practical SQL queries, the Cartesian product is rarely used directly because it often produces large and mostly irrelevant data sets. However, it's an integral part of defining more precise joins with specific conditions. For example, a natural join or an equi-join between two tables is effectively a Cartesian product followed by a selection operation that filters tuples based on a join condition.

To directly execute a Cartesian Product in SQL, you could use the `` `CROSS JOIN` `` clause, which performs a Cartesian product of the specified tables. For instance:

sql Copy code

```sql
SELECT *
FROM Customers
CROSS JOIN Products;
```

This SQL query would produce the same result set as the Cartesian Product example shown, illustrating all possible combinations of customers with products.

The Cartesian-Product Operation

- From that definition, we should already have an intuition about the definition of the Cartesian-product operation.
- However, since the same attribute name may appear in both r1 and r2, we need to devise a naming schema to distinguish between these attributes.
- We do so here by attaching to an attribute the name of the relation from which the attribute originally came.
- For example, the relation schema for r = instructor × teaches is

(*instructor.ID, instructor.name, instructor.dept_name, instructor.salary*
teaches.ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

- With this schema, we can distinguish instructor.ID from teaches.ID.
- For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix.
- This simplification does not lead to any ambiguity.
- We can then write the relation schema for r as:

(*instructor.ID, name, dept_name, salary*
teaches.ID, course_id, sec_id, semester, year)

Rename Operation →

There are a few variations in the syntax for the rename operation, which can be used to rename either the relation itself or its attributes:

1. **Renaming a Relation**:

$$\rho_{\text{NewRelationName}}(\text{Relation})$$

This syntax changes the name of the relation to 'NewRelationName'.

2. **Renaming Attributes**:

$$\rho_{(\text{NewAttr1, NewAttr2, ..., NewAttrN})}(\text{Relation})$$

This syntax changes the names of the attributes without changing the name of the relation.

3. **Renaming Both Relation and Attributes**:

$$\rho_{\text{NewRelationName}(\text{NewAttr1, NewAttr2, ..., NewAttrN})}(\text{Relation})$$

This syntax changes both the name of the relation and the names of its attributes.

Example

Let's use a simple table to illustrate the Rename operation:

Employees Table

EmployeeID	Name	Age	Salary
1	Alice	30	50000
2	Bob	22	48000

Renaming a Relation

Suppose we want to refer to the `Employees` table as `Emp` in a complex query to make it shorter or clearer:

$$\rho_{\text{Emp}}(\text{Employees})$$

Now, the table will be referred to as `Emp` in subsequent operations.

Renaming Attributes

If we want to change the attribute names to be more specific, such as changing `Name` to `EmployeeName` and `Salary` to `AnnualSalary`:

$$\rho_{(\text{EmployeeID, EmployeeName, Age, AnnualSalary})}(\text{Employees})$$

Renaming Both Relation and Attributes

To change the table name to `Emp` and also rename its attributes to `EmployeeName` and `AnnualSalary`:

$$\rho_{\text{Emp}(\text{EmployeeID, EmployeeName, Age, AnnualSalary})}(\text{Employees})$$

Usage

The Rename operation is particularly useful in queries involving multiple relations where attribute names could potentially clash or where clarity is needed in complex SQL joins or subqueries. In SQL, this can often correspond to using aliases with the `AS` keyword, like this:

sql Copy code

```sql
SELECT EmployeeID AS ID, Name AS EmployeeName, Age, Salary AS AnnualSal
FROM Employees AS Emp;
```

In summary, the Rename operation helps in managing the schema of relations during query operations, making them more manageable and preventing conflicts in attribute names, which is crucial for accurate and efficient data manipulation and retrieval.

Set-Intersection Operation →

The **Set Intersection** operation in relational algebra, denoted by ∩, is used to find tuples that are present in both of two given relations. Like the union and set difference operations, set intersection also requires that the two relations be union-compatible, meaning they must have exactly the same attributes with matching types.

Syntax

The set intersection operation is expressed as:
$R \cap S$
where R and S are relations.

Example

Let's consider two tables, `StudentsA` and `StudentsB`, representing students enrolled in two different courses, with some students possibly enrolled in both.

StudentsA Table

StudentID	Name
1	Alice
2	Bob
3	Carol

StudentsB Table

StudentID	Name
3	Carol
4	David
5	Eve

Operation

To find students who are enrolled in both courses:

$$\text{StudentsA} \cap \text{StudentsB}$$

Resulting Table

The result will include only those students present in both `StudentsA` and `StudentsB`.

StudentID	Name
3	Carol

Explanation

- Carol appears in both `StudentsA` and `StudentsB`, so she is included in the result.
- Alice and Bob, who appear only in `StudentsA`, and David and Eve, who appear only in `StudentsB`, are not included in the result.

Usage

The set intersection operation is particularly useful in scenarios where you need to find common elements between two datasets. For instance, it can be used to identify common customers between two regions, students who are taking multiple specific courses, or products that are popular across different sales channels.

In SQL, there is no direct equivalent of set intersection using a single keyword. However, it can be implemented using an `**INNER JOIN**` on all columns or by using a combination of `**UNION**` and `**GROUP BY**` with a `**HAVING**` clause to count duplicates:

sql Copy code

```sql
SELECT StudentID, Name
FROM StudentsA
INTERSECT
SELECT StudentID, Name
FROM StudentsB;
```

The Set-Intersection Operation

- The first additional relational-algebra operation that we shall define is set intersection (∩).

Find the set of all courses taught in both the Fall 2009 and the Spring 2010 semesters.

$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Fall”} \wedge year=2009}\ (section)) \cap \Pi_{course\_id}\ (\sigma_{semester = \text{“Spring”} \wedge year=2010}\ (section))$$

course_id
CS-101

Courses offered in both the Fall 2009 and Spring 2010 semesters

The Set-Intersection Operation

- Rewrite any **relational-algebra expression that uses set intersection**
- by replacing the intersection operation with a pair of set-difference operations

$$r \cap s = r - (r - s)$$

- Set intersection is not a fundamental operation and does not add any power to the relational algebra.
- It is simply more convenient to write r ∩ s than to write r – (r – s).

Joins →

Lec-38: Introduction to Joins and its types | Need of Joins with example | DBMS

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

▶Database Management System(Complete Playlist):

https://www.youtube.com/watch?v=zYH-e6tUYbw

JOINS IN SQL (with example)

If you need personal help in SQL, Mock interviews, You can contact me on work.sadiasiddiqui@gmail.com, Please note it will be a paid.

https://youtu.be/LsRagOpaXEg?si=JnFtq7ZKOaU27U4U

Natural-Join Operation →

The **Natural Join** operation in relational algebra, denoted by $\bowtie$, combines two relations based on their common attributes. The result includes only those combinations of tuples where the values of all attributes with the same name match in both relations. This operation automatically identifies the common columns between the two relations and performs an equi-join on these columns.

Syntax

The natural join is generally expressed as:
$R \bowtie S$
where R and S are relations.

Example

Let's use two tables, `Employees` and `Departments`, to illustrate the Natural Join operation.

Employees Table

EmployeeID	Name	Age	DepartmentID
1	Alice	30	1
2	Bob	22	1
3	Carol	34	2
4	David	29	3

Departments Table

DepartmentID	DepartmentName
1	HR
2	Engineering
3	Marketing

Operation

Performing a Natural Join on `Employees` and `Departments` based on the common attribute `DepartmentID`:

$$\text{Employees} \bowtie \text{Departments}$$

Resulting Table

The result includes all columns from both tables, merged on the common attribute.

EmployeeID	Name	Age	DepartmentID	DepartmentName
1	Alice	30	1	HR
2	Bob	22	1	HR
3	Carol	34	2	Engineering
4	David	29	3	Marketing

Explanation

- The Natural Join matched rows from `Employees` and `Departments` where the `DepartmentID` values are identical.
- Each employee is linked with their corresponding department, and the `DepartmentID` column appears only once in the output because it is the common attribute.

Usage

Natural joins are particularly useful for combining tables that have explicit foreign key relationships without needing to specify the join condition explicitly. They help to simplify SQL queries and are widely used in database operations to maintain data integrity and reduce redundancy.

In SQL, a natural join can be performed using the `NATURAL JOIN` keyword, though it's often recommended to use explicit join conditions (like `INNER JOIN` with `ON` clause) for clarity and to avoid unintentional errors in large or complex schemas:

sql Copy code

```sql
SELECT *
FROM Employees
NATURAL JOIN Departments;
```

This SQL command will produce a result set similar to the one described, seamlessly merging the related data based on the shared `DepartmentID` attribute.

The Natural-Join Operation

- The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.

⋈

- It is denoted by the join symbol
- The natural-join operation forms a Cartesian product of its two arguments,
- Performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

The Natural-Join Operation

ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

The natural join of the instructor relation with the teaches relation

The Natural-Join Operation

- The result relation, has only 13 tuples, the ones that give information about an instructor and a course that that instructor actually teaches.
- We do not repeat those attributes that appear in the schemas of both relations; rather they appear only once.
- Notice **also the order in which the attributes are listed**
  - ***first the attributes common to the schemas of both relations,***
  - ***second those attributes unique to the schema of the first relation, and***
  - ***finally, those attributes unique to the schema of the second relation.***

Assignment Operation →

In relational algebra, the **Assignment Operation** allows for the temporary naming and use of intermediate results in the computation of more complex queries. This operation is symbolized by the left arrow $\leftarrow$.

Syntax

The typical notation for the assignment operation in relational algebra is:

$$\text{Variable} \leftarrow \text{Expression}$$

where `Variable` is the name assigned to the result of the `Expression`.

Example

Let's consider a multi-step process in relational algebra where we need to use intermediate results to answer a more complex query. Suppose we want to find the names of employees who work in the "Engineering" department using our previous `Employees` and `Departments` tables.

Employees Table

EmployeeID	Name	Age	DepartmentID
1	Alice	30	1
2	Bob	22	1
3	Carol	34	2
4	David	29	3

Departments Table

DepartmentID	DepartmentName
1	HR
2	Engineering
3	Marketing

Operations

1. **Join Employees with Departments**:

$$\text{EmpDept} \leftarrow \text{Employees} \bowtie \text{Departments}$$

2. **Select Employees in Engineering**:

$$\text{EngEmp} \leftarrow \sigma_{\text{DepartmentName}=\text{'Engineering'}}(\text{EmpDept})$$

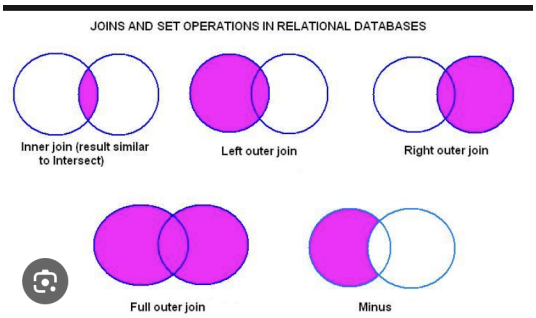
3. **Project the Names of These Employees**:

$$\text{Result} \leftarrow \pi_{\text{Name}}(\text{EngEmp})$$

Explanation

- **EmpDept**: This intermediate relation contains all employee records joined with their corresponding department records.
- **EngEmp**: This is a subset of `EmpDept`, filtered to include only those employees who work in the Engineering department.
- **Result**: This final result contains just the names of the employees from the Engineering department.

Outer join Operations →

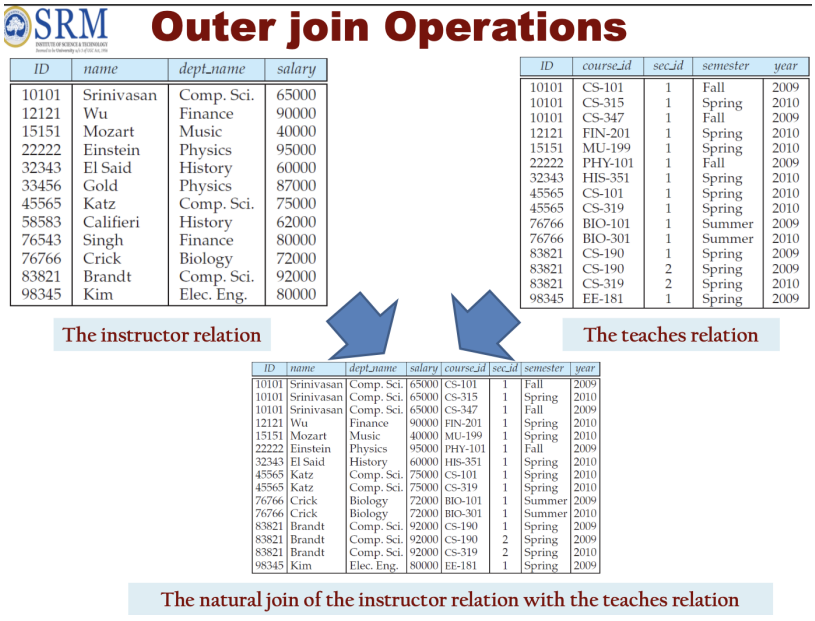


Outer joins in relational algebra are a type of join operation that address missing information in relations. They differ from the regular (inner) join by including rows from one or both tables even if there's no matching data in the other table. Here are the main types of outer joins:

1. **Left outer join (⟕):** This keeps all rows from the left (first mentioned) relation. For rows in the left relation that don't have a match in the right relation, the corresponding columns from the right relation are filled with null values.
2. **Right outer join (⟖):** This keeps all rows from the right (second mentioned) relation. Similar to the left outer join, unmatched rows from the left relation are filled with null values in the resulting table.
3. **Full outer join (⟗):** This combines the results of both left and right outer joins. It includes all rows from both relations, filling null values for unmatched attributes.

Outer join Operations

- An extension of the join operation to deal with missing information.
- Suppose that there is some instructor who teaches no courses.
- The tuple in the ***"instructor relation"*** for that particular instructor would not satisfy the condition of a natural join with the ***"teaches relation"***
- instructor's data would not appear in the result of the natural join, shown in Figure.



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

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Outer join Operations

- For example,
- instructors Califieri, Gold, and Singh do not appear in the result of the natural join, since they do not teach any course.

- More generally, some tuples in either or both of the relations being joined may be “lost” in this way.

- The outer join operation works in a manner similar to the natural join operation we have already studied, but preserves those tuples that would be lost in an join by creating tuples in the result containing null values.

Outer join Operations

- use the outer-join operation to avoid this loss of information.

- actually three forms of the operation:
  - ***left outer join, denoted*** ⟕
  - ***right outer join, denoted*** ***; and*** ⟖
  - ***full outer join, denoted*** ⟗

- All three forms of outer join compute the join, and add extra tuples to the result of the join.

- For example, the results of the expression instructor ⟕ teaches and teaches ⟖ instructor appear in Figures

ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
33456	Gold	Physics	87000	*null*	*null*	*null*	*null*
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
58583	Califieri	History	62000	*null*	*null*	*null*	*null*
76543	Singh	Finance	80000	*null*	*null*	*null*	*null*
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Result of $instructor \rtimes teaches$.

ID	course_id	sec_id	semester	year	name	dept_name	salary
10101	CS-101	1	Fall	2009	Srinivasan	Comp. Sci.	65000
10101	CS-315	1	Spring	2010	Srinivasan	Comp. Sci.	65000
10101	CS-347	1	Fall	2009	Srinivasan	Comp. Sci.	65000
12121	FIN-201	1	Spring	2010	Wu	Finance	90000
15151	MU-199	1	Spring	2010	Mozart	Music	40000
22222	PHY-101	1	Fall	2009	Einstein	Physics	95000
32343	HIS-351	1	Spring	2010	El Said	History	60000
33456	*null*	*null*	*null*	*null*	Gold	Physics	87000
45565	CS-101	1	Spring	2010	Katz	Comp. Sci.	75000
45565	CS-319	1	Spring	2010	Katz	Comp. Sci.	75000
58583	*null*	*null*	*null*	*null*	Califieri	History	62000
76543	*null*	*null*	*null*	*null*	Singh	Finance	80000
76766	BIO-101	1	Summer	2009	Crick	Biology	72000
76766	BIO-301	1	Summer	2010	Crick	Biology	72000
83821	CS-190	1	Spring	2009	Brandt	Comp. Sci.	92000
83821	CS-190	2	Spring	2009	Brandt	Comp. Sci.	92000
83821	CS-319	2	Spring	2010	Brandt	Comp. Sci.	92000
98345	EE-181	1	Spring	2009	Kim	Elec. Eng.	80000

Result of $teaches \ltimes instructor$.

Left Outer join Operations

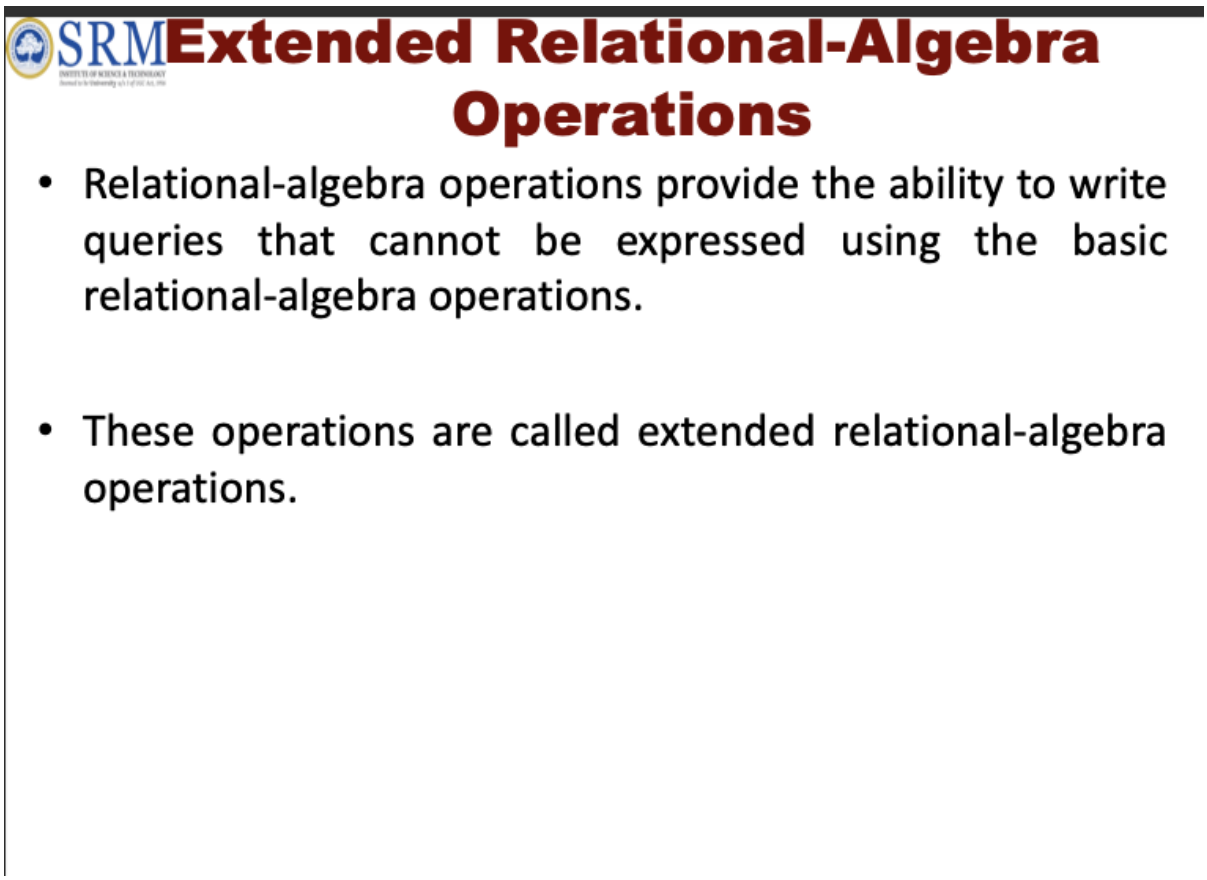
- The left outer join takes all tuples in the left relation
  - ***did not match with any tuple in the right relation,***
  - ***pads the tuples with null values for all other attributes from the right relation, and***
  - ***adds them to the result of the natural join.***
- In Figure, tuple (58583, Califieri, History, 62000, null, null, null, null), is such a tuple.
- All information from the left relation is present in the result of the left outer join.

Right Outer join Operations

- The right outer join is symmetric with the left outer join
- It pads tuples from the right relation that
  - ***did not match any from the left relation with nulls and***
  - ***adds them to the result of the natural join.***
- In Figure 6.18, tuple (58583, null, null, null, null, Califieri, History, 62000), is such a tuple.
- Thus, all information from the right relation is present in the result of the right outer join.

Full Outer join Operations

- The full outer join does both the left and right outer join operations,
  - ***padding tuples from the left relation that did not match any from the right relation,***
  - ***as well as tuples from the right relation that did not match any from the left relation, and***
  - ***adding them to the result of the join.***



Extended relational algebra includes operations beyond the basic set (select, project, union, set difference, and Cartesian product) typically taught in introductory database courses. These extended operations are designed to handle more complex queries and data manipulation tasks. Here are some of the key extended relational algebra operations

Generalized Projection →

Generalized Projection

Generalized Projection allows expressions to be calculated from existing attributes, potentially renaming them and creating new columns.

Example

Suppose we have a table `Employees(Name, Age, Salary)`. We want to create a new table that includes each employee's name and a calculated field "Age in five years."

Using generalized projection, the expression would be:
$\pi_{Name, Age+5 \text{ as } FutureAge}(Employees)$

This operation projects the `Name` and computes `Age + 5` (renaming it as `FutureAge`).

Generalized Projection

- Extends the projection operation by allowing operations such as arithmetic and string functions to be used in the projection list.
- The generalized-projection operation has the form

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

- where
  - ***E is any relational-algebra expression,***
  - ***each of F_1, F_2, . . . , F_n is an arithmetic expression involving constants and attributes in the schema of E.***

Generalized Projection

- Generalized projection also permits operations on other data types, such as concatenation of strings.

$$\Pi_{ID, name, dept\_name, salary \div 12}(instructor)$$

Aggregation →

Aggregation

Aggregation in relational algebra involves applying statistical functions like sum, average, maximum, minimum, and count over specified columns, possibly with grouping.

Example

Consider a table `Orders(CustomerID, OrderAmount)`. If we want to find the total `OrderAmount` for each `CustomerID`:

$$\gamma_{CustomerID;\text{SUM}(OrderAmount)\text{ as }TotalSpent}(Orders)$$

This operation groups the orders by `CustomerID` and calculates the sum of `OrderAmount` for each group, labeling it as `TotalSpent`.

Aggregation

- The second extended relational-algebra operation is the aggregate operation , $\mathcal{G}$
- Permits the use of aggregate functions such as min or average, on sets of values.
- Aggregate functions take a collection of values and return a single value as a result.
  - ***Sum, Average, Count, Min, Max***

The collections on which aggregate functions operate can have multiple occurrences of a value; the order in which the values appear is not relevant. *Such collections are called multisets*. Sets are a special case of multisets where there is only one copy of each element.

Find out the sum of salaries of all instructors

$$\mathcal{G}_{\text{sum}(salary)}(instructor)$$

The result of the expression is a relation with a single attribute, containing a single row with a numerical value corresponding to the sum of the salaries of all instructors.

Aggregation

- There are cases where we **must eliminate multiple occurrences of a value *before computing an aggregate function.***
- **If we do want to eliminate duplicates,**
  - ***use the same function names***
  - ***with the addition of the hyphenated string "distinct" appended to the end of the function name***
  - ***for example, count-distinct***

SRM

Find the total number of instructors who teach a course in the Spring 2010 semester.

$$\mathcal{G}_{\text{count-distinct}(ID)}(\sigma_{semester=\text{“Spring”} \wedge year\ =2010}(teaches))$$

The aggregate function “count-distinct” ensures that even if an instructor teaches ore than one course, she is counted only once in the result.

SRM

To apply the aggregate function not to a single set of tuples, but instead to a group of sets of tuples.

Find the average salary in each department

$$_{dept\_name}\mathcal{G}_{\text{average}(salary)}(instructor)$$

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

Tuples of the instructor relation, grouped by the dept name attribute

dept_name	salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

The result relation for the query “Find the average salary in each department”.

The Tuple Relational Calculus →

Tuple Relational Calculus

Tuple Relational Calculus is a non-procedural query language which, unlike relational algebra, focuses on what to retrieve rather than how to retrieve it. It uses logical formulas to describe queries.

Example

Consider a table `Employees(Name, DepartmentID, Salary)` and another table `Departments(DepartmentID, ManagerID)`. If we want to find the names of employees who work in the same department as their manager:

In tuple relational calculus, the query can be expressed as:
$\{e.Name \mid e \in Employees \land d \in Departments \land e.DepartmentID = d.DepartmentID \land e.ManagerID = d.ManagerID\}$

This expression states: "Select the `Name` from an employee tuple `e` and a department tuple `d` where the `DepartmentID` of both tuples matches and the `ManagerID` of the employee matches the `ManagerID` of the department."

These examples illustrate how these advanced concepts in relational algebra and tuple relational calculus can be applied to handle complex data retrieval needs efficiently.

The Tuple Relational Calculus

- When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query.
- The tuple relational calculus is a nonprocedural query language.
- It describes the desired information without giving a specific procedure for obtaining that information.
- A query in the tuple relational calculus is expressed as

$$\{t \mid P(t)\}$$

The Tuple Relational Calculus

- It is the **set of all tuples "t"** such that ***predicate "P" is true for "t".***
- Following our earlier notation, we use t[A] to denote the value of tuple "t" on attribute A, and we use t ∈ r to denote that tuple t is in relation r.

Find the ID, name, dept name, salary for instructors whose salary is greater than $80,000

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

Pitfalls in Relational Database →

- 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

1.1 Redundancy:

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

1.2 Inconsistency:

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

1.3 Inefficiency:

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

1.4 Complexity:

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

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

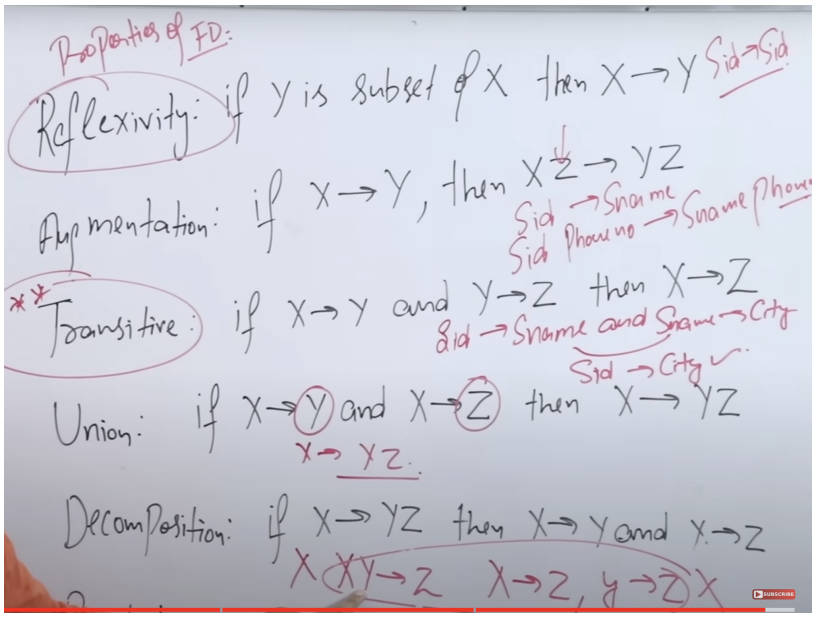
Decomposition process:

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

Functional Dependency →

Lec-23: Functional Dependency & its properties in DBMS in HINDI
👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

0:00 - Functional dependency
https://www.youtube.com/watch?v=qn5neFBpU40



Functional dependency is a fundamental concept in the theory of relational databases, crucial for understanding and designing database schemas, particularly in the normalization process. It describes a relationship between two sets of attributes in a relational database. Here's how it works:

Definition

A functional dependency in a relational database exists when one set of attributes, typically denoted as *A*, is a subset of attributes *B*. This is expressed as $A \rightarrow B$. This means that if two rows in a table have the same value for attribute A, they must also have the same value for attribute B.

Types of Functional Dependencies

1. **Trivial Dependency(intersection is not zero)**: A functional dependency $X \rightarrow Y$ is trivial if Y is a subset of X. Trivial dependencies do not provide new information because the value of Y is always determined by X if Y is part of X.
2. **Non-trivial Dependency(intersection is zero)**: A functional dependency is non-trivial if Y is not a subset of X. These are the dependencies that are typically of interest when normalizing databases.
3. **Transitive Dependency**: A functional dependency is transitive if there is an intermediate attribute Z such that $X \rightarrow Z$ and $Z \rightarrow Y$. Handling transitive dependencies is key in achieving higher normal forms in database normalization.

Importance in Database Design

Functional dependencies are used to ensure the integrity and accuracy of data in a relational database. They are particularly important in the normalization process, where the goal is to divide a database into well-structured tables that reduce redundancy and dependency without losing data integrity or access efficiency.

Library Database Example

Imagine a database with a table named `Books` that includes the following attributes:

- `ISBN`: International Standard Book Number, a unique identifier for books.
- `Title`: The title of the book.
- `AuthorID`: A unique identifier for authors.
- `AuthorName`: The name of the author.
- `Publisher`: The publisher of the book.

Functional Dependencies in the `Books` Table

1. **ISBN → Title, AuthorID, Publisher**
   - The ISBN of a book uniquely determines the Title, AuthorID, and Publisher. If you know the ISBN, you can uniquely identify the title of the book, which author wrote it (through AuthorID), and who published it. This is a non-trivial functional dependency.
2. **AuthorID → AuthorName**
   - The AuthorID uniquely determines the AuthorName. This means that each AuthorID is associated with exactly one AuthorName. If two books have the same AuthorID, they must have the same AuthorName. This is another non-trivial dependency and is crucial for ensuring consistency in the author's name across multiple books.

These functional dependencies are essential for maintaining data integrity in the database. They help ensure that the information is consistent across various entries and helps prevent data anomalies. For instance, by adhering to these dependencies, the database prevents situations where a single ISBN could have multiple titles or authors, which would be incorrect and confusing.

Using such functional dependencies effectively in database design helps in structuring the database efficiently, minimizing redundancy, and enhancing data integrity through normalization processes.

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

To analyze and determine functional dependencies within a database schema, certain rules and techniques are utilized. Among the most fundamental are **Armstrong's Axioms** and the concept of **Attribute Closure**. These tools are essential for reasoning about functional dependencies and for database normalization processes.

Armstrong's Axioms

Armstrong's Axioms are a set of rules that can be used to infer all the functional dependencies on a database schema. These axioms are sound (all dependencies derived from these rules are correct) and complete (they can be used to derive all the functional dependencies). The three basic axioms are:

1. **Reflexivity**: If $Y \subseteq X$ then $X \rightarrow Y$. This axiom states that a set of attributes can determine itself or any subset of itself.
2. **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any attribute set Z. This means if a set of attributes X functionally determines another set Y, then adding another attribute Z to both sides will still maintain the dependency.
3. **Transitivity**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. This rule allows the derivation of a new dependency: if one attribute set determines a second, which in turn determines a third, then the first set determines the third set.

From these basic axioms, several additional rules can be derived, such as **union**, **decom** and **pseudotransitivity**, which provide more tools to work with functional dependencies

Closure of Attributes

The closure of a set of attributes X, denoted as X^+, is the set of all attributes that can b functionally determined by X in a given schema. The attribute closure is helpful for understanding what information a given set of attributes can uniquely identify within a d

Steps to find the attribute closure X^+ include:

1. Start with $X^+ = X$.
2. For each functional dependency $A \rightarrow B$ in the set of all dependencies:
   - If $A \subseteq X^+$, add B to X^+.
3. Repeat step 2 until no more attributes can be added to X^+.

Example of Finding Attribute Closure:
Given the functional dependencies:

- $AB \rightarrow C$
- $C \rightarrow D$
- $D \rightarrow E$

To find the closure of AB, AB^+:

- Start with $AB^+ = AB$.
- Using $AB \rightarrow C$, add C to AB^+ (now $AB^+ = ABC$).
- Using $C \rightarrow D$, add D to AB^+ (now $AB^+ = ABCD$).
- Using $D \rightarrow E$, add E to AB^+ (now $AB^+ = ABCDE$).

This closure tells us that knowing AB allows us to uniquely determine C, D, and E in the database. This process is invaluable in normalization, especially when verifying candidate keys for tables and understanding dependency implications in schema design.

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.

EXAMPLE

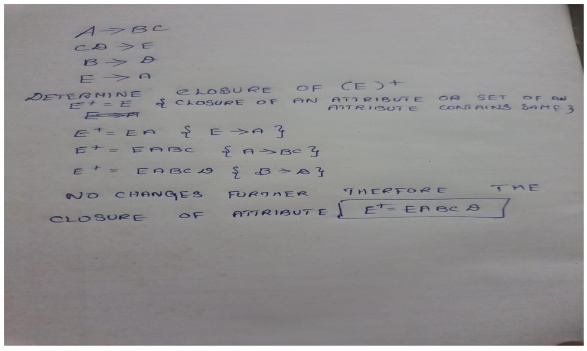
For relation $R = (A, B, C, G, H, I)$ and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. Find out the closure of functional dependency.

- $A \rightarrow H$. 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 $AG \rightarrow I$.

EXAMPLE

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



Decomposition →

In the context of relational databases, **decomposition** refers to the process of breaking down a database table into two or more tables to reduce redundancy and eliminate undesirable characteristics like insertion, update, and deletion anomalies. This is often part of the normalization process. **Lossless decomposition** is a specific type of decomposition that ensures that no information is lost and that the original table can be perfectly reconstructed by joining the decomposed tables.

Decomposition

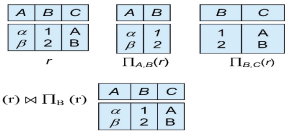
The main goal of decomposition is to achieve higher normal forms in order to reduce redundancy and improve data integrity. Decomposition involves splitting a table into two or more tables such that the new tables are each in a higher normal form or free of certain types of dependencies that cause anomalies.

Characteristics of Good Decomposition:

1. **Redundancy Reduction**: Helps eliminate duplicate data which reduces storage costs and improves update performance.
2. **Elimination of Update Anomalies**: Ensures that insertions, deletions, and updates can be performed without unintended consequences.

Examples of Lossless Decomposition

- Decomposition of $R = (A, B, C)$
  $R_1 = (A, B)$ $R_2 = (B, C)$



$\Pi_{A}(r) \bowtie \Pi_{B}(r)$

A	B	C
α	1	A
β	2	B

Lossless Decomposition

A decomposition is said to be lossless if the original table can be reconstructed without any loss of information by joining the decomposed tables. This is an essential property because it ensures that the decomposition has not distorted the data.

Conditions for Lossless Decomposition:

A decomposition of R into $R1$ and $R2$ is lossless if at least one of the following conditions holds:

1. **Attribute Preservation**: The intersection of $R1$ and $R2$, $R1 \cap R2$, functionally determines $R1$ or $R2$. This is also expressed by the condition:
   $(R1 \cap R2) \rightarrow R1$ or $(R1 \cap R2) \rightarrow R2$
2. **Dependency Preservation**: Every functional dependency $X \rightarrow Y$ that holds over the original schema also holds over one or more of the decomposed schemas, or can be inferred by them.

Example of Lossless Decomposition

Suppose you have a table $R(A, B, C)$ with dependencies $A \rightarrow B$ and $B \rightarrow C$. A possible decomposition into $R1(A, B)$ and $R2(B, C)$ would be lossless if you can join $R1$ and $R2$ on B to get back the original table R. Here, the condition $B \rightarrow B$ (which is trivially true) implies that joining on B will perfectly reconstruct the original table.

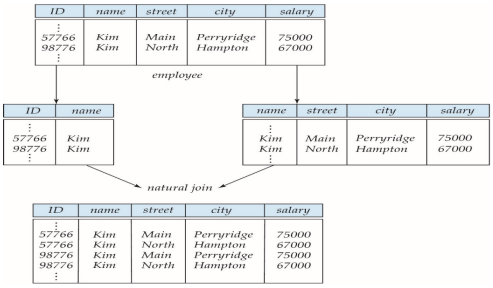
The ability to decompose a database schema losslessly is crucial for ensuring that normalization improves database design without compromising the ability to retrieve original data accurately. This aspect is vital for database designers to consider when structuring or re-structuring a database.

Lossy decomposition is the counterpart to lossless decomposition in the context of database normalization. In lossy decomposition, when a table is split into two or more tables and these tables are then rejoined, some information may be lost or spurious tuples (rows) may be generated. This means that the join of the decomposed tables does not exactly match the original table, which can lead to inaccuracies and data integrity issues.

Characteristics of Lossy Decomposition

1. **Information Loss**: When the decomposed tables are rejoined, the resulting table may contain fewer rows (loss of information) or more rows (introduction of spurious tuples) than the original table.
2. **Spurious Tuples**: These are rows that appear in the joined table but did not exist in the original table. Spurious tuples are usually a result of insufficient overlap in the data used to link the tables, indicating that the common attributes between the tables do not preserve the necessary dependencies.

A Lossy Decomposition



Lossy decomposition is the counterpart to lossless decomposition in the context of database normalization. In lossy decomposition, when a table is split into two or more tables and these tables are then rejoined, some information may be lost or spurious tuples (rows) may be generated. This means that the join of the decomposed tables does not exactly match the original table, which can lead to inaccuracies and data integrity issues.

Characteristics of Lossy Decomposition

1. **Information Loss**: When the decomposed tables are rejoined, the resulting table may contain fewer rows (loss of information) or more rows (introduction of spurious tuples) than the original table.
2. **Spurious Tuples**: These are rows that appear in the joined table but did not exist in the original table. Spurious tuples are usually a result of insufficient overlap in the data used to link the tables, indicating that the common attributes between the tables do not preserve the necessary dependencies.

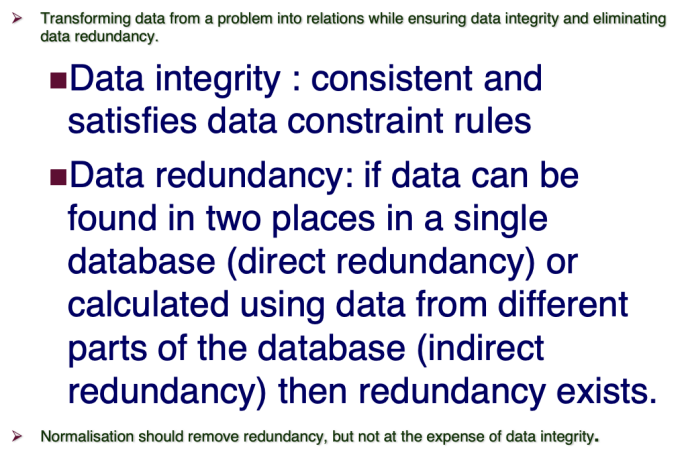
Normalization →

Lec-20: Introduction to Normalization | Insertion, Deletion & Updation Anomaly
👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

0:00 - Introduction
https://youtu.be/5GDTIUVIHB8?si=kiRROvm2s_1Te1PO

Normalization

- The process of normalization is a formal method that identifies relational schemas based upon their primary or candidate keys and the functional dependencies that exists amongst their attributes.
- Normalization is primarily a tool to validate and improve a logical design so that it satisfies certain constraints that *avoid unnecessary duplication of data.*
- Normalization is the process of decomposing relation with anomalies to produce smaller, *well-structured* relations.
- Normalisation should remove redundancy, but not at the expense of data integrity.



Normalization is a systematic approach used in relational database design to minimize redundancy, avoid undesirable characteristics like insertion, update, and deletion anomalies, and ensure data integrity and efficiency. It involves decomposing larger tables into smaller, well-structured tables that better define relationships among the data. This process helps in maintaining consistency and can improve the performance of the database.

Goals of Normalization

1. **Eliminate Redundant Data**: Reduce or eliminate duplicate data to save storage and improve data integrity.
2. **Minimize Data Modification Issues**: Avoid problems such as update, insert, and delete anomalies.
3. **Improve Data Integrity**: Ensure the accuracy and consistency of the database by establishing and maintaining relationships between tables through referential integrity.
4. **Optimize Queries**: Make the database structure more efficient for retrieval operations and reduce the complexity of database queries.

IMP Keys ->

1. **Candidate Key**
   - **Definition**: A candidate key is a minimal set of attributes necessary to uniquely identify a tuple (record) in a table. It is minimal in the sense that omitting any attribute from the candidate key means it will no longer uniquely identify records.
   - **Example**: Consider a table with attributes A, B, C, D where any combination of A and B uniquely identifies a record. Therefore, AB could be a candidate key. If A or B alone can also uniquely identify all records, each would individually be a candidate key as well.
2. **Primary Key**
   - **Definition**: A primary key is a candidate key selected by the database designer to uniquely identify rows in a table. A primary key must have unique and not null values.
   - **Example**: From the candidate keys A, B, and AB in a table, if the designer chooses A as the primary key, then A is used for all unique identification and indexing in the table.

3. **Super Key**
   - **Definition**: A super key is a set of one or more attributes that, taken collectively, can uniquely identify a tuple in a table. A super key may contain additional attributes that are not necessary for unique identification.
   - **Example**: If A is a primary key, then any combination including A — such as A, B, A, C, A, B, C, etc. — is a super key.
4. **Prime Key**
   - **Definition**: A prime attribute is an attribute that is part of any candidate key. An attribute that is part of a candidate key is called a prime attribute.
   - **Example**: If the candidate keys are A and AB, then both A and B are prime attributes.
5. **Non-Prime Key**
   - **Definition**: A non-prime attribute is not part of any candidate key.
   - **Example**: If the candidate keys in a table are A and AB, and the attributes of the table are A, B, C, D, then C and D are non-prime attributes since they are not included in any candidate key.

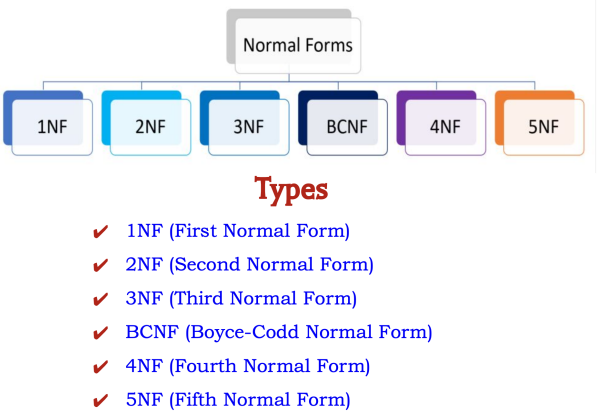
Practical Example Illustration

Imagine a university database with a table called `Students`:

StudentID	Name	Email	Major
1001	Alice	alice@example.com	Math
1002	Bob	bob@example.com	English
1003	Charlie	charlie@example.com	Math

- **Candidate Keys**: Assuming each `StudentID` and `Email` is unique across all students, both `StudentID` and `Email` could serve as candidate keys.
- **Primary Key**: The database designer chooses `StudentID` as the primary key for its simplicity and the fact it is an integer.
- **Super Keys**: Any combination including `StudentID` (e.g., `StudentID, Name`, `StudentID, Email, Major`) or `Email` with any other attributes.
- **Prime Attributes**: `StudentID` and `Email` are prime since they are part of candidate keys.
- **Non-Prime Attributes**: `Name` and `Major` are non-prime as they are not part of any candidate key.

Stages of Normalization



Normalization involves several stages, called "normal forms" (NF), each of which reduces specific types of redundancy and dependency:

1. **First Normal Form (1NF):**

Lec-21: First Normal form in DBMS in HINDI | 1st Normal form क्या होती है ?
👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

▶Database Management System(Complete Playlist):
https://www.youtube.com/watch?v=NIgZy30Dv9A&list=PLxCzCOWd7aiFAN6I8CuViBuCdJgiOkT2Y&index=22

First Normal Form (1NF)

Requirement: Ensure that all table columns have atomic (indivisible) values, and each record needs to be unique.

Example:
Suppose you have a table that tracks customer purchases:

CustomerID	CustomerName	PurchasedItems
1	Alice	Pen, Notebook
2	Bob	Pencil, Eraser, Ruler

This table is not in 1NF because the `PurchasedItems` column contains multiple values. To bring the table into 1NF, you would redesign it to:

CustomerID	CustomerName	PurchasedItem
1	Alice	Pen
1	Alice	Notebook
2	Bob	Pencil
2	Bob	Eraser
2	Bob	Ruler

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

- ✔ 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

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.

2. **Second Normal Form (2NF)**:

Lec-24: Second Normal Form | 2NF | Database Management System

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

▶ Database Management System(Complete Playlist):

https://www.youtube.com/watch?v=tkbAA--wKOc&list=PLxCzCOWd7aiFAN6I8CuViBuCdJgiOkT2Y&index=25

Second Normal Form (2NF)

Requirement: All requirements for 1NF plus removing partial dependencies (a non-key attribute is functionally dependent on part of the primary key).

Example:
Continuing with the modified table from 1NF:

CustomerID	CustomerName	PurchasedItem
1	Alice	Pen
1	Alice	Notebook
2	Bob	Pencil
2	Bob	Eraser
2	Bob	Ruler

Here, `CustomerName` is dependent only on `CustomerID`, which is a partial dependency because `CustomerID` is part of a potential composite key (`CustomerID`, `PurchasedItem`). To normalize to 2NF:

- **Customers Table:**

CustomerID	CustomerName
1	Alice
2	Bob

- **Purchases Table:**

CustomerID	PurchasedItem
1	Pen
1	Notebook
2	Pencil
2	Eraser
2	Ruler

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

3. **Third Normal Form (3NF)**:

Lec-25: Third Normal Form in dbms with examples in Hindi | Normalization

👉Subscribe to our new channel:https://www.youtube.com/@varunainashots

0:00 - Introduction

https://www.youtube.com/watch?v=IeSai2JVm78

3NF in DBMS | Third Normal Form - javatpoint

3NF in DBMS | Third Normal Form with DBMS Overview, DBMS vs Files System, DBMS Architecture, Three schema Architecture, DBMS Language, DBMS Keys, DBMS Generalization, DBMS Specialization, Relational Model concept, SQL Introduction, Advantage of SQL, DBMS Normalization, Functional Dependency, DBMS Schedule, Concurrency Control etc.

https://www.javatpoint.com/dbms-third-normal-form

- **Requirement**: Meet all the criteria of 2NF, and remove transitive dependencies, i.e., non-key attributes must not depend on other non-key attributes.

Third Normal Form (3NF)

Requirement: All requirements for 2NF plus removing transitive dependencies (a non-key attribute depends on another non-key attribute).

Example:
Add a `Location` field to the Customers Table to record the customer's city.

- **Customers Table**:

CustomerID	CustomerName	Location
1	Alice	New York
2	Bob	Los Angeles

Assuming a functional dependency where `Location` is determined by `CustomerName` and not by `CustomerID` directly, this creates a transitive dependency. To normalize to 3NF, create separate tables:

- **Customers Table**:

CustomerID	CustomerName
1	Alice
2	Bob

- **Locations Table**:

CustomerName	Location
Alice	New York
Bob	Los Angeles

project_no	manager	address
p1	Black,B	32 High Street
p2	Smith,J	11 New Street
p3	Black,B	32 High Street
p4	Black,B	32 High Street

Project has more than one non-key field so we must check for transitive dependency:

- **Now we need to store the address only once**
- **If we need to know a manager's address we can look it up in the Manager relation**
- **The manager attribute is the link between the two tables, and in the Projects table it is now a foreign key.**
- **These relations are now in third normal form.**

Project

project_no	manager
p1	Black,B
p2	Smith,J
p3	Black,B
p4	Black,B

Manager

manager	address
Black,B	32 High Street
Smith,J	11 New Street

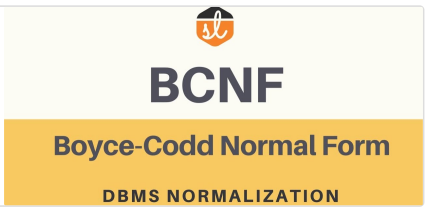
4. **Boyce-Codd Normal Form (BCNF)**:

Boyce-Codd Normal Form (BCNF) | Database Normalization | DBMS

For complete DBMS tutorial: https://www.studytonight.com/dbms/

In this video, you will learn about the Boyce-Codd Normal Form, which is popularly

https://youtu.be/NNjUhvvwOrk?si=m1YdJbbs4KqS_3Az



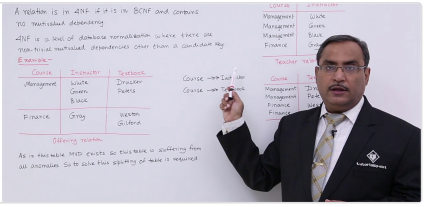
- **Requirement**: A stronger version of 3NF. Every determinant must be a candidate key.
- **Example**: Addressing cases where 3NF is insufficient to eliminate anomalies, especially when there are multiple overlapping candidate keys.

5. **Fourth Normal Form (4NF)**:

DBMS - Fourth Normal Form 4NF

DBMS - Fourth Normal Form 4NF
Watch more Videos at https://www.tutorialspoint.com/videotutorials/index.htm
Lecture By: Mr. Arnab Chakraborty, Tutorials Point India Private Limited

https://youtu.be/OBiNTE14EEg?si=2iylUbdMS-Do1C3m



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

Example

STUDENT

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

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data.

So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_COURSE

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

STUDENT_HOBBY

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

Consider a table `StudentCoursesTeachers` that records which teachers have taught which courses to which students:

StudentID	CourseID	TeacherID
1	Math	Dr. Smith
1	Math	Dr. Jones
2	Science	Dr. Lee
2	Science	Dr. Kahn

Here, `StudentID` and `CourseID` are together a candidate key, but the `TeacherID` has a multi-valued dependency on `CourseID` independent of `StudentID`. This means a course could be taught by multiple teachers, which should not be intermixed with who is taking what course.

To bring this table to 4NF, it should be decomposed into two:

- **StudentCourses**: Records which student takes what course.

StudentID	CourseID
1	Math
2	Science

- **CourseTeachers**: Records which teachers teach what course.

CourseID	TeacherID
Math	Dr. Smith
Math	Dr. Jones
Science	Dr. Lee
Science	Dr. Kahn

- **Requirement**: Meet all the criteria of BCNF, and no table should have multi-valued dependencies.
- **Example**: If a table holds two or more independent multi-valued facts about an entity, split the table.

1. **Fifth Normal Form (5NF)**:

Fifth Normal Form-5NF

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

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

Example

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

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

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

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

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

P2

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

P3

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

- **Requirement**: Meet all the criteria of 4NF, and every join dependency in the table should be implied by the candidate keys.
- **Example**: Further decomposition of tables to eliminate redundancy that is not addressed by 4NF.

Importance of Normalization

Normalization helps in designing efficient databases that are free from many common data management errors. By ensuring that each piece of information is stored only once, the chances of inconsistent data are minimized, which enhances data integrity. Furthermore, normalization can lead to performance improvements, especially in complex queries, by reducing the need for extensive table joins and maintaining a more focused and organized data structure.

Overall, normalization is a crucial process in database design and management, providing a foundation for building robust and scalable databases.

Multivalue Dependency →

A **multivalued dependency** (MVD) occurs in database systems when one attribute in a table can determine multiple independent values of another attribute, without any relation to the other attributes in the table. This concept is particularly important in the context of database normalization, especially when aiming to achieve the Fourth Normal Form (4NF).

Definition of Multivalued Dependency

Formally, a multivalued dependency $X \rightarrow\rightarrow Y$ exists in a relation R if, for each pair of tuples in R that agree on the attributes X, the tuples also agree on the attributes Y. More intuitively, once the value of X is known, the entire set of values for Y can be determined, independently of the values of other attributes in the relation.

Characteristics of Multivalued Dependencies

1. **Independence of Other Attributes**: The key aspect of MVDs is that the attribute Y is dependent on X but this dependency is independent of other attributes in the relation.
2. **Non-Functional**: Unlike functional dependencies where one value of X determines exactly one value of Y, a multivalued dependency allows X to determine a set of values for Y.
3. **Symmetry**: If $X \rightarrow\rightarrow Y$ is a valid MVD, then $X \rightarrow\rightarrow (R - Y - X)$ is also valid, where R is the set of all attributes in the relation.

Example of Multivalued Dependency

Imagine a university database with a table that tracks which professors have taught which courses and which textbooks have been used for each course:

Professor	Course	Textbook
Dr. Smith	Calculus	"Math Basics"
Dr. Smith	Calculus	"Advanced Math"
Dr. Jones	Physics	"Physics 101"
Dr. Jones	Physics	"Quantum Mech"

In this table:

- **MVD 1**: `Course ->-> Textbook`
  - Knowing the course (e.g., Calculus), we can determine all possible textbooks (e.g., "Math Basics", "Advanced Math"), independent of the professor teaching the course.
- **MVD 2**: `Professor ->-> Course`
  - Knowing the professor (e.g., Dr. Smith), we can determine all courses they might teach (e.g., Calculus), independent of the specific textbooks used.

Implications in Database Design

- **Normalization to 4NF**: To resolve multivalued dependencies and reduce redundancy, the relation might be decomposed into smaller relations. Each new relation represents a single MVD.
- **Reduction of Redundancy**: Decomposition helps eliminate redundancy that arises from MVDs. For example, splitting the above table into two—one linking professors to courses and another linking courses to textbooks—would eliminate unnecessary repetition of course or textbook information.
- **Example Decomposition**:
  - **Professor-Courses**:

Professor	Course
Dr. Smith	Calculus
Dr. Jones	Physics

  - **Course-Textbooks**:

Course	Textbook
Calculus	"Math Basics"
Calculus	"Advanced Math"
Physics	"Physics 101"
Physics	ntum Mech"