Database Management System (DBMS)

akc notes

Module 3

1. Basic Operators in Relational Algebra

Relational Algebra is a procedural query language that takes relations as an input and returns relations as an output.

Selection operator (σ): Selection operator is used to selecting tuples from a relation based on some condition.

Syntax: σ (Cond)(Relation Name)

Projection Operator (∏): Projection operator is used to project particular columns from a relation.

Syntax: ∏(Column 1,Column 2….Column n)(Relation Name)

Cross Product(X): Cross product is used to join two relations. For every row of Relation1, each row of Relation2 is concatenated. If Relation1 has m tuples and Relation2 has n tuples, cross product of Relation1 and Relation2 will have m X n tuples.

Syntax: Relation1 X Relation2

Union (U): Union on two relations R1 and R2 can only be computed if R1 and R2 are union compatible (These two relations should have the same number of attributes and corresponding attributes in two relations have the same domain). Union operator when applied on two relations R1 and R2 will give a relation with tuples that are either in R1 or in R2. The tuples which are in both R1 and R2 will appear only once in the result relation.

Syntax: Relation1 U Relation2

Minus (-): Minus on two relations R1 and R2 can only be computed if R1 and R2 are union compatible. Minus operator when applied on two relations as R1-R2 will give a relation with tuples that are in R1 but not in R2.

Syntax: Relation1 - Relation2

Rename(ρ): Rename operator is used to giving another name to a relation.

Syntax: ρ(Relation2, Relation1).

Example: To rename STUDENT relation to STUDENT1, we can use rename operator like: ρ(STUDENT1, STUDENT).

2. Union Compatibility or Type Compatibility

i. We must ensure that the input relations to the union operation have the same number of attributes; the number of attributes of a relation is referred to as its arity.

ii. When the attributes have associated types, the types of the i^{th} attributes of both input relations must be the same, for each i.

Such relations are referred to as compatible relations.

Two relations $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_n)$ are said to be union compatible (or type compatible) if they have the same degree n and if $dom(A_i) = dom(B_i)$ for $1 \leq i \leq n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and S as follows:

UNION: The result of this operation, denoted by R ∪ S, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
INTERSECTION: The result of this operation, denoted by R ∩ S, is a relation that includes all tuples that are in both R and S.
SET DIFFERENCE (or MINUS): The result of this operation, denoted by R – S, is a relation that includes all tuples that are in R but not in S.
Note: R ∩ S = ((R ∪ S) – (R – S)) – (S – R) **[How to perform ∩ operation using basic operators?]**

3. Join Operation

Join is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied.
Theta (θ) Join: Theta join combines tuples from different relations provided they satisfy the theta condition. The join condition is denoted by the symbol θ. The general form of a JOIN operation on two relations $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_m)$ is

$$R \bowtie_{\text{<join condition>}} S$$

The result of the JOIN is a relation Q with (n + m) attributes $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—whenever the combination satisfies the join condition.
Note: Consider relations r(R) and s(S), and let θ be a predicate on attributes in the schema R ∪ S. The join operation $r \bowtie_\theta s$ is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

Equijoin: When Theta join uses only equality comparison operator, it is said to be equijoin. The above example corresponds to equijoin.

Natural Join (⋈): Natural join does not use any comparison operator. It does not concatenate the way a Cartesian product does. We can perform a Natural Join only if there is at least one common attribute that exists between two relations. In addition, the attributes must have the same name and domain.

Outer Joins
An inner join includes only those tuples with matching attributes and the rest are discarded in the resulting relation. Therefore, we need to use outer joins to include all the tuples from the participating relations in the resulting relation. There are three kinds of outer joins – left outer join, right outer join, and full outer join.

Left Outer Join(R ⟕ S): All the tuples from the Left relation, R, are included in the resulting relation. If there are tuples in R without any matching tuple in the Right relation S, then the S-attributes of the resulting relation are made NULL.

Right Outer Join $(R \bowtie S)$: All the tuples from the Right relation, S, are included in the resulting relation. If there are tuples in S without any matching tuple in R, then the R-attributes of resulting relation are made NULL.

Full Outer Join $(R \bowtie S)$: All the tuples from both participating relations are included in the resulting relation. If there are no matching tuples for both relations, their respective unmatched attributes are made NULL.

4. Database Design

Database design may be performed using two approaches: bottom-up or top-down.

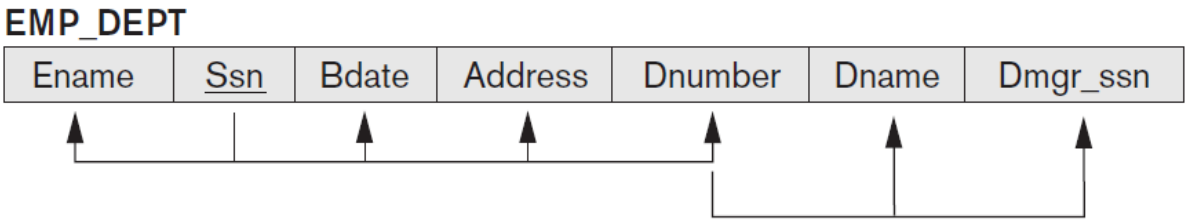
Bottom-up Approach: A bottom-up design methodology (also called **design by synthesis**) considers the basic relationships among individual attributes as the starting point and uses those to construct relation schemas.

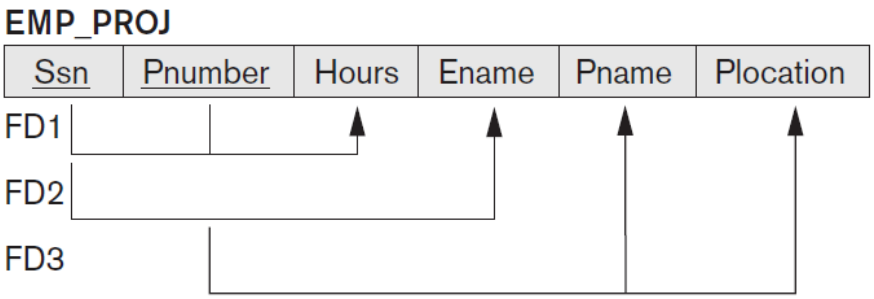
Problems: This approach is not very popular in practice1 because it suffers from the problem of having to collect a large number of binary relationships among attributes as the starting point. For practical situations, it is next to impossible to capture binary relationships among all such pairs of attributes.

Top-down Approach: Top-down design methodology (also called **design by analysis**) starts with a number of groupings of attributes into relations that exist together naturally, for example, on an invoice, a form, or a report. The relations are then analyzed individually and collectively, leading to further decomposition until all desirable properties are met.

5. Anomalies in Database

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space. Storing natural joins of base relations leads to an additional problem referred to as update anomalies. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.





Insertion Anomalies: Insertion anomalies can be differentiated into two types, illustrated by the following examples:

i. To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs (if the employee does not work for a department as yet).
ii. It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the attributes for employee. This violates the entity integrity for EMP_DEPT because its primary key "Ssn" cannot be null. Moreover, when the first employee is assigned to that department, we do not need this tuple with NULL values anymore.

Deletion Anomalies: The problem of deletion anomalies is related to the second insertion anomaly situation just discussed. If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost inadvertently from the database.

Modification Anomalies: In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of all employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.

6. Functional Dependencies

A **functional dependency**, denoted by X → Y, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are determined by, the values of the X component. We also say that there is a functional dependency from X to Y, or that Y is functionally dependent on X. The abbreviation for functional dependency is FD or f.d. The set of attributes X is called the left-hand side of the FD, and Y is called the right-hand side.

Thus, X functionally determines Y in a relation schema R if, and only if, whenever two tuples of r(R) agree on their X-value, they must necessarily agree on their Y-value.

Note:

i. If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation instance r(R)—that is, X is a candidate key of R—this implies that X → Y for any subset of attributes Y of R (because the key constraint implies that no two tuples in any legal state r(R) will have the same value of X). If X is a candidate key of R, then X → R.
ii. If X → Y in R, this does not say whether or not Y → X in R.

7. Normalization

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies.

Normal Form: The normal form of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

The process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:

i. The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation does not occur with respect to the relation schemas created after decomposition.
ii. The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

The nonadditive join property is extremely critical and must be achieved at any cost, whereas the dependency preservation property, although desirable, is sometimes sacrificed.

Denormalization: Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

8. Prime Attribute and Non-Prime Attribute

An attribute of relation schema R is called a prime attribute of R if it is a member of some candidate key of R. An attribute is called nonprime if it is not a prime attribute—that is, if it is not a member of any candidate key.

9. First Normal Form

A domain is atomic if elements of the domain are considered to be indivisible units. We say that a relation schema R is in first normal form (1NF) if the domains of all attributes of R are atomic.

Definition: It states that the domain of an attribute must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute.
Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple. In other words, 1NF disallows relations within relations or relations as attribute values within tuples. The only attribute values permitted by 1NF are single atomic (or indivisible) values.

10. Closure of a Set of Functional Dependencies

Given a relation schema r(R), a functional dependency f on R is logically implied by a set of functional dependencies F on R if every instance of a relation r(R) that satisfies F also satisfies f.
Let F be a set of functional dependencies. The **closure of F**, denoted by F^+, is the set of all functional dependencies logically implied by F.

Steps or Procedure for computing Closure of F (F^+)

$F^+ = F$
apply the reflexivity rule /* Generates all trivial dependencies */
repeat
    for each functional dependency f in F^+
        apply the augmentation rule on f
        add the resulting functional dependencies to F^+
    for each pair of functional dependencies f_1 and f_2 in F^+
        if f_1 and f_2 can be combined using transitivity
            add the resulting functional dependency to F^+
until F^+ does not change any further

11. Armstrong's Axioms

Axioms, or rules of inference, provide a simpler technique for reasoning about functional dependencies. We use αβ to denote α ∪ β.

Reflexivity rule: If α is a set of attributes and β ⊆ α, then α → β holds.
Augmentation rule: If α → β holds and γ is a set of attributes, then γα → γβ holds.
Transitivity rule: If α → β holds and β → γ holds, then α → γ holds.

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

Union rule: If α → β holds and α → γ holds, then α → βγ holds.
Decomposition rule: If α → βγ holds, then α → β holds and α → γ holds.
Pseudotransitivity rule: If α → β holds and γβ → δ holds, then αγ → δ holds.

12. Closure of Attribute Sets

Let α be a set of attributes. We call the set of all attributes functionally determined by α under a set F of functional dependencies the closure of α under F; we denote it by α^+.

Definition: For each such set of attributes X, we determine the set X^+ of attributes that are functionally determined by X based on F; X^+ is called the closure of X under F.

Algorithm: To compute α^+, the input is a set F of functional dependencies and the set α of attributes. The output is stored in the variable result.

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

Uses of the attribute closure algorithm

i. To test if α is a superkey, we compute α^+ and check if α^+ contains all attributes in R.
ii. We can check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), by checking if $\beta \subseteq \alpha^+$. That is, we compute α^+ by using attribute closure, and then check if it contains β.
iii. It gives us an alternative way to compute F^+: For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

13. Equivalence of Sets of Functional Dependencies

A set of functional dependencies F is said to **cover** another set of functional dependencies E if every FD in E is also in F^+; that is, if every dependency in E can be inferred from F; alternatively, we can say that E is covered by F.

Two sets of functional dependencies E and F are **equivalent** if $E^+ = F^+$. Therefore, equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to F if both the conditions—E covers F and F covers E—hold.

14. Canonical Cover

An attribute of a functional dependency is said to be extraneous if we can remove it without changing the closure of the set of functional dependencies.

Definition: We consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
Removal from the left side: Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
Removal from the right side: Attribute A is extraneous in β if $A \in \beta$and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

Let R be the relation schema, and let F be the given set of functional dependencies that hold on R. Consider an attribute A in a dependency $\alpha \rightarrow \beta$.

i. If $A \in \beta$, to check if A is extraneous, consider the set
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
and check if $\alpha \rightarrow A$ can be inferred from F'. To do so, compute α^+ (the closure of α) under F'; if α^+ includes A, then A is extraneous in β.

ii. If $A \in \alpha$, to check if A is extraneous, let $\gamma = \alpha - \{A\}$, and check if $\gamma \rightarrow \beta$ can be inferred from F. To do so, compute γ^+ (the closure of γ) under F; if γ^+ includes all attributes in β, then A is extraneous in α.

Algorithm

$F_c = F$

repeat

Use the union rule to replace any dependencies in F_c of the form $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.

Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous attribute either in α or in β.

/* Note: the test for extraneous attributes is done using F_c, not F */

If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in F_c.

until (F_c does not change)

A canonical cover F_c for F is a set of dependencies such that F logically implies all dependencies in F_c, and Fc logically implies all dependencies in F. Furthermore, Fc must have the following properties:

i. No functional dependency in Fc contains an extraneous attribute.

ii. Each left side of a functional dependency in F_c is unique. That is, there are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in Fc such that $\alpha_1 = \alpha_2$.

15. Algorithm To Determine the Key of a Relation

Input: A relation R and a set of functional dependencies F on the attributes of R.

i. Set K := R.

ii. For each attribute A in K
{compute $(K - A)^+$ with respect to F;
if $(K - A)^+$ contains all the attributes in R, then set K := K - {A} };

16. Dependency Preservation Property of a Decomposition

Definition: Given a set of dependencies F on R, the projection of F on R_i, denoted by $\pi R_i(F)$ where R_i is a subset of R, is the set of dependencies X

→ Y in F^+ such that the attributes in X ∪ Y are all contained in R_i. Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all the left- and right-hand-side attributes of those dependencies are in Ri. We say that a decomposition D = {R_1, R_2, … , R_m} of R is dependency-preserving with respect to F if the union of the projections of F on each R_i in D is equivalent to F; that is,

$$((\pi R_1(F)) \cup K \cup (\pi R_m(F)))^+ = F^+$$

17. Nonadditive (Lossless) Join Property of a Decomposition

Definition: Formally, a decomposition D = {R_1, R_2, … , R_m} of R has the lossless (nonadditive) join property with respect to the set of dependencies F on R if, for every relation state r of R that satisfies F, the following holds, where * is the NATURAL JOIN of all the relations in D: *($\pi R_1(r)$, … , $\pi R_m(r)$) = r.

18. Second Normal Form

Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency X → Y is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold anymore.

A functional dependency X → Y is a **partial dependency** if some attribute A ε X can be removed from X and the dependency still holds; that is, for some A ε X, (X – {A}) → Y.

Definition: A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.

Note: A relation schema R is in second normal form (2NF) if every nonprime attribute A in R is not partially dependent on any key of R.

19. Third Normal Form

Third normal form (3NF) is based on the concept of transitive dependency. A functional dependency X → Y in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, and both X → Z and Z → Y hold.

Definition: A relation schema R is in 3NF if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key.

Note: A relation schema R is in third normal form (3NF) if, whenever a nontrivial functional dependency X → A holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R.

20. Example 1 – Normalization

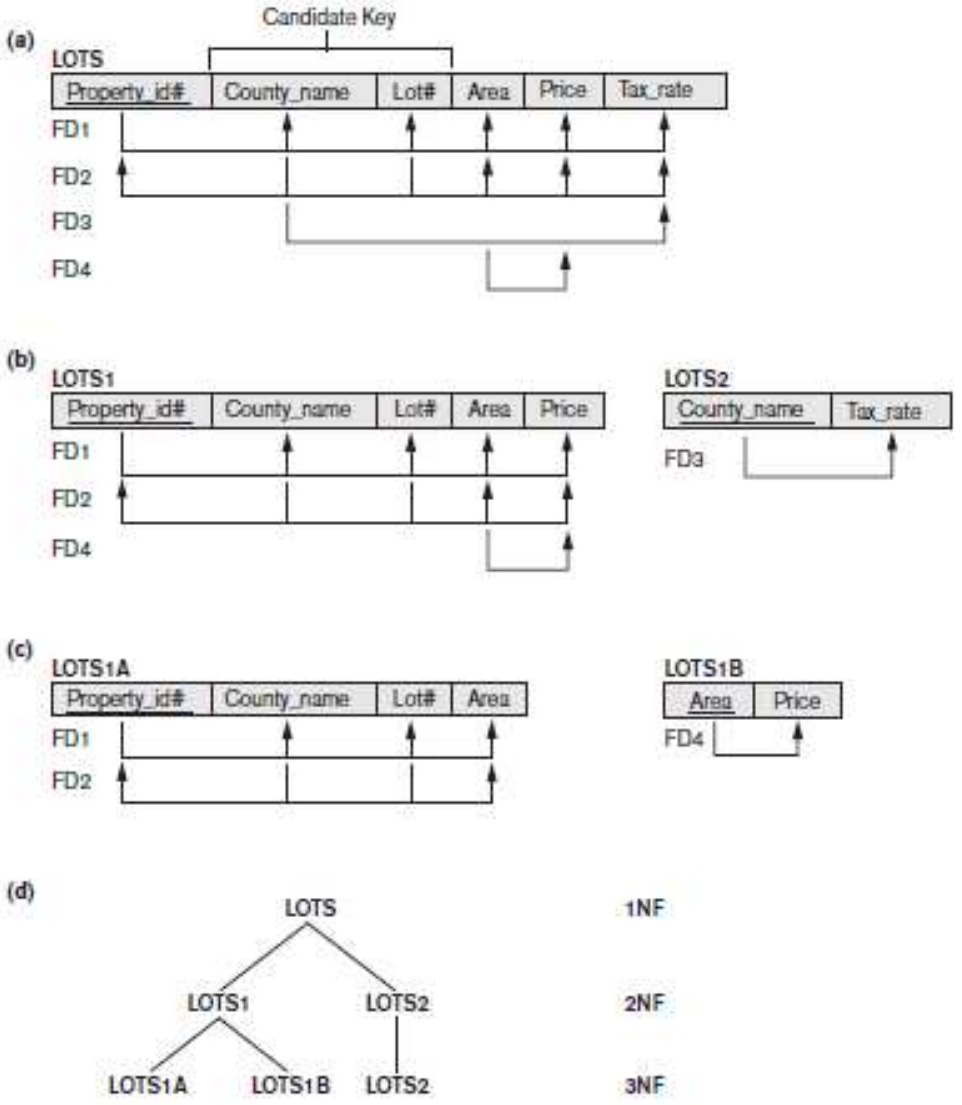
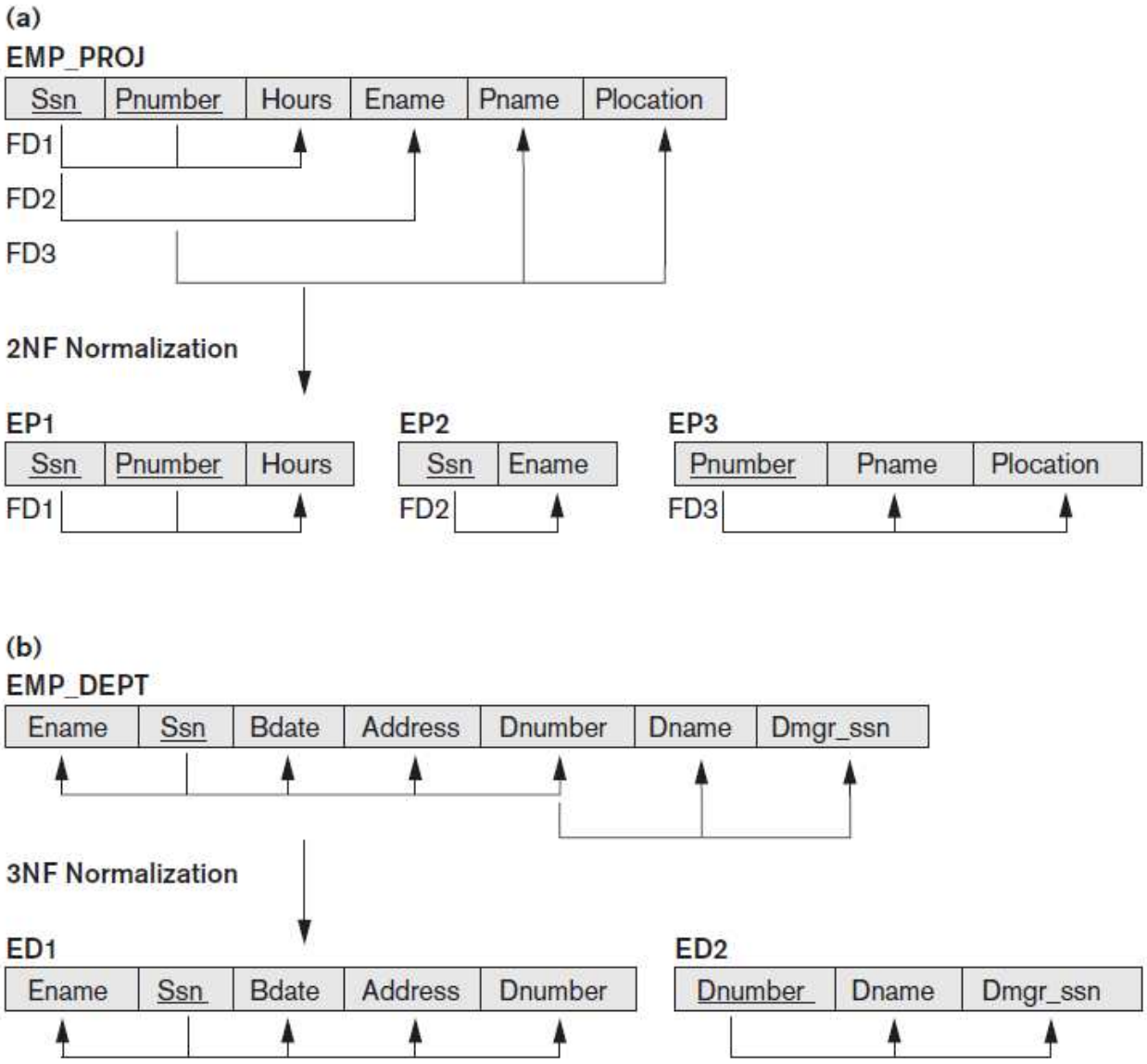


Figure: Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Progressive normalization of LOTS into a 3NF design.

21. Example 2 – Normalization

Figure: Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.



22. Difference between 1NF and 2NF

S.NO.	1NF	2NF
1.	In order to be in 1NF any relation must be atomic and should not contain any composite or multi-valued attributes.	In order to be in 2NF any relation must be in 1NF and should not contain any partial dependency.
2.	The identification of functional dependency is not necessary for first normal form.	The identification of functional dependency is necessary for second normal form.
3.	First Normal form only deals with the schema of the table and it does not handle the update anomalies.	Second normal form handles the update anomalies.
4.	A relation in 1NF may or may not be in 2NF.	A relation in 2NF is always in 1NF.
5.	The primary key in case of first normal form can be a composite key.	The primary key in case of second normal form cannot be a composite

		key in case it arises any partial dependency.
6.	The main goal of first normal form is to eliminate the redundant data within the table.	The main goal of second normal form is to actually ensure the data dependencies.
7.	The first normal form is less stronger than the second normal form.	The second normal form is comparatively more strong than first normal form.

23. Difference between 2NF and 3NF

S.NO.	2NF(Second Normal Form)	3NF(Third Normal Form)
1.	It is already in 1NF.	It is already in 1NF as well as in 2NF also.
2.	In 2NF non-prime attributes are allowed to be functionally dependent on non-prime attributes.	In 3NF non-prime attributes are only allowed to be functionally dependent on Super key of relation.
3.	No partial functional dependency of non-prime attributes are on any proper subset of candidate key is allowed.	No transitive functional dependency of non-prime attributes on any super key is allowed. .
4.	Stronger normal form than 1NF but lesser than 3NF	Stronger normal form than 1NF and 2NF.
5.	It eliminates repeating groups in relation.	It virtually eliminates all the redundancies.
6.	The goal of the second normal form is to eliminate redundant data.	The goal of the third normal form is to ensure referential integrity.

24. Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF.

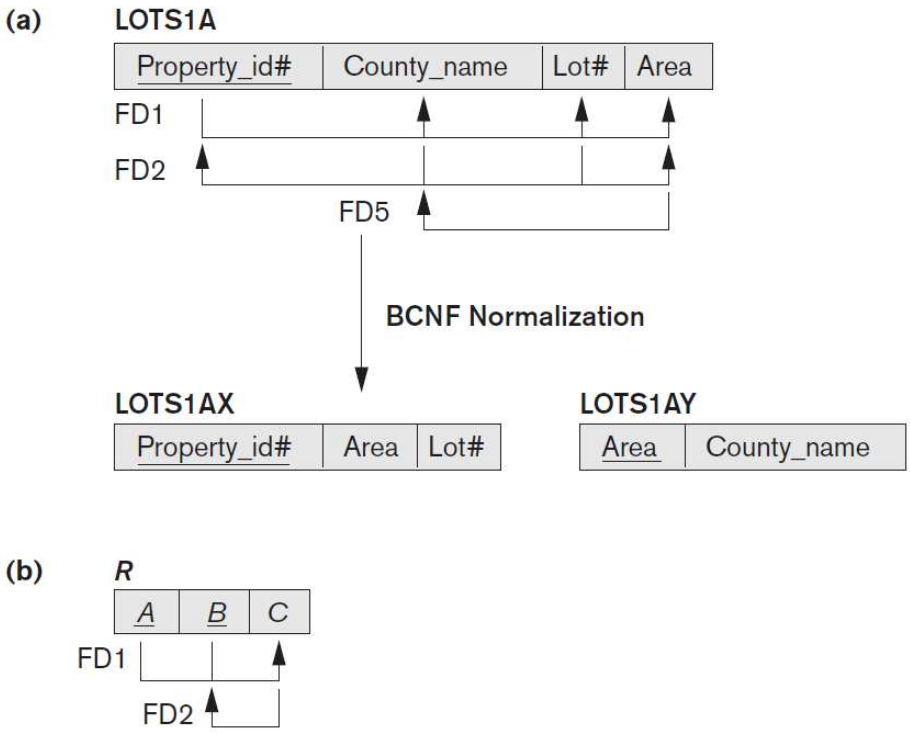
Definition: A relation schema R is in BCNF if whenever a nontrivial functional dependency X → A holds in R, then X is a superkey of R.

Example:

Figure:

(a): BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition.

(b): A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d. C → B.



Note: BCNF is a stronger form of normalization than 3NF because it eliminates the second condition for 3NF, which allowed the right side of the FD to be a prime attribute. Thus, every left side of an FD in a table must be a superkey. Every table that is BCNF is also 3NF, 2NF, and 1NF, by the previous definitions.

25. Difference between 3NF and BCNF

S.NO.	3NF	BCNF
1.	In 3NF there should be no transitive dependency that is no non prime attribute should be transitively dependent on the candidate key.	In BCNF for any relation A->B, A should be a super key of relation.
2.	It is less stronger than BCNF.	It is comparatively more stronger than 3NF.
3.	In 3NF the functional dependencies are already in 1NF and 2NF.	In BCNF the functional dependencies are already in 1NF, 2NF and 3NF.
4.	The redundancy is high in 3NF.	The redundancy is comparatively low in BCNF.
5.	In 3NF there is preservation of all functional dependencies.	In BCNF there may or may not be preservation of all functional dependencies.
6.	It is comparatively easier to achieve.	It is difficult to achieve.
7.	Lossless decomposition can be achieved by 3NF.	Lossless decomposition is hard to achieve in BCNF.