UNIT-3

Data Base Design & Normalization

Normalization

The database normalization is a technique used to refine a database structure after the initial design has been developed. It reduces the level of redundancy in a relational database structure. The key idea behind normalization is to reduce the chance of having multiple different versions of the same data by storing all duplicate data in different tables and linking to them instead of using a copy. Then updating the address in one place will instantly change all the places where the address is used.

Normalization is part of the successful database design without normalization; database systems may be inaccurate, slow and inefficient.

The normalization process was first proposed by CODD, takes a relational schema through a series of tests to certify whether or not it belongs to a certain normal forms. Basically, the normal form of the data indicates how much redundancy is in that data. Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.

Anomalies in Database

Anomaly refers to inconsistent data i.e. usually caused by unnecessary redundancy. Data anomalies exist because any change in any field value must be correctly made in many places. There are three main anomalies:

1. Insertion Anomaly

2. Deletion Anomaly

3. Modification Anomaly

1. Insertion Anomaly

An insertion anomaly is a failure to place information about a new database entry into all the places in the database where information about that new entry needs to be stored.

2. Deletion Anomaly

A deletion anomaly is a failure to remove information about an existing database entry when it is removed from database.

3. Modification Anomaly

A modification anomaly is a failure to modify information about an existing database entry at all places in the database when information about that entry is modified.

Functional Dependencies

A functional dependency, denoted by $X \rightarrow Y$, 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 $t1$ and $t2$ in r that have $t1[X] = t2[X]$, they must also have $t1[Y] = t2[Y]$.
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 Y is **functionally dependent** on X.

A functional dependency is a property of the **semantics** or **meaning of the attributes**. The main use of functional dependencies is to describe further a relation schema R by specifying constraints on its attributes that must hold *at all times.*

Let us consider the relation r

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

Observe that $A \rightarrow C$ is satisfied.
There are two tuples that have an A value of $a1$. These tuples have the same C value—namely, $c1$. Similarly, the two tuples with an A value of $a2$ have the same C value, $c2$. There are no other pairs of distinct tuples that have the same A value. The functional dependency $C \rightarrow A$ is not satisfied, however r satisfies $AB \rightarrow C$.

Some functional dependencies are said to be **trivial** because they are satisfied by all relations. For example, $A \rightarrow A$ is satisfied by all relations involving attribute A. Reading the definition of functional dependency literally, we see that, for all tuples $t1$ and $t2$ such that $t1[A] = t2[A]$, it is the case that $t1[A] = t2[A]$. Similarly, $AB \rightarrow A$ is satisfied by all relations involving attribute A. In general, a functional dependency of the form $\alpha \rightarrow \beta$ is **trivial** if $\beta \subseteq \alpha$.

Closure of a Set of Functional Dependencies

Given a set F of functional dependencies, we can prove that certain other functional dependencies hold. We say that such functional dependencies are "logically implied" by F.

More formally, given a relational schema R, a functional dependency f on R is **logically implied** by a set of functional dependencies F on R if every relation instance $r(R)$ 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. We can use the following three rules to find logically implied functional dependencies. By applying these rules *repeatedly*, we can find all of F^+, given F.

Armstrong's axioms

• **Reflexivity rule**. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.

• **Augmentation rule**. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.

• **Transitivity rule**. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

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

Additional Rules

• **Union rule**. If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.

• **Decomposition rule**. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.

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

Example: Given schema $R = (A, B, C, G, H, I)$ and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$.

We list several members of F^+ here:
• $A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule. Observe that it was much easier to use Armstrong's axioms to show that $A \rightarrow H$ holds.
• $CG \rightarrow HI$. 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 pseudotransitivity rule implies that $AG \rightarrow I$ holds.

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

Closure of Attribute Sets

We say that an attribute β is functionally determined by α, if $\alpha \rightarrow \beta$. To Test whether a set α is a super key, we have to device an algorithm for computing the set of attributes functionally determined by α.

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^+.

Algorithm:

Result:= α
While(changes to result) do
    For each functional dependency $\beta \rightarrow \gamma$ in F do
    Begin
        If $\beta \subseteq$ *result then result:=result* $\cup \gamma$*;*
    End

Example: Given schema $R = (A, B, C, G, H, I)$ and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$.

We shall use it to compute $(AG)^+$ with the functional dependencies. We start with *result* = AG. The first time that we execute the **while** loop to test each functional dependency, we find that

• $A \rightarrow B$ causes us to include B in *result*. To see this fact, we observe that $A \rightarrow B$ is in F, $A \subseteq$ *result* (which is AG), so *result*:= *result* $\cup$ B.

• $A \rightarrow C$ causes *result* to become $ABCG$.

• $CG \rightarrow H$ causes *result* to become $ABCGH$.

• $CG \rightarrow I$ causes *result* to become $ABCGHI$.

The second time that we execute the **while** loop, no new attributes are added to *result*, and the algorithm terminates.

To test if α is a super key, we compute α^+, and check if α^+ contains all attributes of R.

Canonical Cover

A **minimal cover** of a set of functional dependencies E is a set of functional dependencies F that satisfies the property that every dependency in E is in the closure F^+ of F. In addition, this property is lost if any dependency from the set F is removed; F must have no redundancies in it, and the dependencies in F are in a standard form. To satisfy these properties, we can formally define a set of functional dependencies F to be **minimal** if it satisfies the following conditions:

1. Every dependency in F has a single attribute for its right-hand side.

2. We cannot replace any dependency $X \rightarrow A$ in F with a dependency $Y \rightarrow A$, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.

3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F.

We can always find *at least one* minimal cover F for any set of dependencies E using Algorithm.

Algorithm. Finding a Minimal Cover F for a Set of Functional Dependencies E

Input: A set of functional dependencies E.

1. Set F:= E.

2. Replace each functional dependency $X \rightarrow \{A_1, A_2, ..., A_n\}$ in F by the n functional dependencies $X \rightarrow A_1, X \rightarrow A_2, ..., X \rightarrow A_n$.

3. For each functional dependency $X \rightarrow A$ in F

for each attribute B that is an element of X

if { {F – {$X \rightarrow A$} } ∪ { (X – {B}) →A} } is equivalent to F

then replace $X \rightarrow A$ with (X – {B}) →A in F.

4. For each remaining functional dependency $X \rightarrow A$ in F

if {F – {$X \rightarrow A$} } is equivalent to F,

then remove $X \rightarrow A$ from F.

Example: Let the given set of FDs be E: {$B \rightarrow A$, $D \rightarrow A$, $AB \rightarrow D$}. We have to find the minimal cover of E.

■ All above dependencies are in canonical form (that is, they have only one attribute on the right-hand side), so we have completed step 1 of Algorithm and can proceed to step 2. In step 2 we need to determine if $AB \rightarrow D$ has any redundant attribute on the left-hand side; that is, can it be replaced by $B \rightarrow D$ or $A \rightarrow D$?

■ Since B →A, by augmenting with B on both sides (IR2), we have $BB \rightarrow AB$, or $B \rightarrow AB$ (i). However, $AB \rightarrow D$ as given (ii).

■ Hence by the transitive rule (IR3), we get from (i) and (ii), $B \rightarrow D$. Thus $AB \rightarrow D$ may be replaced by $B \rightarrow D$.

■ We now have a set equivalent to original E, say E': {$B \rightarrow A$, $D \rightarrow A$, $B \rightarrow D$}. No further reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.

■ In step 3 we look for a redundant FD in E'. By using the transitive rule on $B \rightarrow D$ and $D \rightarrow A$, we derive $B \rightarrow A$. Hence $B \rightarrow A$ is redundant in E' and can be eliminated.

■ Therefore, the minimal cover of E is {$B \rightarrow D$, $D \rightarrow A$}.

Properties of Relational Decompositions

1. The non-additive (or lossless) join decomposition

2. The dependency preservation

1. The non-additive (or lossless) join decomposition

Let R be the relation schema with instance 'r' decomposed into R_1, R_2, ... , R_n with instances r_1, r_2, ... , r_n respectively. We say that the decomposition is a lossless decomposition if for all legal database instances

$$r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n = r$$

In other words, if we compute the natural join of the decomposed relation instances (r_1, r_2, ... , r_n) then we get back exactly r.

A decomposition that is not lossless decomposition is called lossy decomposition, if

$$r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n \subseteq r$$

then it is a lossy join decomposition.

For Example:
Let there be a relational schema R(A, B, C). R1(A, B) and R2(A, C) be it's decompositions.

R

A	B	C
a1	b1	c1
a2	b1	c1
a1	b2	c2
a1	b3	c3

R1

A	B
a1	b1
a2	b1
a1	b2
a1	b3

R2

A	C
a1	c1
a2	c1
a1	c2
a1	c3

R1 ⋈ R2

A	B	C
a1	b1	c1
a1	b1	c2
a1	b1	c3
a2	b1	c1
a1	b2	c1
a1	b2	c2
a1	b2	c3
a1	b3	c1
a1	b3	c2
a1	b3	c3

This decomposition is lossy as $R \subset R1 \bowtie R2$.

We can use functional dependency to show when certain decompositions are lossless. Let F be a set of functional dependency on a relation schema R, and R1 & R2 be the relation decompositions of R. This decomposition is a lossless-join decomposition of *R* if at least one of the following functional dependencies is in F^+:

• $R1 \cap R2 \rightarrow R1$

• $R1 \cap R2 \rightarrow R2$

In other words, if $R1 \cap R2$ forms a superkey of either *R*1 or *R*2, the decomposition of *R* is a lossless-join decomposition.

Example: R = (A, B, C, D, E). We decompose it into R1 = (A, B, C), R2 = (A, D, E). The set of functional dependencies is: A → BC, CD → E, B → D, E → A. Show that this decomposition is a lossless-join decomposition.

R1 ∩ R2 = A
A^+ = {A, B, C, D}

A is a super key of R1. So, this decomposition is lossless-join decomposition.

Example: R = (V, W, X, Y, Z). We decompose it into R1 = (V, W, X), R2 = (V, Y, Z). The set of functional dependencies is: Z → V, W → Y, XY → Z, V→ WX.

R1 ∩ R2 = V
V^+ = {V, W, X, Y, Z}
V is a super key of both R1 and R2. So, this decomposition is lossless-join decomposition.

Example: R = (V, W, X, Y, Z). We decompose it into R1 = (V, W, X), R2 = (X, Y, Z). The set of functional dependencies is: Z → V, W → Y, XY → Z, V→ WX.

R1 ∩ R2 = X
X^+ = {X}

V is not a super key of either R1 or R2. So, this decomposition is lossy.

Matrix Method

Algorithm. Testing for Non-additive Join Property

Input: A universal relation *R*, a decomposition *D* = {*R*1, *R*2, ..., *Rm*} of *R*, and a set *F* of functional dependencies.
Note: Explanatory comments are given at the end of some of the steps. They follow the format: (* *comment* *).

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

4. Repeat the following loop until a *complete loop execution* results in no changes to *S*
{for each functional dependency *X*→*Y* in *F*
{for all rows in *S that have the same symbols* in the columns corresponding to attributes in *X*
{make the symbols in each column that correspond to an attribute in *Y* be the same in all these rows as follows: If any of the rows has an *a* symbolfor the column, set the other rows to that *same a* symbol in the column. If no *a* symbol exists for the attribute in any of the rows, choose one of the *b* symbols that appears in one of the rows for the attribute and set the other rows to that same *b* symbol in the column ;} ; } ;};

5. If a row is made up entirely of *a* symbols, then the decomposition has the non-additive join property; otherwise, it does not.

Figure 16.1
Nonadditive join test for *n*-ary decompositions. (a) Case 1: Decomposition of EMP_PROJ into EMP_PROJ1 and EMP_LOCS fails test. (b) A decomposition of EMP_PROJ that has the lossless join property. (c) Case 2: Decomposition of EMP_PROJ into EMP, PROJECT, and WORKS_ON satisfies test.

(a) R = {Ssn, Ename, Pnumber, Pname, Plocation, Hours} $D = \{R_1, R_2\}$
R_1 = EMP_LOCS = {Ename, Plocation}
R_2 = EMP_PROJ1 = {Ssn, Pnumber, Hours, Pname, Plocation}

F = {Ssn → Ename; Pnumber → {Pname, Plocation}; {Ssn, Pnumber} → Hours}

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R_2	a_1	b_{22}	a_3	a_4	a_5	a_6

(No changes to matrix after applying functional dependencies)

(b)

EMP

Ssn	Ename

PROJECT

Pnumber	Pname	Plocation

WORKS_ON

Ssn	Pnumber	Hours

(c) R = {Ssn, Ename, Pnumber, Pname, Plocation, Hours} $D = \{R_1, R_2, R_3\}$
R_1 = EMP = {Ssn, Ename}
R_2 = PROJ = {Pnumber, Pname, Plocation}
R_3 = WORKS_ON = {Ssn, Pnumber, Hours}

F = {Ssn → Ename; Pnumber → {Pname, Plocation}; {Ssn, Pnumber} → Hours}

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

(Original matrix S at start of algorithm)

	Ssn	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	~~b_{32}~~ a_2	a_3	~~b_{34}~~ a_4	~~b_{35}~~ a_5	a_6

(Matrix S after applying the first two functional dependencies; last row is all "a" symbols so we stop)

2. The dependency preservation

It would be useful if each functional dependency $X \rightarrow Y$ specified in F either appeared directly in one of the relation schemas R_i in the decomposition D or could be inferred from the

dependencies that appear in some R_i. This is the *dependency preservation condition.* We want to preserve the dependencies because each dependency in F represents a constraint on the database. If one of the dependencies is not represented in some individual relation R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation.

It is not necessary that the exact dependencies specified in F appear themselves in individual relations of the decomposition D.

Let F be a set of functional dependencies on a schema R, and let R_1, R_2, . . . , R_n be a decomposition of R with functional dependency set F_1, F_2, . . . , F_n respectively.

In general, $\{F_1 \cup F_2 \cup \ldots \cup F_n\} \subseteq F$

If $\{F_1 \cup F_2 \cup \ldots \cup F_n\} = F$, then it is dependency preserving.

If $\{F_1 \cup F_2 \cup \ldots \cup F_n\} \subset F$, then it is not dependency preserving.

If decomposition is not dependency preserving, some dependency is lost in the decomposition.

Problem: Let a relation R (A, B, C, D) and functional dependency {AB –> C, C –> D, D –> A}. Relation R is decomposed into R1(A, B, C) and R2(C, D). Check whether decomposition is dependency preserving or not.

R1(A, B, C) and R2(C, D)

Let us find closure of F1 and F2

To find closure of F1, consider all combination of ABC. i.e., find closure of A, B, C, AB, BC and AC

Note ABC is not considered as it is always ABC

closure(A) = { A } // Trivial

closure(B) = { B } // Trivial

closure(C) = {C, A, D} but D can't be in closure as D is not present R1.

= {C, A}

C--> A // Removing C from right side as it is trivial attribute

closure(AB) = {A, B, C, D}

= {A, B, C}

AB --> C // Removing AB from right side as these are trivial attributes

closure(BC) = {B, C, D, A}

= {A, B, C}

BC --> A // Removing BC from right side as these are trivial attributes

closure(AC) = {A, C, D}

AC --> D // Removing AC from right side as these are trivial attributes

F1 {C--> A, AB --> C, BC --> A}.

Similarly F2 { C--> D }

In the original Relation Dependency { AB --> C , C --> D , D --> A}.

AB --> C is present in F1.

C --> D is present in F2.

D --> A is not preserved.

F1 U F2 is a subset of F. So given decomposition is not dependency preserving.

Some Important Terminologies

1. Fully Functional Dependency

Given a relation R and functional dependency X → Y, Y is fully functional dependent on X when there should not be any Z → Y, where Z is a proper set of X.

2. Partial Functional Dependency

If any proper subsets of the key determine any of the non-key attributes then there exists a partial dependency.

Example: R(A, B, C, D, E)

AB → CDE , here key is AB

Then,

A → C, A → D, A→ E, B → C, B→ D, B → E all are partial functional dependencies.

3. Transitive Dependency

If in a given relation where the determinant consists of non-key attributes and the determined attribute is also a non-key attributes then this dependency is transitive dependency.

Example: R(A, B, C, D, E)

Then dependency like A → B, B → C is a transitive dependency since A → C is implied.

4. Prime Attributes

An attribute of relation schema R is called a prime attribute of R if it is a member of some candidate key of R.

5. Non-Prime Attributes

An attribute of relation schema R is called a non-prime attribute of R if it is not a member of any candidate key of R.

6. Trivial and Non-trivial Functional Dependency

A functional dependency Y → X is trivial functional dependency, if X ⊆ Y. Otherwise, it is non-trivial.

Normal Forms

Normalization of data can be looked upon as a process of analyzing the given relation schemas based on their functional dependencies and primary keys to achieve the desirable properties of: Minimizing redundancy and minimizing the insertion, deletion and updation.

The normal forms 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 together, should possess the two properties:

1. The lossless join or non-additive join property, which guarantees that the spurious tuple generation problems does not occur with respect to the relation schemas created after decomposition.

2. The dependency preservation which ensure that each functional dependency is represented in some individual relations resulting after decomposition.

The non-additive join property is extremely critical and must be achieved at any cost whereas the dependency preservation property, although desirable, is sometimes sacrifices.

First Normal Form

First normal form (1NF) is considered to be part of the formal definition of a relation in the basic (flat) relational model;
It disallows multivalued attributes, composite attributes, and their combinations.

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**.
For Example: Consider the DEPARTMENT relation; this is not in 1NF because Dlocations is not an atomic attribute.

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

1NF version of the same relation with redundancy.

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it.*

For Example:

(a)

EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

Each tuple represents an employee entity, and a relation PROJS(Pnumber, Hours) *within each tuple* represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

EMP_PROJ(Ssn, Ename, {PROJS(Pnumber, Hours)})

To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation.

EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

NOTE: 1NF has high redundancy. It is not removing redundancy but it is the base of all normal forms.

Second Normal Form

Second normal form (2NF) is based on the concept of *full functional dependency.*

A relation schema *R* is in 2NF:

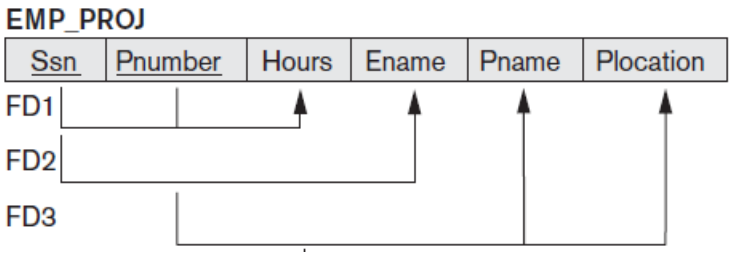
- If it satisfies all the characteristics of 1NF.
- if every nonprime attribute *A* in *R* is *fully functionally dependent* on the primary key of *R.*

OR

- if every non-prime attribute A in R is not partially dependent on any key of R.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all.

For Example:



The above functional dependencies can be restated as:

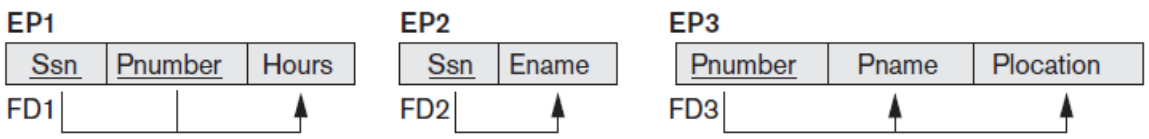
FD1: ssn, Pnumber → Hours

FD2: ssn → Ename

FD3: Pnumber → Pname, Plocation

Here key is {ssn Pnumber}. The functional dependency FD2 and FD3 make Ename, Pname and Plocation partially dependent on the key {ssn Pnumber}. Thus, violating the 2NF test.

If a relation schema is not in 2NF, it can be *second normalized* or *2NF normalized* into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3, each of which is in 2NF.



To summarize the process of decomposing non 2NF relation schema into many 2NF relation schema follow the following steps:

1. Create a new relation by using attributes from the offending functional dependency as attributes of new relation.

2. The attributes on the RHS of functional dependencies are eliminated from original relation.

3. If there are more than one offending functional dependency with same LHS then relations corresponding to these functional dependency are combined to have only one relation with all the attributes of these functional dependencies.

Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ 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.

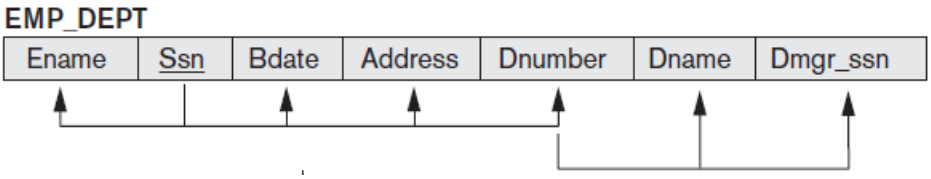
A relation R is said to be in 3NF:

- if it is in 2NF.
- If no non-prime attribute of R is transitively dependent on the key.

OR

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

For Example:



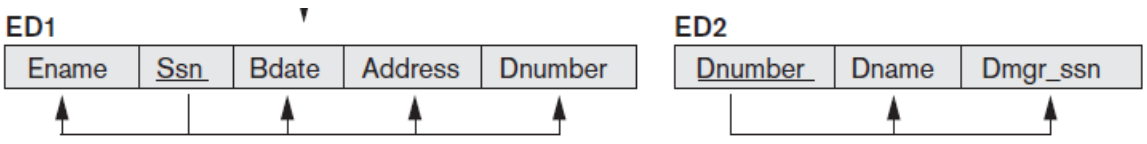
The above functional dependencies can be restated as:

FD1: ssn → Ename, bdate, address, Dnumber

FD2: Dnumber → Dname, Dmgr_ssn

Candidate key for the relation EMP_DEPT is {ssn}

The above relation is in 2NF as there is no partial dependency. EMP_DEPT is not in 3NF because of the transitive dependency of Dnumber → Dname, Dmgr_ssn. We can normalize EMP_DEPT by decomposing it into the two relations ED1 and ED2.

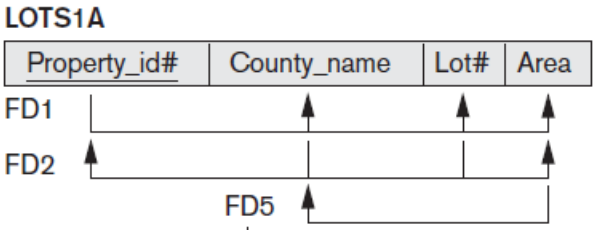


Boyce-Codd Normal Form

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.

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*.

For Example:



The above functional dependencies can be restated as:

FD1: Property_id# → county_name, Lot#, Area

FD2: County_name Lot# → Area, Property_id#

FD3: Area → County_name

Keys for the relation are {property_id, (county_name Lot#)}. As there is no partial dependency and transitive dependency, so the relation is in 2NF & 3NF respectively. FD3 Area → County_name violates BCNF because Area is not a superkey . So, we decompose LOTS1A into two BCNF relations.

This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3 NF are also in BCNF. Only if X→A holds in a relation schema R with X not being a super key and A being a prime attribute will R be in 3NF but not in BCNF.

Multivalued Dependency

Multivalued dependencies are a consequence of first normal form (1NF), which disallows an attribute in a tuple to have a *set of values*, and the accompanying process of converting an unnormalized relation into 1NF.

If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For Example: consider the relation EMP shown in Figure(a). A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname. An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another. To keep the relation state consistent, and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is specified as a multivalued dependency on the EMP relation, which we define in this section. Informally, whenever two *independent* 1:N relationships *A*:*B* and *A*:*C* are mixed in the same relation, *R*(*A*, *B*, *C*), an MVD may arise.

(a) **EMP**

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

EMP relation has two MVDs Ename →→ Pname and Ename →→ Dname.

Definition of Multivalued Dependency

A multivalued dependency $X \rightarrow\rightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties where we use Z to denote $(R - (X \cup Y))$:

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

Whenever $X \rightarrow\rightarrow Y$ holds, we say that X **multidetermines** Y.

An MVD X →→ Y in R is called a **trivial MVD** if (a) Y is a subset of X, or (b) $X \cup Y = R$.

Note: Relations containing nontrivial MVDs tend to be **all-key relations**—that is, their key is all their attributes taken together. And, an all key relation is always in BCNF.

Inference Rules Multivalued Dependencies

As with functional dependencies (FDs), inference rules for multivalued dependencies (MVDs) have been developed. Assume that all attributes are included in a "universal" relation schema $R = \{A_1\ A_2, \ldots, A_n\}$ and that X, Y, Z, and W are subsets of R.

IR1: (Complementation rule for MVDs): $\{X \rightarrow\rightarrow Y\} \Rightarrow \{X \rightarrow\rightarrow (R - (X \cup Y))\}$.

IR2: (Augmentation rule for MVDs): If $X \rightarrow\rightarrow Y$ and $W \supseteq Z$, then $WX \rightarrow\rightarrow YZ$.

IR3 (Transitive rule for MVDs): {X→→ Y, Y→→ Z} ⇒ X →→ (Z - *Y*).

IR4 (Replication rule for FD to MVD): {X → Y} ⇒ X →→ Y.

IR5 (Coalescence rule for FDs and MVDs): If X →→ Y and there exists W with the properties that (a) W ∩ Y is empty, (b) W → Z, and (c) Y ⊇ Z, then X →→ Z.

Fourth Normal Form

A relation is in 4NF, if it is in BCNF and it contains no multi-valued dependencies.

A relation schema R is in 4NF with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every nontrivial multivalued dependency X →→ Y in F^+, X is a superkey for R.

Note: The definition of 4NF differs from the definition of BCNF in only the use of multivalued dependencies.

Example: The EMP relation of figure (a) has no functional dependency since it is an all-key relation, an all key relation is always in BCNF by default. Hence, EMP is in BCNF. However, EMP is not in 4NF, because in the non-trivial MVDs Ename →→ Pname and Ename →→ Dname, Ename is not a super key of EMP.

We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS.

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs Ename →→ Pname in EMP_PROJECTS and Ename →→ Dname in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS.

Join Dependency

A **join dependency** (**JD**), denoted by JD(*R*1, *R*2, ..., *Rn*), specified on relation schema *R*, specifies a constraint on the states *r* of *R*. The constraint states that every legal state *r* of *R* should have a nonadditive join decomposition into R_1, R_2, ..., R_n. Hence, for every such *r* we have

$$\bowtie(\pi_{R1}(r), \pi_{R2}(r), \ldots, \pi R_n(r)) = r$$

OR

$$\pi_{R1}(r) \bowtie \pi_{R2}(r) \bowtie \ldots \bowtie \pi_{Rn}(r) = r$$

Notice that an MVD is a special case of a JD where $n = 2$. That is, a JD denoted as JD(R_1, R_2) implies an MVD ($R_1 \cap R_2$) $\rightarrow\!\!\rightarrow$ ($R_1 - R_2$). A join dependency JD(R_1, R_2, ..., R_n), specified on relation schema R, is a **trivial** JD if one of the relation schemas R_i in JD(R_1, R_2, ..., R_n) is equal to R. Such a dependency is called trivial because it has the nonadditive join property for any relation state r of R and thus does not specify any constraint on R.

JD arises when a relation is decomposed into a set of projected relations that can be joined back to yield the original relation.

Fifth normal form

A relation schema R is in **fifth normal form (5NF)** (or **project-join normal form (PJNF)**) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R_1, R_2, ..., R_n) in F^+ (that is, implied by F), every R_i is a superkey of R.

Discovering JDs in practical databases with hundreds of attributes is next to impossible. It can be done only with a great degree of intuition about the data on the part of the designer.

Example1:

Factory produces component that is delivered to project.

R

Factory	Component	Project
GM	Engine	MPC
GM	Gearbox	125A
Honda	Engine	125A
GM	Engine	125A

R1

Factory	Component
GM	Engine
GM	Gearbox
Honda	Engine

R2

Component	Project
Engine	MPC
Gearbox	125A
Engine	125A

R3

Factory	Project
GM	MPC
GM	125A
Honda	125A

R1 ⋈ R2

Factory	Component	Project
GM	Engine	MPC
GM	Engine	125A
GM	Gearbox	125A
Honda	Engine	MPC
Honda	Engine	125A

(R1 ⋈ R2) ⋈ R3

Factory	Component	Project
GM	Engine	MPC
GM	Engine	125A
GM	Gearbox	125A
Honda	Engine	125A

(R1 ⋈ R2) ⋈ R3=R; So, there exists join dependency in the original relation R. Thus, the relation R is not in 5NF. Hence, the relation R is decomposed into R1, R2 and R3.

Example2:

R

Factory	Component	Project
GM	Engine	MPC
GM	Gearbox	125A
Honda	Engine	125A
Honda	Gearbox	MPC
GM	Engine	125A

R1

Factory	Component
GM	Engine
GM	Gearbox
Honda	Engine
Honda	Gearbox

R2

Component	Project
Engine	MPC
Gearbox	125A
Engine	125A
Gearbox	MPC

R3

Factory	Project
GM	MPC
GM	125A
Honda	125A
Honda	MPC

R1 ⋈ R2

Factory	Component	Project
GM	Engine	MPC
GM	Engine	125A
GM	Gearbox	125A
GM	Gearbox	MPC
Honda	Engine	MPC
Honda	Engine	125A
Honda	Gearbox	125A
Honda	Gearbox	MPC

(R1 ⋈ R2) ⋈ R3

Factory	Component	Project
GM	Engine	MPC
GM	Engine	125A
GM	Gearbox	125A
GM	Gearbox	MPC
Honda	Engine	MPC
Honda	Engine	125A
Honda	Gearbox	125A
Honda	Gearbox	MPC

As (R1 ⋈ R2) ⋈ R3 !=R, there exists no join dependency in the relation R. Hence, R is already in 5NF.

Inclusion Dependencies

Inclusion dependencies were defined in order to formalize two types of interrelational constraints:

■ The foreign key (or referential integrity) constraint cannot be specified as a functional or multivalued dependency because it relates attributes across relations.

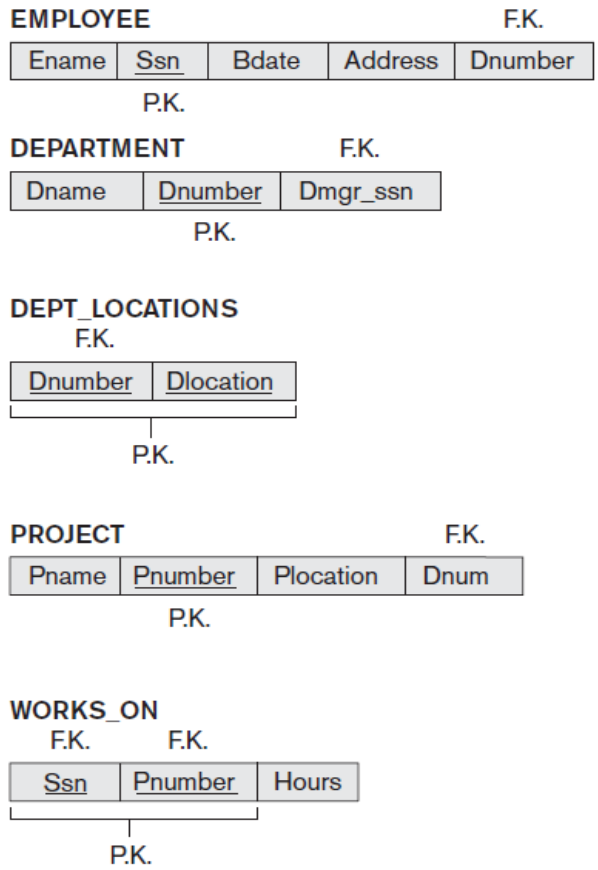
■ The constraint between two relations that represent a class/subclass relationship also has no formal definition in terms of the functional, multivalued, and join dependencies.

Definition. An **inclusion dependency** $R.X < S.Y$ between two sets of attributes— X of relation schema R, and Y of relation schema S—specifies the constraint that, at any specific time when r is a relation state of R and s a relation state of S, we must have

$$\pi_X(r(R)) \subseteq \pi_Y(s(S))$$

The domains for each pair of corresponding attributes should be compatible.

Example: Consider the following COMPANY relational schema:



We can specify the following inclusion dependencies on the relational schema in Figure:

DEPARTMENT.Dmgr_ssn < EMPLOYEE.Ssn

WORKS_ON.Ssn < EMPLOYEE.Ssn

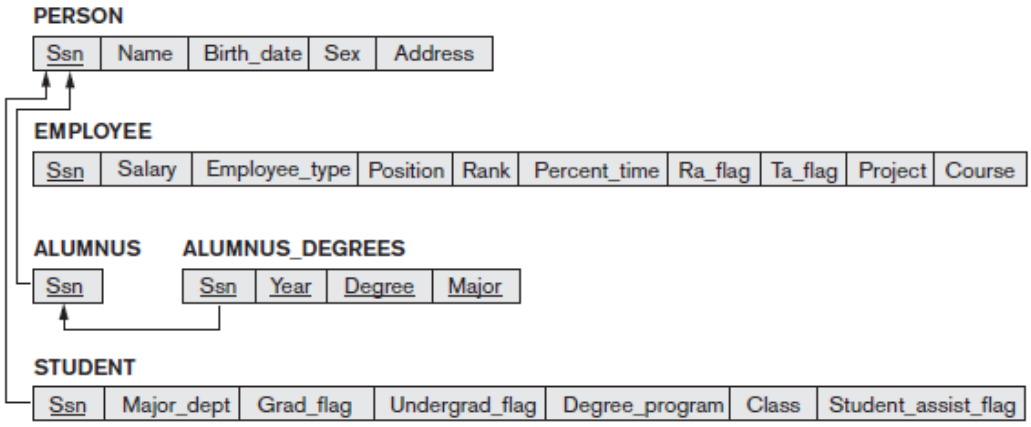
EMPLOYEE.Dnumber < DEPARTMENT.Dnumber

PROJECT.Dnum < DEPARTMENT.Dnumber

WORKS_ON.Pnumber < PROJECT.Pnumber

DEPT_LOCATIONS.Dnumber < DEPARTMENT.Dnumber

All the preceding inclusion dependencies represent **referential integrity constraints**. We can also use inclusion dependencies to represent **class/subclass relationships**. For example, in the relational schema of Figure, we can specify the following inclusion dependencies:



EMPLOYEE.Ssn < PERSON.Ssn

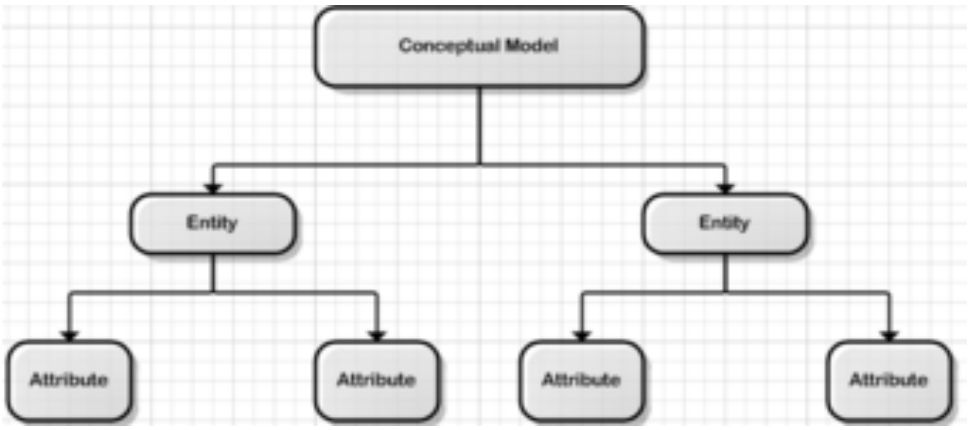
ALUMNUS.Ssn < PERSON.Ssn

STUDENT.Ssn < PERSON.Ssn

Alternative Approaches to Database Design

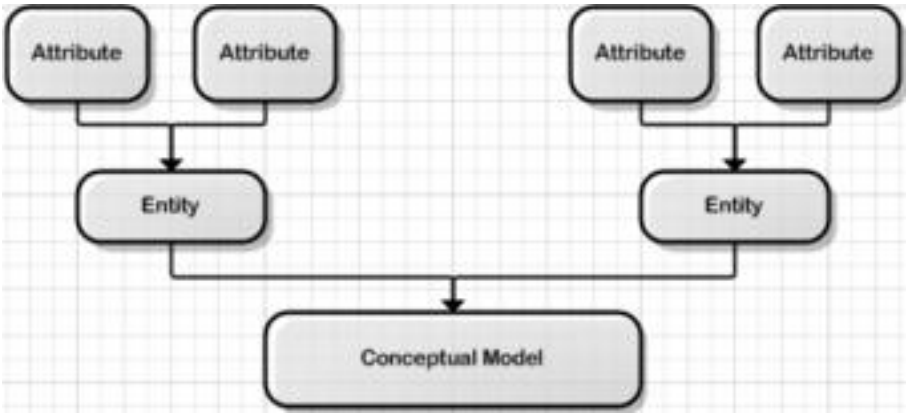
1. Top – down design method

The top-down design method starts from the general and moves to the specific. In other words, you start with a general idea of what is needed for the system and then work your way down to the more specific details of how the system will interact. This process involves the identification of different entity types and the definition of each entity's attributes.



2. Bottom – up design method

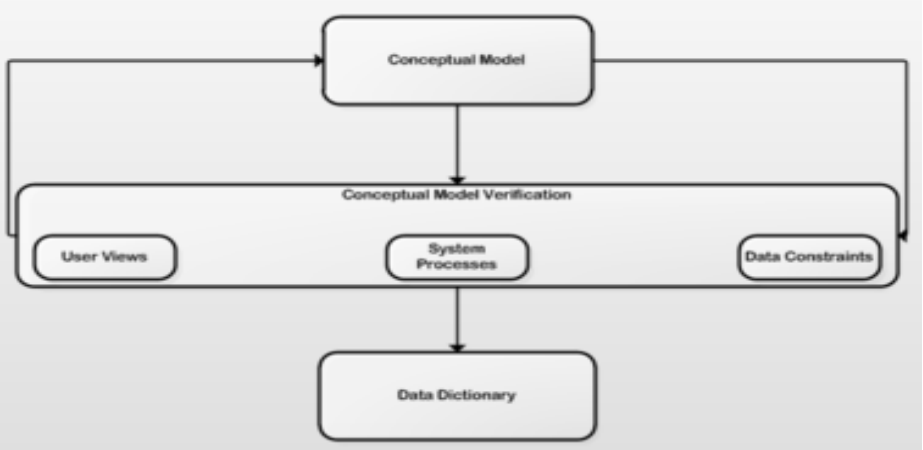
The bottom-up approach begins with the specific details and moves up to the general. This is done by first identifying the data elements (items) and then grouping them together in data sets. In other words, this method first identifies the attributes, and then groups them to form entities.



Two general approaches (top – down and bottom – up) to the design of the databases can be heavily influenced by factors like scope, size of the system, the organizations management style, and the organizations structure. Depending on such factors, the design of the database might use two very different approaches, centralized design and decentralized design.

Centralized design

Centralized design is most productive when the data component is composed of a moderately small number of objects and procedures. The design can be carried out and represented in a somewhat simple database. Centralized design is typical of a simple or small database and can be successfully done by a single database administrator or by a small design team. This person or team will define the problems, create the conceptual design, verify the conceptual design with the user views, and define system processes and data constraints to ensure that the design complies with the organizations goals. That being said, the centralized design is not limited to small companies. Even large companies can operate within the simple database environment.



Decentralized design

Decentralized design might best be used when the data component of the system has a large number of entities and complex relations upon which complex operations are performed. This is also likely to be used when the problem itself is spread across many operational sites and the elements are a subset of the entire data set. In large and complex projects a team of carefully selected designers are employed to get the job done. This is commonly accomplished by several teams that work on different subsets or modules of the system. Conceptual models are created by these teams and compared to the user views, processes, and constraints for each module. Once all the teams have completed their modules they are all put aggregated into one large conceptual model.