Module 04: Normalization

Chapter 14: Basics of Functional Dependencies and Normalization for Relational Databases

Goodness of relation schemas are discussed at 2 levels:

- Logical level
- Implementation level

Logical level :- how users interpret the relation schemas and the meaning of their attributes. Having good relation schemas at this level enables user to understand the meaning od data in the relations clearly and hence to formulate query.

Implementation level:- how the tuples in the base relations are stored and updated. This level applies only to schemas of base relations – which will be physically stored as files whereas at the logical level we are interested in schemas of both base relations and views.

Database Design approaches

There are two approaches:

- Bottom-up design methodology
- Top-down design methodology

Bottom-up design methodology – also called as '**design by synthesis'**. This consider basic relationship[s among individual attributes as the starting point and uses those to construct relation schemas.

Top-down design methodology - also called as '**design by analysis'** . This starts with number of groupings of attributes into relations that exists together naturally. The relations are then analyzed individually and collectively leading to further decomposition until all desirable properties are met.

Note: The implicit and main goal of design activity is information preservation and minimum redundancy.

14.1 Informal Design Guidelines for Relation Schemas

There are four informal guidelines that may be used as measures to determine the quality of relation schema design:

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples.

Each of these are discussed now.

14.1.1 Imparting Clear Semantics to Attributes in Relations

The semantics of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple. If the semantics of the relation is easier to understand/explain then the relation schema design is considered as better.

Consider an example below,

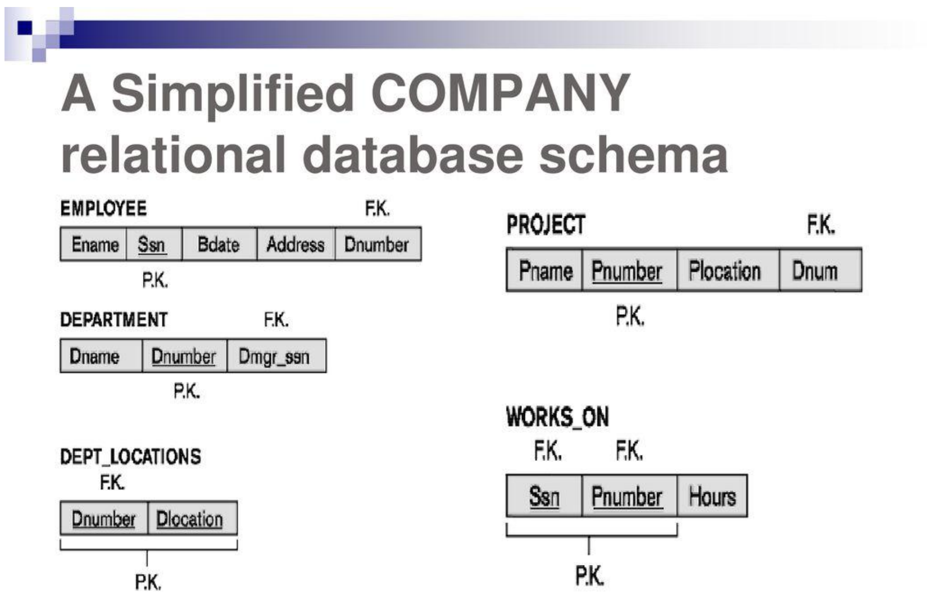


Figure 14.1: Simplified COMPANY relational database schema

Looking at the figure above, we can easily understand about COMPANY database. Hence, all the relation schemas in Figure 14.1 may be considered as easy to explain and therefore good from the standpoint of having clear semantics. We can thus formulate the following informal design guideline.

Guideline 1: Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation.

14.1.2 Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations. Grouping attributes into relation schemas has a significant effect on storage space.

Ex :-

EMP_DEPT

Ename	**<u>Ssn</u>**	**Bdate**	**Address**	**Dnumber**	**Dname**	**Dmgr_ssn**

EMP_PROJ

<u>Ssn</u>	**<u>Pnumber</u>**	**Hours**	**Ename**	**Pname**	**Plocation**

Figure 14.2 : Schema of EMP_PROJ and EMP_DEPT

Here, the relation EMP_DEPT is the result of Natural Join operation on EMPLOYEE and DEPARTMENT table. In this, the attribute values pertaining to a particular department(Dnumber,Dname,Dmgr_ssn) are repeated for every employee who works for that department. In contrast, each department's information appears only once in the Department relation. Only the department number is repeated in the EMPLOYEE relation for each employee who works in that department.

Storing natural joins of base relations leads to an additional problem referred to as update anomalies. These can be classified into three categories:

- Insertion anomalies
- Deletion anomalies and
- Modification anomalies.

Insertion Anomalies

To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for that department that the employee works for, or nulls. It's difficult to insert a new department that has no employee as yet in the EMP_DEPT relation. The only way to do this is to place null values in the attributes for employee. This causes a problem because SSN is the primary key of EMP_DEPT, and each tuple is supposed to represent an employee entity - not a department entity.

Deletion Anomalies

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 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.

Guideline 2: Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

14.1.3 NULL Values in Tuples

If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This causes many problems:

- Leads to waste of space at the storage level.
- Leads to problem in understanding the meaning of attributes.

- Leads to problem in specifying JOIN operations at logical level
- How to account / consider them when aggregate operations such as COUNT or SUM are applied.
- SELECT and JOIN operations involve comparisons. If NULL values are present, the result may become unpredictable.
- NULL can be interpreted in 3 different ways, which one to consider?

Guideline 3: As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLs are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

14.1.4 Generation of Spurious Tuples

Consider the following schemas

EMP_LOCS

<u>Ename</u>	<u>Plocation</u>

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation

Figure 14.3 : Schema of EMP_LOCS and EMP_PROJ1

Here, we have two relation schemas EMP_LOCS and EMP_PROJ1 which can be used instead of single EMP_PROJ relation. Suppose that we use EMP_LOCS and EMP_PROJ1 as the base relations instead of EMP_PROJ . this produces a bad schema design. Hence, if we attempt a Natural JOIN operation on these two relations, the result will contain many more tuples than the original number of tuples in EMP_PROJ. They are called **Spurious tuples**, because they represent spurious information that is not valid.

Guideline 4: Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated.

14.2 Functional Dependencies

14.2.1 Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database.

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 t1 and t2 in r that have t1[X] = t2[X], they must also have t1[Y] = t2[Y].

In X->Y, the left side of FD is known as a **Determinant** and the right side of the production is known as **Dependent**.

A functional dependency is a property of the semantics or meaning of the attributes. Consider the relation schema EMP_PROJ in Figure 14.2; from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

(a) Ssn → Ename
(b) Pnumber → {Pname, Plocation}
(c) {Ssn, Pnumber} → Hours

These functional dependencies specify that

(a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename),

(b) the value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation)

(c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours).

14.3 Normal Forms Based on Primary Keys

Most practical relational design projects take one of the following two approaches:

■ Perform a conceptual schema design using a conceptual model such as ER or EER and map the conceptual design into a set of relations.

■ Design the relations based on external knowledge derived from an existing implementation of files or forms or reports.

14.3.1 Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to certify whether it satisfies a certain **normal form**.

Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively.

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.

An unsatisfactory relation schema that does not meet the condition for a normal form—the normal form test—is decomposed into smaller relation schemas that contain a subset of the attributes and hence possess desirable properties.

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.

During the process of normalization , it also confirms the existence of additional properties that a relation schema possess. They are,

■ The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.

■ The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

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

14.3.3 Definitions of Keys and Attributes Participating in Keys

A **superkey** of a relation schema R = {A1, A2, ... , An} is a set of attributes S ⊆ R with the property that no two tuples t1 and t2 in any legal relation state r of R will have t1[S] = t2[S].

A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey anymore.

If a relation schema has more than one key, each is called a **candidate key.**

One of the candidate keys is arbitrarily designated to be the **primary key**, and the others are called **secondary keys**.

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

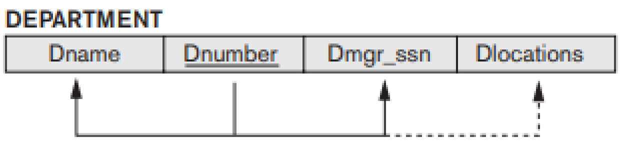
An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

14.3.4 First Normal Form

"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. The only attribute values permitted by 1NF are single **atomic** (or indivisible) values.

(a)



(b)

DEPARTMENT

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

(c)

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

Figure 14.4

Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

Consider the DEPARTMENT relation schema shown in Figure 14.4(a), whose primary key is Dnumber. As we can see, this is not in 1NF because Dlocations is not an atomic attribute. There are two ways we can look at the Dlocations attribute:

■ The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber.

■ The domain of Dlocations contains sets of values and hence is nonatomic. In this case, Dnumber → Dlocations because each set is considered a single member of the attribute domain.

There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute Dlocations that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this newly formed relation is the combination {Dnumber, Dlocation}, as shown in Figure 14.4(b). A distinct tuple in DEPT_LOCATIONS exists for each location of a department. This decomposes the non-1NF relation into two 1NF relations.
2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 14.4(c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing redundancy in the relation and hence is rarely adopted.
3. 3. If a maximum number of values is known for the attribute—for example, if it is known that at most three locations can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing NULL values if most departments have fewer than three locations.

Among three options first one founds suitable.

14.3.5 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.

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.

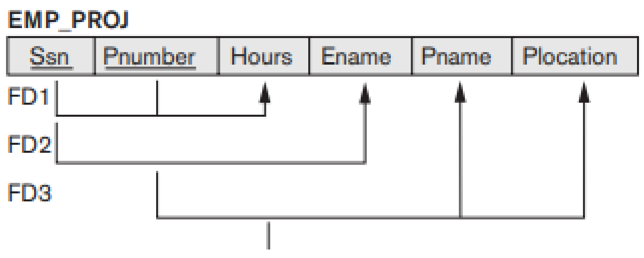


Figure 14.5 Schema of EMP_PROJ

Here, we have 3 FDs considering that (Ssn,Pnumber) acts as a primary key.

- {Ssn, Pnumber } -> Hours
- {Ssn, Pnumber} → Ename
- {Ssn, Pnumber} -> Pname,Plocation

Among these ,

{Ssn, Pnumber} → Hours is a full dependent (neither Ssn → Hours nor Pnumber → Hours holds). However, the dependency {Ssn, Pnumber} → Ename is partial because Ssn → Ename holds. Similarly, {Ssn, Pnumber} -> Pname,Plocation is also partially dependent because Pnumber alone can determine Pname and Plocation.i.e. Pnumber -> Pname,Plocation.

Therefore , EMP_PROJ relation is not in 2NF as the nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3.

Solution : 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 in Figure 14.5 lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 14.6, each of which is in 2NF.

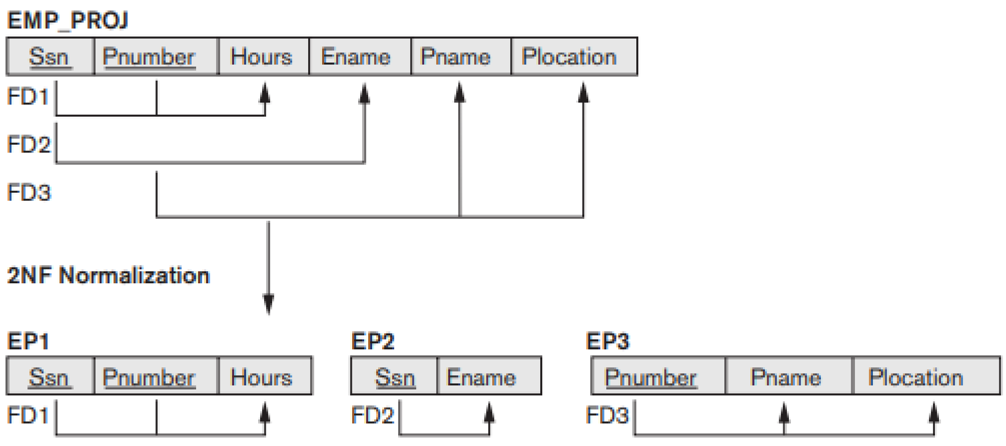


Figure 14.6 : Normalizing to 2NF

14.3.6 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: According to Codd's original 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.

Consider an example shown in figure 14.7

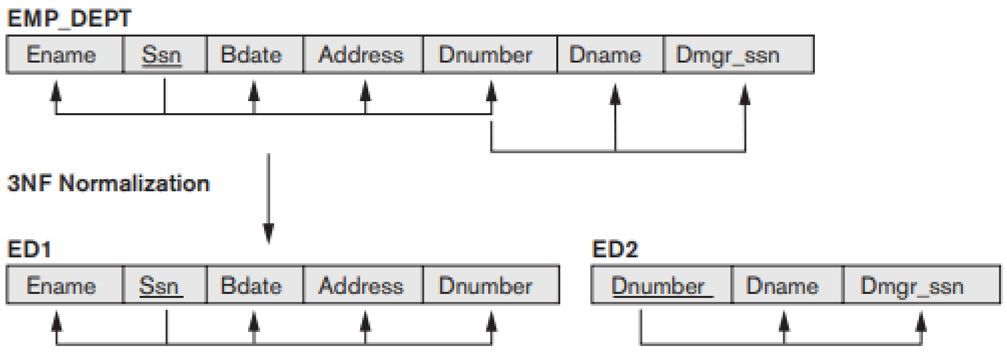


Figure 14.7: Normalizing to 3NF

The dependency Ssn → Dmgr_ssn is **transitive** through Dnumber in EMP_DEPT , because both the dependencies Ssn → Dnumber and Dnumber → Dmgr_ssn hold and Dnumber is neither a key itself nor a subset of the key of EMP_DEPT.

The relation schema EMP_DEPT is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2.

14.4 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.

Let's assume there is a company where employees work in more than one department.

In the table of figure 14.8, Functional dependencies are as follows:

EMP_ID → EMP_COUNTRY
EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Figure 14.8 : BCNF

Functional dependencies:

EMP_ID → EMP_COUNTRY
EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

14.5 Multi-valued Dependency and Fourth Normal Form

A relation R is in 4NF if and only if the following conditions are satisfied:

1. It should be in the Boyce-Codd Normal Form (BCNF).
2. the table should not have any Multi-valued Dependency.

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

Figure 14.9 : 4th Normal Form

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

14.6 Join Dependencies and Fifth Normal Form

- 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	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

Figure 14.20:5th Normal Form

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:

P1

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

Chapter 15: Normalization Algorithms

15.1 Inference Rules, Equivalence, and Minimal Cover

15.1.1 Inference Rules for Functional Dependencies

In real life, it is impossible to specify all possible functional dependencies for a given situation.

For example, if each department has one manager, so that Dept_no uniquely determines Mgr_ssn. It is represented by Functional Dependencies.

FD1 : Dept_no → Mgr_ssn

A manager has a unique phone number called Mgr_phone

FD2 : Mgr_ssn → Mgr_phone

Then these two dependencies together imply that

FD3 : Dept_no → Mgr_phone.

This is an inferred or implied FD and need not be explicitly stated in addition to the two given FDs. Therefore, we define a concept called **closure** that includes all possible dependencies that can be inferred from the given set F.

Closure

Definition. Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the **closure** of F; it is denoted by F+.

For example, suppose that we specify the following set F of obvious functional dependencies on the relation schema

F = {Ssn → {Ename, Bdate, Address, Dnumber}, Dnumber → {Dname, Dmgr_ssn} }

Some of the additional functional dependencies that we can infer from F are the following:

Ssn → {Dname, Dmgr_ssn}

Ssn → Ssn

Dnumber → Dname

The closure F+ of F is the set of all functional dependencies that can be inferred from F.

To determine a systematic way to infer dependencies, we must discover a set of inference rules that can be used to infer new dependencies from a given set of dependencies. We consider some of these inference rules next. We use the notation F |=X → Y to denote that the functional dependency X → Y is inferred from the set of functional dependencies F.

Inference Rules

“an inference rule is an assertion that we can apply to a set of functional dependencies to derive other functional dependencies.”

IR1 (reflexive rule) : If X ⊇ Y, then X →Y.

IR2 (augmentation rule) : {X → Y} |=XZ → YZ.

IR3 (transitive rule): {X → Y, Y → Z} |=X → Z.

IR4 (decomposition, or projective, rule): {X → YZ} |=X → Y.

IR5 (union, or additive, rule): {X → Y, X → Z} |=X → YZ.

IR6 (pseudotransitive rule): {X → Y, WY → Z} |=WX → Z.

IR1 (Reflexive rule) states that a set of attributes always determines itself or any of its subsets. Because IR1 generate dependencies that are always true, such dependencies are called **Trivial.**

Definition: A functional dependency X -> Y is trivial if X ⊇ Y. Otherwise, it is non-trivial.

IR2 (augmentation rule) states that adding the same set of attributes to both left and right hand side of a dependency results in another valid dependency.

IR3 (transitive rule) states that Functional Dependencies are transitive.

IR4 (decomposition, or projective, rule) says that we can remove attributes from the right-hand side of a dependency. By applying this rule repeatedly, we can decompose given FD X->{A1,A2,A3,...An} into set of dependencies..i.e. {X->A1, X->A2, X->A3,.......X->An}.

IR5 (union, or additive, rule) allows us to combine set of dependencies into a single FD. i.e. . {X->A1, X->A2, X->A3,.......X->An} into X->{A1,A2,A3,...An}.

IR6 (pseudotransitive rule): allows us to replace a set of attributes Y on the left hand side of FD with another set X that functionally determines Y.

Each of the preceding inference rules can be proved from the definition of functional dependency either by direct proof or by contradiction.

Consider the relation EMPLOYEE_DEPARTMENT

Ssn	Fname	Lname	Dno	Dname

Examples of Armstrong axioms:

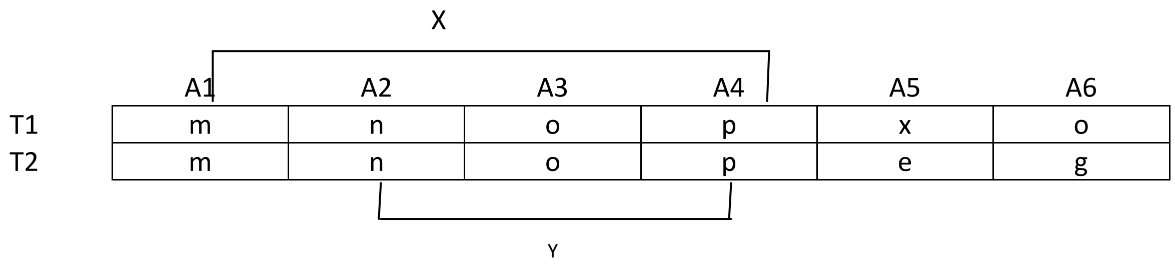
1. Reflexivity Rule : if X ⊇ Y then X->Y
   {Fname,Lname}->Fname

2. Augmentation Rule : If X->Y, then XZ->YZ.
   If ssn ->{Fname} then {Ssn,Dname} -> {Fname,Dname}
3. Transitive Rule : If X -> Y, Y -> Z, then X ->Z.
   If Ssn -> Dno and Dno -> Dname then Ssn -> Dname.

Proof of IR1

Suppose that X ⊇ Y and that two tuples t1 and t2 exist in some relation instance r of R such that t1[X] = t2[X] then t1[Y] = t2[Y]. Hence x->y must hold in r.

The following diagram shows graphically why t1[Y] must be equal to t2[Y].



	A1	A2	A3	A4	A5	A6
T1	m	n	o	p	x	o
T2	m	n	o	p	e	g

T1[X] = (m,n,o,p)

T2[X] = (m,n,o,p)

T1[Y] = (n,o,p)

T2[Y] = (n,o,p)

Proof of IR2

Assume that

- X -> Y is true.
- But XZ -> YZ is false.

From the fact that XZ -> YZ is false.

Then there must exist two tuples t1 and t2 in r such that

(1) t1 [X] = t2 [X]
(2) t1 [Y] = t2 [Y]
(3) t1 [XZ] = t2 [XZ] and
(4) t1 [YZ] ≠ t2 [YZ]

This is not possible because from (1) and (3) we deduce

(5) t1 [Z] = t2 [Z]

from (2) and (5) we deduce

(6) t1 [YZ] = t2 [YZ] which contradicts (4)

Proof of IR3

Assume

(1) X → Y and

(2) Y → Z both hold in a relation r.

Then for any two tuples t1 and t2 in r such that t1 [X] = t2 [X], we must have

(3) t1 [Y] = t2 [Y], from assumption (1);

Hence we must also have

(4) t1 [Z] = t2 [Z] from (3) and assumption (2);

Thus X→Z must hold in r.

Proof of IR4 (Using IR1 through IR3)

1. X→YZ (given)
2. YZ→Y (using IR1 and knowing that YZ ⊇ Y)
3. X→Y (using IR3 on 1 and 2).

Proof of IR5 (using IR1 through IR3).

1. X→Y (given)
2. X→Z (given)
3. X→XY (using IR2 on 1 by augmenting with X; notice that XX = X)
4. XY→YZ (using IR2 on 2 by augmenting with Y)
5. X→YZ (using IR3 on 3 and 4).

Proof of IR6 (using IR1 through IR3)

1. X→Y (given)
2. WY→Z (given)
3. WX→WY (using IR2 on 1 by augmenting with W)
4. WX→Z (using IR3 on 3 and 2).

Inference Rules from IR1 through IR3 are known as Armstrong's inference rules. These are Sound and Complete.

Sound – For the given set of FDs F, any dependencies that we can infer from F by using IR1 through IR3 satisfies the dependencies in F.

Complete – Just with the help of these three IRs we can infer all possible FDs in given relation.

Closure of X

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 15.1. Determining X+, the Closure of X under F

Input: A set F of FDs on a relation schema R, and a set of attributes X, which is a subset of R.

```
X+ := X;
 Repeat
        oldX+ := X+;
        for each functional dependency Y → Z in F do
        if X+ ⊇ Y then X+ := X+ ∪ Z;
        until (X+ = oldX+);
```

Explaination :

Start by setting X+ to all the attributes in X. By IR1, we know that all these attributes are functionally dependent on X. Using inference rules IR3 and IR4, we add attributes to X+, using each functional dependency in F. We keep going through all the dependencies in F (the repeat loop) until no more attributes are added to X+ during a complete cycle (of the for loop) through the dependencies in F.

The closure concept is useful in understanding the meaning and implications of attributes or sets of attributes in a relation.

EMP_PROJ

<u>**Ssn**</u>	**Ename**	**Pnumber**	**Pname**	**Plocation**	**Hours**

Set of FDs here are,

{ Ssn -> Ename,

Pnumber -> {Pname,Plocation},

{Ssn, Pnumber} -> Hours }.

Now, we calculate closure set for each of these Fds.

{Ssn}+ = {Ssn,Ename}

{Pnumner}+ = { Pnumber, Pname, Plocation}

{ Ssn,Pnumber}+ = {Ssn,Pnumber,Ename,Pname,Plocation,Hours}

15.1.2 Equivalence of Sets of Functional Dependencies

Given a relation with two different FD sets, we have to find out whether one FD set is subset of other or both are equal.

Cover

Definition: 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.

Equivalence

Definition: 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.

NOTE : We can determine whether F covers E by calculating X+ with respect to F for each FD X → Y in E, and then checking whether this X+ includes the attributes in Y. If this is the case for every FD in E, then F covers E. We determine whether E and F are equivalent by checking that E covers F and F covers E.

Example :- To check whether F covers G.

F = {A → C, AC → D, E → AD, E → H} and

G = {A → CD, E → AH}

F+ :	A+ = {ACD}	G+:	A+ = {ACD}
	AC+ = {ACD}		E+ = {EAHCD}
	E+ = {EADHC}		

G ⊇ F

⇨ F covers G.

15.1.3 Minimal Sets of Functional Dependencies

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.

Extraneous Attribute

An attribute in a functional dependency is considered an **extraneous attribute** if we can remove it without changing the closure of the set of dependencies.

Example: let us consider a relation R with schema R(A,B,C) and set of functional dependencies

F = { AB->C, A->C}

The closure for F is F+ = { AB->C , A->C}

In AB->C, B is extraneous attribute.

The reason is, there is another Functional Dependency A->C which means A alone can determine C. So, the use of B is not necessary. Now, new F+ = {A->C}.

Minimal Sets

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 → A in F with a dependency Y → 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.

Condition (1) just represents that every dependency in a canonical form with a single attribute on the right-hand side of FDs.

Condition(2) and (3) ensure that there are no redundancies in the dependencies either by having redundant attributes on the left hand side of dependency or by having a dependency that can be inferred from the remaining FDs in F.

NOTE: There can be many such minimal covers for a set of functional dependencies F.

A **minimal cover** of a set of functional dependencies E is a minimal set of dependencies that is equivalent to E.

Algorithm 15.2. 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 → {A1, A2, … , An} in F by
   the n functional dependencies  X →A1, X →A2, … , X → An.
3. For each functional dependency X → A in F
        for each attribute B that is an element of X
            if { {F − {X → A} } ∪ { (X − {B} ) → A} } is equivalent to F
                then replace X → A with (X − {B} ) → A in F.
4. For each remaining functional dependency X → A in F
        if {F − {X → A} } is equivalent to F,
        then remove X → A from F.
```

Let us understand this algorithm with an Example.

Problem 1

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

Step 1: All above dependencies are in canonical form that is, they have only one attribute on the right-hand side. So, first step is not necessary.

Step 2: Here, we need to determine if AB → D has any redundant (extraneous) attribute on the left-hand side; that is, can it be replaced by B → D or A → D?

- Since B → A, by augmenting with B on both sides (IR2), we have BB → AB, or B → AB .....(i).
- However, AB → D as given ....(ii).
- Hence by the transitive rule (IR3), we get from (i) and (ii), B → D. Thus AB → D may be replaced by B → D.

We now have a set equivalent to original E, say E′: {B → A, D → A, B → D}.

No further reduction is possible in step 2 since all FDs have a single attribute on the left-hand side.

Step 3: In step 3 we look for a redundant FD in E′.

By using the transitive rule on B → D and D → A, we derive B → A.

Hence B → A is redundant in E′ and can be eliminated.

Therefore, the minimal cover of E is F: {B → D, D → A}.

Algorithm 15.2(a): Finding a Key K for R Given a Set F of Functional Dependencies

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

```
1. Set K := R.
2. For each attribute A in K
   {compute (K − A) + with respect to F;
   if (K − A) + contains all the attributes in R, then set K := K − {A} };
```

Problem

R(A,B,D)

F= { B->A, D->A,AB->D}

Step 1: K:= R i.e. K = (A,B,D)

For A, Compute (K-A)+ i.e. (BD)+ -> BDA (This contains all attributes of R.)

So, Set K:=K-{A} => K:=(BD)

Step 2: Now K = (BD)

For B , Compute (K-B)+ i.e. D+ ->DA (Does not contain all attributes of R)

Step 3: Now K= (BD)

For D, Compute (K-d)+ i.e. B+ ->BAD (contains all attributes of R)

So, Set K:= K – {D} i.e. K:=(B)

- ⇨ **B is a key for given relation R.**

15.2 Properties of Relational Decompositions

All these days , we decomposed given relations in order to get rid of unwanted dependencies and to achieve higher normal forms. But looking at an individual relation to test whether it is in a higher normal form does not guarantee a good design. Rather, a set of relations that together form the relational database schema must possess certain additional properties to ensure good design.

- Dependency Preservation Property
- Non-additive or Lossless Join Property

15.2.1 Relation Decomposition and Insufficiency of Normal Forms

The relational database design algorithms that we present in Section 15.3 start from a single universal relation schema R = {A1, A2, … , An} that includes all the attributes of the database. We implicitly make the universal relation assumption, which states that every attribute name is unique. The set F of functional dependencies that should hold on the attributes of R is specified by the database designers and is made available to the design algorithms. Using the functional dependencies, the algorithms decompose the universal relation schema R into a set of relation schemas D = {R1, R2, … , Rm} that will become the relational database schema; D is called a decomposition of R.

We must make sure that each attribute in R will appear in at least one relation schema Ri in the decomposition so that no attributes are lost; formally, we have R R i i m = = 1 U This is called the attribute preservation condition of a decomposition.

Another goal is to have each individual relation Ri in the decomposition D be in BCNF or 3NF. However, this condition is not sufficient to guarantee a good database design on its own. We must consider the decomposition of the universal relation as a whole, in addition to looking at the individual relations.

15.2.2 Dependency Preservation Property of a Decomposition

The **dependency preservation decomposition** is another property of decomposed relational database schema D in which each functional dependency X -> Y specified in F either appeared directly in one of the relation schemas R_i in the decomposed D or could be inferred from the dependencies that appear in some R_i.

Decomposition D = { R_1 , R_2, R_3,,.., ,R_m} of R is said to be dependency-preserving with respect to F if the union of the projections of F on each R_i , in D is equivalent to F. In other words, R ⊂ join of R_1, R_1 over X. The dependencies are preserved because each dependency in F represents a constraint on the database. If decomposition is not dependency-preserving, some dependency is lost in the decomposition.

Example:

Let a relation R(A,B,C,D) and set a FDs F = { A -> B , A -> C , C -> D} are given. A relation R is decomposed into –

```
R₁ = (A, B, C) with FDs F₁ = {A -> B, A -> C}, and
R₂ = (C, D) with FDs F₂ = {C -> D}.
      F' = F₁ ∪ F₂ = {A -> B, A -> C, C -> D}
      so, F' = F.
      And so, F'⁺ = F⁺.
```

Thus, the decomposition is dependency preserving decomposition.

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

15.2.3 Nonadditive (Lossless) Join Property of a Decomposition

Another property that a decomposition D should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition.

Example

We have **EMP_PROJ** relation

<u>SSN</u>	ENAME	PNUMBER	PNAME	PLOCATION	HOURS

FD = { SSN -> ENAME,

PNUMBER -> { PNAME, PLOCATION},

{SSN, PNUMBER} -> HOURS }

If we decompose this relation into two relations R1 and R2 as below,

R1 -> EMP_LOC = { ENAME, PLOCATION}

R2 -> EMP_PROJ1 = { SSN, PNUMBER, HOURS, PNAME, PLOCATION}

Now, we try to perform Natural Join on these two relations, we will get spurious tuples.

Instead, if we can divide given relation into three relations as below,

EMP = { SSN,ENAME}

PROJ = { PNUMBER, PNAME, PLOCATION}

WORKS_ON = { SSN, PNUMBER, HOURS}

When we try to join them, we will get result without spurious tuples.

Algorithm 15.3. Testing for Nonadditive Join Property

Input: A universal relation R, a decomposition D = {R1, R2, … , Rm} of R, and a set F of functional dependencies.

```
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
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;};};
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 symbol for the column, set
       the other rows to that same a symbol in the column. If no a symbol exists for the
       attribute in any of the rows, choose one of the b symbols that 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 symbols, then the decomposition has the nonadditive join
   property; otherwise, it does not.
```

15.2.4 Testing Binary Decompositions for the Nonadditive Join Property

Binary Decomposition: decomposition of a relation *R* into two relations**.**

PROPERTY (lossless join test for binary decompositions):

A decomposition $D = \{R_1, R_2\}$ of *R* has the lossless join property with respect to a set of functional dependencies *F* on *R if and only if* either

- The f.d. $((R_1 \cap R_2) \text{ -> } (R_1 - R_2))$ is in F^+, or
- The f.d. $((R_1 \cap R_2) \text{ -> } (R_2 - R_1))$ is in F^+.

15.2.5 Successive Nonadditive Join Decompositions

We decompose relations successively during the process of second and third normalization. So, during this process we need to check whether it is ensuring nonadditive join property at every iteration or step of decomposition.

15.3 Algorithms for Relational Database Schema Design

15.3.1 Dependency-Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas

Algorithm 15.4

Relational Synthesis into 3NF with Dependency Preservation and Nonadditive Join Property

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

1. Find a minimal cover G for F .

2. For each left-hand-side X of a functional dependency that appears in G, create a relation schema in D with attributes $\{X \cup \{A1\} \cup \{A2\} \dots \cup \{Ak\}\}$, where $X \rightarrow A1, X \rightarrow A2, \dots, X \rightarrow Ak$ are the only dependencies in G with X as lefthand side (X is the key of this relation).

3. If none of the relation schemas in D contains a key of R, then create one more relation schema in D that contains attributes that form a key of R.

4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation R is considered redundant if R is a projection of another relation S in the schema; alternately, R is subsumed by S.

Example 1 of Algorithm 15.4. Consider the following universal relation:

U (Emp_ssn, Pno, Esal, Ephone, Dno, Pname, Plocation)

Emp_ssn, Esal, and Ephone refer to the Social Security number, salary, and phone number of the employee. Pno, Pname, and Plocation refer to the number, name, and location of the project. Dno is the department number.

The following dependencies are present:

FD1: Emp_ssn → {Esal, Ephone, Dno}

FD2: Pno → { Pname, Plocation}

FD3: Emp_ssn, Pno → {Esal, Ephone, Dno, Pname, Plocation}

By applying the minimal cover Algorithm 15.2, in step 3

we see that Pno is an extraneous attribute in Emp_ssn, Pno → Esal, Ephone, Dno. Moreover, Emp_ssn is extraneous in Emp_ssn, Pno → Pname, Plocation. Hence the minimal cover consists of FD1 and FD2 only (FD3 being completely redundant) as follows (if we group attributes with the same left-hand side into one FD):

Minimal cover G: {Emp_ssn → Esal, Ephone,

Dno; Pno → Pname, Plocation}

The **second step** of Algorithm 15.4 produces relations R1 and R2 as:

R1 (Emp_ssn, Esal, Ephone, Dno)

R2 (Pno, Pname, Plocation)

In **step 3**, we generate a relation corresponding to the key {Emp_ssn, Pno} of U. Hence, the resulting design contains:

R1 (Emp_ssn, Esal, Ephone, Dno)

R2 (Pno, Pname, Plocation)

R3 (Emp_ssn, Pno)

This design achieves both the desirable properties of dependency preservation and nonadditive join.

15.3.2 Nonadditive Join Decomposition into BCNF Schemas

This algorithm decomposes a universal relation schema R = {A1, A2, … , An} into a decomposition D = {R1, R2, … , Rm} such that each Ri is in BCNF and the decomposition D has the lossless join property with respect to F.

Algorithm 15.5. Relational Decomposition into BCNF with Nonadditive Join Property

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

```
1. Set D := {R} ;
2. While there is a relation schema Q in D that is not in BCNF do
        {
        choose a relation schema Q in D that is not in BCNF;
        find a functional dependency X → Y in Q that violates BCNF;
        replace Q in D by two relation schemas (Q − Y) and (X ∪ Y);
        } ;
```

Each time through the loop in Algorithm 15.5, we decompose one relation schema Q that is not in BCNF into two relation schemas. According to property NJB for binary decompositions and claim 2, the decomposition D has the nonadditive join property. At the end of the algorithm, all relation schemas in D will be in BCNF.

In step 2 of Algorithm 15.5, it is necessary to determine whether a relation schema Q is in BCNF or not. One method for doing this is to test, for each functional dependency X → Y in Q, whether X+ fails to include all the attributes in Q, thereby determining whether or not X is a (super) key in Q.

Example of algorithm 15.5

Consider the following relationship R(A,B,C,D) and set of functional dependencies F = { A -> BCD, BC -> AD, D->B}

1. Set Q:= R i.e.Q:=(A,B,C,D)
2. We have F = { A -> BCD, BC -> AD, D->B}
   A+ ->ABCD
   (BC)+ -> BCAD
   D+ -> DB => Here, D is not a key. So, it is not in BCNF.

Now, consider D -> DB .(X ->Y). We have Q(ABCD)

(Q-Y) => (AC)

(X U Y) => (DB)

Replace Q by (Q-Y) => R1

And X U Y =>R2.

i.e.

R1 => (AC)

R2 => (BD)

15.4 About Nulls, Dangling Tuples , and Alternative Relational Designs

15.4.1 Problems with NULL Values and Dangling Tuples

We must carefully consider the problems associated with NULLs when designing a relational database schema. There is no fully satisfactory relational design theory as yet that includes NULL values.

Problem 1 :- When some tuples have NULL values for attributes that will be used to join individual relations in the decomposition.

Ex:-

EMPLOYEE

Ename	**Ssn**	**Bdate**	**Address**	**Dnum**
Smith	101	-	RYTYYQE	5
John	102	-	WADSDGF	5
Vivek	103	-	GGJH	4
Anu	104	-	VJJKJJLK	NULL
Sindhu	105	-	JKJLKJLJHK	NULL

DEPARTMENT

Dname	**Dnum**	**Dmgr_ssn**
Research	5	3456

Administration	4	3452
Head Quarters	1	3510

Here, we have two relations, EMPLOYEE and DEPARTMENT. When we try to join these two tables, the last two tuples in EMPLOYEE represents newly hired employees who have not yet been assigned to a department. Now, suppose we want to retrieve all employees list and apply natural join, the last two tuples will not appear in the result. On the contrary, the outer join operation will address this issue.

Note:- whenever a relational database schema is designed in which two or more relations are interrelated via foreign keys, particular care must be taken about having potential NULL values. Also, should avoid having NULL values for attributes on which we may apply aggregate functions.

Problem 2:- Dangling tuples

A tuple that does not participate in a natural join is called a Dangling tuple. Dangling tuples may indicate a consistence problem in the database.

Suppose we decompose our table into many tables like emp1,emp2,...emp4.

Here, emp3 do not include a tuple which has not been assigned a department. If we use JOIN on these two tables, the tuples without values for Dnum will not appear in the result. These are called Dangling tuples. Because they are represented in only one of the two relations and hence are lost if we apply JOIN operation.

15.4.2 Discussion of Normalization algorithms and alternative Relational Designs.

1. one of the main issue in the algorithms discussed so far is that database designer must specify all the relevant functional dependencies among the database attributes. As the size of database increases, it becomes tough to do so. Failing to specify one or two important dependencies may result in undesirable design.

2. Another issue is that, these algorithms are not deterministic in general.

In synthesis algorithm, it requires the specification of a minimal cover G for given set of FDs. But, we already know that we can have different minimal cover for same set of FDs. Some of these designs may not be desirable. So algorithm can give different designs depending upon the minimal cover used.

15.5 Further Discussion of Multivalued Dependencies and 4NF

15.5.1 Inference Rules for Functional and Multivalued Dependencies

The following inference rules IR1 through IR8 form a sound and complete set for inferring functional and multivalued dependencies from a given set of dependencies. Assume that all attributes are included in a universal relation schema R = {A1, A2, … , An} and that X, Y, Z, and W are subsets of R.

IR1 (reflexive rule for FDs): If X ⊇ Y, then X → Y.

IR2 (augmentation rule for FDs): {X → Y} |= XZ → YZ.

IR3 (transitive rule for FDs): {X → Y, Y → Z} |= X → Z.

IR4 (complementation rule for MVDs): {X →→ R} |= {X →→(R − (X ∪))}.

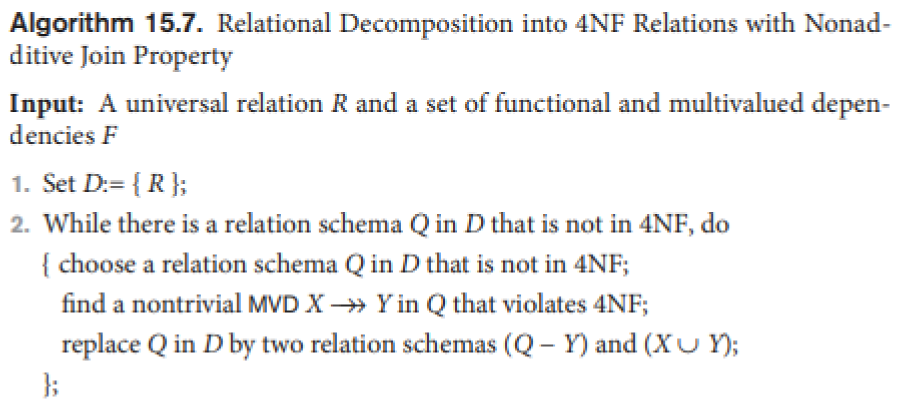
IR5 (augmentation rule for MVDs): If X →→ Y and W ⊇ Z, then WX →→ YZ.

IR6 (transitive rule for MVDs): {X →→ Y, Y →→ Z} | = X →→ (X − Y).

IR7 (replication rule for FD to MVD): {X → Y} | = X →→ Y.

IR8 (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.

IR1 through IR3 are Armstrong's inference rules for FDs alone.



Algorithm 15.7. Relational Decomposition into 4NF Relations with Nonadditive Join Property

Input: A universal relation R and a set of functional and multivalued dependencies F

```
1. Set D:= { R };
2. While there is a relation schema Q in D that is not in 4NF, do
   { choose a relation schema Q in D that is not in 4NF;
     find a nontrivial MVD $X \twoheadrightarrow Y$ in Q that violates 4NF;
     replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
   };
```