Basics of Functional Dependencies and Normalization for Relational Databases.

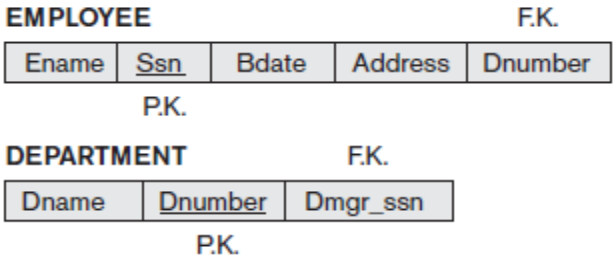
1. Informal Design Guidelines for Relation Schemas.

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

1.1 Imparting Clear Semantics to Attributes in Relations.

- The group of attributes belonging to one relation have certain real-world meaning and a proper interpretation associated with them.
- The **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple.
- If the conceptual design done carefully and the mapping procedure is followed systematically, the relational schema design should have a clear meaning.
- The meaning of the EMPLOYEE relation schema is quite simple: Each tuple represents an employee, with values for the employee's name (Ename), Social Security number (Ssn), birth date (Bdate), and address (Address), and the number of the department that the employee works for (Dnumber).



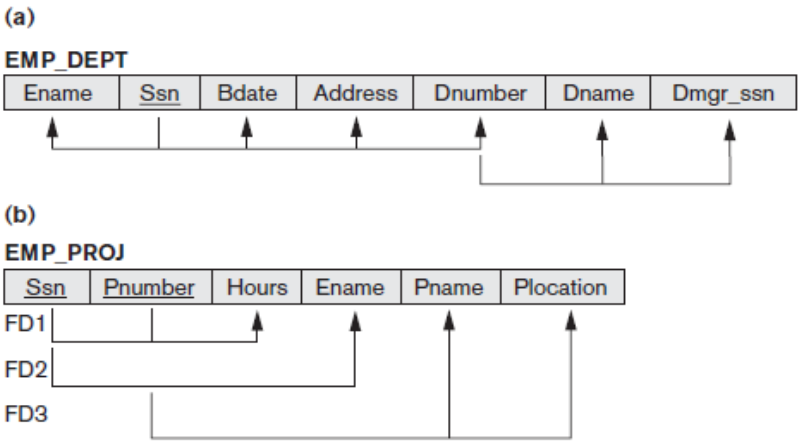
Guideline 1

- 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.
- If a relation schema corresponds to one entity type or one relationship type, it is straightforward to interpret and to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Examples of Violating Guideline 1.

- The following relation schema EMP_DEPT and EMP_PROJ have clear semantics but they violate Guideline 1 by mixing attributes from distinct real-world entities: EMP_DEPT mixes attributes of employees and departments, and EMP_PROJ mixes attributes of employees and projects and the WORKS_ON relationship. Hence, they fare poorly against the above measure of design quality.

Figure 15.3
Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.



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.
- For example, The space used by the two base relations **EMPLOYEE** and **DEPARTMENT** is less compared to **EMP_DEPT** .

EMPLOYEE

Ename	Ssn	Bdate	Address	Dnumber
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5

DEPARTMENT

Dname	Dnumber	Dmgr_ssn
Research	5	333445555
Administration	4	987654321

1. In **EMP_DEPT**, 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.

Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555

2. EMP_DEPT base relation is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. Storing natural joins of base relations leads to an additional problem referred to as **update anomalies**.

Update anomalies can be classified into **insertion anomalies, deletion anomalies, and modification anomalies.**

Insertion Anomalies.

Insertion anomalies can be differentiated into two types, based on the EMP_DEPT relation:

1. To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs (if the employee does not work for a department as yet).
2. It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the attributes for employee. This violates the entity integrity for EMP_DEPT because Ssn is its primary key.

Deletion Anomalies.

The problem of deletion anomalies is related to the second insertion anomaly situation.

1. 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.
2. This problem does not occur in DEPARTMENT relation since tuples are stored separately.

Modification Anomalies.

1. 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.
2. 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.

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 can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes.
- SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable.
- NULLs can have multiple interpretations, such as the following:
  1. The attribute does not apply to this tuple. For example, Visa_status may not apply to U.S. students.
  2. The attribute value for this tuple is unknown. For example, the Date_of_birth may be unknown for an employee.
  3. The value is known but absent; For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Guideline 3

- 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.

1.4 Generation of Spurious Tuples

- Consider the two relation schemas EMP_LOCS and EMP_PROJ1. Suppose if we perform NATURAL JOIN operation on EMP_PROJ1 and EMP_LOCS, the result produces many more tuples than the original set of tuples .
- These additional tuples are called **spurious tuples** because they represent spurious information that is not valid. The spurious tuples are marked by asterisks (*) in Figure 15.6.
- Decomposing EMP_PROJ into EMP_LOCS and EMP_PROJ1 is undesirable because when we JOIN them back using NATURAL JOIN, we do not get the correct original information. This is because in this case Plocation is the attribute that relates EMP_LOCS and EMP_PROJ1, and Plocation is neither a primary key nor a foreign key in either EMP_LOCS or EMP_PROJ1.

EMP_LOCS

Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston

	Ssn	Pnumber	Hours	Pname	Plocation	Ename
	123456789	1	32.5	ProductX	Bellaire	Smith, John B.
*	123456789	1	32.5	ProductX	Bellaire	English, Joyce A.
	123456789	2	7.5	ProductY	Sugarland	Smith, John B.
*	123456789	2	7.5	ProductY	Sugarland	English, Joyce A.
*	123456789	2	7.5	ProductY	Sugarland	Wong, Franklin T.
	666884444	3	40.0	ProductZ	Houston	Narayan, Ramesh K.

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.
- Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

2 Functional Dependencies

- **Definition :**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 tuples in 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].
- A **functional dependency**, denoted by X → Y means that the values of the Y are determined by the values of X .
- A functional dependency is a property of the **semantics** or **meaning of the attributes**. The database designers will use their understanding of the semantics of the attributes of R to specify the functional dependencies in a relation.
- Consider the relation schema EMP_PROJ from the semantics of the attributes and the relation, the following functional dependencies should hold:
  a. Ssn→Ename
  b. Pnumber →{Pname, Plocation}
  c. {Ssn, Pnumber}→Hours
- These functional dependencies specifies 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), and
(c) Combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours).

- Types of functional dependency :
  1. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more;
     Ex: **{Ssn, Pnumber}** → **Hours** is a full dependency (neither Ssn → Hours nor Pnumber→Hours holds).
  2. A functional dependency $X \rightarrow Y$ is a **partial functional dependency** if removal of any attribute A from X and the dependency still holds;
     Ex: **{Ssn, Pnumber}→Ename** is partial because Ssn→Ename holds.
  3. 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 such that $X \rightarrow Z$ and $Z \rightarrow Y$ hold.
     Ex: The dependency Ssn→Dmgr_ssn is transitive in EMP_DEPT, because of the dependencies Ssn → Dnumber and Dnumber → Dmgr_ssn.
  4. **Trivial Functional Dependency**. If a **functional dependency** (FD) X → Y holds, where Y is a subset of X, then it is called a **trivial** FD.
     Non-**trivial** − If an FD X → Y holds, where Y is not a subset of X, then it is called a non-**trivial** FD.

3 Normal Forms Based on Primary Keys

3.1 Normalization of Relations

- The normalization process, as first proposed by Codd (1972). Codd proposed three normal forms, which he called first, second, third normal form and Boyce-Codd normal form (BCNF). All these normal forms are based on 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.
- It can be considered as a "filtering" or "purification" process to make the design have successively better quality.
- **Definition.** 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.

3.2 Definitions of Keys and Attributes Participating in Keys.

- **Definition.** A **superkey** of a relation schema R is a set of attributes $S \subseteq R$ 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 any more.
- The difference between a key and a superkey is that a key has to be *minimal;* that is, if we have a key $K = \{A1, A2, ..., Ak\}$ of R, then $K - \{Ai\}$ is not a key of R.
  Ex: {Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename}, {Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys.
- 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. {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary key.
- **Definition.** 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.
  Ex: Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

3.3 First Normal Form

- **First normal form (1NF**) 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.
- Consider the following DEPARTMENT relation, It is not in 1NF. Because the domain of Dlocations contains sets of values and hence is nonatomic.

DEPARTMENT

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

- 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 relation is the combination {Dnumber, Dlocation}.

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT. In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing *redundancy* in the relation.

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

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.

- 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.*
- Figure 15.10 the EMP_PROJ relation represents an employee entity, and a relation PROJS(Pnumber, Hours) *within each tuple*.
- The schema of this EMP_PROJ relation can be represented as follows: EMP_PROJ(Ssn, Ename, {PROJS(Pnumber, Hours)}). The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses ( ).
- To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2, as shown in Figure 15.10(c).

(a)
EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

(b)
EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0

(c)
EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

Figure 15.10 Normalizing nested relations into 1NF.
(a) Schema of the EMP_PROJ relation with a *nested relation* attribute PROJS.
(b) Sample extension of the EMP_PROJ relation showing nested relations within each tuple.
c) Decomposing EMP_PROJ into EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

3.4 Second normal form (2NF)

- **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*.**

- 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 any more.
  Example1: {Ssn, Pnumber} → Hours is a full dependency (neither Ssn → Hours nor Pnumber→Hours holds).
- A functional dependency X → Y is a **partial functional dependency** if removal of any attribute A from X and the dependency still holds;
  Example2: The dependency {Ssn, Pnumber}→Ename is partial because Ssn→Ename holds.
- The EMP_PROJ relation is in 1NF but is not in 2NF. The nonprime attribute Ename violates 2NF because of FD2 , Pname and Plocation violates 2NF because of FD3.
- The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.

- If a relation schema is not in 2NF, it can be *second normalized by* decomposing EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 15.11(a), each of which is in 2NF.

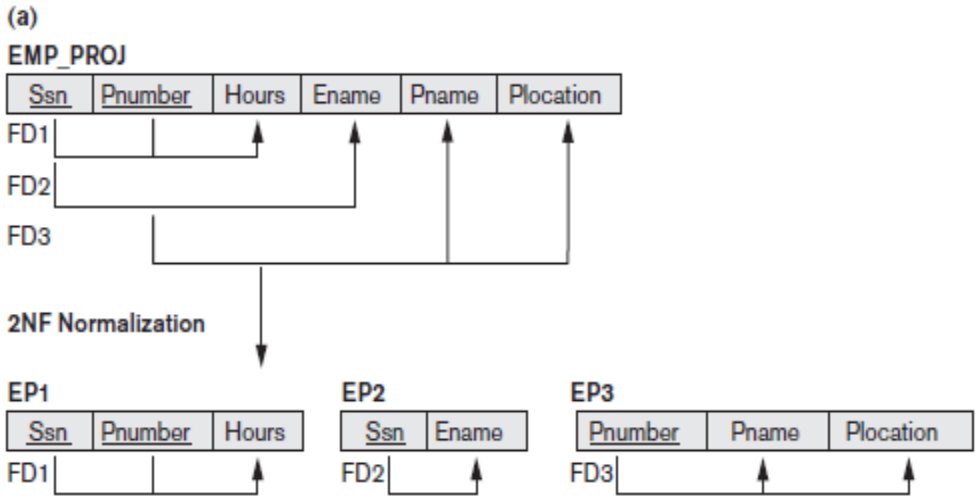


Figure 15.11- Normalizing into 2NF and 3NF.
(a) Normalizing EMP_PROJ into 2NF relations.

Example 2:

- Consider the relation schema LOTS shown in Figure 15.12(a). There are two candidate keys: Property_id# and {County_name, Lot#}; lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.

- There are two candidate keys Property_id# and {County_name, Lot#}. We choose Property_id# as the primary key, so it is underlined in Figure 15.12(a).

- The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key {County_name, Lot#}, due to FD3.

- To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 15.12(b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF.

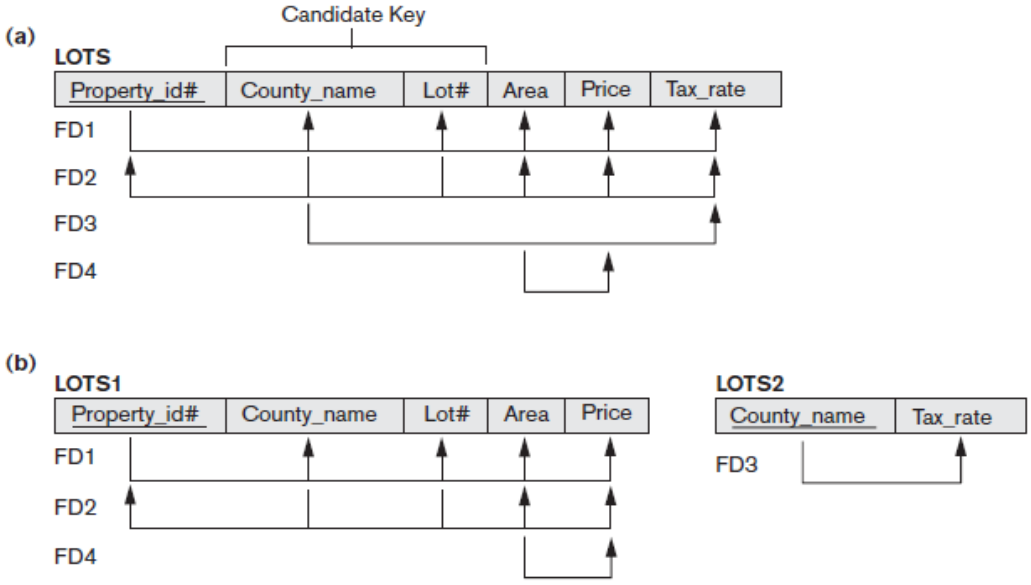


Figure 15.12

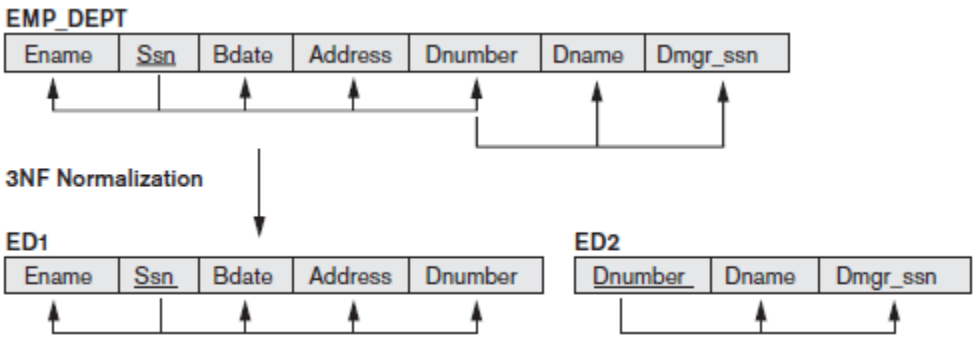
(a) The LOTS relation with its functional dependencies FD1 through FD4.

(b) Decomposing into the 2NF relations LOTS1 and LOTS2.

3.5 Third Normal Form (3NF)

- **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.**
- 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 such that $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

  Ex: The dependency Ssn→Dmgr_ssn is transitive in EMP_DEPT, because of the dependencies Ssn → Dnumber and Dnumber → Dmgr_ssn.



Normalizing EMP_DEPT into 3NF relations.

- The relation schema EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn and Dname on Ssn via Dnumber. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in the above figure.

Example 2:

- FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1.
- To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 15.12(c).We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the lefthand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

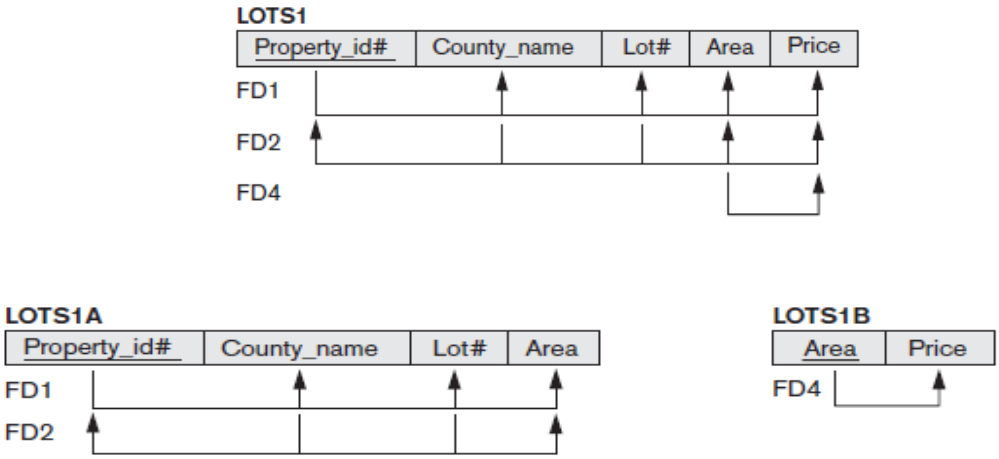


Figure 15.12(c). Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B.

4. Boyce-Codd Normal Form.

- **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*.**
- The BCNF is based on the concept non trivial dependency. If an FD X → Y holds, where Y is not a subset of X, then it is called a non-**trivial** FD.
- FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 15.13(a).

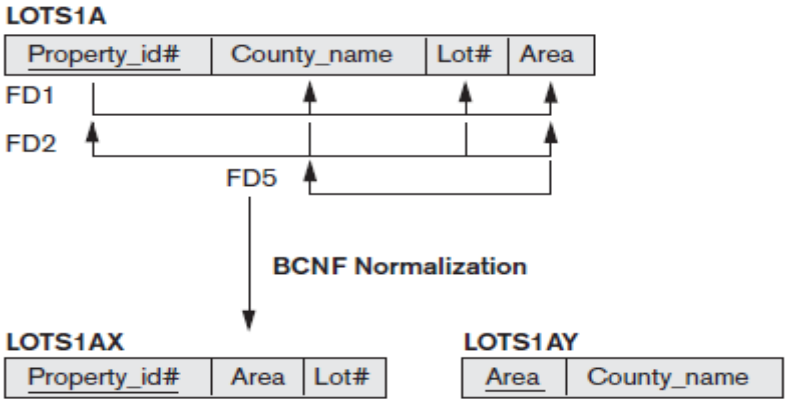


Figure 15.13
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition.

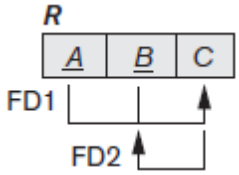
- Consider Figure 15.14, which shows a relation TEACH with the following dependencies:
  FD1: {Student, Course} → Instructor
  FD2: Instructor → Course

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Figure 15.14 A relation TEACH that is in 3NF but not BCNF.

- {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 15.13(b), with Student as *A*, Course as *B*, and Instructor as *C*. Hence this relation is in 3NF but not BCNF.



- The relation can be decomposed into one of the three following possible pairs:

  1. {Student, Instructor} and {Student, Course}.
  2. {Course, Instructor} and {Course, Student}.
  3. {Instructor, Course} and {Instructor, Student}.

- All three decompositions *lose* the functional dependency FD1. The *desirable* decomposition is (Instructor, Course) and (Instructor, Student), because it is nonadditive join decomposition
- The relation schemas $R1$ and $R2$ form a nonadditive join decomposition of R with respect to a set F of functional dependencies if and only if $(R1 \cap R2) \rightarrow (R1 - R2)$ or, $(R1 \cap R2) \rightarrow (R2 - R1)$.

5 . Multivalued Dependency and Fourth Normal Form

- **Definition: 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 $t1$ and $t2$ exist in r such that $t1[X] = t2[X]$. Then two tuples t3 and t4 should also exist in r with the following properties, where we use Z to denote $(R - (X \cup Y))$**
  t3[X] = t4[X] = t1[X] = t2[X].
  t3[Y] = t1[Y] and t4[Y] = t2[Y].
  t3[Z] = t2[Z] and t4[Z] = t1[Z].

- An MVD X →→ Y in R is called a **trivial MVD** if (a) *Y* is a subset of *X*, or (b) *X* ∪ Y = *R*. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**. For example, the relation EMP_PROJECTS in Figure 15.15(b) has the trivial MVD Ename →→ Pname.

EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

15.15(b)

- **Definition. 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*.**

- An all-key relation is always in BCNF since it has no FDs. An all-key relation such as the EMP relation in Figure 15.15(a), which has no FDs but has the MVD Ename→→ Pname | Dname, is not in 4NF.

- A relation that is not in 4NF due to a nontrivial MVD must be decomposed To convert it into a set of relations in 4NF. The decomposition removes the redundancy caused by the MVD.
- Consider the EMP relation in Figure 15.15(a). EMP is not in 4NF because of the nontrivial MVDs Ename→→ Pname and Ename →→ Dname, and Ename is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure 15.15(b). 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.

(a) **EMP**

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

(b) **EMP_PROJECTS**

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

Figure 15.15
(a) The EMP relation with two MVDs: Ename →→ Pname and Ename →→ Dname.
(b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.

- Whenever we decompose a relation schema R into $R1 = (X \cup Y)$ and $R2 = (R - Y)$ based on an MVD $X \rightarrow\rightarrow Y$ that holds in R, the decomposition has the nonadditive join property.
- The following algorithm shows 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 $X \rightarrow\rightarrow Y$ in Q that violates 4NF;
     replace Q in D by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
     };

6. Join Dependencies and Fifth Normal Form

- A relation schema R when divided in to R_1 and R_2 has the lossless property and if the natural join is applied $(R_1 * R_2)$ we will get the original relation R.
- In some cases there may be no lossless join decomposition of R if the number of decomposition is equal to two. But if the same relation is decomposed in to more than two relations we have a lossless decomposition. This dependency depends on the number of decomposition and hence referred as join dependency
- **Definition.** 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 $R1$, $R2$, ..., Rn. Hence, for every such r we have $*(\pi R1\ (r), \pi R2\ (r), ..., \pi Rn\ (r)) = r$.
- A join dependency JD($R1$, $R2$, ..., Rn), specified on relation schema R, is a **trivial** JD if one of the relation schemas Ri in JD($R1$, $R2$, ..., Rn) 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.
- **Definition.** 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($R1$, $R2$, ..., Rn) in F^+, every Ri is a superkey of R.

SUPPLY

Sname	Part_name	Proj_name
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

Figure 15.15 (c)The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD(*R*1, *R*2, *R*3).

R_1

Sname	Part_name
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R_2

Sname	Proj_name
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R_3

Part_name	Proj_name
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Figure 15.15 (d) Decomposing the relation SUPPLY into the 5NF relations *R*1, *R*2, *R*3.

- For example of a JD, consider once again the SUPPLY all-key relation in Figure 15.15(c). Suppose that the following additional constraint always holds: Whenever a supplier *s* supplies part *p*, *and* a project *j* uses part *p*, *and* the supplier *s* supplies *at least one* part to project *j*, *then* supplier *s* will also be supplying part *p* to project *j*. This constraint can be restated in other ways and specifies a join dependency JD(*R*1, *R*2, *R*3) among the three projections *R*1(Sname, Part_name), *R*2(Sname, Proj_name), and *R*3(Part_name, Proj_name) of SUPPLY.
- Figure 15.15(d) shows how the SUPPLY relation with the join dependency is decomposed into three relations R1, R2, and R3 that are each in 5NF. Notice that applying a natural join to any two of these relations produces spurious tuples, but applying a natural join to all three together does not.

16. Relational Database Design Algorithms and Dependencies

1 Inference Rules for Functional Dependencies

- The set of functional dependencies are specified by F on relation schema R, other functional dependencies can be inferred or deduced from the FDs in F.
- For example, Department has one manager, the Dept_no uniquely determines Mgr_ssn, and manager uniquely determines phone number called Mgr_phone then these two dependencies together imply that Dept_no → Mgr_phone.
  (Dept_no → Mgr_ssn),
  (Mgr_ssn→Mgr_phone),
  Dept_no → Mgr_phone
- This is an inferred FD and need not be explicitly stated in addition to the two given FDs. Therefore, it is useful to define a concept called closure formally that includes all possible dependencies that can be inferred from the given set F.
- **Definition-** 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 in Figure 15.3(a):
  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}
  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, The set of **inference rules** are used to infer new dependencies from a given set of dependencies.
- The notation F |=X → Y to denote that the functional dependency X→Y is inferred from the set of functional dependencies F. The FD {X,Y}→Z is abbreviated to XY→Z, and the FD {X, Y, Z} → {U, V} is abbreviated to XYZ → UV.

The six inference rules IR1 through IR6 for functional dependencies:

1. **IR1 (reflexive rule): If X ⊇ Y, then X→Y.** The **reflexive rule** states that a set of attributes always determines itself or any of its subsets.
   Ex: **{fname, lname} → {fname}**

2. **IR2 (augmentation rule): {X→Y} |=XZ→YZ.** The **augmentation rule (IR2)** says that adding the same set of attributes to both the left- and right-hand sides of a dependency results in another valid dependency.
   Ex: **If {SSN} → {fname} then: {SSN, DName} → {fname, DName}**

3. **IR3 (transitive rule): {X→Y, Y→Z} |= X→Z. This F**unctional dependencies are transitive.
   Ex: **If: {SSN} → {DNO} {DNO} → {DName}**
   Then:
   {SSN} → {DName}

4. **IR4 (decomposition, or projective, rule): {X→YZ} |=X→Y.** The **decomposition rule (IR4)** says that we can remove attributes from the right-hand side of a dependency; applying this rule repeatedly can decompose the FD ***X → YZ*** to ***X → Y*** and ***X → Z.***
   Ex: **If {SSN} → {fname, DNO} Then**
   {SSN} → {fname}
   {SSN} → { DNO}
   .
5. **IR5 (union, or additive, rule): {X→Y, X→Z} |=X→YZ.** allows to combine a set of dependencies {X→A1, X→A2, ..., X→An} into the single FD X→{A1, A2, ..., An}.

6. **IR6 (pseudotransitive rule): {X→Y,WY→Z} |=WX→Z. , A**llows us to replace a set of attributes Y on the left hand side of a dependency with another set X that functionally determines Y, and can be derived from IR2 and IR3 if we augment the first functional dependency X→Y with W (the augmentation rule) and then apply the transitive rule.

The inference rules can be proved based on contradiction.

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] because X ⊇ Y; hence, X→Y must hold in r.

Proof of IR2 (by contradiction).
Assume that X→Y holds in a relation instance r of R but that XZ→YZ does not hold. 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], and from (2) and (5) we deduce (6) t1 [YZ] = t2 [YZ], contradicting (4).

Proof of IR3.
Assume that (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).

A systematic way to determine these additional functional dependencies is first to determine each set of attributes *X* that appears as a left-hand side of some functional dependency in *F* and then to determine the set of *all attributes* that are dependent on *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 16.1. 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
        old X+  := X+;
        for each functional dependency Y→Z in F do
            if X+  ⊇ Y then X+:= X+ ∪ Z;
until (X+  = old X+);
```

Algorithm starts by setting X^+ to all the attributes in X. By IR1, we know that all these attributes are functionally dependent on X. 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.

Example 1, consider the relation schema EMP_PROJ from the semantics of the attributes, the following set *F* of functional dependencies are identified.
F = {Ssn → Ename,

Pnumber → {Pname, Plocation},
{Ssn, Pnumber} → Hours}

Using Algorithm 16.1, we calculate the following closure sets with respect to *F*:
$\{Ssn\}^+ = \{Ssn, Ename\}$
$\{Pnumber\}^+ = \{Pnumber, Pname, Plocation\}$
$\{Ssn, Pnumber\}^+ = \{Ssn, Pnumber, Ename, Pname, Plocation, Hours\}$

Example 2. Consider the following relation schema.
CLASS(Classid,Course#,Instr_name,Credit_hrs,Text,Publisher,Classroom , Capacity).
Let F, the set of functional dependencies for the above relation
FD1: Classid → Course#, Instr_name, Credit_hrs, Text, Publisher, Classroom, Capacity;
FD2: Course# → Credit_hrs;
FD3: {Course#, Instr_name} → Text, Classroom;
FD4: Text → Publisher
FD5: Classroom → Capacity

Using the inference rules about the FDs and applying the definition of closure, we can define the following closures:
$\{Classid\}^+ =\{Classid,Course\#,Instr_name, Credit_hrs, Text, Publisher,Classroom, Capacity\}$
$\{Course\#\}^+ = \{Course\#, Credit_hrs\}$
$\{Course\#, Instr_name\}^+ = \{Course\#, Credit_hrs, Text, Publisher, Classroom, Capacity\}$

2. Equivalence of Sets of Functional Dependencies:

- A set of functional dependencies F is said to cover another set of functional dependencies E if every FD in E is also in F^+; that is, if every dependency in E can be inferred from F; alternatively, we can say that E is covered by F.
- Two sets of functional dependencies E and F are equivalent if $E^+ = F^+$. Therefore, equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to F if both the conditions E covers F and F covers E hold.
- 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.

Problem 1:
Consider two sets of FDs, F and G, F = {A → B, B → C, AC→D} and G = {A→B, B→C, A→D} Are F and G equivalent?
Take the attributes from the LHS of FDs in F and compute attribute closure for each using FDs in G:

A^+ using G = ABCD; A → A; **A → B;** A →C; A → D (Augmenting A to both sides of A→C, AA→AC, ie A→AC from transitive rule A→AC and AC→D we get A→D) B^+ using G = BC; B → B; **B -> C**;
AC^+ using G = ABCD; AC → A; AC→ B; AC → C; **AC → D**;

Notice that all FDs in F (highlighted) can be inferred using FDs in G. To see if all FDs in G are inferred by F, compute attribute closure for attributes on the LHS of FDs in G using FDs in F:

A+ using F = ABCD; A → A; **A → B**; A→ C; **A→ D**;
B+ using F = BC; B → B; **B → C**;

Since all FDs in F can be obtained from G and vice versa, we conclude that F and G are equivalent.

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. 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.
- The 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*.

Algorithm 16.2. 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 $X \rightarrow \{A1, A2, ..., An\}$ in *F* by the *n* functional dependencies $X \rightarrow A1$, $X \rightarrow A2$, ..., $X \rightarrow An$.
3. For each functional dependency $X \rightarrow A$ in *F* for each attribute *B* that is an element of *X* if $\{ \{F - \{X \rightarrow A\} \} \cup \{ (X - \{B\}) \rightarrow A\} \}$ is equivalent to *F* then replace $X \rightarrow A$ with $(X - \{B\}) \rightarrow A$ 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 1: 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*.

- Step1:**Break down** the **RHS** of each **functional dependency** into a ***single*** **attribute ,here all RHS attribute are single.** E : $\{B \rightarrow A, D \rightarrow A, AB \rightarrow D\}$.
- In step 2 we need to determine if $AB \rightarrow D$ *h*as any redundant attribute on the left-hand side; that is, can it be replaced by $B \rightarrow D$ *o*r $A \rightarrow D$?
  Consider B →A, by augmenting B on both sides , we have $BB \rightarrow AB$, or $B \rightarrow AB$.
  Hence by the transitive rule we get $B \rightarrow D$. Because $B \rightarrow AB$ *and* $AB \rightarrow D$.
- We now have $E = \{B \rightarrow A, D \rightarrow A, B \rightarrow D\}$. 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→D, D→A*}.**

Example 2: Given the set of FDs G: {A → BCDE, CD → E}. Find a minimal cover for G.

- The given FDs are NOT in the canonical form. So we first convert them into:
  E: {A → B, A→ C, A→ D, A→ E, CD → E}.
- In step 2 of the algorithm, for CD → E, neither C nor D is extraneous on the left-hand side, since we cannot show that C → E or D → E from the given FDs. Hence we cannot replace it with either.
- In step 3, we want to see if any FD is redundant. Since A→ CD and CD → E, by transitive rule (IR3), we get A→ E. Thus, A→ E is redundant in G.
- So we are left with the set F, equivalent to the original set G as: {A → B, A→ C, A→ D, CD → E}. F is the minimum cover. The first three FDs can be combined using the union rule (IR5). Minimum cover of G is F: {A → BCD, CD → E}.

Example 3: Find a **minimal cover** for the following set of **functional dependencies:** Relation R = (A, B, C, D, E, F)
F= { A → C, AC → D, E → ADF }

- Step 1: **Break down** the **RHS** of each **functional dependency** into a ***single*** **attribute** :
  F_{min} = { A → C, AC → D, E → A, E → D, E → F }
- Step 2, **Consider functional dependencies** whose **LHS** has ≥ **2 attributes. Replace** AC → D to A → D or C → D ?
  By augmenting A to both sides of A → C ,We get AA →AC ie A→AC , Hence by transitive rule we get A→D (A→AC and AC→D).
  F_{min} = { A → C, A → D, E → A, E → D, E → F }
- Step 3 **:Minimize** the **redundancy** by **removing** ***unnecessary*** **functional dependencies.**
  By using the transitive rule on E → A and A → D, we derive E → D Hence E → D is redundant and can be eliminated.
  F_{min} = { A → C, A → D, E → A, E → F }
- **Minimal cover = { A → CD, E → AF }**

Algorithm 16.2(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} };

1. Suppose that you are given a relation R = (A,B,C,D,E) with the following functional dependencies: {CE → D,D → B,C → A}.

Find all candidate keys.

Answer: From the given set of functional dependencies, we can observe that only the attributes C and E are present only on LHS. Hence, we can try with C and E attributes to find candidate keys.

C alone cannot determine all the other attributes.

Hence, C → CA

E alone cannot determine all the other attributes.

Hence, E → E

C and E together can form a candidate key.

CE → ABCDE. Hence, **CE is the only candidate key** for the given relation R.

Example: Let R=(A,B,C) a relation and F={A-->B, B-->C} a set of dependencies for this relation.

1) Find a canditate key in R.
2) Is R in BCNF? Justify your answer.
3) Find the possible violations of BCNF for R.
4) If R is not in BCNF, give a decomposition of R in relations that will be in BCNF.

Solution:

1) We apply the properties of functional dependencies to prove that A→{A,B,C}. Thus, A is a candidate key in R.
2) R is not in BCNF because it is not in 3NF
3) There is transitive dependency B→C.
4) A decomposition in BCNF is : R1=(A,B) and R2(B,C).

16.2 Properties of Relational Decompositions

16.2.1 Relation Decomposition and Insufficiency of Normal Forms.

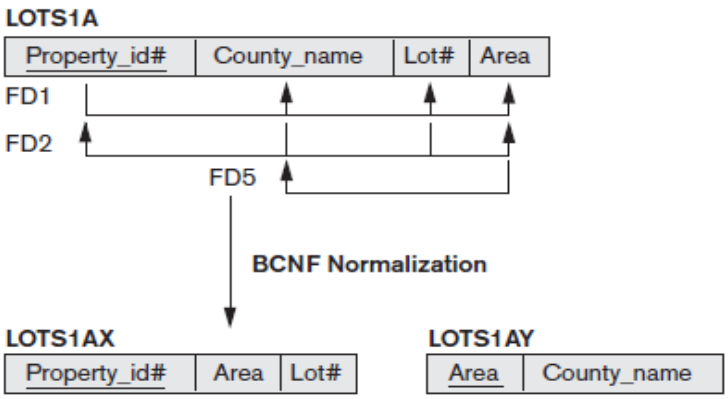
- The relational database design algorithm starts from a single **universal relation schema** *R* = {*A*1, *A*2, ..., *An*} that includes *all* the attributes of the database.
- The **universal relation assumption** is that every attribute name is unique.
- The set *F* of functional dependencies that 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* = {*R*1, *R*2, ..., *Rm*} that will become the relational database schema; *D* is called a **decomposition** of *R*.

$$\bigcup_{i=1}^{m} R_i = R$$

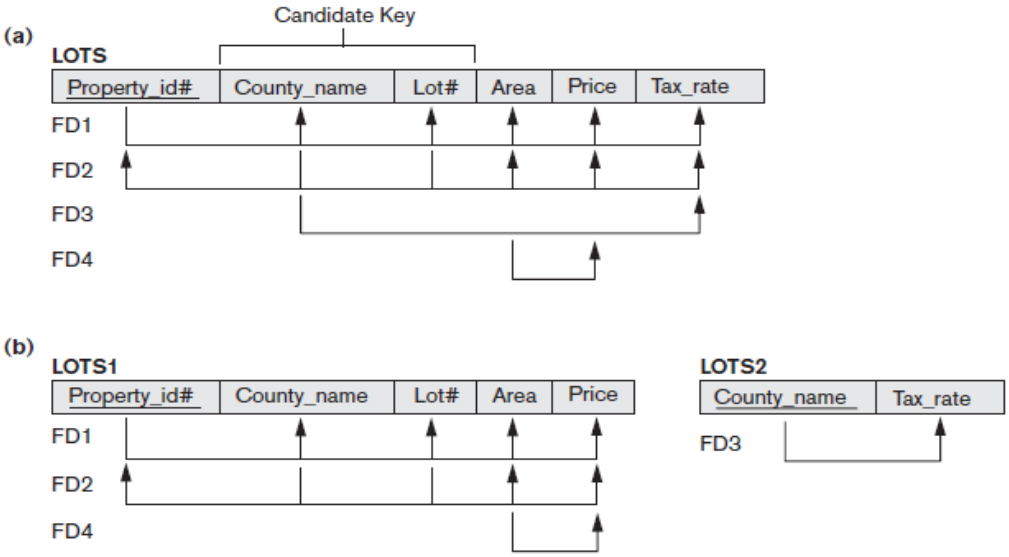
- Each attribute in *R* will appear in at least one relation schema *Ri* in the decomposition so that no attributes are *lost*; This is called the **attribute preservation** condition of a decomposition.
- Another goal of decomposition is to have each individual relation Ri in the decomposition D be in BCNF or 3NF. Additional properties of decomposition are needed to prevent from generating spurious tuples

16.2.2 Dependency Preservation Property of a Decomposition

- Functional dependency $X \rightarrow Y$ specified in *F* either appeared directly in one of the relation schemas *Ri* in the decomposition *D* or could be inferred from the dependencies that appear in some *Ri*. Informally, this is the ***dependency preservation condition.***
- **Definition.** Given a set of dependencies *F* on *R*, the **projection** of *F* on *Ri*, denoted by $\pi Ri(F)$ where *Ri* is a subset of *R*, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in *Ri*.
  The decomposition ***D* = {*R*1, *R*2, ..., *Rm*}** of *R* is **dependency-preserving** with respect to *F* if the union of the projections of *F* on each *Ri* in *D* is equivalent to *F*; that is, $((\pi R1(F)) \cup ... \cup (\pi Rm(F)))^+ = F^+$.
- If a decomposition is not dependency-preserving, some dependency is **lost** in the decomposition. To check that a lost dependency holds, we must take the JOIN of two or more relations in the decomposition to get a relation
- An example of a decomposition that does not preserve dependencies is shown in the following Figure , in which the functional dependency FD2 is lost when LOTS1A is decomposed into {LOTS1AX, LOTS1AY}.



- The decompositions in follwoing Figure are dependency-preserving.



- **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.

16.2.3 Nonadditive (Lossless) Join Property of a Decomposition

- Decomposition *D* should possess 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.
- **Definition.** Formally, a decomposition $D = \{R_1, R_2, ..., R_m\}$ of *R* has the **lossless (nonadditive) join property** with respect to the set of dependencies *F* on *R* if, for *every* relation state *r* of *R* that satisfies *F*, the following holds, where * is the NATURAL JOIN of all the relations in *D*: $*(\pi_{R1}(r), ..., \pi_{Rm}(r)) = r$.
- **Algorithm 16.3.** 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 *a* symbols, then the decomposition has the nonadditive join property; otherwise, it does not.

- Figure 16.1(a) shows how Algorithm16.3 can be appiled for the decomposition of the EMP_PROJ relation schema into two relation schemas EMP_PROJ1 and EMP_LOCS. The loop in step 4 of the algorithm cannot change any *b* symbols to *a* symbols; hence, the resulting matrix *S* does not have a row with all *a* symbols, and so the decomposition does not have the nonadditive join property.
- Figure 16.1(b) shows another decomposition of EMP_PROJ (into EMP, PROJECT, and WORKS_ON) that does have the nonadditive join property.
- Figure 16.1(c) shows how we apply the algorithm to that decomposition. Once a row consists only of *a* symbols, we conclude that the decomposition has the nonadditive join property, and we can stop applying the functional dependencies (step 4 in the algorithm) to the matrix *S*.

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)

16.2.4 Testing Binary Decompositions for the Nonadditive Join Property

- Algorithm for testing non additive join property allows us to test whether a particular decomposition *D* into *n* relations obeys the nonadditive join property with respect to a set of functional dependencies *F*. There is a special case of a decomposition called a **binary decomposition**— decomposition of a relation *R* into two relations.

- **Property NJB (Nonadditive Join Test for Binary Decompositions).** A decomposition $D = \{R1, R2\}$ of R has the lossless (nonadditive) join property with respect to a set of functional dependencies F on R *if and only if* either
  The FD $((R1 \cap R2) \rightarrow (R1 - R2))$ is in $F+$, or
  The FD $((R1 \cap R2) \rightarrow (R2 - R1))$ is in $F+$.

16.2.5 Successive Nonadditive Join Decompositions

Claim 2 (Preservation of Nonadditivity in Successive Decompositions).
If a decomposition $D = \{R1, R2, ..., Rm\}$ of R has the nonadditive (lossless) join property with respect to a set of functional dependencies F on R, and if a decomposition $Di = \{Q1, Q2, ..., Qk\}$ of Ri has the nonadditive join property with respect to the projection of F on Ri, then the decomposition $D2 = \{R1, R2,..., Ri-1, Q1, Q2, ..., Qk, Ri+1, ..., Rm\}$ of R has the nonadditive join property with respect to F.

16.3 Algorithms for Relational Database Schema Design

16.3.1 Dependency-Preserving Decomposition into 3NF Schemas

Algorithm 16.4. Relational Synthesis into 3NF with Dependency Preservation 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 $\{X \cup \{A1\} \cup \{A2\} ... \cup \{Ak\}\}$, where $X \rightarrow A1$, $X \rightarrow A2$, ..., $X \rightarrow Ak$ are the only dependencies in G with X as the left-hand-side (X is the key of this relation);
3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema to ensure the attribute preservation property.

Example: Consider the following universal relation:
U(Emp_ssn, Pno, Esal, Ephone, Dno, Pname, Plocation)

The following dependencies are present:
FD1: Emp_ssn → {Esal, Ephone, Dno}
FD2: Pno → { Pname, Plocation}
FD3: Emp_ssn, Pno → {Esal, Ephone, Dno, Pname, Plocation}

- The set of given FDs **F**= {Emp_ssn → Esal, Ephone, Dno; Pno→Pname, Plocation; Emp_ssn, Pno→Esal, Ephone, Dno, Pname, Plocation}.
- **Step 1:** By applying the minimal cover Algorithm in step 3 we see that Pno is a redundant attribute in Emp_ssn, Pno → Esal, Ephone, Dno. Moreover, Emp_ssn is redundant in

Emp_ssn, Pno→Pname, Plocation. Hence the minimal cover consists of FD1 and FD2 only (FD3 being completely redundant) as follows.
Minimal cover *G*: {Emp_ssn → Esal, Ephone, Dno**;** Pno → Pname, Plocation}

- **Step 2: We** get a 3NF design consisting of two relations with keys Emp_ssn and Pno as follows:
  *R*1 (Emp_ssn, Esal, Ephone, Dno)
  *R*2 (Pno, Pname, Plocation)
- These two relations might have lost the original information contained in the key of the universal relation U (namely, that there are certain employees working on certain projects in a many-to-many relationship).
- The **relational synthesis algorithm**, preserve the original dependencies, it makes no guarantee of preserving all of the information. Hence, the resulting design is a *lossy* design.

16.3.2 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. 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 $X \rightarrow Y$ in *Q* that violates BCNF;
replace *Q* in *D* by two relation schemas $(Q - Y)$ and $(X \cup Y)$;
} ;

16.3.3 Dependency Preserving and Nonadditive (Lossless) Join Decomposition into 3NF Schemas.

This algorithm yields a decomposition *D* of *R* that does the following:

- Preserves dependencies
- Has the nonadditive join property
- Is such that each resulting relation schema in the decomposition is in 3NF

Algorithm 16.6. 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 $\{X \cup \{A1\} \cup \{A2\} \ldots \cup \{Ak\}\}$, where $X \rightarrow A1$, $X \rightarrow A2$, ..., $X \rightarrow Ak$ are the only dependencies in G with X as left-hand-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;

Example: Consider the following universal relation:
U(Emp_ssn, Pno, Esal, Ephone, Dno, Pname, Plocation)

The following dependencies are present:
FD1: Emp_ssn → {Esal, Ephone, Dno}
FD2: Pno → { Pname, Plocation}
FD3: Emp_ssn, Pno → {Esal, Ephone, Dno, Pname, Plocation}

- **Step 1:** By applying the minimal cover Algorithm in step 3 we see that Pno is a redundant attribute in Emp_ssn, Pno → Esal, Ephone, Dno. Moreover, Emp_ssn is redundant in Emp_ssn, Pno→Pname, Plocation. Hence the minimal cover consists of FD1 and FD2 only (FD3 being completely redundant) as follows.
  Minimal cover G: {Emp_ssn → Esal, Ephone, Dno**;** Pno → Pname, Plocation}
- **Step 2: We** get a 3NF design consisting of two relations with keys Emp_ssn and Pno as follows:
  $R1$ (Emp_ssn, Esal, Ephone, Dno)
  $R2$ (Pno, Pname, Plocation)
- **Step 3**, we will generate a relation corresponding to the key {Emp_ssn, Pno}. 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.

Example 2 Consider the relation schema LOTS1A. Assume that this relation is given as a universal relation with the following functional dependencies:
FD1: Property_id→Lot#, County, Area
FD2: Lot#, County→Area, Property_id
FD3: Area→County

- For ease of reference, let us abbreviate the above attributes with the first letter for each and represent the functional dependencies as the set
  ***F* : { P → LCA, LC → AP, A → C }.**

- **Step 1**: If we apply the minimal cover Algorithm to *F*, We get
  ***F* : {P → L, P → C, P → A, LC → A, LC → P, A → C}.**
  In the set *F*, **P→A** can be inferred from **P→LC** and **LC→A**; hence **P→A** by transitivity and is therefore redundant.
  Thus, one possible minimal cover is Minimal cover ***GX*: {P → LC, LC → AP, A → C** }.
- **Step 2**: We produce design *X* (before removing redundant relations) using the above minimal cover as
  Design *X*: ***R*1 (<u>P</u>, L, C), R2 (<u>L</u>, <u>C</u>, A, P),** and **R3 (<u>A</u>, C).**
- In step 4 of the algorithm, we find that *R*3 is subsumed by *R*2 (that is, *R*3 is always a projection of *R*2 and *R*1 is a projection of *R*2 as well. Hence both of those relations are redundant. Thus the 3NF schema that achieves both of the desirable properties is (after removing redundant relations)
  Design *X*: *R*2 (L, C, A, P). or, in other words it is identical to the relation **LOTS1A (Lot#, County, Area, Property_id**) that we had determined to be in 3NF .

16.4 About Nulls, Dangling Tuples,

- The tuple which affects the resultant of the relation on applying different forms of Join is referred to as **dangling tuple**.
- When some tuples have NULL values for attributes that will be used to join individual relations leads to problem. To illustrate this, consider the database shown in Figure 16.2(a), where two relations EMPLOYEE and DEPARTMENT are shown. The last two employee tuples 'Berger' and 'Benitez' represent newly hired employees who have not yet been assigned to a department. If we apply the NATURAL JOIN operation on EMPLOYEE and DEPARTMENT (Figure 16.2(b)), two employee tuples 'Berger' and 'Benitez will *not* appear in the result. These are called Dangling tuples
- The OUTER JOIN operation, is used ie LEFT OUTER JOIN of EMPLOYEE with DEPARTMENT, tuples in EMPLOYEE that have NULL for the join attribute will still appear in the result, joined with an *imaginary* tuple in DEPARTMENT that has NULLs for all its attribute values. Figure 16.2(c) shows the result.

EMPLOYEE

Ename	Ssn	Dnum
Smith, John B.	123456789	5
Wong, Franklin T.	333445555	5
Zelaya, Alicia J.	999887777	4
Borg, James E.	888665555	1
Berger, Anders C.	999775555	NULL
Benitez, Carlos M.	888664444	NULL

DEPARTMENT

Dname	Dnum	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

Figure 16.2 (a) Some EMPLOYEE tuples have NULL for the join attribute Dnum.

Ename	Ssn	Bdate	Address	Dnum	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

Figure 16.2 (b) Result of applying NATURAL JOIN to the EMPLOYEE and DEPARTMENT relations.

Ename	Ssn	Bdate	Address	Dnum	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555
Berger, Anders C.	999775555	1965-04-26	6530 Braes, Bellaire, TX	NULL	NULL	NULL
Benitez, Carlos M.	888665555	1963-01-09	7654 Beech, Houston, TX	NULL	NULL	NULL

Figure 16.2
(c) Result of applying LEFT OUTER JOIN to EMPLOYEE and DEPARTMENT.

16.6 Other Dependencies and Normal Forms

16.6.1 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 $\boldsymbol{\pi_X(r(R)) \subseteq \pi_Y(s(S))}$.
- For example, we can specify the following inclusion dependencies on the relational schema in Figure 15.1:

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 9.6, we can specify the following inclusion dependencies:

EMPLOYEE.Ssn < PERSON.Ssn
ALUMNUS.Ssn < PERSON.Ssn
STUDENT.Ssn < PERSON.Ssn

16.6.2 Template Dependencies

.

- The idea behind template dependencies is to specify a template or example that defines each constraint or dependency. There are two types of templates: tuple-generating templates and constraint generating templates.
- A template consists of a number of **hypothesis tuples** that are meant to show an example of the tuples that may appear in one or more relations. The other part of the template is the **template conclusion**.
- For tuple-generating templates, the conclusion is a *set of tuples* that must also exist in the relations if the hypothesis tuples are there. For constraint-generating templates, the template conclusion is a *condition* that must hold on the hypothesis tuples. Using constraint generating templates, we are able to define **semantic constraints** those that are beyond the scope of the relational model in terms of its data definition language and notation.
- Figure 16.5 shows how we may define functional, multivalued, and inclusion dependencies by templates.
- Figure 16.6 shows how we may specify the constraint that an *employee's salary cannot be higher than the salary of his or her direct supervisor*on the relation schema EMPLOYEE in Figure 3.5.

Figure 16.5
Templates for some common type of dependencies.
(a) Template for functional dependency $X \rightarrow Y$.
(b) Template for the multivalued dependency $X \twoheadrightarrow Y$.
(c) Template for the inclusion dependency $R.X < S.Y$.

(a) R = {A, B, C, D}

Hypothesis

a_1	b_1	c_1	d_1
a_1	b_1	c_2	d_2

Conclusion $c_1 = c_2$ and $d_1 = d_2$

X = {A, B}
Y = {C, D}

(b) R = {A, B, C, D}

Hypothesis

a_1	b_1	c_1	d_1
a_1	b_1	c_2	d_2

Conclusion

a_1	b_1	c_2	d_1
a_1	b_1	c_1	d_2

X = {A, B}
Y = {C}

(c) R = {A, B, C, D} S = {E, F, G}

Hypothesis

a_1	b_1	c_1	d_1

Conclusion

c_1	d_1	g

X = {C, D}
Y = {E, F}

Figure 16.6
Templates for the constraint that an employee's salary must be less than the supervisor's salary.

EMPLOYEE = {Name, Ssn, . . . , Salary, Supervisor_ssn}

	a	b	c	d
Hypothesis	e	d	f	g
Conclusion			c < f	

16.6.3 Functional Dependencies Based on Arithmetic Functions and Procedures

- Sometimes some attributes in a relation may be related via some arithmetic functional relationship. If unique value of *Y* is associated with every *X*, we can still consider that the FD $X \rightarrow Y$ exists.
- For example,
  ORDER_LINE (Order#, Item#, Quantity, Unit_price, Extended_price, Discounted_price)

- In this relation, **(Quantity,Unit_price) → Extended** price by the formula **Extended_price = Unit_price * Quantity**. Hence, there is a unique value for Extended_price for every pair (Quantity, Unit_price), and thus it conforms to the definition of functional dependency.

16.6.4 Domain-Key Normal Form

- The idea behind **domain-key normal form** (**DKNF**) is to specify the *ultimate normal form* that takes into account all possible types of dependencies and constraints.
- A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation.
- For a relation in DKNF, all database constraints are by simply checking that each attribute value in a tuple is of the appropriate domain and that every key constraint is enforced.
- For example, consider a relation CAR(Make, Vin#) (where Vin# is the vehicle identification number) and another relation MANUFACTURE(Vin#,Country) (where Country is the country of manufacture). A general constraint may be of the following form: *If the Make is either 'Toyota' or 'Lexus,' then the first character of the Vin# is a 'J' if the country of manufacture is 'Japan';*

Exercise:1

Let R = (A,B,C,D) a relation and F= {C→D, C→A, B→C} a set of dependencies for this relation. 1) Find the candidate keys in R. 2) Identify the NF that R satisfies. Justify your anwer. 3) if R is not in BCNF, decompose it into a set of BCNF relations that preserve the dependencies.

Solution

1. Candidate key : B
2. R is in 2NF but not 3NF
3. C→D and C→A both because of violations of BCNF. One way to obtain a (lossless) join preserving decomposition is to decompose R into AC, BC, and CD.

Exercise:2

Let R = (A,B,C,D) a relation and F= {B→C, D→A} a set of dependencies for this relation. 1) Find the canditate keys in R. 2) Identify the NF that R satisfies. Justify your anwer. 3) if R is not in BCNF, decompose it into a set of BCNF relations that preserve the dependencies.

Solution

1. Candidate key: BD
2. R is in 1NF but not 2NF.
3. Both B→C and D→A cause BCNF violations. The decompositions: AD, BC, BD (obtained by first decomposing to AD, BCD) are losseless and join preserving