Chapter 5: Database Constraints and Relational Database Design

INTEGRITY CONSTRAINT An integrity constraint is a mechanism used by dbms to prevent invalid data entry into the table. It has enforcing the rules for the columns in a table. The types of the integrity constraints are: a) Domain Integrity b) Entity Integrity c) Referential Integrity.
a) Domain Integrity This constraint sets a range and any violations that take place will prevent the user from performing the manipulation that caused the breach. It includes:
Not Null constraint: While creating tables, by default the rows can have null value .the enforcement of not null constraint in a table ensure that the table contains values.
Principle of null values:

- Setting null value is appropriate when the actual value is unknown, or when a value would not be meaningful.
- A null value is not equivalent to a value of zero.
- A null value will always evaluate to null in any expression.
- When a column name is defined as not null, that column becomes a mandatory i.e., the user has to enter data into it.
- Not null Integrity constraint cannot be defined using the alter table command when the table contain rows.

Check Constraint: Check constraint can be defined to allow only a particular range of values .when the manipulation violates this constraint, the record will be rejected. Check condition cannot contain sub queries.
E.g: Create table student (regno int, mark int, constraint b check (mark >=0 and mark <=100));

b) Entity Integrity Maintains uniqueness in a record. An entity represents a table and each row of a table represents an instance of that entity. To identify each row in a table uniquely we need to use this constraint. There are 2 entity constraints:
Unique key constraint It is used to ensure that information in the column for each record is unique, as with telephone or drivers license numbers. It prevents the duplication of value with rows of a specified column in a set of column. A column defined with the constraint can allow null value. If unique key constraint is defined in more than one column i.e., combination of column cannot be specified.Maximum combination of columns that a composite unique key can contain is 16.
Primary Key Constraint A primary key avoids duplication of rows and does not allow null values. It can be defined on one or more columns in a table and is used to uniquely identify each row in a table. These values should never be changed and should never be null. A table should have only one primary key. If a primary key constraint is assigned to more than one column or combination of column is said to be composite primary key, which can contain 16 columns.

c) Referential Integrity It enforces relationship between tables. To establish parent-child relationship between 2 tables having a common column definition, we make use of this constraint. To implement this, we should define the column in the parent table as primary key and same column in the child table as foreign key referring to the corresponding parent entry.

Foreign key A column or combination of column included in the definition of referential integrity, which would refer to a referenced key.**Referenced key** It is a unique or primary key upon which is defined on a column belonging to the parent table.

e.g :

```
CREATE TABLE department
(D_id int,
 D_name varchar(30),
 CONSTRAINT pk_nid Primary key(D_id));

CREATE TABLE teacher
(
T_id int  PRIMARY KEY,
D_id int,
T_name varchar(50) NOT NULL,
T_address varchar(50),
T_email varchar(25),
CONSTRAINT fk_td FOREIGN KEY (D_id) REFERENCES department(D_id))
```

Cascading Actions in SQL

create table *teacher*
. . .
foreign key*(D_id)* **references** *department(D_id))*
on delete cascade
on update cascade
. . . **)**

- Due to the **on delete cascade** clauses, if a delete of a tuple in *branch* results in referential-integrity constraint violation, the delete "cascades" to the *account* relation, deleting the tuple that refers to the branch that was deleted.
- Cascading updates are similar.
- If there is a chain of foreign-key dependencies across multiple relations, with **on delete cascade** specified for each dependency, a deletion or update at one end of the chain can propagate across the entire chain.
- If a cascading update to delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.
- Referential integrity is only checked at the end of a transaction
  - ✓ Intermediate steps are allowed to violate referential integrity provided later steps remove the violation

- ✓ Otherwise it would be impossible to create some database states, e.g. insert two tuples whose foreign keys point to each other (e.g. *spouse* attribute of relation *marriedperson)*

- Alternative to cascading:
    - **on delete set null**
- Null values in foreign key attributes complicate SQL referential integrity semantics, and are best prevented using **not null**
    - ✓ if any attribute of a foreign key is null, the tuple is defined to satisfy the foreign key constraint!

Assertion:

. An *assertion* is a predicate expressing a condition that we wish the database always to satisfy.

- ✓ An assertion in SQL takes the form

    create assertion <assertion-name> **check** <predicate>

. When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion

- ✓ This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

.Asserting for all X, P(X) is achieved in a round-about fashion using not exists X such that not P(X)

Assertion Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
create assertion sum-constraint check
   (not exists (select * from branch
                         where (select sum(amount) from loan
                                      loan.branch-name = branch.branch-name)
                            >= (select sum(amount) from account
                                     where loan.branch-name = branch.branch-name)))
```

Triggers

A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

To design a trigger mechanism, we must:

- Specify the conditions under which the trigger is to be executed.
- Specify the actions to be taken when the trigger executes.

Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

Statement Level Triggers

Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a single transaction

- Use **for each statement** instead of **for each row**
- Use **referencing old table** or **referencing new table** to refer to temporary tables containing the affected rows
- Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g. total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

Trigger Example:

For creating a new trigger, we need to use the CREATE TRIGGER statement. Its syntax is as follows −

```
CREATE TRIGGER trigger_name trigger_time trigger_event

ON table_name

FOR EACH ROW

BEGIN

...

END;
```

Here,

- **Trigger_name** is the name of the trigger which must be put after the CREATE TRIGGER statement. The naming convention for trigger_name can be like [trigger time]_[table name]_[trigger event]. For example, before_student_update or after_student_insert can be the name of the trigger.
- **Trigger_time** is the time of trigger activation and it can be BEFORE or AFTER. We must have to specify the activation time while defining a trigger. We must use BEFORE if we want to process action prior to the change made on the table and AFTER if we want to process action post to the change made on the table.
- **Trigger_event** can be INSERT, UPDATE, or DELETE. This event causes the trigger to be invoked. A trigger only can be invoked by one event. To define a trigger that is invoked by multiple events, we have to define multiple triggers, one for each event.
- **Table_name** is the name of the table. Actually, a trigger is always associated with a specific table. Without a table, a trigger would not exist hence we have to specify the table name after the 'ON' keyword.
- **BEGIN…END** is the block in which we will define the logic for the trigger.
  e.g: Suppose we want to apply trigger on the table Student_age which is created as follows –

```
Create table Student_age(age INT, Name Varchar(35));
```

Now, the following trigger will automatically insert the age = 0 if someone tries to insert age < 0.

```
DELIMITER //

Create Trigger before_inser_studentage BEFORE INSERT ON student_age
FOR EACH ROW

BEGIN

IF NEW.age < 0 THEN SET NEW.age = 0;

END IF;

END //
```

Now, for invoking this trigger, we can use the following statements –

```
INSERT INTO Student_age(age, Name) values(30, 'Rahul');

INSERT INTO Student_age(age, Name) values(-10, 'Harshit');

Select * from Student_age;
```

```
+------+---------+
| age  | Name    |
+------+---------+
| 30   | Rahul   |
| 0    | Harshit |
+------+---------+
```

The above result set shows that on inserting the negative value in the table will lead to insert 0 by a trigger.

CLOSURE OF A SET OF FUNCTIONAL DEPEDENCIES

Given a relational schema R, a functional dependencies f on R is logically implied by a set of functional dependencies F on R if every relation instance r(R) that satisfies F also satisfies f.
The closure of F, denoted by F_+, is the set of all functional dependencies logically implied by F.
The closure of F can be found by using a collection of rules called **Armstrong axioms.**

- **Reflexivity rule:** If A *is a set of attributes and B is subset or equal to A, then A*→B holds.
- **Augmentation rule:** If A→B holds and C is a set of attributes, then CA→CB holds
- **Transitivity rule:** If A→B holds and B→C holds, then A→C holds.
- **Union rule:** If A→B holds and A→C then A→BC holds
- **Decomposition rule:** If A→BC holds, then A→B holds and A→C holds.
- **Pseudo transitivity rule:** If A→B holds and BC→D holds, then AC→D holds.

Suppose we are given a relation schema R=(A,B,C,G,H,I) and the set of function dependencies

A→B,A→C,CG→H,CG→I,B→H

- We list several members of F_+ here:

☐ A→H, since A→B and B→H hold, we apply the transitivity rule.

☐ CG→HI. Since CG→H and CG→I , the union rule implies that CG→HI

☐ AG→I, since A→C and CG→I, the pseudo transitivity rule implies that AG→I holds

. A→BC, since A→B and A→C hold, we apply union rule

. AG→H, since A→C and CG→H, the pseudo transitivity rule implies that AG→H holds

Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has *n* attributes *A*1, *A*2, ...,*An*; let us think of the whole database as being described by a single **universal** relation schema *R* = {*A*1, *A*2, ... , *An*}. We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data 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 possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t1 and t2 in r that have t1[X] = t2[X], they must also have t1[Y] = t2[Y].*

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

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

Normalization:

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of

(1) minimizing redundancy and

(2) minimizing the insertion, deletion, and update anomalies.

It can be considered as a “filtering” or “purification” process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—the normal form tests—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database designers with the following:

- A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes
- A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be normalized to any desired degree.

First Normal Form

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple.* In other words, 1NF disallows *relations within relations* or *relations as attribute values within tuples*. The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

Consider the DEPARTMENT relation schema shown in Figure below, whose primary key is Dnumber. We assume that each department can have *a number of* locations. As we can see, this is not in 1NF because Dlocations is not an atomic attribute.

DEPARTMENT

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

To Convert it into 1NF:

There are two 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}, as shown in Figure below. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.

DEPARTMENT

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

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, as shown in Figure 15.9(c). In this case, the primary key becomes the combination {Dnumber, Dlocation}. This solution has the disadvantage of introducing *redundancy* in the relation.

DEPARTMENT

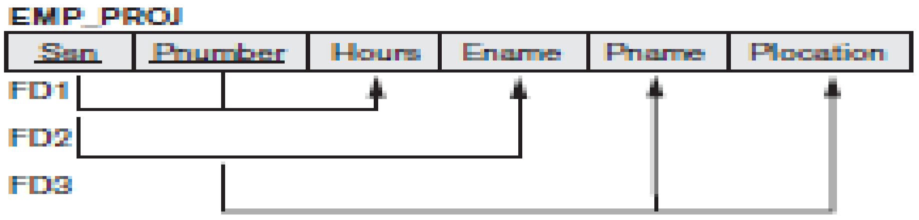
Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Second Normal Form:

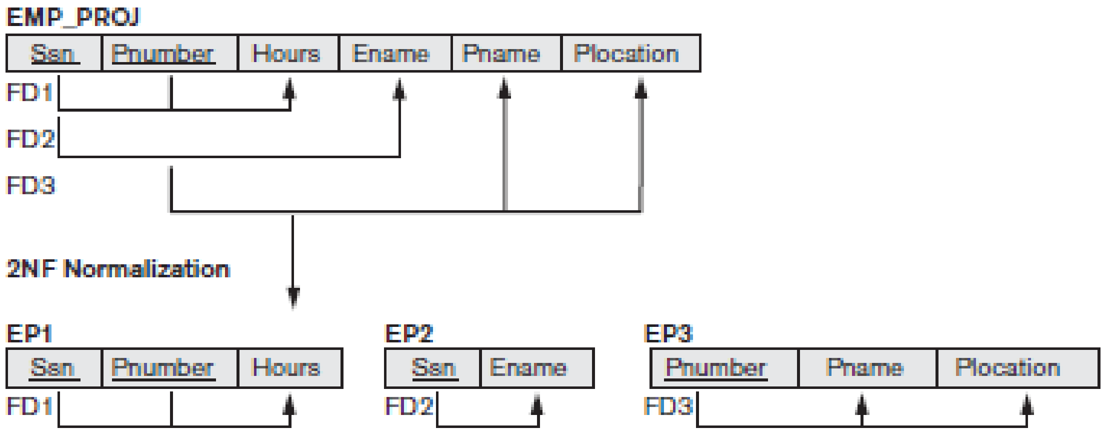
Second normal form (2NF) is based on the concept of *full functional dependency.* A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does *not* functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure EMP_PROJ, {Ssn, Pnumber} → Hours is a full dependency (neither Ssn → Hours nor Pnumber→Hours holds). However, the dependency {Ssn, Pnumber}→Ename is partial because Ssn→Ename 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.*

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure below is in 1NF but is not in 2NF. The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation 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* 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 of above Figure lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure below each of which is in 2NF.

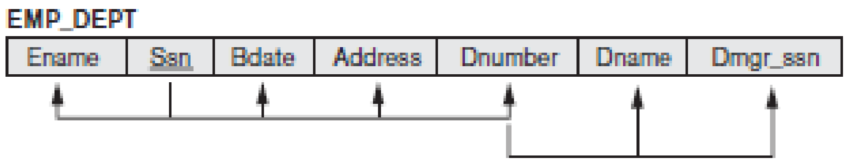


Third Normal Form:

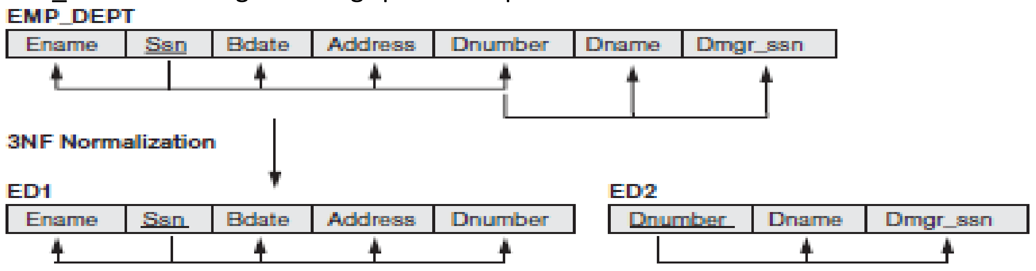
Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency *X*→*Y* in a relation schema *R* is a **transitive dependency** if there exists a set of attributes *Z* in *R* that is neither a candidate key nor a subset of any key of *R*, and both *X*→*Z* and *Z*→*Y* hold. The dependency Ssn→Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure below, 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. Intuitively, we can see that the dependency of Dmgr_ssn on Dnumber is undesirable in EMP_DEPT since Dnumber is not a key of EMP_DEPT.

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.

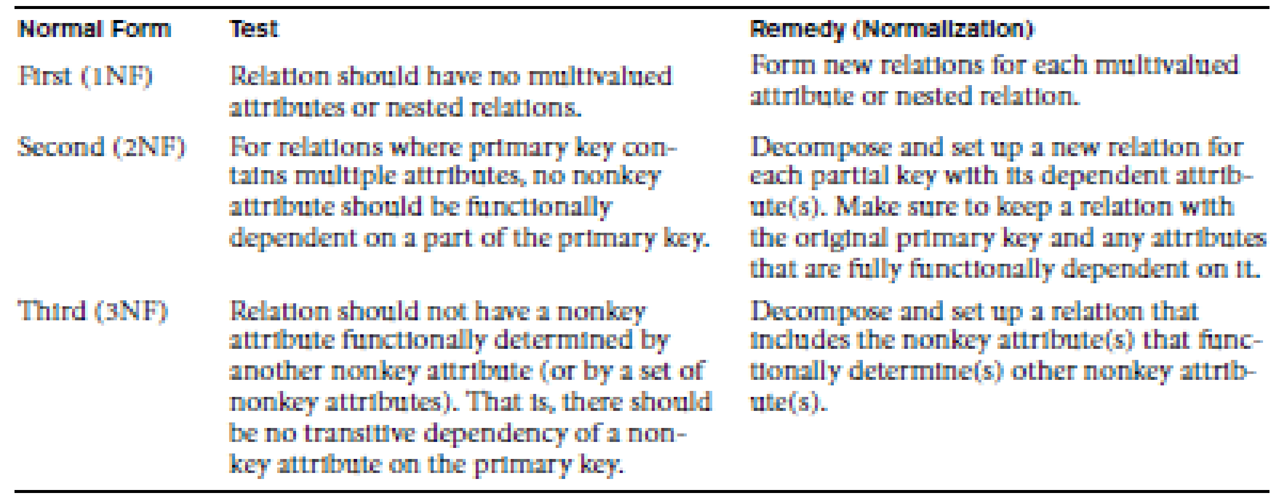
The relation schema EMP_DEPT in Figure below is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber.



We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure below. Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.



Summary of Normal Forms Based on Primary Key and Corresponding Normalization



Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

Boyce-Codd Normal Form (BCNF):

- A relation is in BCNF if every determinant is a candidate key
- Those determinants that are keys we initially call *candidate keys*.
- Eventually, we select a single candidate key to be *the key* for the relation.
- **Consider the following example:**
    - **Funds consist of one or more Investment Types.**
    - **Funds are managed by one or more Managers**
    - **Investment Types can have one more Managers**
    - **Managers only manage one type of investment.**
- **Relation: FUNDS (FundID, InvestmentType, Manager)**

Fund ID	**Investment Type**	**Manager**
99	**Common Stock**	**Smith**
33	**Common Stock**	**Green**
22	**Growth Stocks**	**Brown**

11	Municipal Bonds	Smith

- FD1: FundID, InvestmentType → Manager
- FD2: FundID, Manager → InvestmentType
- FD3: Manager → InvestmentType
- In this case, the combination FundID and InvestmentType form a *candidate key* because we can use FundID,InvestmentType to uniquely identify a tuple in the relation.
- Similarly, the combination FundID and Manager also form a *candidate key* because we can use FundID, Manager to uniquely identify a tuple.
- Manager by itself is not a candidate key because we cannot use Manager alone to uniquely identify a tuple in the relation.
- Is this relation FUNDS(FundID, InvestmentType, Manager) in 1NF, 2NF or 3NF ?

Given we pick FundID, InvestmentType as the *Primary Key:* 1NF for sure. 2NF because all of the non-key attributes (Manager) is dependant on all of the key. 3NF because there are no transitive dependencies.

- Therefore, while FUNDS relation is in 1NF, 2NF and 3NF, it is in BCNF because not all determinants (Manager in FD3) are candidate keys.
- The following are steps to normalize a relation into BCNF:
    1. List all of the determinants.
    2. See if each determinant can act as a key (candidate keys).
    3. For any determinant that is *not* a candidate key, create a new relation from the functional dependency. Retain the determinant in the original relation.
- For our example:

FUNDS (FundID, InvestmentType, Manager)

1. The determinants are:

FundID, InvestmentType
FundID, Manager
Manager

2. Which determinants can act as keys ?

FundID, InvestmentType *YES*
FundID, Manager *YES*
Manager *NO*

3. Create a new relation from the functional dependency:

MANAGERS(<u>Manager</u>, InvestmentType)
FUND_MANAGERS(<u>FundID</u>, <u>Manager</u>)

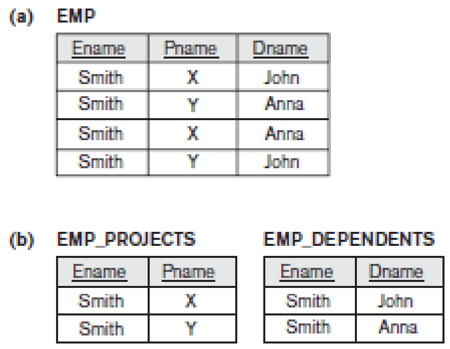
In this last step, we have retained the determinant “Manager” in the original relation MANAGERS.

- Each of the new relations sould be checked to ensure they meet the definitions of 1NF, 2NF, 3NF and BCNF

Multivalued Dependency and Fourth Normal Form:

If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.
For example, consider the relation EMP shown in Figure 1(a). A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname. An employee may work on several projects and may have several

dependents, and the employee's projects and dependents are independent of one another. To keep the relation state consistent, and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is spec fied as a multivalued dependency on the EMP relation, which we define in this section. Informally, whenever two *independent* 1:N relationships *A*:*B* and *A*:*C* are mixed in the same relation, *R*(*A*, *B*, *C*), an MVD may arise.



(a) EMP

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

(b) EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

Figure 1:Multivalued dependency and fourth normal form.(a) The Emp relation with two MVDs:Ename→→PNAME and ENAME→→DNAME. (b) Decomposing EMP into two relations in 4NF

Formal Definition of Multivalued Dependency:

Definition. A multivalued dependency *X*→→*Y* specified on relation schema *R*, where *X* and *Y* are both subsets of *R*, specifies the following constraint on any relation state *r* of *R*: If two tuples *t*1 and *t*2 exist in *r* such that *t*1[X] = *t*2[X], then two tuples *t*3 and *t*4 should also exist in *r* with the following properties, where we use *Z* to denote (*R* – (*X* ∪Y)):

■ *t*3[*X*] = *t*4[*X*] = *t*1[*X*] = *t*2[*X*].
■ *t*3[*Y*] = *t*1[*Y*] and *t*4[*Y*] = *t*2[*Y*].
■ *t*3[*Z*] = *t*2[*Z*] and *t*4[*Z*] = *t*1[*Z*].

Whenever *X*→→*Y* holds, we say that *X* **multidetermines** *Y*. Because of the symmetry in the definition, whenever *X* →→ *Y* holds in *R*, so does *X* →→ *Z*. Hence, *X* →→ *Y* implies *X*→→*Z*, and therefore it is sometimes written as *X*→→*Y*|*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*. For example, the relation EMP_PROJECTS in Figure 1(b) has the trivial MVD Ename →→ Pname. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**.A trivial MVD will hold in *any* relation state *r* of *R*; it is called trivial because it does not specify any significant or meaningful constraint on *R*. If we have a *nontrivial MVD* in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure 1(a), the values 'X' and 'Y' of Pname are repeated with each value of Dname (or, by symmetry, the values 'John' and 'Anna' of Dname are repeated with each value of Pname).

Fourth Normal Form:

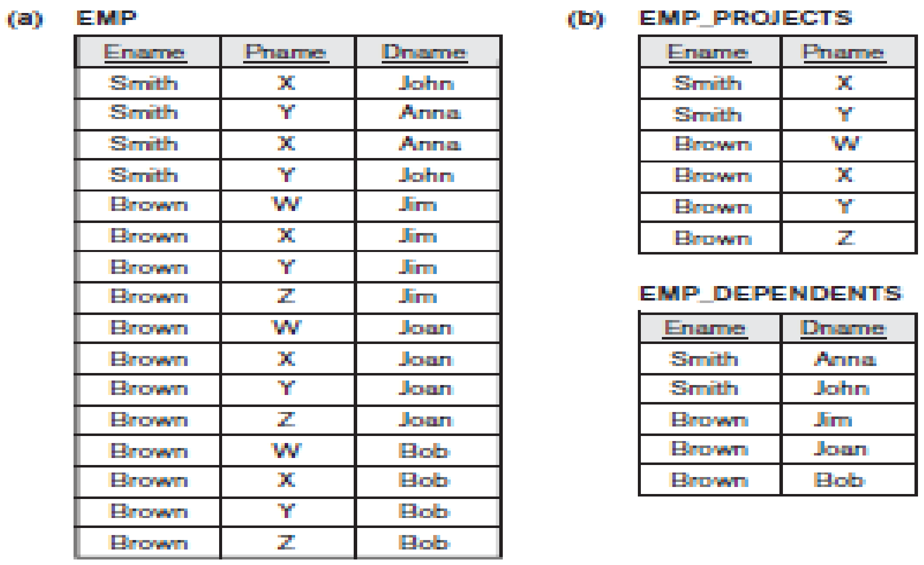
Definition. A relation schema *R* is in 4**NF** with respect to a set of dependencies *F* (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency *X* →→ *Y* in F^+ , *X* is a superkey for *R*.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD. Consider the EMP relation in Figure 1(a). EMP is not in 4NF because in 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 1(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. No other nontrivial MVDs hold in either

EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

To illustrate the importance of 4NF, Figure 16.4(a) shows the EMP relation in Figure 1(a) with an additional employee, 'Brown', who has three dependents ('Jim', 'Joan', and 'Bob') and works on four different projects ('W', 'X', 'Y', and 'Z'). There are 16 tuples in EMP in Figure 16.4(a). If we decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, as shown in Figure 16.4(b), we need to store a total of only 11 tuples in both relations. Not only would the decomposition save on storage, but the update anomalies associated with multivalued dependencies would also be avoided. For example, if 'Brown' starts working on a new additional project 'P,'we must insert *three* tuples in EMP—one for each dependent. If we forget to insert any one of those, the relation violates the MVD and becomes inconsistent in that it incorrectly implies a relationship between project and dependent. If the relation has nontrivial MVDs, then insert, delete, and update operations on single tuples may cause additional tuples to be modified besides the one in question. If the update is handled incorrectly, the meaning of the relation may change. However, after normalization into 4NF, these update anomalies disappear. For example, to add the information that 'Brown' will be assigned to project 'P', only a single tuple need be inserted in the 4NF relation EMP_PROJECTS.

Figure 16.4
Decomposing a relation state of EMP that is not in 4NF. (a) EMP relation with additional tuples. (b) Two corresponding 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.



(a) **EMP**

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John
Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

(b) **EMP_PROJECTS**

Ename	Pname
Smith	X
Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

EMP_DEPENDENTS

Ename	Dname
Smith	Anna
Smith	John
Brown	Jim
Brown	Joan
Brown	Bob

Some Solved Example of Normalization:

Q1. Examine the table shown below.

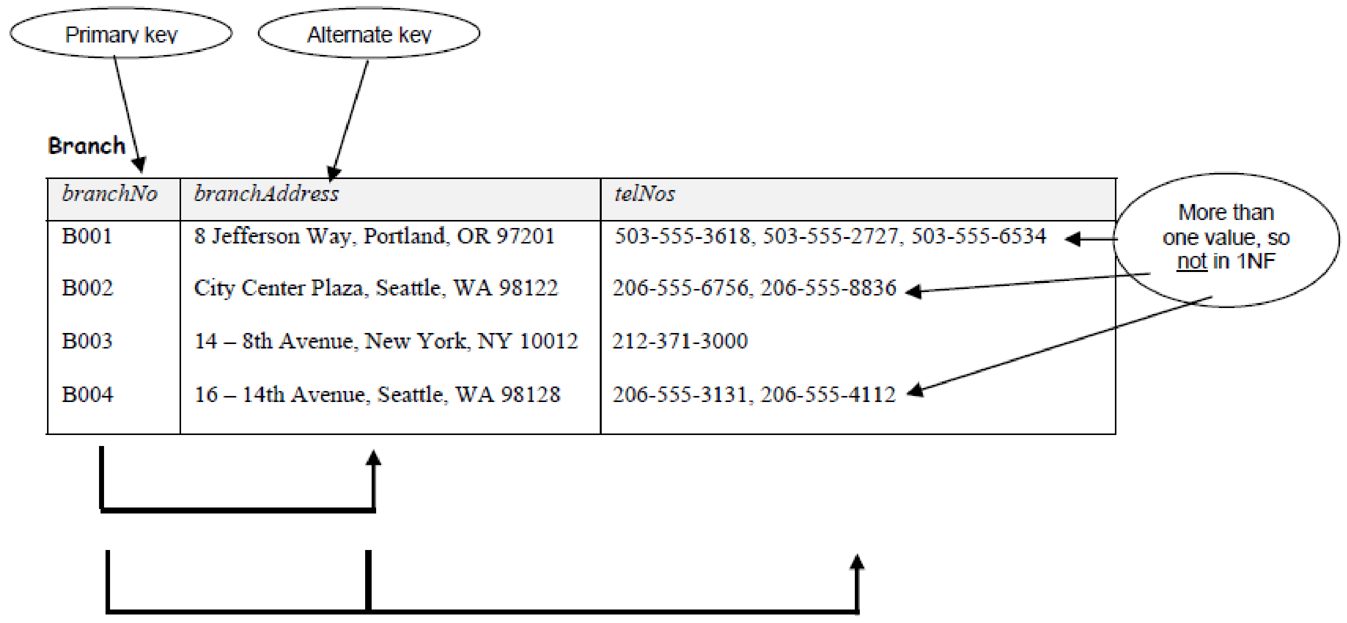
branchNo	*branchAddress*	*telNos*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618, 503-555-2727, 503-555-6534
B002	City Center Plaza, Seattle, WA 98122	206-555-6756, 206-555-8836
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131, 206-555-4112

(a) Why is this table not in 1NF?

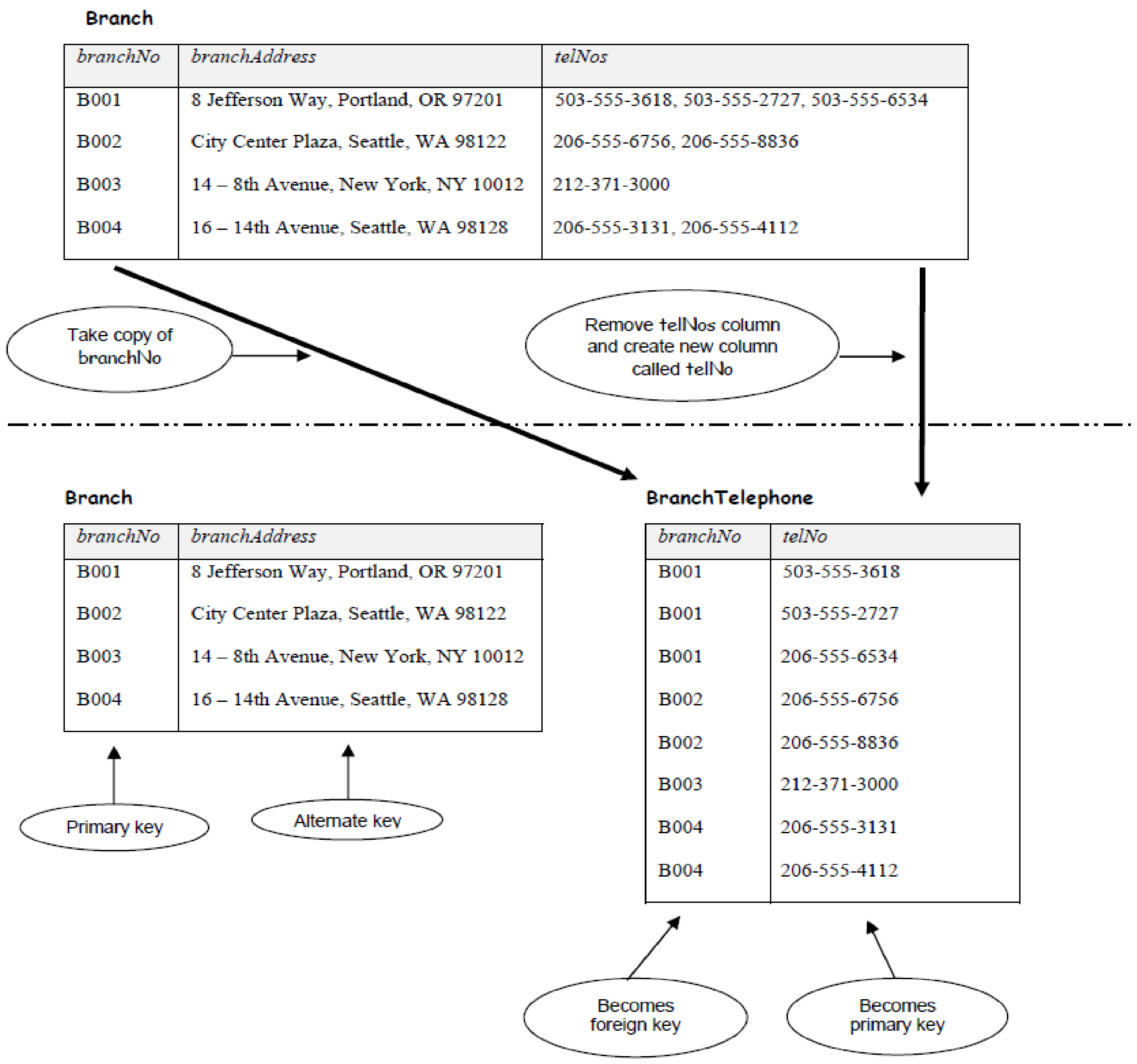
(b) Describe and illustrate the process of normalizing the data shown in this table to third normal form (3NF).

(c) Identify the primary, alternate and foreign keys in your 3NF relations.

Answer:



branchNo	*branchAddress*	*telNos*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618, 503-555-2727, 503-555-6534
B002	City Center Plaza, Seattle, WA 98122	206-555-6756, 206-555-8836
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131, 206-555-4112



Branch

branchNo	*branchAddress*	*telNos*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618, 503-555-2727, 503-555-6534
B002	City Center Plaza, Seattle, WA 98122	206-555-6756, 206-555-8836
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131, 206-555-4112

Branch

branchNo	*branchAddress*
B001	8 Jefferson Way, Portland, OR 97201
B002	City Center Plaza, Seattle, WA 98122
B003	14 – 8th Avenue, New York, NY 10012
B004	16 – 14th Avenue, Seattle, WA 98128

BranchTelephone

branchNo	*telNo*
B001	503-555-3618
B001	503-555-2727
B001	206-555-6534
B002	206-555-6756
B002	206-555-8836
B003	212-371-3000
B004	206-555-3131
B004	206-555-4112

Q. 2 Examine the table shown below.

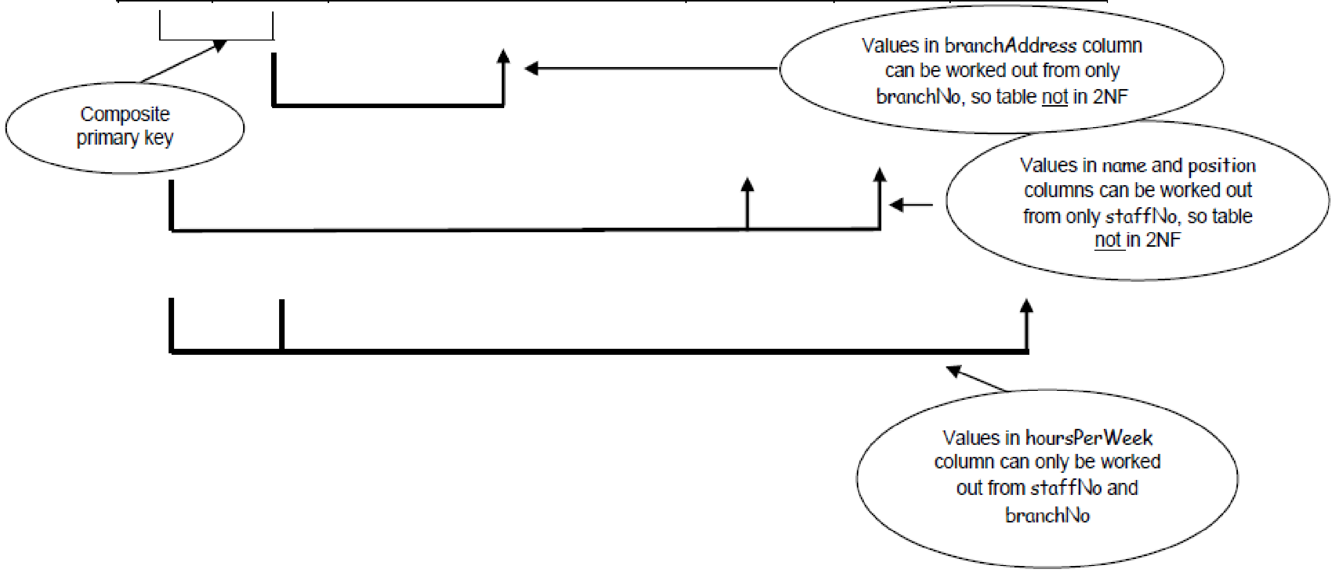
staffNo	*branchNo*	*branchAddress*	*name*	*position*	*hoursPerWeek*
S4555	B002	City Center Plaza, Seattle, WA 98122	Ellen Layman	Assistant	16
S4555	B004	16 – 14th Avenue, Seattle, WA 98128	Ellen Layman	Assistant	9
S4612	B002	City Center Plaza, Seattle, WA 98122	Dave Sinclair	Assistant	14
S4612	B004	16 – 14th Avenue, Seattle, WA 98128	Dave Sinclair	Assistant	10

(a) Why is this table not in 2NF?
(b) Describe and illustrate the process of normalizing the data shown in this table to third normal form (3NF).
(c) Identify the primary, (alternate) and foreign keys in your 3NF relations.

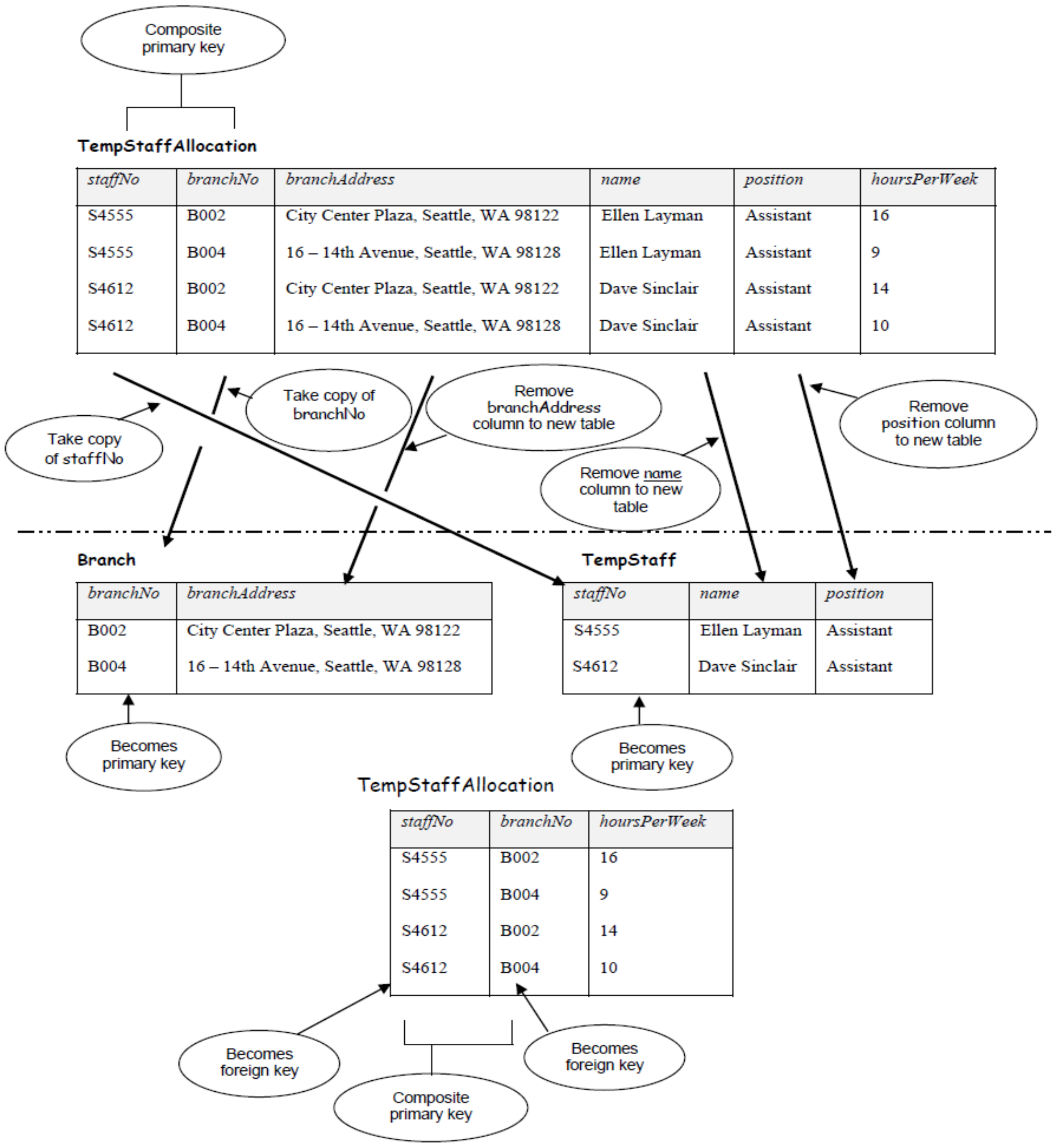
Answer:

TempStaffAllocation

staffNo	*branchNo*	*branchAddress*	*name*	*position*	*hoursPerWeek*
S4555	B002	City Center Plaza, Seattle, WA 98122	Ellen Layman	Assistant	16
S4555	B004	16 – 14th Avenue, Seattle, WA 98128	Ellen Layman	Assistant	9
S4612	B002	City Center Plaza, Seattle, WA 98122	Dave Sinclair	Assistant	14
S4612	B004	16 – 14th Avenue, Seattle, WA 98128	Dave Sinclair	Assistant	10



Composite primary key
Values in branchAddress column can be worked out from only branchNo, so table not in 2NF
Values in name and position columns can be worked out from only staffNo, so table not in 2NF
Values in hoursPerWeek column can only be worked out from staffNo and branchNo



TempStaffAllocation

staffNo	branchNo	branchAddress	name	position	hoursPerWeek
S4555	B002	City Center Plaza, Seattle, WA 98122	Ellen Layman	Assistant	16
S4555	B004	16 – 14th Avenue, Seattle, WA 98128	Ellen Layman	Assistant	9
S4612	B002	City Center Plaza, Seattle, WA 98122	Dave Sinclair	Assistant	14
S4612	B004	16 – 14th Avenue, Seattle, WA 98128	Dave Sinclair	Assistant	10

Branch

branchNo	branchAddress
B002	City Center Plaza, Seattle, WA 98122
B004	16 – 14th Avenue, Seattle, WA 98128

TempStaff

staffNo	name	position
S4555	Ellen Layman	Assistant
S4612	Dave Sinclair	Assistant

TempStaffAllocation

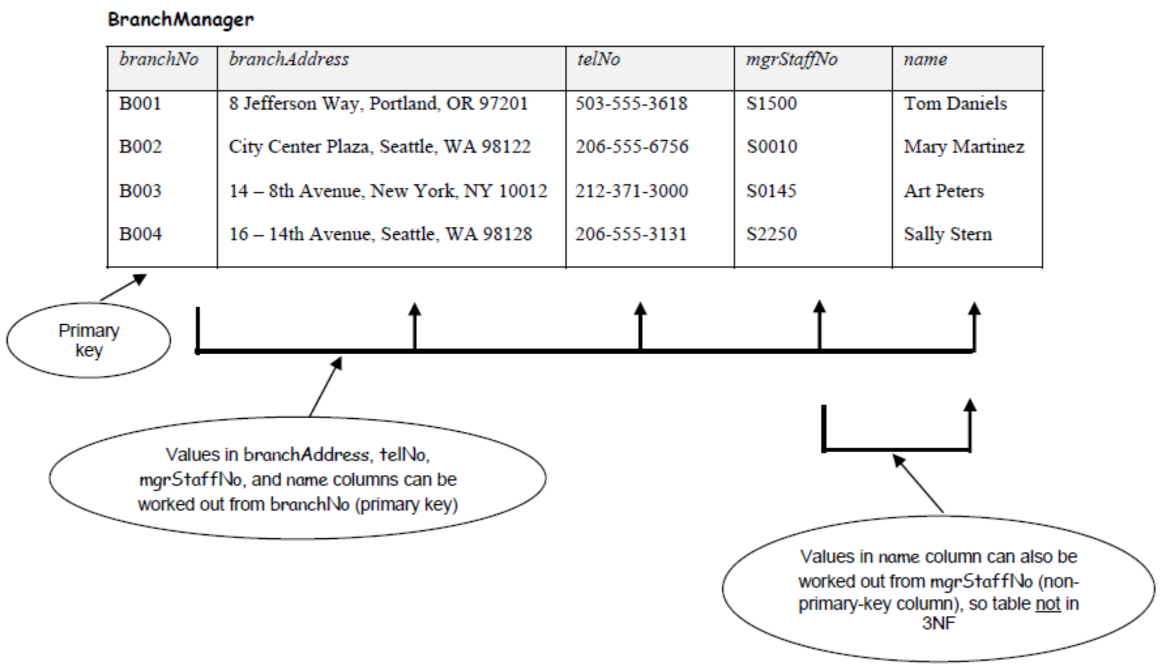
staffNo	branchNo	hoursPerWeek
S4555	B002	16
S4555	B004	9
S4612	B002	14
S4612	B004	10

Q.3 Examine the table shown below.

branchNo	*branchAddress*	*telNo*	*mgrStaffNo*	*name*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618	S1500	Tom Daniels
B002	City Center Plaza, Seattle, WA 98122	206-555-6756	S0010	Mary Martinez
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000	S0145	Art Peters
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131	S2250	Sally Stern

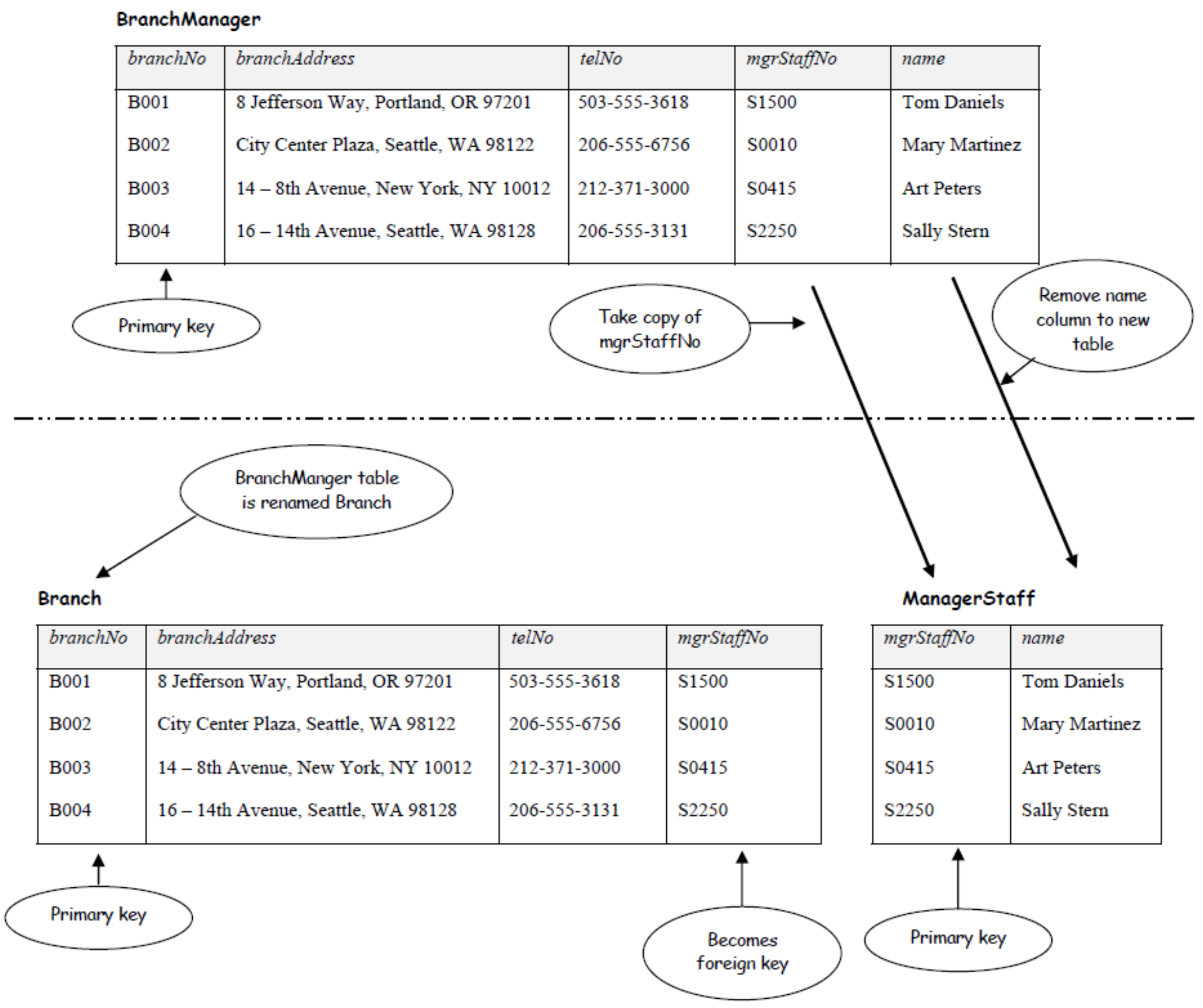
(a) Why is this table not in 3NF?
(b) Describe and illustrate the process of normalizing the data shown in this table to third normal form (3NF).
(c) Identify the primary, (alternate) and foreign keys in your 3NF relations.

Answer:


BranchManager

branchNo	*branchAddress*	*telNo*	*mgrStaffNo*	*name*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618	S1500	Tom Daniels
B002	City Center Plaza, Seattle, WA 98122	206-555-6756	S0010	Mary Martinez
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000	S0145	Art Peters
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131	S2250	Sally Stern

Primary key
Values in branchAddress, telNo, mgrStaffNo, and name columns can be worked out from branchNo (primary key)
Values in name column can also be worked out from mgrStaffNo (non-primary-key column), so table not in 3NF



BranchManager

branchNo	*branchAddress*	*telNo*	*mgrStaffNo*	*name*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618	S1500	Tom Daniels
B002	City Center Plaza, Seattle, WA 98122	206-555-6756	S0010	Mary Martinez
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000	S0415	Art Peters
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131	S2250	Sally Stern

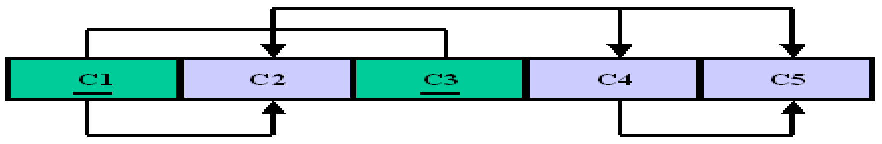
Branch

branchNo	*branchAddress*	*telNo*	*mgrStaffNo*
B001	8 Jefferson Way, Portland, OR 97201	503-555-3618	S1500
B002	City Center Plaza, Seattle, WA 98122	206-555-6756	S0010
B003	14 – 8th Avenue, New York, NY 10012	212-371-3000	S0415
B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131	S2250

ManagerStaff

mgrStaffNo	*name*
S1500	Tom Daniels
S0010	Mary Martinez
S0415	Art Peters
S2250	Sally Stern

Q4. Given the dependency diagram shown in the following Figure, answer the following questions.



a. Identify and discuss each of the indicated dependencies.
Answer:

- C1 → C2 represents a partial dependency, because C2 depends only on C1, rather than on the entire primary key composed of C1 and C3.
- C4 → C5 represents a transitive dependency, because C5 depends on an attribute (C4) that is not part of a primary key.
- C1, C3 → C2, C4, C5 represents a functional dependency, because C2, C4, and C5 depend on the primary key composed of C1 and C3.