Database Management System (MDS 505)

Jagdish Bhatta

Unit -3

Database Constraints and Normalization

Database Constraints

- Already covered in unit 2.
- Refer 2.1 Relational Model.

Assertions

- ASSERTION, which can be used to specify additional types of constraints that are outside the scope of the *built-in relational model constraints* (primary and unique keys, entity integrity, and referential integrity).
- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- An **assertion** is a predicate expressing a condition we wish the database to always satisfy.
- Domain constraints, functional dependency and referential integrity are special forms of assertion.
- **Syntax:**

  create assertion assertion-name check predicate

Assertions

- For example, to specify the constraint that ***the salary of an employee must not be greater than the salary of the manager of the department that the employee works for*** in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK
    (
    NOT EXISTS ( SELECT *
    FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
    WHERE E.Salary>M.Salary AND E.Dno = D.Dnumber AND D.Mgr_ssn =
    M.Ssn )
    );
```

Assertions

- The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied.
- The constraint name can be used later to disable the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.
- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**. The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.
- By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE. Thus, the assertion is violated if the result of the query is not empty. In the preceding example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.

Assertions

- A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated* in a specific table. Hence, constraint checking can be implemented more efficiently by the DBMS in these cases. The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples*.
- On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that simple checks are implemented more efficiently by the DBMS.
- When an assertion is created, the system tests it for validity. If the assertion is valid, any further modification to the database is allowed only if it does not cause that assertion to be violated. This testing may result in significant overhead if the assertions are complex. Because of this, the **assert** should be used with great care. Some system developer omits support for general assertions or provides specialized form of assertions that are easier to test.

Triggers

- In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to **monitor** the database. Other actions may be specified, such as executing a specific *stored procedure* or triggering other updates. The CREATE TRIGGER statement is used to implement such actions in SQL.

Triggers

- Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database.
- Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure SALARY_VIOLATION,5 which will notify the supervisor.

```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
ON EMPLOYEE
FOR EACH ROW
WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
WHERE SSN = NEW.SUPERVISOR_SSN ) )
INFORM_SUPERVISOR (NEW.Supervisor_ssn,
NEW.Ssn );
```

Triggers

- The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger which is regarded as an ECA (Event, Condition, Action) rule has three components:
  - The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. In some cases, it may be necessary to write more than one trigger to cover all possible cases. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.
  - The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.
  - The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Triggers

- The following AFTER trigger simply prints ‘In the After Trigger’ when the trigger is executed:

```
USE Family;
CREATE TRIGGER TriggerOne ON Person
AFTER INSERT
AS
PRINT ‘In the After Trigger’;
```

- With the AFTER trigger enforced, the following code inserts a sample row:

```
INSERT Person(PersonID, LastName, FirstName, Gender)
VALUES (50, ‘Ebob’, ‘Bill’,‘M’);
```

- Result:

In the After Trigger

Stored Procedures

- A **stored procedure** (also termed **proc**, **storp**, **sproc**, **StoPro**, **StoredProc**, **StoreProc**, **sp**, or **SP**) is a subroutine available to applications that access a relational database management system (RDBMS). Such procedures are stored in the database data dictionary.
- Uses for stored procedures include data-validation (integrated into the database) or access-control mechanisms.
- Stored routines can be particularly useful in certain situations when multiple client applications are written in different languages or work on different platforms, but need to perform the same database operations.

Stored Procedures

- **Create, alter and drop:**
- CREATE must be the first command in a batch; the termination of the batch ends the creation of the stored procedure. The following example creates a very simple stored procedure that retrieves data from the ProductCategory table :

```
CREATE PROCEDURE CategoryList
AS
SELECT ProductCategoryName, ProductCategoryDescription
FROM ProductCategory;
```

Executing a Stored Procedure

- When calling a stored procedure within a SQL batch, the EXECUTE or CALL command executes the stored procedure with a few special rules. EXECUTE is typically coded as EXEC.

- For Eg.: EXEC/CALL dbo.CategoryList;

Passing Data to Stored Procedure

- A stored procedure is more useful when it can be manipulated by parameters. The CategoryList stored procedure created previously returns all the product categories, but a procedure that performs a task on an individual row requires a method for passing the row ID to the procedure.

Passing Data to Stored Procedure

- **Input parameters**
- You can add input parameters that pass data to the stored procedure by listing the parameters after the procedure name in the CREATE PROCEDURE command
- Each parameter must begin with an @ sign, and becomes a local variable within the procedure. Like local variables, the parameters must be defined with valid data types. When the stored procedure is called, the parameter must be included (unless the parameter has a default value).
- The following code sample creates a stored procedure that returns a single product category. The value passed by means of the parameter is available within the stored procedure as the variable @CategoryName in the WHERE clause:

Passing Data to Stored Procedure

- **Input parameters**

```
CREATE PROCEDURE CategoryGet (@CategoryName VARCHAR(35))
AS
SELECT ProductCategoryName, ProductCategoryDescription
FROM ProductCategory
WHERE ProductCategoryName = @CategoryName;
```

Passing Data to Stored Procedure

- When the following code sample is executed, the Unicode string literal ‘Kite’ is passed to the stored procedure and substituted for the variable in the WHERE clause:

  EXEC CategoryGet **’Kite’;**

- Result:

```
ProductCategoryName    ProductCategoryDescription
---------------------- ----------------------------------
Kite                   a variety of kites, from simple to
                       stunt, to Chinese, to novelty kites
```

Passing Data to Stored Procedure

- If multiple parameters are involved, the parameter name can be specified in any order, or the parameter values listed in order. If the two methods are mixed, then as soon as the parameter is provided by name, all the following parameters must be as well.
- The next four examples each demonstrate calling a stored procedure and passing the parameters by original position and by name:

```
EXEC Schema.StoredProcedure
@Parameter1 = n,
@Parameter2 = 'n';

EXEC Schema.StoredProcedure
@Parameter2 = 'n',
@Parameter1 = n;

EXEC Schema.StoredProcedure  n, 'n';

EXEC Schema.StoredProcedure  n, @Parameter2 = 'n';
```

Passing Data to Stored Procedure

- **Parameter defaults**
- You must supply every parameter when calling a stored procedure, unless that parameter has been created with a default value. You establish the default by appending an equal sign and the default to the parameter, as follows;

  CREATE PROCEDURE *StoredProcedure*

  (

  @Variable DataType **= *Default Value***

)

Passing Data to Stored Procedure

- **Parameter defaults**
- The following code, demonstrates a stored procedure default. If a product category name is passed in this stored procedure, the stored procedure returns only the selected product category. However, if nothing is passed, the NULL default is used in the WHERE clause to return all the product categories:

```
CREATE PROCEDURE pProductCategory_Fetch2
( @Search VARCHAR(50) = NULL)
AS
SELECT ProductCategoryName, ProductCategoryDescription
FROM ProductCategory
WHERE ProductCategoryName = @Search OR @Search IS NULL;
```

Passing Data to Stored Procedure

- **Parameter defaults**
- The first execution passes a product category:

  EXEC dbo.pProductCategory_Fetch2 'OBX';
- Result:

Result:

ProductCategoryName	ProductCategoryDescription
OBX	OBX stuff

- When pProductCategory_Fetch executes without a parameter, the @Search parameter's default of NULL allows the WHERE clause to evaluate to true for every row, as follows:

  EXEC dbo.pProductCategory_Fetch2;

ProductCategoryName	ProductCategoryDescription
Accessory	kite flying accessories
Book	Outer Banks books
Clothing	OBX t-shirts, hats, jackets
Kite	a variety of kites, from simple to stunt, to Chinese, to novelty kites
Material	Kite construction material
OBX	OBX stuff
Toy	Kids stuff
Video	stunt kite contexts and lessons, and Outer Banks videos

Returning Data from Stored Procedure

- **Output Parameter defaults**
- Output parameters enable a stored procedure to return data to the calling client procedure. The keyword OUTPUT is required both when the procedure is created and when it is called. Within the stored procedure, the output parameter appears as a local variable. In the calling procedure or batch, a variable must have been created to receive the output parameter. When the stored procedure concludes, its current value is passed to the calling procedure's local variable.
- **Eg:**

```
CREATE PROC GetProductName (
@ProductCode CHAR(10),
@ProductName VARCHAR(25) OUTPUT
)
AS
SELECT @ProductName = ProductName
FROM Product
WHERE Code = @ProductCode;
RETURN;
```

Returning Data from Stored Procedure

- **Output Parameter defaults**

```
DECLARE @ProdName VARCHAR(25);
EXEC GetProductName '1001', @ProdName OUTPUT;
PRINT @ProdName;
```

- Result;

  Basic Box Kite 21 inch

Informal Design Guidelines for Relational Databases

- The 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

Informal Design Guidelines for Relational Databases

- **Imparting Clear Semantics to Attributes in Relations**
- Whenever we group attributes to form a relation schema, we assume that 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.
- **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. Intuitively, if a relation schema corresponds to one entity type or one relationship type, it is straightforward 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.

Informal Design Guidelines for Relational Databases

- **Redundant Information in Tuples and Update Anomalies**
- One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space.
- Redundant information storage leads to extra space and chances of having anomalies.

Informal Design Guidelines for Relational Databases

- **Redundant Information in Tuples and Update Anomalies**
- **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.

Informal Design Guidelines for Relational Databases

- **NULL Values in Tuples**
- In some schema designs we may group many attributes together into a "fat" relation. 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 and with specifying JOIN operations at the logical level. Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable. Moreover, NULLs can have multiple interpretations, such as the following:
  - The attribute *does not apply* to this tuple.
  - The attribute value for this tuple is *unknown*.
  - The value is *known but absent*; that is, it has not been recorded yet.

Informal Design Guidelines for Relational Databases

- **NULL Values in Tuples**
- Having the same representation for all NULLs compromises the different meanings they may have.

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

Informal Design Guidelines for Relational Databases

- **Generation of Spurious Tuples**
- Generation of unnecessary tuples while using joins, which actually does not exist in base relations.

- **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.

Informal Design Guidelines for Relational Databases

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values
- Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

Functional Dependency

- Functional dependency is a constraint between two sets of attributes from the database. The functional dependency between two sets of attributes X & Y in a relation R is a constraint on possible tuples that can form a relation instance r of R. This constraint is that for any two tuples t1 & t2 in r if t1[X]=t2[X] then t1[Y]=t2[Y]. This functional dependency is represented by $X \rightarrow Y$.
- 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**.

Functional Dependency

- This functional dependency generalizes the concept of key. If K is a key of R, then K functionally determines all attributes in R (since we never have two distinct tuples with t1[K]=t2[K])
- 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*—that is, how they relate to one another—to specify the functional dependencies that should hold on *all* relation states (extensions) *r* of *R*. Relation extensions *r*(*R*) that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of *R*. Hence, the main use of functional dependencies is to describe further a relation schema *R* by specifying constraints on its attributes that must hold *at all times*.

Functional Dependency

- **Trivial** − If a functional dependency (FD) X → Y holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.
- **Non-trivial** − If an FD X → Y holds, where Y is not a subset of X, then it is called a non-trivial FD.

Examples of FD Constraints

- In the database schema COMPANY;
  - social security number determines employee name

    SSN -> ENAME
  - project number determines project name and location

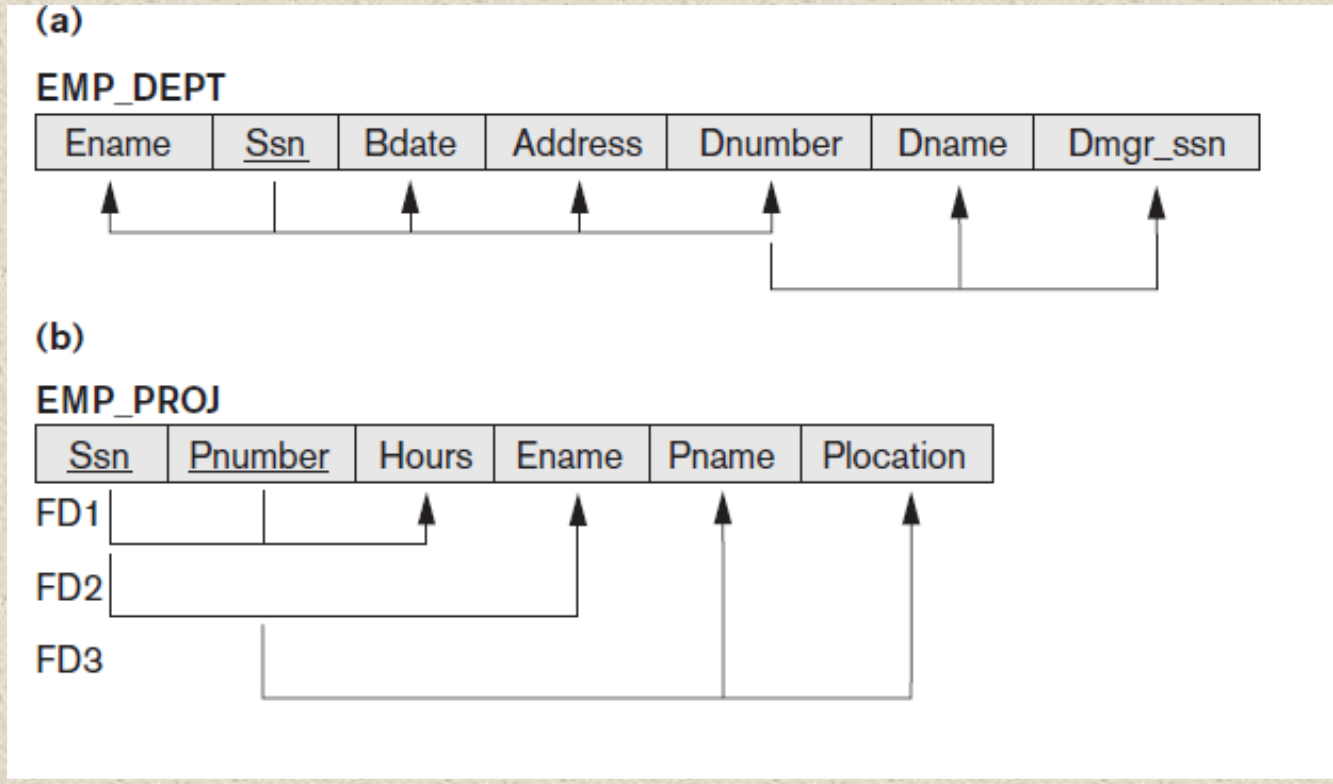
    PNUMBER -> {PNAME, PLOCATION}
  - employee ssn and project number determines the hours per week that the employee works on the project

    {SSN, PNUMBER} -> HOURS

- A functional dependency is a *property of the relation schema R*, not of a particular legal relation state *r* of *R*. Therefore, an FD *cannot* be inferred automatically from a given relation extension *r* but must be defined explicitly by someone who knows the semantics of the attributes of *R*.

Examples of FD Constraints

- Following figure introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, whereas the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.



Inference Rules for FDs

- Given a set of FDs F, we can *infer* additional FDs that hold whenever the FDs in F hold

  Armstrong's inference rules:

  IR1. (**Reflexive**) If Y *subset-of* X, then X -> Y

  IR2. (**Augmentation**) If X -> Y, then XZ -> YZ

  (Notation: XZ stands for X ∪ Z)

  IR3. (**Transitive**) If X -> Y and Y -> Z, then X -> Z

- IR1, IR2, IR3 form a *sound* and *complete* set of inference rules. *Sound* means that from a given set of FDs F specified on relation schema R, any dependency that we can infer from F holds in every instance r of R that satisfies in F & *complete* means we can infer all possible dependencies from given set of FDs F.

Inference Rules for FDs

Some **additional inference rules** that are useful:

(**Decomposition**) If X -> YZ, then X -> Y and X -> Z

(**Union**) If X -> Y and X -> Z, then X -> YZ

(**Psuedotransitivity**) If X -> Y and WY -> Z, then WX -> Z

Inference Rules for FDs

- **Closure of a set F of FDs** is the set F^+ of all FDs that can be inferred from F. Using the inference rules F^+ can be formed.

- **Closure of a set of attributes** X with respect to F is the set X^+ of all attributes that are functionally determined by X.

  X^+ can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in F

  Algorithm: Determining X^+ , the closure of X under F

  $X^+ = X$;

  repeat

  oldX$^+$ = X$^+$;

  for each functional dependency Y -> Z in F do

  if Y subset-of X^+ then $X^+ = X^+ \cup Z$;

  until (X^+ = old X^+);

Example: Closure of set of FDs

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - from $CG \rightarrow H$ *and* $CG \rightarrow I$: "union rule" can be inferred from

Example: Closure of set of Attributes

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Now for $(AG)^+$, using the algorithm we have

  1. $result = \{A,G\}$
  2. $result = \{A,B,C,G\}$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = \{A,B,C,G,H\}$ $(CG \rightarrow H$ and $CG \subseteq ABCG)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq ABCGH)$

  And so on…

Equivalence of sets of FDs

- Two sets of FDs F and G are **equivalent** if:
  - every FD in F can be inferred from G, *and*
  - every FD in G can be inferred from F

- Hence, F and G are equivalent if $F^+ = G^+$
  i.e. if both of the conditions F covers G and G covers F holds.

- A set of functional dependencies F is said to cover another set of functional dependencies G, F ***covers*** G, if every FD in G can be inferred from F (i.e., if G^+ *subset-of* F^+).

Normalization of Relations

- **Normalization**: The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations.
- Normalization of data can be looked upon as 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.
- **Normalization** is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
- We have 1NF, 2NF, 3NF, BCNF based on keys and FDs of a relation schema and 4NF based on keys, multi-valued dependencies : MVDs; 5NF based on keys, join dependencies : JDs.

Importance of Normalization

- **Data consistency:** Data consistency means that the data is always real and it is not ambiguous.

- **Data becomes nonredundant:** Non-redundant means that only copy original copy of data is available for each user and for every time. There are no multiple copies of the same data for different persons. So when data is changed in one file and stay in one file. Then of course data is consistent and non-redundant. Here redundant is not the same as a backup of data, both are different things.

Importance of Normalization

- **Reduce insertion, deletion and updating anomalies:**
  - **Insertion anomaly** is an anomaly that occurs when we want to insert data into the database but the data is not completely or correctly inserted in the target attributes. If completely inserted in the database then not correctly entered.
  - **Deletion anomaly** is an anomaly that occurs when we want to delete data in the database but the data is not completely or correctly deleted in the target attributes.
  - **Update anomaly** is an anomaly that occurs when we want to update data in the database but the data is not completely or correctly updated in the target attributes.

Importance of Normalization

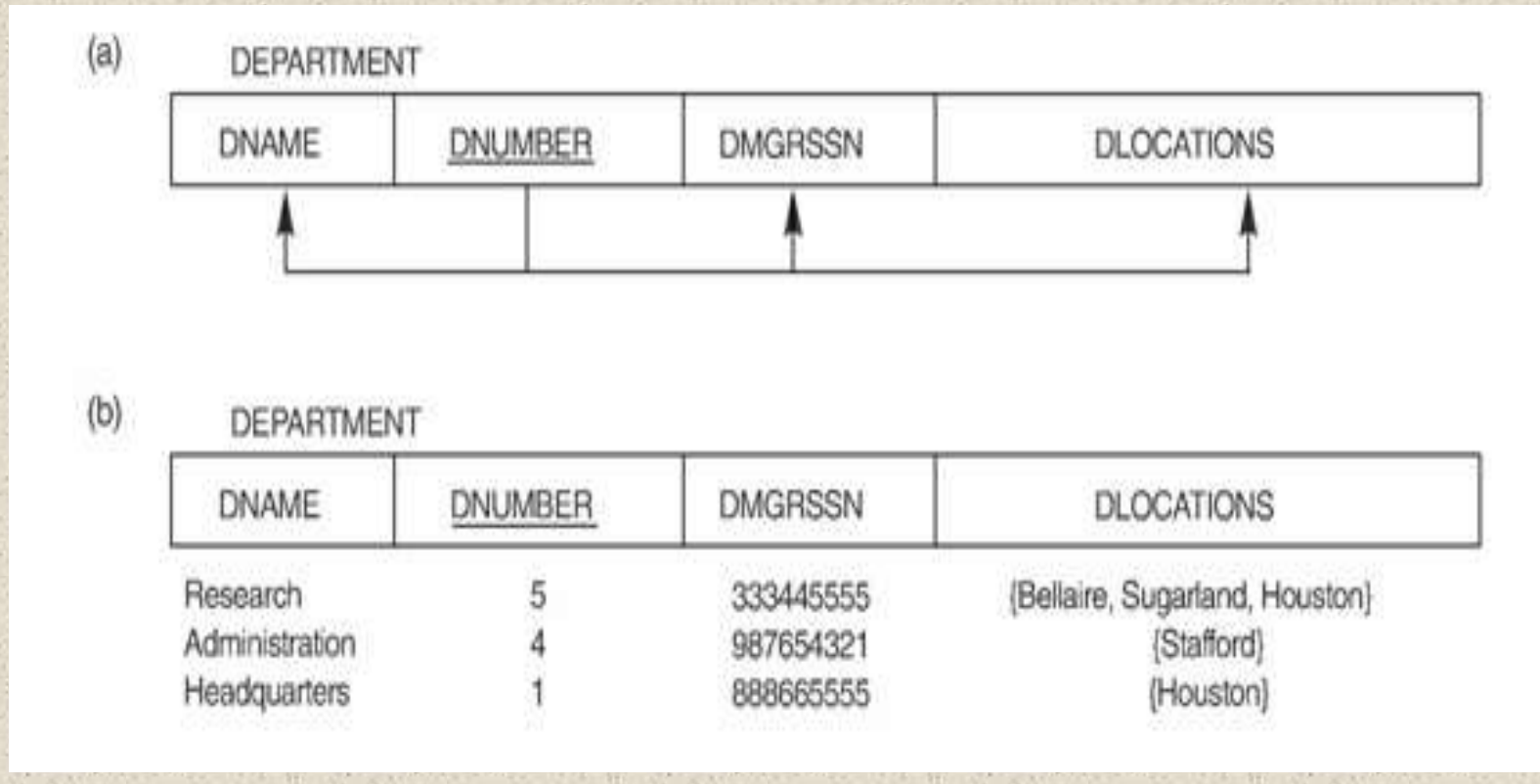
- **Database table compaction:** When we normalize the database, we convert the large table into a smaller table that leads to data and table compaction. Compaction means to have the least and required size.
- **Isolation of Data:** A good designed database states that the changes in one table or field do not affect other. This is achieved through Normalization.
- **Better performance**

First Normal Form(1NF)

- It states that domain of an attribute must include only atomic values.
- Disallows composite attributes, multivalued attributes and **nested relations**; attributes whose values *for an individual tuple* are non-atomic.
- It is considered to be part of the definition of relation

1NF: Example

- Following example is not in normal form as attribute dlocation is multivalued.



1NF:Example

- To resolve it into 1NF, remove the attribute DLocation that voilates the 1NF & 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}. Hence the normalized relations will be as:

  DEPARTMENT (Dnumber, Dname, MgrSSN)

  Dept_locations (Dnumber, Dlocation)

1NF:Example (Normalizing nested relations into 1NF)

(a) **EMP_PROJ**

SSN	ENAME	PROJS	
		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
333445555	Wong,Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya,Alicia J.	30	30.0
		10	10.0
987987987	Jabbar,Ahmad V.	10	35.0
		30	5.0
987654321	Wallace,Jennifer S.	30	20.0
		20	15.0
888665555	Borg,James E.	20	null

(c) **EMP_PROJ1**

SSN	ENAME

EMP_PROJ2

SSN	PNUMBER	HOURS

Decomposing EMP_PROJ into 1NF relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

Second Normal Form(2NF)

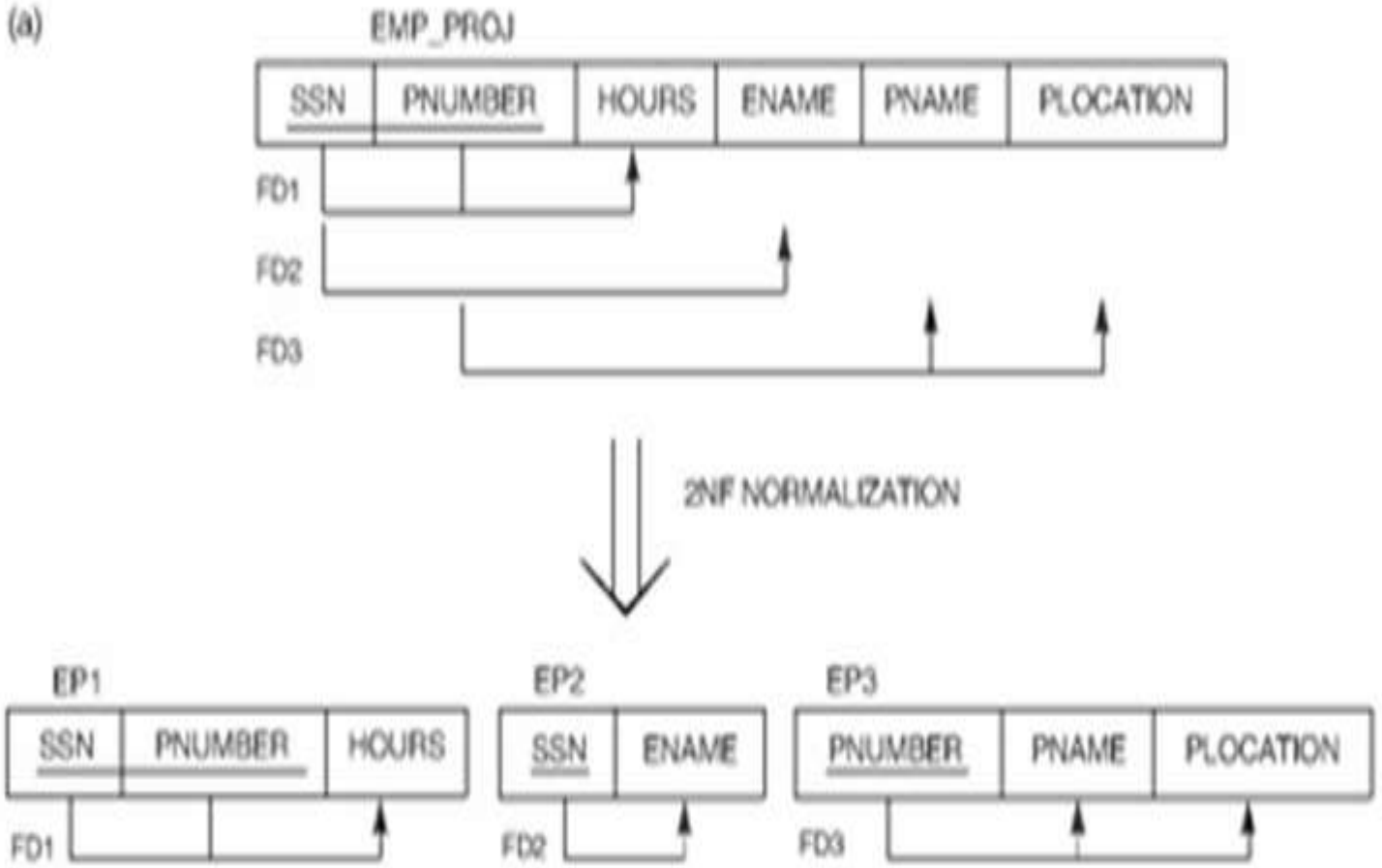
- A relation schema R is in **second normal form** (**2NF**) if every non-prime attribute A in R is fully functionally dependent on the primary key.
- R can be decomposed into 2NF relations via the process of 2NF normalization.

- **Prime attribute** - attribute that is member of the primary key K
- **Full functional dependency** - a FD Y -> Z where removal of any attribute from Y means the FD does not hold any more

  Examples:- {SSN, PNUMBER} -> HOURS is a full FD since neither SSN -> HOURS nor PNUMBER -> HOURS hold

  - {SSN, PNUMBER} -> ENAME is *not* a full FD (it is called a *partial dependency*) since SSN -> ENAME also holds

2NF: Example


(a)
EMP_PROJ
SSN
PNUMBER
HOURS
ENAME
PNAME
PLOCATION
FD1
FD2
FD3
2NF NORMALIZATION
EP1
SSN
PNUMBER
HOURS
FD1
EP2
SSN
ENAME
FD2
EP3
PNUMBER
PNAME
PLOCATION
FD3

Third Normal Form(3NF)

- **Transitive functional dependency -** a FD X -> Z that can be derived from two FDs X -> Y and Y -> Z

  Examples:

  SSN -> DMGRSSN is a *transitive* FD since

  SSN -> DNUMBER and DNUMBER -> DMGRSSN hold

  SSN -> ENAME is *non-transitive* since there is no set of attributes X where SSN -> X and X -> ENAME

Third Normal Form(3NF)

- A relation schema R is in **third normal form** (**3NF**) if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key
- R can be decomposed into 3NF relations via the process of 3NF normalization

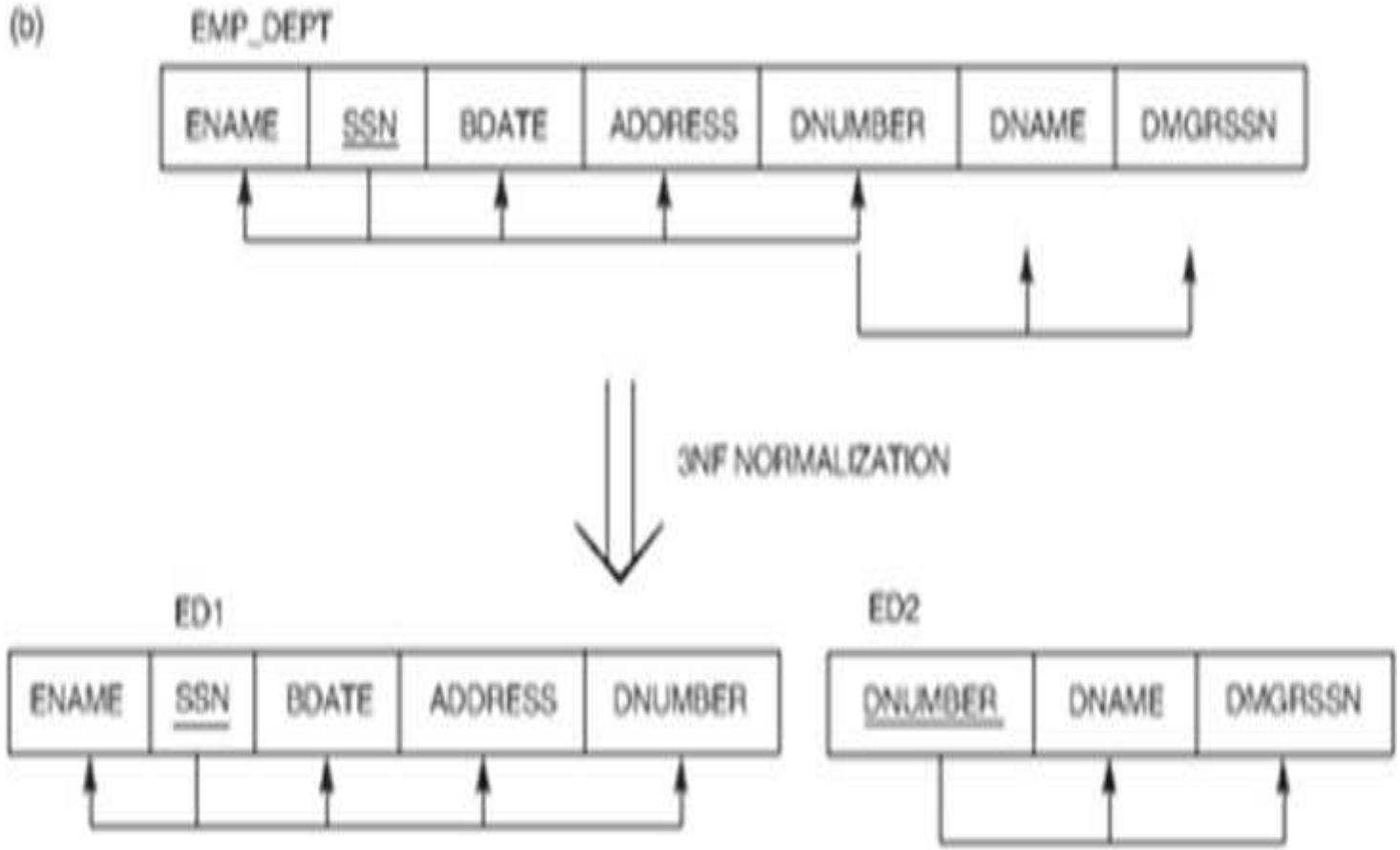
NOTE:

In X -> Y and Y -> Z, with X as the primary key, we consider this a problem only if Y is not a candidate key. When Y is a candidate key, there is no problem with the transitive dependency .

E.g., Consider EMP (SSN, Emp#, Salary).

Here, SSN -> Emp# -> Salary and Emp# is a candidate key.

3NF: Example


(b)
EMP_DEPT
ENAME
SSN
BDATE
ADDRESS
DNUMBER
DNAME
DMGRSSN
3NF NORMALIZATION
ED1
ENAME
SSN
BDATE
ADDRESS
DNUMBER
ED2
DNUMBER
DNAME
DMGRSSN

Summary of 1NF, 2NF, 3NF

Normal Forms	Test	Remedy(Normalization)
1NF	Relation should have no nonatomic attributes of nested relations.	Form new relations for each nonatomic attribute or nested relation.
2NF	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose & 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.

Summary of 1NF, 2NF, 3NF

Normal Forms	Test	Remedy(Normalization)
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 & 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 schema R is in **Boyce-Codd Normal Form** (**BCNF**) if whenever any **non-trivial functional dependency** X → A holds in R, then X is a superkey of R.
- BCNF was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. i.e every relation in BCNF is also in 3NF ;however, a relation in 3NF is not necessarily in BCNF. Thus a relation in BCNF should be in 3NF.
- In practice, most relation schemas that are in 3NF are also in BCNF. Only if there exists some functional dependency $X \rightarrow A$ that holds in a relation schema R with X not being a superkey *and* A being a prime attribute will R be in 3NF but not in BCNF.

Boyce-Codd Normal Form (BCNF)

- Example: Relation TEACH with {student, course} as a candidate key and having following two dependencies;

  FD1: {Student, Course}→ Instructor **FD2:** Instructor → Course

Figure 14.13 A relation TEACH that is in 3NF but not in BCNF.

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

Boyce-Codd Normal Form (BCNF)

- The relation Teach is in 3NF but not in BCNF.
- A relation **NOT** in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations
- Three possible decompositions for relation TEACH
  1. {student, instructor} and {student, course}
  2. {course, instructor } and {course, student}
  3. {instructor, course } and {instructor, student}
- All three decompositions will lose FD1. We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.
- Out of the above three, only the **3rd** decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).

Boyce-Codd Normal Form (BCNF)

- In general, a relation *R* not in BCNF can be decomposed so as to meet the nonadditive join property by the following procedure. It decomposes *R* successively into a set of relations that are in BCNF:

- Let *R* be the relation not in BCNF, let $X \subseteq R$, and let $X \rightarrow A$ be the FD that causes a violation of BCNF. *R* may be decomposed into two relations:
  - $R - A$
  - XA
- If either $R - A$ or XA. is not in BCNF, repeat the process.

- Hence, in previous example in slide number 38, the proper decomposition of TEACH into BCNF relations is:

  TEACH1 (Instructor, Course) and TEACH2 (Instructor, Student)

Decomposition

- Consider a universal relation schema R={A_1, A_2, …, A_n}.
- **Decomposition**: The process of decomposing the universal relation schema R into a set of relation schemas D = {R_1,R_2, …, R_m} that will become the relational database schema; D is called a ***decomposition of R***.
- Each attribute in R will appear in at least one relation schema R_i in the decomposition so that no attributes are "lost".

  i.e R = R1 ∪ R2 ∪ R_3 ∪…… ∪ R_m
- Another goal of decomposition is to have each individual relation R_i in the decomposition D be in BCNF or 3NF.
- Additional properties of decomposition are needed to prevent from generating spurious tuples

Properties of Relational Decomposition

- **Dependency Preservation Property of a Decomposition**

Definition:

- Given a set of dependencies F on R, the **projection** of F on R_i, denoted by $\pi_{Ri}(F)$ where R_i is a subset of R, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \rightarrow Y$ are all contained in R_i. Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all their left- and right-hand-side attributes are in R_i.

Properties of Relational Decomposition

- **Dependency Preservation Property of a Decomposition:**
  - a decomposition $D = \{R_1, R_2, ..., R_m\}$ of R is **dependency-preserving** with respect to F if the union of the projections of F on each R_i in D is equivalent to F; that is, $((\pi_{R1}(F)) \cup . . . \cup (\pi_{Rm}(F)))^+ = F^+$
  - If the decomposition is not dependency preserving some dependency is lost in the decomposition.

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

Properties of Relational Decomposition

- **Lossless (Nonadditive) Join Property:**
  - 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,

    $* (\pi_{R1}(r), ..., \pi_{Rm}(r)) = r$; where * is the natural join of all the relations in D:
  - In other words, if we can recover original relation from decomposed relations then lossless property holds.

    Note: The word loss in *lossless* refers to *loss of information*, not to loss of tuples. In fact, for "loss of information" a better term is "addition of spurious information". So, if a decomposition does not have this property then we may get the spurious tuples after the project and natural join operations.

Properties of Relational Decomposition

- **Lossless (Nonadditive) Join Property:**
  - Thus a decomposition is lossless if the joining attribute is a superkey in at least one of the new relations obtained after decomposition.
  - Formally,

    A decomposition of R into R_1 and R_2 is lossless join if and only if at least one of the following dependencies is in F^+:

    $R_1 \cap R_2 \rightarrow R_1$

    $R_1 \cap R_2 \rightarrow R_2$

    i.e. $R_1 \cap R_2$ forms a superkey of either R_1 or R_2

Multivalued Dependency

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The ***multivalued dependency***

  $$\alpha \rightarrow\!\!\!\rightarrow \beta$$

  holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

  $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
  $t_3[\beta] = t_1[\beta]$
  $t_3[Z] = t_2[Z]$
  $t_4[\beta] = t_2[\beta]$
  $t_4[Z] = t_1[Z]$; where $Z = R - (\alpha \cup \beta)$

A multivalued dependency can be trivial or non-trivial. It is trivial if $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$. Otherwise it is non-trivial.

Multivalued Dependency

- A relation is said to have multi-valued dependency, if the following conditions are true;
  - For a dependency A → B, if for a single value of A, multiple value of B exists, then the relation may have multi-valued dependency, A$\rightarrow\!\!\rightarrow$ B.
  - Also, a relation should have at-least 3 columns for it to have a multi-valued dependency.
  - And, for a relation R(A,B,C), if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.
- If all these conditions are true for any relation(table), it is said to have multi-valued dependency.

Multivalued Dependency

- Below we have a college enrolment table with columns SID, Course and Hobby.

SID	Course	Hobby
1	Science	Cricket
1	Maths	Hockey
2	DBMS	Cricket
2	PHP	Hockey

- As you can see in the table above, student with SID **1** has opted for two courses, **Science** and **Maths**, and has two hobbies, **Cricket** and **Hockey**.
- Here it exists multivalued dependency SID $\rightarrow\rightarrow$ Course & SID$\rightarrow\rightarrow$ Hobby

Multivalued Dependency

- Example:

Emp

Ename	Pname	Dependent_name
Ram	X	Rakesh
Ram	Y	Rao
Ram	X	Rao
Ram	Y	Rakesh

- Here we have Ename $\rightarrow\!\!\rightarrow$ Pname & Ename $\rightarrow\!\!\rightarrow$ Dependent_name

Fourth Normal Form(4NF)

- 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 $X \rightarrow\rightarrow Y$ in *F*+,21 *X* is a superkey for *R*.
- Or, equivalently;
- A relation schema *R* is in 4NF with respect to a set *D* of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a superkey for schema *R*
- If a relation is in 4NF it is in BCNF.

Fourth Normal Form(4NF)

- **4NF** is violated when a relation has undesirable multivalued dependencies and hence can be used to identify and decompose such relations.
- For a relation to be in 4NF, it should satisfy the following two conditions:
  - It should be in the **Boyce-Codd Normal Form**.
  - It should not have any **Multi-valued Dependency**.

4NF: Example

- Example:

SID	Course	Hobby
1	Science	Cricket
1	Maths	Hockey
2	DBMS	Cricket
2	PHP	Hockey

SID	Course
1	Science
1	Maths
2	DBMS
2	PHP

SID	Hobby
1	Cricket
1	Hockey
2	Cricket
2	Hockey

Fig: Decomposing the relation into two 4NF relationss

4NF: Example

- Example:

Emp

Ename	Pname	Dependent_name
Ram	X	Rakesh
Ram	Y	Rao
Ram	X	Rao
Ram	Y	Rakesh

Emp_Projects

Ename	Pname
Ram	X
Ram	Y

Emp_Dependents

Ename	Dependent_name
Ram	Rakesh
Ram	Rao

Fig: Decomposing the Emp relation into two 4NF relationss

Properties of Relational Decompositions

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

Properties of Relational Decompositions

- **Relation Decomposition and Insufficiency of Normal Forms**
- We must make sure that each attribute in *R* will appear in at least one relation schema *Ri* in the decomposition so that no attributes are *lost*; formally, we have

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

- This is called the **attribute preservation** condition of a decomposition.

Properties of Relational Decompositions

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

Properties of Relational Decompositions

- **Dependency Preservation Property of a Decomposition**
- It would be useful if each 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.* We want to preserve the dependencies because each dependency in F represents a constraint on the database. If one of the dependencies is not represented in some individual relation Ri of the decomposition, we cannot enforce this constraint by dealing with an individual relation. We may have to join multiple relations so as to include all attributes involved in that dependency.

Properties of Relational Decompositions

- **Dependency Preservation Property of a Decomposition**
- It is not necessary that the exact dependencies specified in *F* appear themselves in individual relations of the decomposition *D*. It is sufficient that the union of the dependencies that hold on the individual relations in *D* be equivalent to *F*. We now define these concepts more formally.

Properties of Relational Decompositions

- **Dependency Preservation Property of a Decomposition**
- 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. Hence, the projection of F on each relation schema Ri in the decomposition D is the set of functional dependencies in $F+$, the closure of F, such that all the left- and right-hand-side attributes of those dependencies are in Ri. We say that a decomposition $D = \{R1, R2, \ldots, Rm\}$ 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, $((\pi R1(F)) \cup K \cup (\pi Rm(F)))+ = F+$.

Properties of Relational Decompositions

- **Dependency Preservation Property of a Decomposition**
- 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 that includes all left- and right-hand-side attributes of the lost dependency, and then check that the dependency holds on the result of the JOIN—an option that is not practical.

Properties of Relational Decompositions

- **Nonadditive (Lossless) Join Property of a Decomposition**
- Another property that a decomposition *D* should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition.
- Because this is a property of a decomposition of relation *schemas*, the condition of no spurious tuples should hold on *every legal relation state*—that is, every relation state that satisfies the functional dependencies in *F*. Hence, the lossless join property is always defined with respect to a specific set *F* of dependencies.

Properties of Relational Decompositions

- **Nonadditive (Lossless) Join Property of a Decomposition**
- Formally, a decomposition $D = \{R1, R2, \ldots, Rm\}$ 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: $*(\pi R1(r), \ldots, \pi Rm(r)) = r$.

Properties of Relational Decompositions

- **Nonadditive (Lossless) Join Property of a Decomposition**
- The word loss in *lossless* refers to *loss of information*, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT (π) and NATURAL JOIN (*) operations are applied; these additional tuples represent erroneous or invalid information. We prefer the term *nonadditive join* because it describes the situation more accurately.
- The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations. We may, however, sometimes use the term **lossy design** to refer to a design that represents a loss of information.

Examples of Normalization

1NF: Example

MP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

1NF: Solution Example

EMP_ID	EMP_NAME	EMP_STATE
14	John	UP
20	Harry	Bihar
12	Sam	Punjab

EMP_ID	EMP_PHONE
14	7272826385
14	9064738238
20	8574783832
12	7390372389
12	8589830302

2NF: Example

Employee No	Department No	Employee Name	Department
1	101	Amit	OBIEE
2	102	Divya	COGNOS
3	101	Rama	OBIEE

Given FD:
{Employee NO, Department No}-> Employee Name
Department No-> Department

2NF: Solution Example

Employee No	Department No	Employee Name
1	101	Amit
2	102	Divya
3	101	Rama

Department No	Department
101	OBIEE
102	COGNOS

3NF: Example

Employee No	Salary Slip No	Employee Name	Salary
1	0001	Amit	50000
2	0002	Divya	40000
3	0003	Rama	57000

Given FD:
Employee No-> Employee Name, Salary Slip No

Salary Slip No -> Salary

3NF: Solution Example

Employee No	Salary Slip No	Employee Name
1	0001	Amit
2	0002	Divya
3	0003	Rama

Salary Slip No	Salary
0001	50000
0002	40000
0003	57000

4NF: Example

Pizza Delivery Permutations		
Restaurant	Pizza Variety	Delivery Area
A1 Pizza	Thick Crust	Springfield
A1 Pizza	Thick Crust	Shelbyville
A1 Pizza	Thick Crust	Capital City
A1 Pizza	Stuffed Crust	Springfield
A1 Pizza	Stuffed Crust	Shelbyville
A1 Pizza	Stuffed Crust	Capital City
Elite Pizza	Thin Crust	Capital City
Elite Pizza	Stuffed Crust	Capital City
Vincenzo's Pizza	Thick Crust	Springfield
Vincenzo's Pizza	Thick Crust	Shelbyville
Vincenzo's Pizza	Thin Crust	Springfield
Vincenzo's Pizza	Thin Crust	Shelbyville

4NF: Example

- The dependencies are:

  {Restaurant} $\rightarrow\rightarrow$ {Pizza Variety}

  {Restaurant} $\rightarrow\rightarrow$ {Delivery Area}

4NF: Solution Example

Restaurant	Pizza Variety
A1 Pizza	Thick Crust
A1 Pizza	Stuffed Crust
Elite Pizza	Thin Crust
Elite Pizza	Stuffed Crust
Vincenzo's Pizza	Thick Crust
Vincenzo's Pizza	Thin Crust

Restaurant	Delivery Area
A1 Pizza	Springfield
A1 Pizza	Shelbyville
A1 Pizza	Capital City
Elite Pizza	Capital City
Vincenzo's Pizza	Springfield
Vincenzo's Pizza	Shelbyville

Normalization Example : 1NF

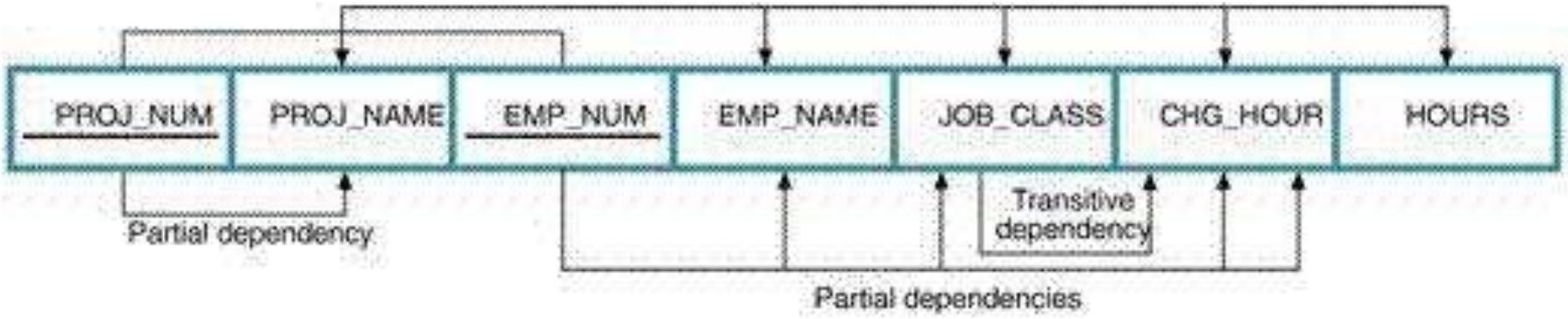


FIGURE 5.4 A DEPENDENCY DIAGRAM: FIRST NORMAL FORM (1NF)

Normalization Example: 2NF Results

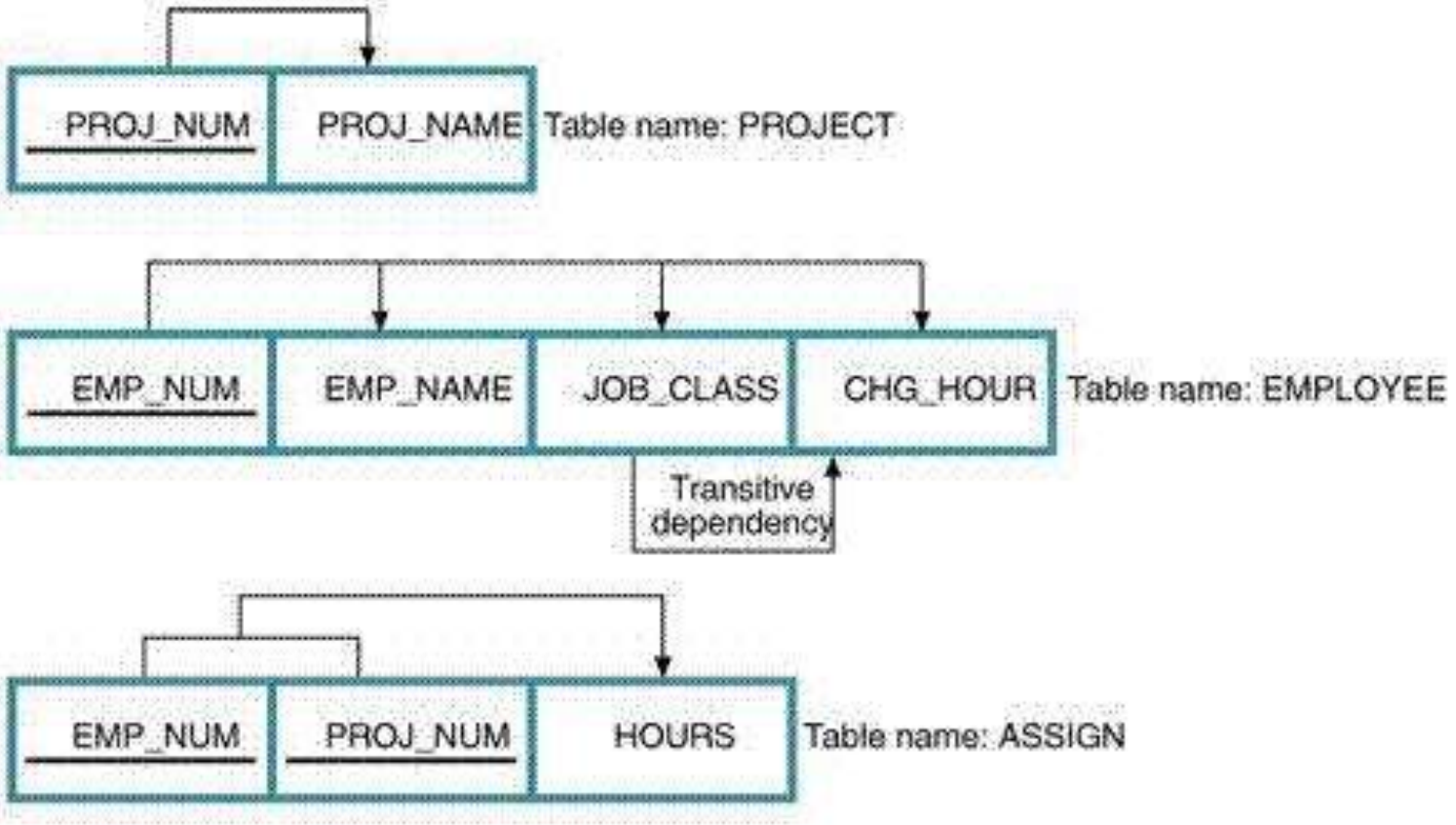


FIGURE 5.5 SECOND NORMAL FORM (2NF) CONVERSION RESULTS

Normalization Example: 3NF Results

Project (Proj_Num, Proj_name)
Employee(Emp_Num, Emp_name, Job_Class)
Job(Job_Class, CHG_Hour)
Assign(Emp_Num, Proj_Num, Hours)