Unit 3

Integrity Constraints and Relational Database Design

Prof. V. V. Kheradkar

Introduction

- ensure that changes made to the database by authorized users do not result in a loss of data consistency.
- integrity constraints guard against accidental damage to the database.
- seen two forms of integrity constraints for the E-R model
- **Key declarations** - certain attributes form a candidate key for a given entity set.
- **Form of a relationship -** many to many, one to many, one to one.

- integrity constraints with minimal overhead
- Domain constraints
- Referential Integrity
- Functional Dependency
- Normalization

Domain Constraints

- domain of possible values must be associated with every attribute
- standard domain types, such as integer types, character types, and date/time types defined in SQL
- Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take
- Domain constraints are the most elementary form of integrity constraint.
- They are tested easily by the system whenever a new data item is entered into the database

- for several attributes may have the same domain.
- For example, the attributes *customer-name* and *employee-name* might have the same domain: the set of all person names
- whether *customer-name* and *branch-name* should have the same domain.
- At the implementation level, both customer names and branch names are character strings.
- at the conceptual, rather than the physical level, *customer-name* and *branch-name* should have distinct domains

- **Equivalent Statement in Oracle**
- CREATE TABLE gender_domain
  - (gender VARCHAR2(1) PRIMARY KEY, CONSTRAINT ch_gen CHECK (gender IN ('M', 'F')));

Referential Integrity

- value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
- This condition is called **referential integrity**.

Referential Integrity in SQL

- Foreign keys can be specified as part of the SQL **create table** statement by using the **foreign key** clause

create table *account*
(*. . .*
foreign key (*branch-name*) **references** *branch(name)*
on delete cascade
on update cascade,
. . .)

- create table suppliers1(
- id number,name varchar2(50),CONSTRAINT FK_SP1 foreign key(id) references consumers(p_id) on delete cascade)

- **on delete cascade**
- if a delete of a tuple in *branch* results in this referential-integrity constraint being violated, the system does not reject the delete.
- Instead, the delete “cascades” to the *account* relation, deleting the tuple that refers to the branch that was deleted
- Similarly, the system does not reject an update to field referenced by the constraint
- instead, the system updates the field branch-name in referencing tuples in *account* to new value

- actions other than **cascade**, if the constraint is violated:
- The referencing field (branch-name) can be set to null by using **set null** in place of **cascade**

- create table suppliers2
(id number,name varchar2(50),
CONSTRAINT FK_SP2 foreign key(id) references consumers(p_id) on delete set null)

Database Modification

- Database modifications can cause violations of referential integrity.

$$\Pi_{\alpha}(r_2) \subseteq \Pi_K(r_1)$$

- **Insert**. If a tuple t_2 is inserted into r_2, the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] = t_2[\alpha]$. That is,

$$t_2[\alpha] \in \Pi_K(r_1)$$

- **Delete**. If a tuple t_1 is deleted from r_1, the system must compute the set of tuples in r_2 that reference t_1:

$$\sigma_{\alpha = t_1[K]}(r_2)$$

Database Modification

- **Update**. We must consider two cases for update: updates to the referencing relation (r_2), and updates to the referenced relation (r_1).
  - If a tuple t_2 is updated in relation r_2, and the update modifies values for the foreign key α, then a test similar to the insert case is made. Let t_2' denote the new value of tuple t_2. The system must ensure that

$$t_2'[\alpha] \in \Pi_K(r_1)$$

  - If a tuple t_1 is updated in r_1, and the update modifies values for the primary key (K), then a test similar to the delete case is made. The system must compute

$$\sigma_{\alpha = t_1[K]}(r_2)$$

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- Domain constraints and referential-integrity constraints are special forms of assertions.

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.
- Every loan has at least one customer who maintains an account with a minimum balance of $1000.00.

- When an assertion is created, the system tests it for validity.
- If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.
- This testing may introduce a significant amount of overhead if complex assertions have been made.
- Hence, assertions should be used with great care.
- The high overhead of testing and maintaining assertions has led some system developers to omit support for general assertions,
- or to provide specialized forms of assertions that are easier to test

Triggers

- A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database.
- To design a trigger mechanism, we must meet two requirements:

1. Specify when a trigger is to be executed.

This is broken up into an event that causes the trigger to be checked and a condition that must be satisfied for trigger execution to proceed.

2. Specify the *actions* to be taken when the trigger executes.

- This model of triggers is referred to as the **event-condition-action model** for triggers.
- The database stores triggers just as if they were regular data, so that they are persistent and are accessible to all database operations.
- Once we enter a trigger into the database, the database system takes on the responsibility of executing it whenever the specified event occurs and the corresponding condition is satisfied.

Need for Triggers

- Triggers are useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met.
- Example
- Instead of allowing negative account balances,
- the bank deals with overdrafts by setting the account balance to zero,
- and creating a loan in the amount of the overdraft.
- The bank gives this loan a loan number identical to the account number of the overdrawn account.

- For this example, the condition for executing the trigger is an update to the *account* relation that results in a negative *balance* value.
- Suppose that Jones' withdrawal of some money from an account made the account balance negative.
- Let *t* denote the account tuple with a negative *balance* value.
- The actions to be taken are:

•Insert a new tuple s in the loan relation with
s[loan-number] = t[account-number]
s[branch-name] = t[branch-name]
s[amount] = −t[balance]

(since t[balance] is negative, we negate t[balance] to get the loan amount—a positive number.)

•Insert a new tuple u in the borrower relation with
u[customer-name] = “Jones”
u[loan-number] = t[account-number]

• Set t[balance] to 0.

Triggers in SQL

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
    insert into borrower
            (select customer-name, account-number
             from depositor
             where nrow.account-number = depositor.account-number);
     insert into loan values
            (nrow.account-number, nrow.branch-name, – nrow.balance);
    update account set balance = 0
            where account.account-number = nrow.account-number
end
```

Authorization

- data stored in the database need protection from unauthorized access and malicious destruction or alteration
- Among the forms of malicious access are:
- Unauthorized reading of data (theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data

- **Database security** refers to protection from malicious access
- To protect the database, we must take security measures at several levels:
- **Database system**
- **Operating system**
- **Network**
- **Physical**
- **Human**

Authorization

- **Read authorization** allows reading, but not modification, of data.
- **Insert authorization** allows insertion of new data, but not modification of existing data.
- **Update authorization** allows modification, but not deletion, of data.
- **Delete authorization** allows deletion of data.

Audit Trails

- An audit trail is a log of all changes (inserts/deletes/updates) to the database,
- along with information such as which user performed the change and when the change was performed.
- Many secure database applications require an **audit trail** be maintained.

- The audit trail aids security in several ways.
- For instance, if the balance on an account is found to be incorrect,
- the bank may wish to trace all the updates performed on the account, to find out incorrect (or fraudulent) updates, as well as the persons who carried out the updates.
- The bank could then also use the audit trail to trace all the updates performed by these persons, in order to find other incorrect or fraudulent updates

- It is possible to create an audit trail by defining appropriate triggers on relation updates
- many database systems provide built-in mechanisms to create audit trails,
- Which are much more convenient to use.
- Details of how to create audit trails vary across database systems,
- Can be found in database system manuals

Authorization in SQL

- **Granting Privileges**
- **grant** <privilege list> **on** <relation name or view name> **to** <user/role list>
- **grant select on** *account* **to** *U*1, *U*2, *U*3
- **grant update** (*amount*) **on** *loan* **to** *U*1, *U*2, *U*3
- **grant references** (*branch-name*) **on** *branch* **to** *U*1

- **Roles**
- Roles can be created as follows
- **create role** *teller*
- Roles can then be granted privileges just as the users can:
- **grant select on** *account* **to** *teller*

grant *teller* **to** john
create role *manager*
grant *teller* **to** *manager*
grant *manager* **to** mary

- **The Privilege to Grant Privileges**
- **grant select on** *branch* **to** *U*1 **with grant option**

- **revoke** <privilege list> **on** <relation name or view name> **from** <user/role list> [**restrict | cascade**]

- Restrict -

  system returns an error if there are any cascading revokes

Pitfalls in Relational-Database Design

- Repetition of information
- Complicating to update
- Inability to represent certain information

Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount*)

t[*assets*] is the asset figure for the branch named *t*[*branch-name*].
t[*branch-city*] is the city in which the branch named *t*[*branch-name*] is located.
t[*loan-number*] is the number assigned to a loan made by the branch named *t*[*branch-name*] to the customer named *t*[*customer-name*].
t[*amount*] is the amount of the loan whose number is *t*[*loan-number*].

Pitfalls in Relational-Database Design

- Add a new loan to our database
- loan is made by the Perryridge branch to Adams in the amount of $1500. Let the loan-number be L-31
- Repeat the asset and city data for the Perryridge branch
- (Perryridge, Horseneck, 1700000, Adams, L-31, 1500)
- Asset of branch downtown changes from 17000 to 19000.
- Expected only one tuple need to change value.
- But In alternate design, more tuple gets changed & its costly

Pitfalls in Relational-Database Design

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Sample *lending* relation.

Pitfalls in Relational-Database Design

- Another problem with the Lending-schema design is that we cannot represent directly the information concerning a branch (branch-name, branch-city, assets) unless there exists at least one loan at the branch.

Functional Dependencies

- the goal of a relational-database design is to generate a set of relation schemas
- that allows us to store information without unnecessary redundancy,
- yet also allows us to retrieve information easily.

One approach is to design schemas that are in an appropriate normal form.

- Functional dependencies play a key role in differentiating good database designs from bad database designs.
- A **functional dependency** is a type of constraint that is a generalization of the notion of key
- Functional dependencies are constraints on the set of legal relations

- **Definition:**
- A functional dependency occurs when one attribute in a relation uniquely determines another attribute.
- This can be written A -> B
- which would be the same as stating "B is functionally dependent upon A."

- Let R be the relation, and
- let x and y be the arbitrary subset of the set of attributes of R.
- Then we say that Y is functionally dependent on x – in symbol.
- $$X \rightarrow Y$$
- (Read x functionally determines y)
- If and only if each x value in R has associated with it precisely one y value in R
- In other words
- Whenever two tuples of R agree on their x value, they also agree on their Y value.

- In a table listing employee characteristics including Social Security Number (SSN) and name,
- it can be said that name is functionally dependent upon SSN (or SSN -> name)
- because an employee's name can be uniquely determined from their SSN.
- However, the reverse statement (name -> SSN) is not true because more than one employee can have the same name but different SSNs.

Basic Concepts

- Consider a relation schema R, and
- let $\alpha \subseteq R$ and $\beta \subseteq R$.
- The **functional dependency** $\alpha \rightarrow \beta$ holds on schema R if,
- in any legal relation r(R),
- for all pairs of tuples t1 and t2 in r such that
- $t1[\alpha] = t2[\alpha]$, then
- $t1[\beta] = t2[\beta]$.

1:1 relationship between attribute(s) on left and right-hand side of a dependency; hold for all time;

- **X → Y means**
- Given any two tuples in r, if the X values are the same,
- then the Y values must also be the same.
- (but not vice versa)

- **Read "→" as "determines"**

if "K → all attributes of R"

then K is a superkey for R

- **FDs are a generalization of keys.**

- Loan-info-schema = (loan-number, branch-name, customer-name, amount)
- The set of functional dependencies that we expect to hold on this relation schema is
- loan-number →amount
- loan-number →branch-name
- We would not, however, expect the functional dependency
- loan-number →customer-name
- in general, a given loan can be made to more than one customer (for example, to both members of a husband–wife pair).

- We shall use functional dependencies in two ways:
- **1.** To test relations to see whether they are legal under a given set of functional dependencies.
- If a relation r is legal under a set F of functional dependencies, we say that r **satisfies** F.

- **2.** To specify constraints on the set of legal relations.
- concern with only those relations that satisfy a given set of functional dependencies.
- If we wish to constrain ourselves to relations on schema R that satisfy a set F of functional dependencies,
- we say that F **holds** on R.

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

- An FD is trivial if and only if
- the right hand side is a subset of the left hand side.
- e.g. <S#, P#> $\rightarrow$ <S#>. (Trivial)
- Nontrivial dependencies are the one, which are not trivial.

- when we design a relational database,
- we first list those functional dependencies that must always hold.
- In the banking example, list of dependencies includes the following:

- Branch-schema = (branch-name, branch-city, assets)
- Customer-schema = (customer-name, customer-street, customer-city)
- Loan-schema = (loan-number, branch-name, amount)
- Borrower-schema = (customer-name, loan-number)
- Account-schema = (account-number, branch-name, balance)
- Depositor-schema = (customer-name, account-number)

- On Branch-schema:
  - branch-name →branch-city
  - branch-name →assets
- On Customer-schema:
  - customer-name→ customer-city
  - customer-name→ customer-street
- On Loan-schema:
  - loan-number → amount
  - loan-number → branch-name

- On Borrower-schema:
  - No functional dependencies

- On Account-schema:
  - account-number $\rightarrow$ branch-name
  - account-number $\rightarrow$ balance

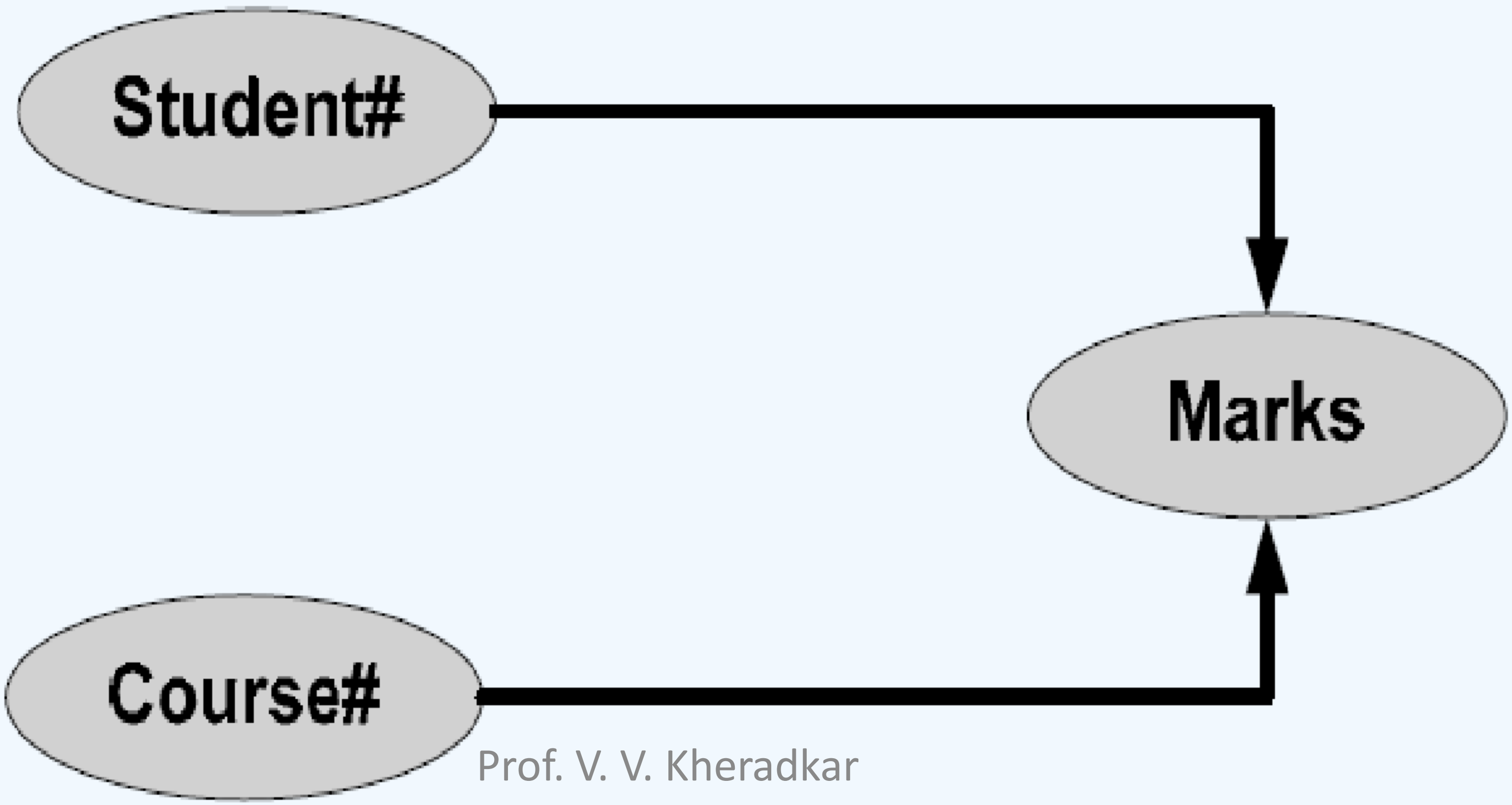
- On Depositor-schema:
  - No functional dependencies

Types of functional dependencies:

- Full Functional dependency
- Partial Functional dependency
- Transitive dependency

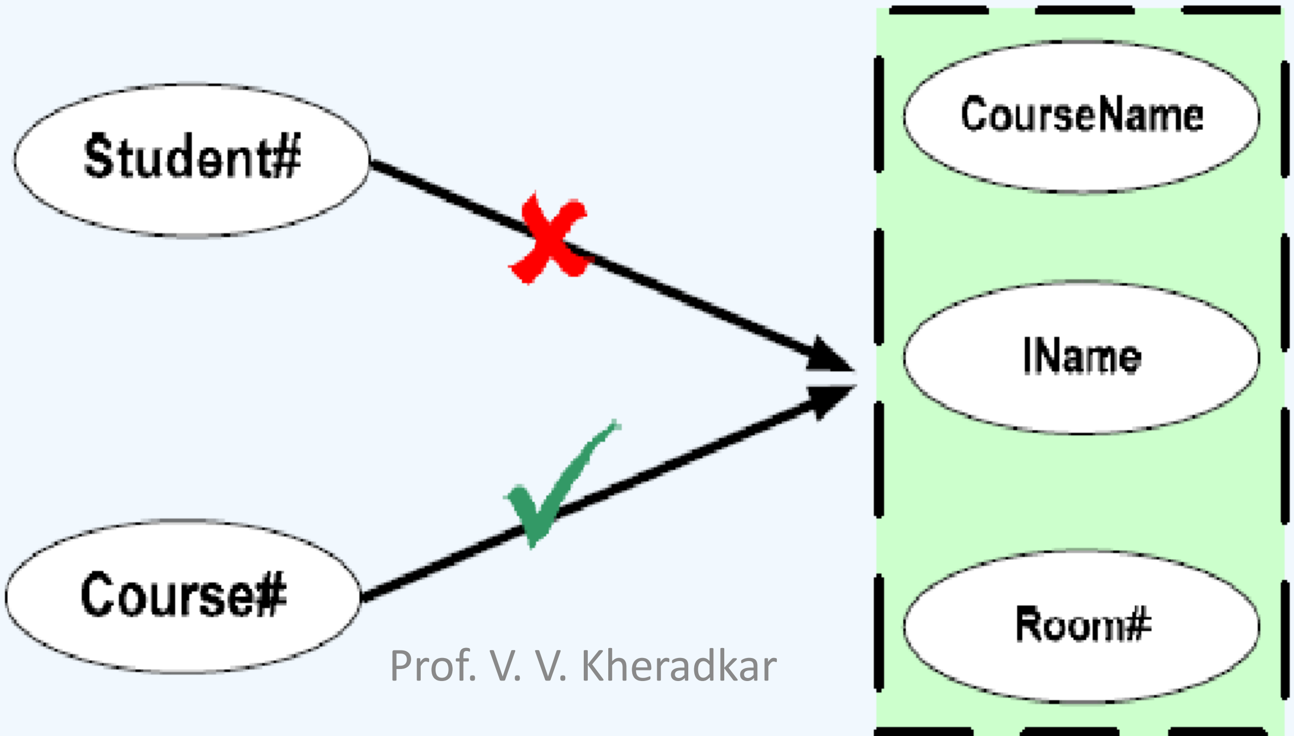
Full dependencies

X and Y are attributes.
X Functionally determines Y
Note: Subset of X should not functionally determine Y



Partial dependencies

X and Y are attributes.
Attribute Y is partially dependent on the attribute X only if it is dependent on a sub-set of attribute X.



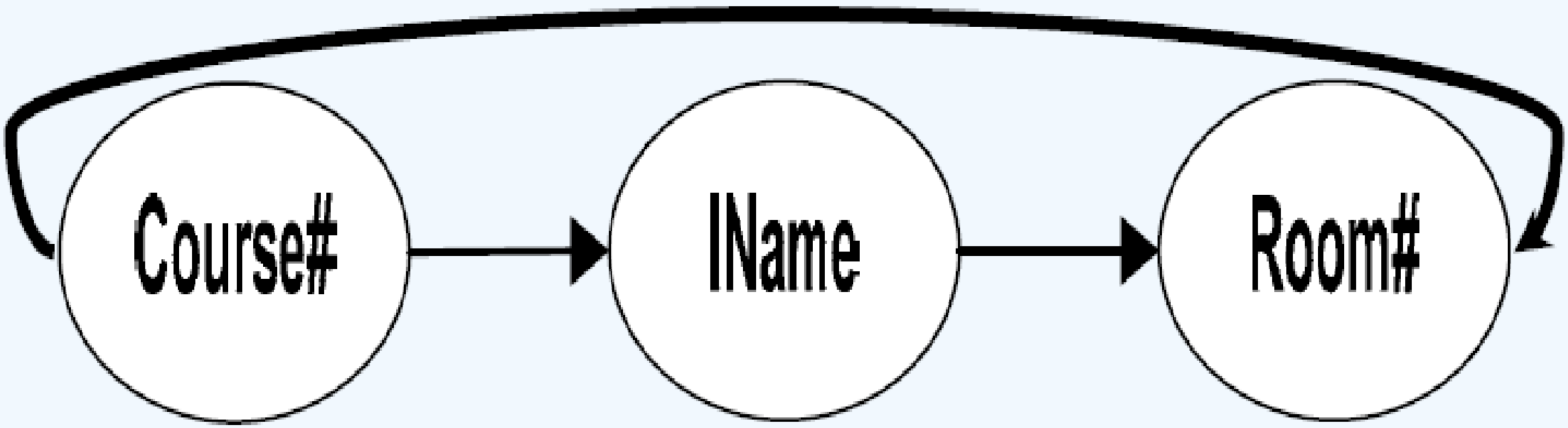
Transitive dependencies

X Y and Z are three attributes.

X -> Y

Y-> Z

=> X -> Z



Closure of a Set of Functional Dependencies

- The set of all FDs that are implied by a given set S of FDs
- is called the closure of S, denoted by S^+
- It is not sufficient to consider the given set of functional dependencies.
- We need to consider all functional dependencies that hold

- given a relation schema $R = (A, B, C, G, H, I)$
- and the set of functional dependencies
- $A \rightarrow B$
- $A \rightarrow C$
- $CG \rightarrow H$
- $CG \rightarrow I$
- $B \rightarrow H$
- The functional dependency $A \rightarrow H$ is logically implied.

- Suppose
- t1 and t2 are tuples such that
- t1[A] = t2[A]
- Since we are given that A→B,
- it follows from the definition of functional dependency that
- t1[B] = t2[B]
- Then, since we are given that B → H,
- it follows from the definition of functional dependency that
- t1[H] = t2[H]

- Let F be a set of functional dependencies.
- The **closure** of F, denoted by F+,
- is the set of all functional dependencies logically implied by F.
- Given F, we can compute F+ directly from the formal definition of functional dependency.
- If F is large, this process would be lengthy and difficult.

- **Axioms**, or rules of inference, provide a simpler technique for reasoning about functional dependencies.

- We can use the following three rules to find logically implied functional dependencies.

- By applying these rules repeatedly, we can find all of $F+$, given F.
- This collection of rules is called **Armstrong's axioms** in honuor of the person who first proposed it.

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

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

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

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

- Although Armstrong’s axioms are complete,
- it is tiresome to use them directly for the computation of F^+.
- To simplify matters further, we list additional rules.

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

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

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

1. **Reflexivity:** if B is a subset of A, then $A \rightarrow B$.
2. **Augmentation:** if $A \rightarrow B$ then $AC \rightarrow BC$
3. **Transitivity:** it $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.
4. **Self – determination:** $A \rightarrow A$.
5. **Decomposition:** If $A \rightarrow BC$, then $A \rightarrow B, A \rightarrow C$.
6. **Union:** it $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$
7. **Composition:** if $A \rightarrow B$, $C \rightarrow D$ then $AC \rightarrow BD$.
8. If $A \rightarrow B$ and $C \rightarrow D$, then All $(C – B) \rightarrow BD$

- Let us apply our rules to the example of schema
- $R = (A, B, C, G, H, I)$ and
- the set F of functional dependencies
- $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$.
- We list several members of F+ here:

- $A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule.
- $CG \rightarrow HI$. Since $CG \rightarrow H$ and $CG \rightarrow I$, the union rule implies that $CG \rightarrow HI$.

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

$F^+ = F$

repeat

    for each functional dependency f in F^+

        apply reflexivity and augmentation rules on f

        add the resulting functional dependencies to F^+

    for each pair of functional dependencies f_1 and f_2 in F^+

        if f_1 and f_2 can be combined using transitivity

            add the resulting functional dependency to F^+

until F^+ does not change any further

Figure 7.6 A procedure to compute F^+.

Closure of Attribute Sets

- To test whether a set α is a superkey,
- we must devise an algorithm for computing the set of attributes functionally determined by α.
- One way of doing this is to compute $F+$,
- take all functional dependencies with α as the left-hand side, and take the union of the right-hand sides of all such dependencies.
- However, doing so can be expensive, since $F+$ can be large.

Closure of Attribute Sets

- Algorithm to compute a+, the closure of a under F

$result := \alpha;$
while (changes to $result$) **do**
  for each functional dependency $\beta \rightarrow \gamma$ **in** F **do**
    begin
      if $\beta \subseteq result$ **then** $result := result \cup \gamma;$
    end

- compute $(AG)^+$ with the functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$.
- We start with result = AG.
- $A \rightarrow B$ causes us to include B in result.
- we observe that $A \rightarrow B$ is in F, $A \subseteq$ result (which is AG), so result := result $\cup$ B.
- $A \rightarrow C$ causes result to become ABCG.
- $CG \rightarrow H$ causes result to become ABCGH.
- $CG \rightarrow I$ causes result to become ABCGHI.

Canonical Cover

- Suppose that we have a set of functional dependencies F on a relation schema.
- Whenever a user performs an update on the relation,
- the database system must ensure that the update does not violate any functional dependencies,
- that is, all the functional dependencies in F are satisfied in the new database state.
- The system must roll back the update if it violates any functional dependencies in the set *F*.

- We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set.
- Any database that satisfies the simplified set of functional dependencies will also satisfy the original set, and vice versa,
- since the two sets have the same closure.
- However, the simplified set is easier to test.

- An attribute of a functional dependency is said to be **extraneous**
- if we can remove it without changing the closure of the set of functional dependencies.
- The formal definition of **extraneous attributes** is as follows.
- For example, suppose we have the functional dependencies $AB \rightarrow C$ and $A \rightarrow C$ in F.
- Then, B is extraneous in $AB \rightarrow C$.

- A **canonical cover** Fc for F is a set of dependencies such that F logically implies all dependencies in Fc,
- and Fc logically implies all dependencies in F.
- Furthermore, Fc must have the following properties:
- No functional dependency in Fc contains an extraneous attribute.
- Each left side of a functional dependency in Fc is unique.
- That is, there are no two dependencies $\alpha 1 \rightarrow \beta 1$ and $\alpha 2 \rightarrow \beta 2$ in Fc such that $\alpha 1 = \alpha 2$.

$F_c = F$

repeat

Use the union rule to replace any dependencies in F_c of the form

$\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.

Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous

attribute either in α or in β.

/* Note: the test for extraneous attributes is done using F_c, not F */

If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$.

until F_c does not change.

Figure 7.8 Computing canonical cover

- Consider the following set F of functional dependencies on schema (A,B,C):
- $A \rightarrow BC$
- $B \rightarrow C$
- $A \rightarrow B$
- $AB \rightarrow C$
- Let us compute the canonical cover for *F*.

- There are two functional dependencies with the same set of attributes on the left side of the arrow:
- $A \rightarrow BC$
- $A \rightarrow B$
- We combine these functional dependencies into $A \rightarrow BC$.

- A is extraneous in $AB \rightarrow C$ because F logically implies $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$.
- This assertion is true because $B \rightarrow C$ is already in our set of functional dependencies.

- C is extraneous in $A \rightarrow BC$, since $A \rightarrow BC$ is logically implied by $A \rightarrow B$ and $B \rightarrow C$.

- Thus, our canonical cover is
- $A \rightarrow B$
- $B \rightarrow C$

- A canonical cover might not be unique.

Testing if an Attribute is Extraneous

Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.

To test if attribute $A \in \alpha$ is extraneous in α

1. compute $(\{\alpha\} - A)^+$ using the dependencies in *F*
2. check that $(\{\alpha\} - A)^+$ contains β; if it does, *A* is extraneous in α

To test if attribute $A \in \beta$ is extraneous in β

1. compute α^+ using only the dependencies in

   $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
2. check that α^+ contains *A;* if it does, *A* is extraneous in β

Computing a Canonical Cover

Q.1. Let R = (*A, B, C) and F = {AB* $\rightarrow$ *C, A* $\rightarrow$ *C*}

Find canonical cover?

Check A is extraneous in *AB* $\rightarrow$ *C?*

compute $(AB - A)^+$ *under F*

if contains C then extraneous

Check B is extraneous in *AB* $\rightarrow$ *C?*

compute $(AB - B)^+$ *under F*

if contains C then extraneous

Q.2. Let R = (*A, B, C,D,E) and F = {AB* $\rightarrow$ *CD, A* $\rightarrow$ E, *E* $\rightarrow$ C }

Find canonical cover?

Check A is extraneous in *AB* $\rightarrow$ *CD?*

Check B is extraneous in *AB* $\rightarrow$ *CD?*

Decomposition

- Lending-schema = (branch-name, branch-city, assets, customer-name, loan-number, amount)
- The set F of functional dependencies that we require to hold on Lending-schema are

- branch-name → branch-city, assets
- loan-number → amount, branch-name

- Lending-schema is an example of a bad database design.
- Assume that we decompose it to the following three relations:

- Branch-schema = (branch-name, branch-city, assets)
- Loan-schema = (loan-number, branch-name, amount)
- Borrower-schema = (customer-name, loan-number)

Normalization

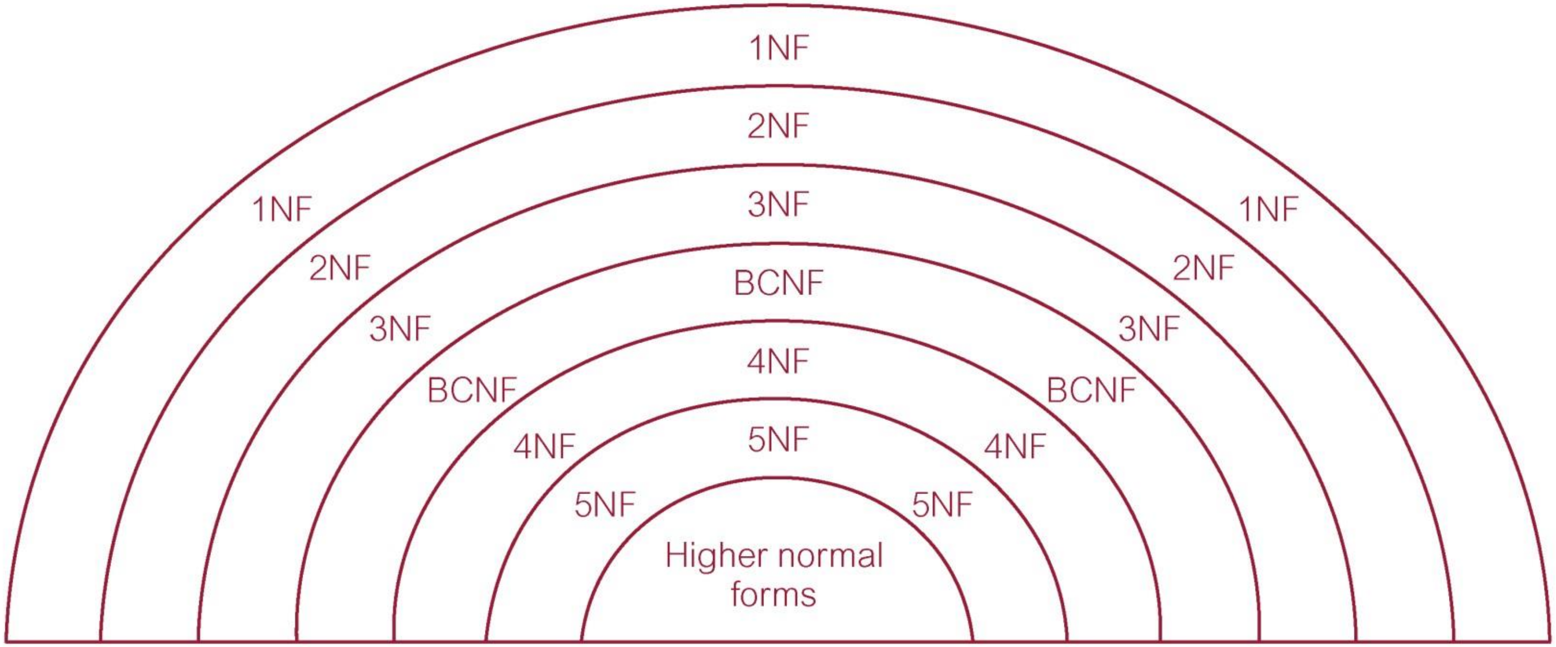
- the goal of a relational-database design is to generate a set of relation schemas
- that allows us to store information without unnecessary redundancy,
- yet also allows us to retrieve information easily.
- One approach is to design schemas that are in an appropriate normal form.

- design standards are called as normal forms

- Database design theory includes design standards called *normal forms*.
- The process of making your data and tables match these standards is called *normalizing data* or *data normalization*.
- By normalizing your data, you eliminate redundant information
- and organize your table to make it easier to manage the data and make future changes to the table and database structure.
- This process removes the insertion, deletion, and modification anomalies.

- In normalizing data, we usually divide large tables into smaller, easier to maintain tables.
- we can then use the technique of adding foreign keys to enable connections between the tables.
- Data normalization is part of the database design process and is not specific nor unique to any particular RDBMS.
- There are first, second, third, Boyce-Codd, fourth, and fifth normal forms.
- Each normal form represents an increasingly stringent set of rules;

Relationship Between Normal Forms



Unnormalized Form (UNF)

- A table that contains one or more repeating groups.

- To create an unnormalized table:
- transform data from information source (e.g. form) into table format with columns and rows.

First Normal Form

- **First normal form**, imposes a very basic requirement on relations;
- unlike the other normal forms, it does not require additional information such as functional dependencies.
- **A relation in which intersection of each row and column contains one and only one value.**
- A table is in first normal form (1NF) if there are no repeating groups.
- A repeating group is a set of logically related fields or values that occur multiple times in one record.

- Under first normal form, all occurrences of a record type must contain the same number of fields.
- For all attributes only one value
- All the attributes are atomic.
- Atomic- the smallest level to which data may be broken down and remain meaningful.

-
- example
- 1. if the schema of a relation employee included an attribute children whose domain elements are sets of names,
- the schema would not be in first normal form.
- 2. Composite attributes, such as an attribute address with component attributes street and city, also have nonatomic domains.

EmployeeID	Name	Project	Time
EN1-26	Sean O'Brien	30-452-T3, 30-457-T3, 32-244-T3	0.25, 0.40, 0.30
EN1-33	Amy Guya	30-452-T3, 30-382-TC, 32-244-T3	0.05, 0.35, 0.60
EN1-35	Steven Baranco	30-452-T3, 31-238-TC	0.15, 0.80
EN1-36	Elizabeth Roslyn	35-152-TC	0.90
EN1-38	Carol Schaaf	36-272-TC	0.75
EN1-40	Alexandra Wing	31-238-TC, 31-241-TC	0.20, 0.70

EmpID	Last Name	First Name	Project1	Time1	Project2	Time2	Project3	Time3
EN1-26	O'Brien	Sean	30-452-T3	0.25	30-457-T3	0.40	32-244-T3	0.30
EN1-33	Guya	Amy	30-452-T3	0.05	30-382-TC	0.35	32-244-T3	0.60
EN1-35	Baranco	Steven	30-452-T3	0.15	31-238-TC	0.80		
EN1-36	Roslyn	Elizabeth	35-152-TC	0.90				
EN1-38	Schaaf	Carol	36-272-TC	0.75				
EN1-40	Wing	Alexandra	31-238-TC	0.20	31-241-TC	0.70		

* EmployeeID	Last Name	First Name
EN1-26	O'Brien	Sean
EN1-33	Guya	Amy
EN1-35	Baranco	Steven
EN1-36	Roslyn	Elizabeth
EN1-38	Schaaf	Carol
EN1-40	Wing	Alexandra

* ProjectNum	* EmployeeID	Time
30-328-TC	EN1-33	0.35
30-452-T3	EN1-26	0.25
30-452-T3	EN1-33	0.05
30-452-T3	EN1-35	0.15
31-238-TC	EN1-35	0.80
30-457-T3	EN1-26	0.40
31-238-TC	EN1-40	0.20
31-241-TC	EN1-40	0.70
32-244-T3	EN1-33	0.60
35-152-TC	EN1-36	0.90
36-272-TC	EN1-38	0.75

Second Normal Form (2NF)

- Suppose a table is in first normal form and
- each non-key field is functionally dependent on the entire primary key.
- Look for values that occur multiple times in a non-key field.
- This tells you that you have too many fields in a single table.
- No Partial dependency between non-key attributes and key attributes

*EmployeeID	LastName	FirstName	*ProjectNumber	ProjectTitle
EN1-26	O'Brien	Sean	30-452-T3	STAR manual
EN1-26	O'Brien	Sean	30-457-T3	ISO procedures
EN1-26	O'Brien	Sean	31-124-T3	Employee handbook
EN1-33	Guya	Amy	30-452-T3	STAR manual
EN1-33	Guya	Amy	30-482-TC	Web Site
EN1-33	Guya	Amy	31-241-TC	New catalog
EN1-35	Baranco	Steven	30-452-T3	STAR manual
EN1-35	Baranco	Steven	31-238-TC	STAR prototype
EN1-36	Roslyn	Elizabeth	35-152-TC	STAR pricing
EN1-38	Schaaf	Carol	36-272-TC	Order system
EN1-40	Wing	Alexandra	31-238-TC	STAR prototype
EN1-40	Wing	Alexandra	31-241-TC	New catalog

- Solution to this lies in breaking the table into smaller tables.
- more tables is the solution to most problems encountered during data normalisation.
- This removes the modification anomaly of having the repeated values.

Complying with second normal form

design new tables that will eliminate the repeated data in the non-key fields.

- 1. Decide what fields belong together in a table, think about which field determines the values in other fields.
- Create a table for those fields and enter the sample data.
- 2.Think about what the primary key for each table would be and about the relationship between the tables.
- If necessary, add foreign keys.
- 3.Mark the primary key for each table and make sure that you don't have repeating data in non-key fields.

EMPLOYEES

*EmployeeID	Last Name	First Name
EN1-26	O'Brien	Sean
EN1-33	Guya	Amy
EN1-35	Baranco	Steven
EN1-36	Roslyn	Elizabeth
EN1-38	Schaaf	Carol
EN1-40	Wing	Alexandra

- Employee_Projects

*EmployeeID	*ProjectNum
EN1-26	30-452-T3
EN1-26	30-457-T3
EN1-26	31-124-T3
EN1-33	30-328-TC
EN1-33	30-452-T3
EN1-33	32-244-T3
EN1-35	30-452-T3
EN1-35	31-238-TC
EN1-36	35-152-TC
EN1-38	36-272-TC
EN1-40	31-238-TC
EN1-40	31-241-TC

- Projects

*ProjectNum	ProjectTitle
30-452-T3	STAR manual
30-457-T3	ISO procedures
30-482-TC	Web site
31-124-T3	Employee handbook
31-238-TC	STAR prototype
31-238-TC	New catalog
35-152-TC	STAR pricing
36-272-TC	Order system

Third Normal Form (3NF)

- Consider a table is in second normal form (2NF) and there are no transitive dependencies.
- A **transitive dependency** is a type of functional dependency in which
- the value in a non-key field is determined by the value in another non-key field and that field is not a candidate key.
- **3NF - A relation that is in 1NF and 2NF and in which no non-key attribute is transitively dependent on the primary key.**

*ProjectNum	ProjectTitle	ProjectMgr	Phone
30-452-T3	STAR manual	Garrison	2756
30-457-T3	ISO procedures	Jacanda	2954
30-482-TC	Web site	Friedman	2846
31-124-T3	Employee handbook	Jones	3102
31-238-TC	STAR prototype	Garrison	2756
31-241-TC	New catalog	Jones	3102
35-152-TC	STAR pricing	Vance	3022
36-272-TC	Order system	Jacanda	2954

- The phone number is repeated each time a manager name is repeated.
- This is because the phone number is only a second cousin to the project number.
- It's dependent on the manager,
- which is dependent on the project number
- (a transitive dependency).

- The ProjectMgr field is not a candidate key because
- the same person manages more than one project.
- Again, the solution is to remove the field with repeating data to a separate table.

- Complying with third normal form
- 1. Think about which fields belong together and create new tables to hold them.
- 2. Enter the sample data and check for unnecessarily (not part of primary key) repeated values.
- 3. Identify the primary key for each table and, if necessary, add foreign keys.

- # Projects

*ProjectNum	ProjectTitle	ProjectMgr
30-452-T3	STAR manual	Garrison
30-457-T3	ISO procedures	Jacanda
30-482-TC	Web site	Friedman
31-124-T3	Employee handbook	Jones
31-238-TC	STAR prototype	Garrison
31-241-TC	New catalog	Jones
35-152-TC	STAR pricing	Vance
36-272-TC	Order system	Jacanda

- Manager

*ProjectMgr	Phone
Friedman	2846
Garrison	2756
Jacanda	2954
Jones	3102
Vance	3022

- there are no unnecessarily repeating values in non-key fields
- and the value in each non-key field is determined by the value(s) in the key field(s).

Boyce-Codd Normal Form

- A table is in third normal form (3NF) and all determinants are candidate keys.
- Boyce-Codd normal form (BCNF) can be thought of as a "new" third normal form.
- It was introduced to cover situations that the "old" third normal form did not address.
- determinant - determines the value in another field
- candidate keys -qualify for designation as primary key.
- This normal form applies to situations where you have overlapping candidate keys.

Candidate Key

- A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely an entity in the entity set.
- superkeys for which no proper subset is a superkey.
- Such minimal superkeys are called **candidate keys**.
- several distinct sets of attributes could serve as a candidate key.
- **primary key** denote a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set.

Student#	EmailID	Course#	Marks
101	Davis@myuni.edu	M4	82
102	Daniel@myuni.edu	M4	62
101	Davis@myuni.edu	H6	79
103	Sandra@myuni.edu	C3	65
104	Evelyn@myuni.edu	B3	77
102	Daniel@myuni.edu	P3	68
105	Susan@myuni.edu	P3	89
103	Sandra@myuni.edu	B4	54
105	Susan@myuni.edu	H6	87
104	Evelyn@myuni.edu	M4	65

- In the above table, we have two candidate keys namely
- **STUDENT# COURSE#**
- **COURSE# EmailId**.

- COURSE# is overlapping among those candidate keys.
- Hence these candidate keys are called as **“Overlapping Candidate Keys”**.

- there are four determinants in this relation namely:
- STUDENT# (STUDENT# decides EmailiD)
- EMailID (EmailID decides STUDENT#)
- STUDENT# COURSE# (decides Marks)
- COURSE# EMailID (decides Marks).

- Only combination of STUDENT# COURSE# and COURSE# EMailID are candidate keys.

STUDENT TABLE

Student#	EmailID
101	Davis@myuni.edu
102	Daniel@myuni.edu
103	Sandra@myuni.edu
104	Evelyn@myuni.edu
105	Susan@myuni.edu

Student#	Course#	Marks
101	M4	82
102	M4	62
101	H6	79
103	C3	65
104	B3	77
102	P3	68
105	P3	89
103	B4	54
105	H6	87
104	M4	65

- Difference between 3NF and BCNF is that for a functional dependency $A \rightarrow B$,
- BCNF insists that for this dependency to remain in a relation, A must be a candidate key.
- Every relation in BCNF is also in 3NF. However, relation in 3NF may not be in BCNF.

Fourth Normal Form:

- A table is in Boyce-Codd normal form (BCNF) and there are no multi-valued dependencies.
- A multi-valued dependency occurs when,
- for each value in field A, there is a set of values for field B and a set of values for field C but fields B and C are not related.

- Although BCNF removes anomalies due to functional dependencies,
- another type of dependency called a multi-valued dependency (MVD) can also cause data redundancy.
- Possible existence of MVDs in a relation is due to 1NF and can result in data redundancy.

- Look for repeated or null values in non-key fields.
- A multi-valued dependency occurs when the table contains fields that are not logically related.

*Movie	*Star	*Producer
Once Upon a Time	Julie Garland	Alfred Brown
Once Upon a Time	Mickey Rooney	Alfred Brown
Once Upon a Time	Julie Garland	Muriel Humphreys
Once Upon a Time	Mickey Rooney	Muriel Humphreys
Moonlight	Humphrey Bogart	Alfred Brown
Moonlight	Julie Garland	Alfred Brown

- A movie can have more than one star and more than one producer.
- A star can be in more than one movie.
- A producer can produce more than one movie.
- The primary key would have to include all three fields and so this table would be in BCNF.
- have unnecessarily repeated values, with the data maintenance problems that causes,
- and will have trouble with deletion anomalies.

- The Star and the Producer really aren't logically related.
- The Movie determines the Star and the Movie determines the Producer.
- The answer is to have a separate table for each of those logical relationships - one holding Movie and Star and the other with Movie and Producer, as shown below:

*Movie	*Star
Once Upon a Time	Julie Garland
Once Upon a Time	Mickey Rooney
Moonlight	Humphrey Bogart
Moonlight	Julie Garland

*Movie	*Producer
Once Upon a Time	Alfred Brown
Once Upon a Time	Muriel Humphreys
Moonlight	Alfred Brown

Fifth Normal Form

- A table is in fourth normal form (4NF) and there are no cyclic dependencies.
- A cyclic dependency can occur only when you have a multi-field primary key consisting of three or more fields.
- For example, let's say your primary key consists of fields A, B, and C.
- A cyclic dependency would arise if the values in those fields were related in pairs of A and B, B and C, and A and C.

- Fifth normal form is also called projection-join normal form.
- A projection is a new table holding a subset of fields from an original table.
- When properly formed projections are joined, they must result in the same set of data that was contained in the original table

- Look for the number of records that will have to be added or maintained
- Following is some sample data about buyers, the products they buy, and the companies they buy from on behalf of MegaMall, a large department store.

Buyer	*Product	*Company
Chris	jeans	Levi
Chris	jeans	Wrangler
Chris	shirts	Levi
Lori	jeans	Levi

- a table with cyclic dependencies
- The primary key consists of all three fields.
- One data maintenance problem that occurs is that you need to add a record for every buyer who buys a product for every company that makes that product or they can't buy from them.
- That may not appear to be a big deal in this sample of 2 buyers, 2 products, and 2 companies (2 X 2 X 2 = 8 total records).
- But what if you went to 20 buyers, 50 products, and 100 companies (20 X 50 X 100 = 100,000 potential records)?

*Buyer	*Product
Chris	jeans
Chris	shirts
Lori	jeans

Product	*Company
jeans	Wrangler
jeans	Levi
shirts	Levi

- However, if you joined the two tables above on the Product field,
- it would produce a record not part of the original data set
- it would say that Lori buys jeans from Wrangler.
- This is where the projection-join concept comes in.

*Buyer	*Product
Chris	jeans
Chris	shirts
Lori	jeans

*Product	*Company
jeans	Wrangler
jeans	Levi
shirts	Levi

*Buyer	*Company
Chris	Levi
Chris	Wrangler
Lori	Levi

- Above, tables that comply with 5NF
- When the first two tables are joined by Product and the result joined to the third table by Buyer and Company, the result is the original set of data.
- In our scenario of 20 buyers, 50 products, and 100 companies, you would have, at most, 1000 records in the Buyers table (20 X 50), 5000 records in the Products table (50 X 100), and 2000 records in the Companies table (20 X 100). With a maximum of 8000 records, these tables would be much easier to maintain than the possible 100,000 records we saw earlier.

Denormalization

- Normalization is one of many database design goals
- Normalized table requirements

  Additional processing
  Loss of system speed

Normalization purity is difficult to sustain due to conflict in:

Design efficiency
Information requirements
Processing

Unnormalized Table Defects

- Data updates less efficient
- Indexing more cumbersome
- No simple strategies for creating views