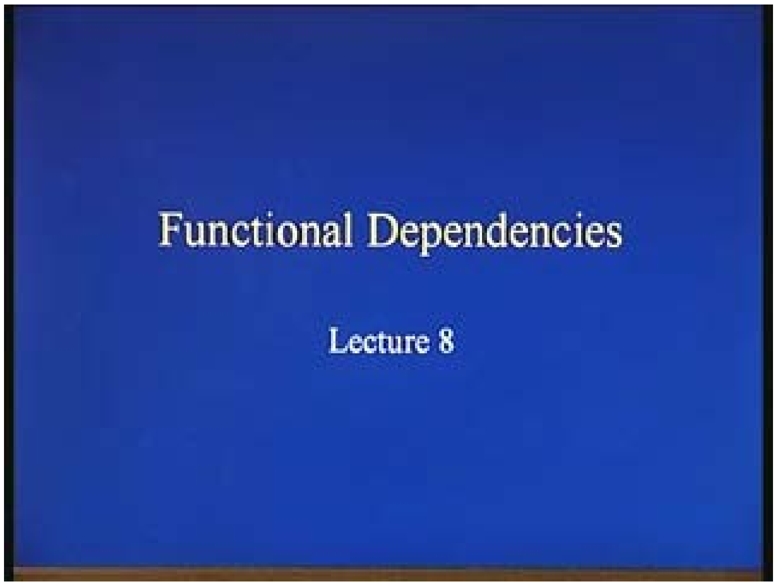
**Database Management System**
**Dr. S. Srinath**
**Department of Computer Science & Engineering**
**Indian Institute of Technology, Madras**
**Lecture No. # 8**

# Functional Dependencies and Normal Forms

Two different kinds of database models now, when we are talking about how to design a database system. The first data model that we saw was the entity relationship model which is especially used for building conceptual schema.

(Refer Slide Time: 00:01:31)



## Functional Dependencies

Lecture 8

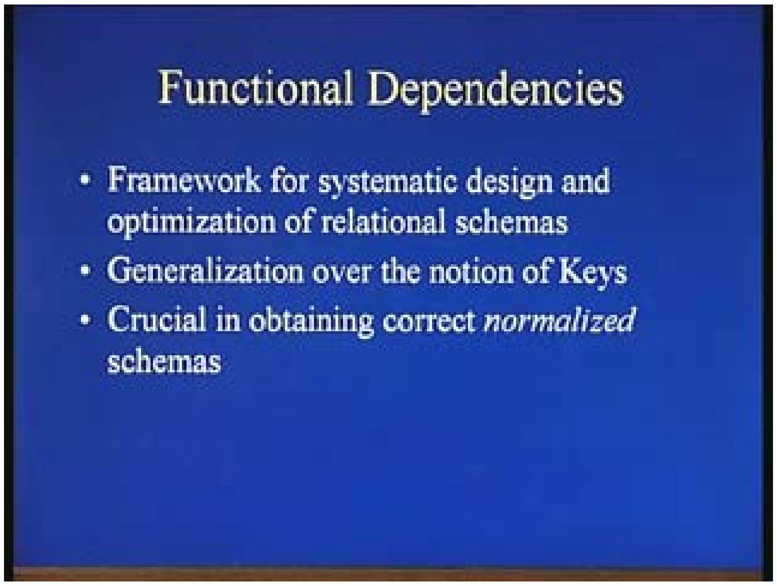
So as we have defined a conceptual schema is something that is meant for human consumption that is it is meant for communication with end users so that they can understand what the database designers have really understood about their domain, what kinds of entities exist, what kinds of relationship exist and so on. But an entity relationship schema is not really the best data model for storing data in a computer. And for storing data that is the internal schema or the physical schema it is usually the relational model that is used and the relational model in some sense has some kind of common terms with the entity schema like we have seen the concept of keys, foreign keys and referential integrity and so on which hold both for entity schema and the relational schema.

However the relational schema is mainly meant for the physical schema or the internal schema while the conceptual schema is meant for human consumption, while the physical schema is meant for the computer.

Now usually what happens in a database design process is that we use some kind of tools or some kind of case tools to build an ER model or of the information system that we are trying to design and there are number of automated techniques by which or number of automated tools which can take a ER representation of a schema and then convert it into a relational schema. However while this is automatic, this need not always result in the most optimal form of the relational schema. And what do we mean by optimal? We are going to try to formulize this notion today and try to see which kinds of relational schema or schemas are better than which other kinds and are there some kind of techniques, formal techniques that I can use by which without trying to understand the semantics, just given a relational schema doesn't matter in what context can I try to optimize it into a fashion that can help in efficient storage and retrieval of data.

So today in this session we are going to look at an important concept of relational database design namely the idea of functional dependencies. The notion of functional dependencies is fundamental to the design of different normal forms which we are going to see in a relational database schema. So what are functional dependencies? Functional dependencies are a frame work for systematic design and optimization of relational schemas. Of course there are many more nonsystematic options for optimization. For example if we know something extra about the operational domain or the specific database in use, we can up always optimize something more than what we can do with the relational, with functional dependencies.

(Refer Slide Time: 00:04:20)

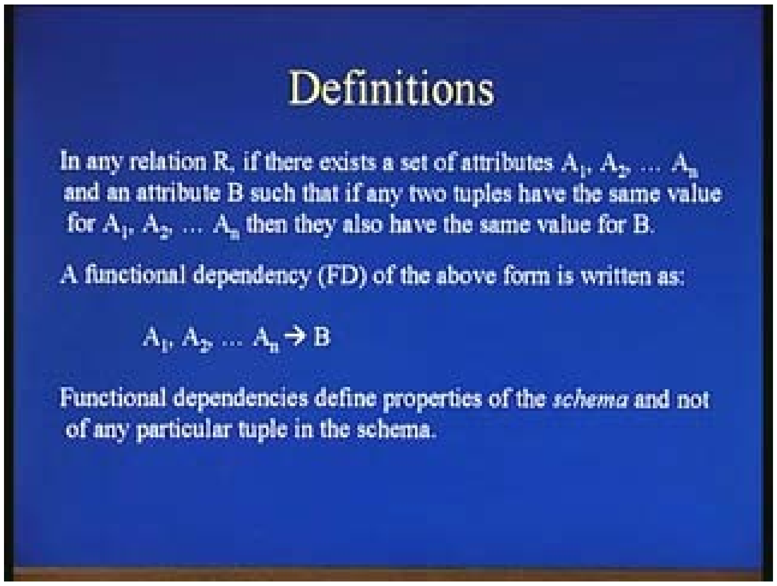


## Functional Dependencies

- Framework for systematic design and optimization of relational schemas
- Generalization over the notion of Keys
- Crucial in obtaining correct *normalized* schemas

However functional dependencies are some kinds of techniques that are database independent and can be used in a formal fashion in order to achieve some level of optimization or some level of efficiency in terms of database design. We are going to see how we can define keys. We have looked at the notion of super keys, primary keys and so on. How we can define the notion of keys using functional dependencies? In fact functional dependencies are more generalization or is a generalization over the concept of keys and they are crucial in obtaining the correct normalized schemas when we are trying to design database systems.

(Refer Slide Time: 00:05:33)



## Definitions

In any relation R, if there exists a set of attributes $A_1, A_2, \dots A_n$ and an attribute B such that if any two tuples have the same value for $A_1, A_2, \dots A_n$ then they also have the same value for B.

A functional dependency (FD) of the above form is written as:

$$A_1, A_2, \dots A_n \rightarrow B$$

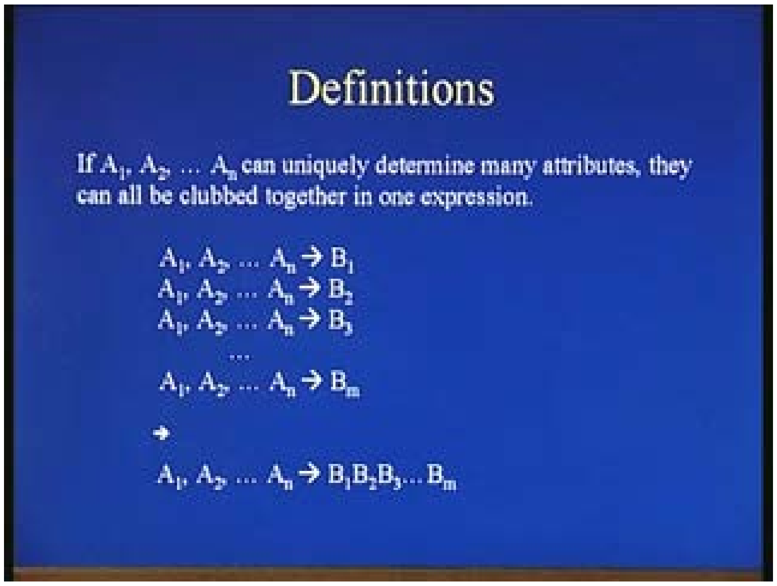
Functional dependencies define properties of the *schema* and not of any particular tuple in the schema.

So let us first define the notion of a functional dependency. Let me first take the definition here, the formal definition here and try to explain it in formal terms. The definition reads in any relation R, if there exists a set of attributes $A_1$ to $A_n$ and an attribute B such that if any tuples in the relation have the same value for $A_1$ to $A_n$ then they also have the same value for B. So essentially what it means? This is a formal way of saying that these set of attributes $A_1$ to $A_n$ uniquely determine the value of B. That means if two or more tuples have the same value of $A_1$ to $A_n$, it’s not possible essentially that means to say that is they should have the same value of B. They can't have different values of B because these sets of attributes $A_1$ to $A_n$ uniquely determine the value of B.

So this is called a functional dependency and a functional dependences is written as shown in the slide, here it is written as $A_1$ $A_2$ extra $A_n$ and there is a right arrow to B. So an important thing to note here is that functional dependencies define properties of the schema and not of the tuples in the schema. What is that mean? Essentially this kind of dependency that exists between $A_1$ to $A_n$ and this attribute B should be a property of the entire set of tuples that are there in the schema. For example I might say that the employee number uniquely determines the name of the employee in any relational database schema.

However we cannot say that let us say the employee name uniquely determines the age of an employee because there could be two or more employees having the same name but different age. But it could well be possible that there is certain employee with very rare name and there is nobody else who has that name and therefore for this particular employee, the name uniquely determines the age, uniquely determines his or her age. But this is not a property of the schema in itself, this is a property of that particular tuple of this particular employee who has this rare name so that there is nobody else with that name therefore we can uniquely identify the person and the age of that person. So when we are talking about functional dependencies, we are talking about properties of the schema which holds for all relations or all tuples in the schema.

(Refer Slide Time: 00:08:21)



## Definitions

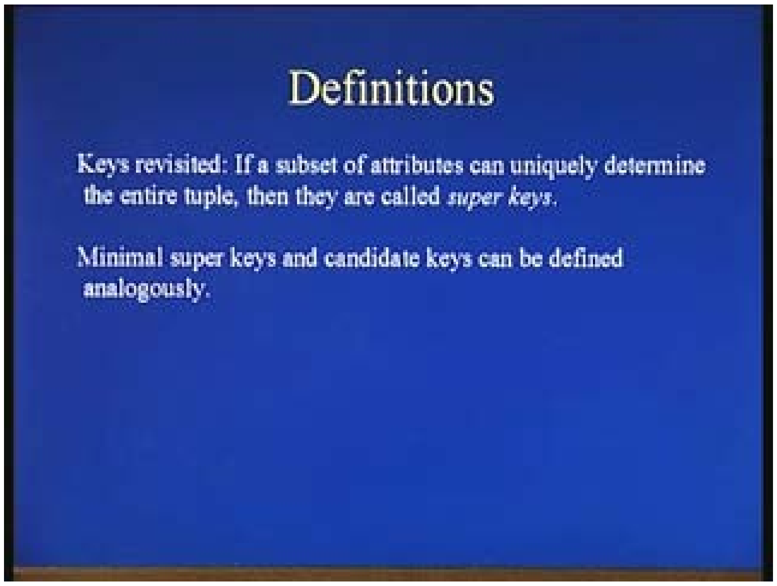
If $A_1, A_2, \ldots A_n$ can uniquely determine many attributes, they can all be clubbed together in one expression.

$$A_1, A_2, \ldots A_n \rightarrow B_1$$
$$A_1, A_2, \ldots A_n \rightarrow B_2$$
$$A_1, A_2, \ldots A_n \rightarrow B_3$$
$$\ldots$$
$$A_1, A_2, \ldots A_n \rightarrow B_m$$
$$\Rightarrow$$
$$A_1, A_2, \ldots A_n \rightarrow B_1B_2B_3\ldots B_m$$

In any given schema there could be many functional dependencies, let us say the set of attributes $A_1$ to $A_n$ can uniquely define, it's also called uniquely determine B or defines B or so on. So $A_1$ to $A_n$ can uniquely determine $B_1$ or $B_2$ or $B_3$ until $B_m$ so this is written as shown in the slide here that is $A_1$ to $A_n$ and right arrow $B_1$ to $B_m$. Now why is this called a functional dependencies, what is so functional about functional dependency? If you are familiar with the mathematical notion of a function which is a special kind of mapping or a special kind of relation between sets, you can see here that the notation that is used for example $A_1$ to $A_n$ or uniquely determining some attribute B is precisely the notation used for a function.

It basically determines because $A_1$ to $A_n$ uniquely determines a particular attribute B, it acts as a mapping or it act as a function which maps this set of attributes $A_1$ to $A_n$ uniquely to each B that it defines. This precisely acts as the mathematical notion of a function. However there need not be any computation that determines this unique definition. For example the employee number may uniquely determine name, however just given the employee number I mean I won't be able to compute, I won't be able to determine what is the name. It's just that each employee number has a different or identifies an employee with a different name or identifies uniquely each employee. However, it is not possible to determine what is the name of the employee given just the employee number, we still have to access a database to do that.

(Refer Slide Time: 00:10:24)



## Definitions

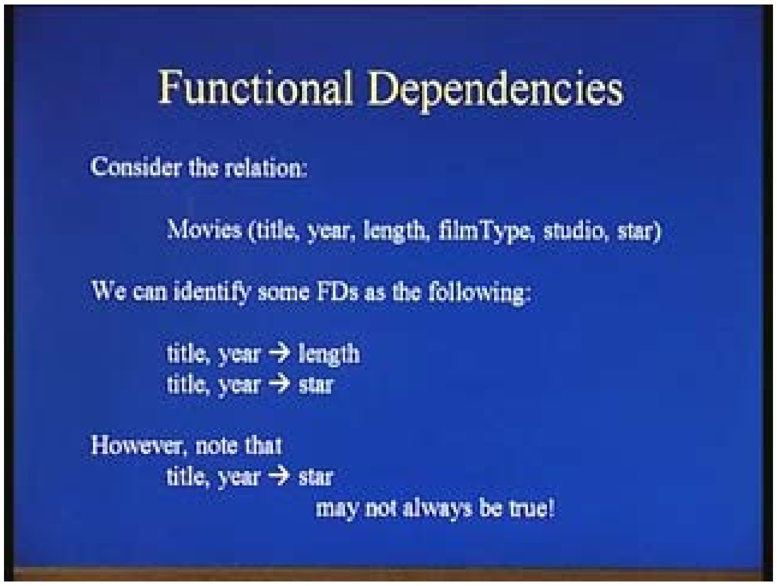
Keys revisited: If a subset of attributes can uniquely determine the entire tuple, then they are called *super keys*.

Minimal super keys and candidate keys can be defined analogously.

So let us revisit the notion of keys now when we say functional dependencies. As you might have noticed the notion of functional dependency is actually a generalization over the notion of keys. That is in the previous slide here if $A_1$ to $A_n$ uniquely determine the set of all attributes which form the relation let us say $A_1$ to $A_n$ combined with $B_1$ to $B_m$ forms the entire relation. Then we see that $A_1$ to $A_n$ is nothing but a key for this relation that is for the entire relation comprising of $A_1$ to $A_n$ and $B_1$ to $B_m$ and similarly minimal super keys or candidate keys can be defined analogously.

Let us take some examples now. Consider a relation of the form that is shown in the slide here. The slide shows a relation called movies which has the following set of attributes, it has title, year, length, film type, studio and star. So, going by the standard definitions of what is a movie and what is title and what is the year of its release, the length of the movie and so on. We can from common sense reasoning, identify some kinds of functional dependencies.

(Refer Slide Time: 00:11:11)



## Functional Dependencies

Consider the relation:

Movies (title, year, length, filmType, studio, star)

We can identify some FDs as the following:

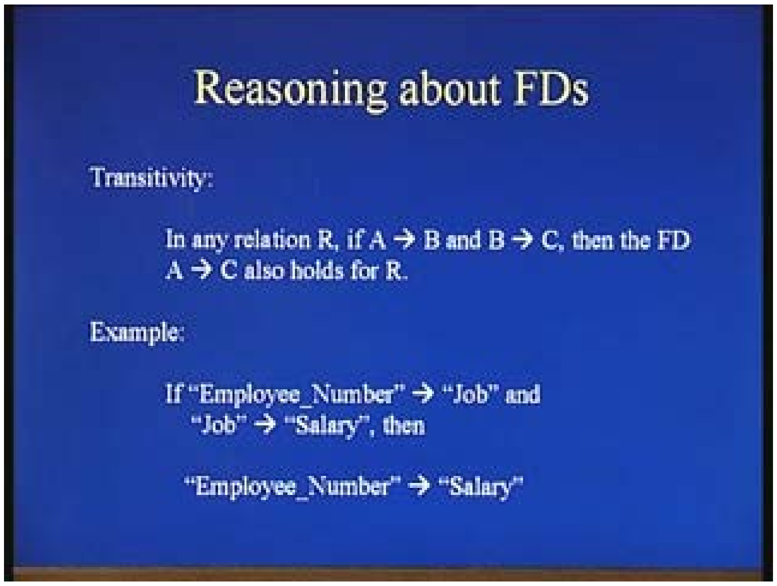
title, year → length
title, year → star

However, note that
title, year → star
may not always be true!

For example given the title of a movie and the year in which it was released, I can probably uniquely determine what is the length of that movie and similarly given there is a bug in the second slide here, second functional dependency here. Given a title and the year of release of a particular movie, I can probably determine what is the film type. Given a movie of a particular type, of a particular name and the year in which it was released, I can probably say well this is an art movie, this is a commercial movie, this is social documentary, this is a comedy tragedy whatever.

Similarly I can probably also say title, year uniquely determines say film studio. However title, year may not uniquely determine the star that is the actors who star in that movie because there could be more than one actor who have stared in that movie and you cannot uniquely determine given the title and the year you cannot uniquely say well this is the movie star who acted in that movie because it need not be complete, there could be other stars as well.

(Refer Slide Time: 00:13:03)



## Reasoning about FDs

Transitivity:

In any relation R, if A → B and B → C, then the FD A → C also holds for R.

Example:

If "Employee_Number" → "Job" and "Job" → "Salary", then

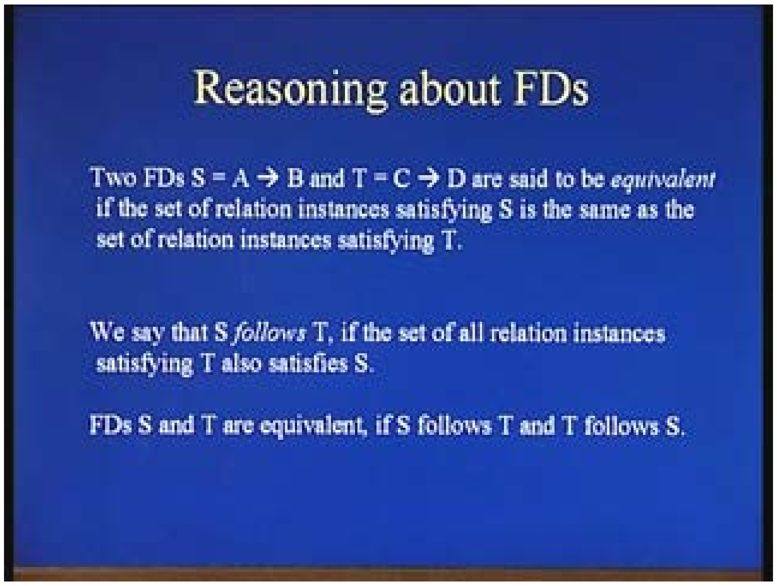
"Employee_Number" → "Salary"

So let us see what kinds of properties we can identify about functional dependencies and what we can do with these properties. Note that if in any given relation R, if I have a functional dependency of the form A to B where A is a set of attributes. Now I am using the term A that is without a subscript to determine actually a set of attributes. So if the set of attributes A defines a set of attributes B and the set of attributes B uniquely define another set of attribute C then we can easily identify the functional dependency saying A also defines C. So there is an example here, if the employee number is given in such a way that it also defines the job of the employee.

Let us say all employees who do supplies work are given numbers between 100 to 200 and all employees who do administrative works or a clerical job is given numbers between 201 to 300 and so on. So employee number suppose uniquely defines job and let us say job also uniquely define salary that is every employee of a particular job has the same salary. Then we can easily say that the employee number actually defines salary that is given just the employee number, I will be able to determine what is the salary for that employee.

Some more definitions. Suppose we have two sets of functional dependencies A and B. Now again A and B are sets here and we have two sets of functional dependencies A and B and that is A defining B and C defining D. Now these functional dependencies are said to be equivalent, if the set of all relations that satisfy the first functional dependency is the same as the set of all relations satisfying the second functional dependencies. Similarly a small generalization over this definition, we say that a functional dependency S follows another FD called T, if the set of all relation instances satisfying T also satisfies S.

(Refer Slide Time: 00:14:44)



## Reasoning about FDs

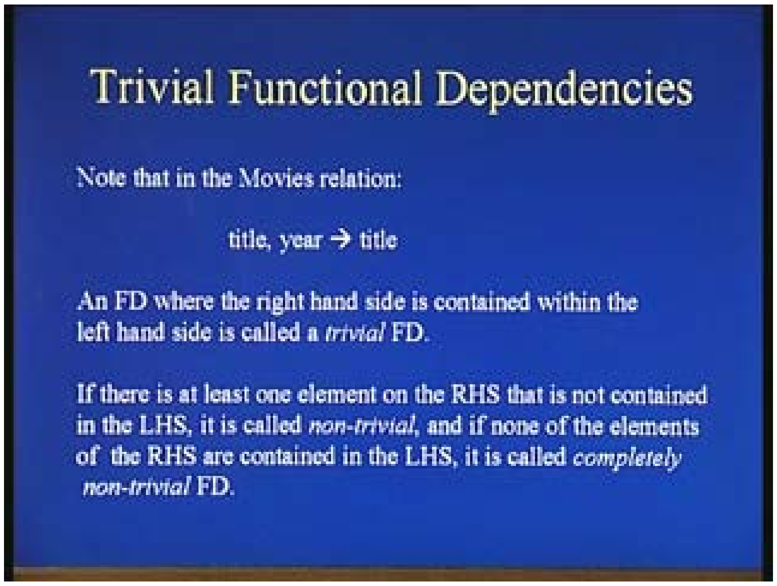
Two FDs S = A → B and T = C → D are said to be *equivalent* if the set of relation instances satisfying S is the same as the set of relation instances satisfying T.

We say that S *follows* T, if the set of all relation instances satisfying T also satisfies S.

FDs S and T are equivalent, if S follows T and T follows S.

So we will take up examples here, the follows property of a functional dependency is extremely important by which we will be able to compute what is called as the closure of a functional dependencies. That is suppose I give some kind of functional dependencies saying A defines B. What else can I say about that? What else can I infer from this functional dependencies is what we are going to take up. Before we move on to inferring functional dependencies, we need to complete a few more definitions. The next definition that we are going to take up is the notion of a trivial functional dependency. Now a trivial functional dependency may look of course trivial but it is quiet important when we are trying to define let us say normal forms.

(Refer Slide Time: 00:16:07)



## Trivial Functional Dependencies

Note that in the Movies relation:

title, year → title

An FD where the right hand side is contained within the left hand side is called a *trivial* FD.

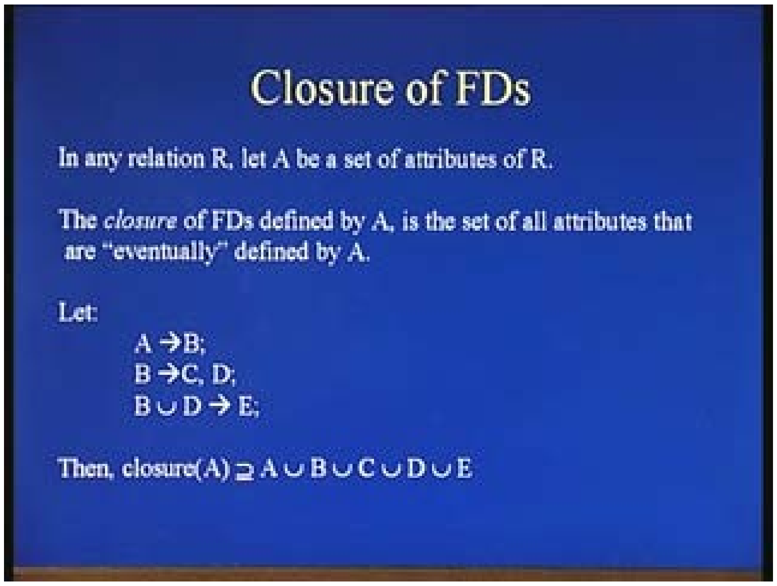
If there is at least one element on the RHS that is not contained in the LHS, it is called *non-trivial*, and if none of the elements of the RHS are contained in the LHS, it is called *completely non-trivial* FD.

Now have a look at the example in the slide here. This slide shows a functional dependency of the form title, year defines title which is obviously true. Every attributes defines itself that is every attribute uniquely identifies itself, therefore any functional dependency in which the right hand side is contained within the left hand side. That is the antecedent or the right hand side of a rule is a subset of the left hand side of a rule then it is said to be a trivial functional dependency.

If there is at least one element in the right hand side which is not part of the left hand side then it is called a non-trivial functional dependency and usually we are interested in what are called as completely non-trivial functional dependencies which says that none of the elements in the right hand side belong to the left hand side. We will see that this notion of triviality is not so simple because essentially when there are circular dependencies, it becomes quiet tricky to handle functional dependencies which are removed triviality from functional dependencies.

We now come to an important aspect of functional dependencies, the notion of closure of FDs. What do we intuitively understand by the term closure? If we are familiar with the notion of closure and say discrete mathematics, you would probably recall the definition that the closure we say that a particular algebra is closed when I say, when I take certain elements from or certain operands from a set that defines the universe and then perform an operator that performs some kind of a function on these operands and the result of this function also goes back into this set.

(Refer Slide Time: 00:17:42)



## Closure of FDs

In any relation R, let A be a set of attributes of R.

The *closure* of FDs defined by A, is the set of all attributes that are "eventually" defined by A.

Let:

$A \rightarrow B;$
$B \rightarrow C, D;$
$B \cup D \rightarrow E;$

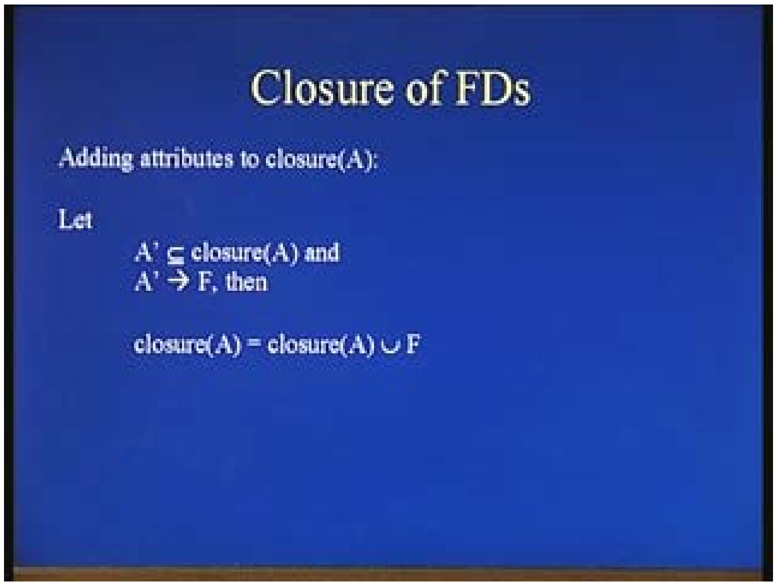
Then, closure(A) $\supseteq A \cup B \cup C \cup D \cup E$

For example if I take two integers and add them, the result is again an integer. If I take two integers and subtract them, the result is again an integer. Now this is what is called as closure that is I take certain operands or take certain elements from a universe, perform a function on them and the result of that function also belongs to the same universe. So let us define the function, the notion of closure on functional dependencies in an analogies fashion. The closure of functional dependencies defined by A that is A is a set of attributes in any given relation R is the set of all attributes that are eventually defined by A. That is how we can eventually define or uniquely determine what all attributes can be uniquely determined eventually by A.

For example let in a given relation, let the set of attributes A uniquely define a set of attributes B. Similarly let the set of attributes B uniquely define two other sets of attributes C and D. Now let the combined sets of attributes B and D uniquely determine another set of attributes called C. Then we say that the closure of A contains all of these attributes that is A union B union C union D union E that is all of that and actually because all of these attributes can be eventually defined from A. From A we can eventually define B, from B we can uniquely define D and combining B and D we can uniquely define E. So all of these attributes contained in all of these sets can be eventually uniquely defined by A.

What about elements that lie within A, can they be uniquely define by A? The answer is yes. This is because of the trivial property of functional dependencies that is every attribute in A uniquely defines itself which is a triviality rule. however closure also just A union B union C union D union E that is shown in the slide does not complete the closure of A.

(Refer Slide Time: 00:20:52)



## Closure of FDs

Adding attributes to closure(A):

Let

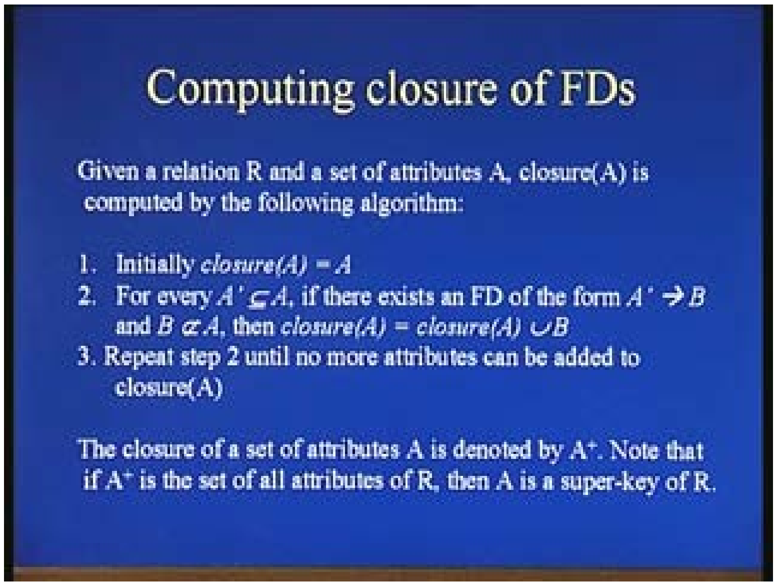
A' ⊆ closure(A) and

A' → F, then

closure(A) = closure(A) ∪ F

If for example there is a subset of A or subset of any element of closure of A and if that subset of attributes defines some other sets of attributes F such that F is not part of the closure of A then we also add F to the closure of A.

(Refer Slide Time: 00:21:13)



## Computing closure of FDs

Given a relation R and a set of attributes A, closure(A) is computed by the following algorithm:

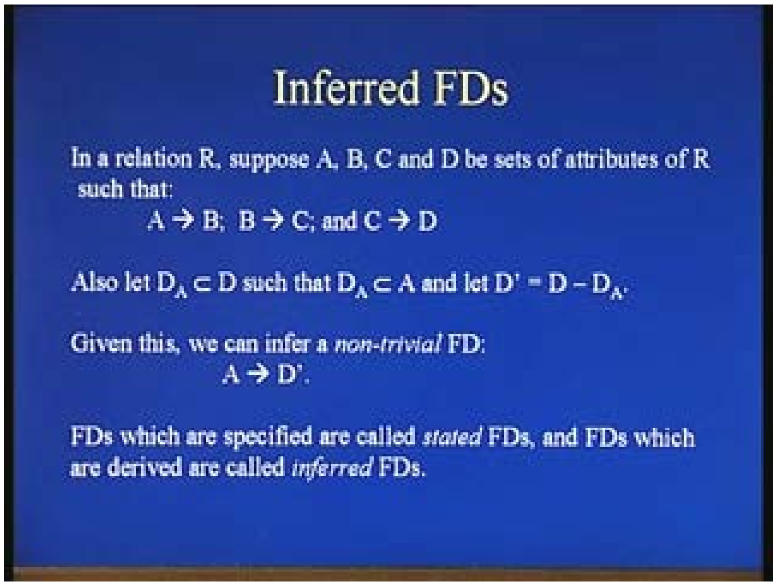
1. Initially *closure(A) = A*
2. For every $A' \subseteq A$, if there exists an FD of the form $A' \rightarrow B$ and $B \not\subset A$, then $closure(A) = closure(A) \cup B$
3. Repeat step 2 until no more attributes can be added to closure(A)

The closure of a set of attributes A is denoted by $A^+$. Note that if $A^+$ is the set of all attributes of R, then A is a super-key of R.

So, let us first see look at an algorithm by which we can compute the closure of a functional dependencies then we look at a few examples of closures. So given a relation R and a set of attributes A, how do we compute closure of A? So initially we start with the set saying the closure of A equal to A that is equal to all elements of A. This is true because of the trivial functional dependencies, every element of A trivially defines itself. Now for every A prime that is a subset of A, if there exists a functional dependency of the form A prime defines B and B does not belong to A that is B is a set of attributes which does not belong to A as of now then just add B to the set of elements in the closure of A and repeat step two until no more attributes can be added to closure of A.

So the closure of a set of attributes A is also denoted by A with a superscript of plus that is A plus and also note that suppose we run this algorithm and A plus contains the set of all attributes of R, what can we say about the set of elements in A? A is of course a super key of R because using A, we can eventually uniquely define all elements of R. So we now come to the next property of functional dependencies, the notion of inferring functional dependencies or following that is how do we find out functional dependencies which follow from one another.

(Refer Slide Time: 00:22:53)



## Inferred FDs

In a relation R, suppose A, B, C and D be sets of attributes of R such that:

$A \rightarrow B$; $B \rightarrow C$; and $C \rightarrow D$

Also let $D_A \subset D$ such that $D_A \subset A$ and let $D' = D - D_A$.

Given this, we can infer a *non-trivial* FD:

$A \rightarrow D'$.

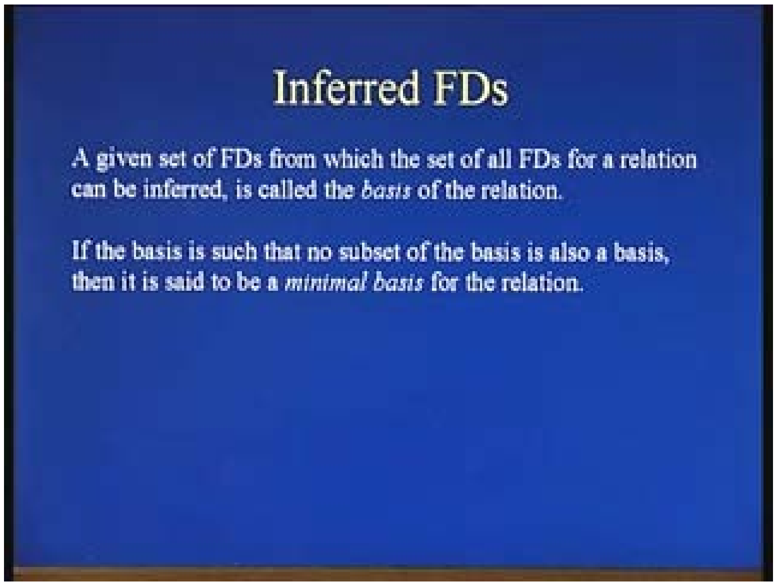
FDs which are specified are called *stated* FDs, and FDs which are derived are called *inferred* FDs.

Given a set of functional dependencies and given no more knowledge about the domain in which a relation exists, what can we infer about any more functional dependencies that exist in R. So, we have already seen one rule for inferring which is a transitivity rule which we will revisit again now and before that note another rule that is shown here.

Suppose A B C and D are sets of attributes of R such that the following functional dependencies exist that is A uniquely defines B, B defines C and C defines D. Now based on the closure property and transitivity, we can easily say that A defines D. However there could be some elements of D which are actually contained in A that is D need not be completely disjoined from A. So let D subscript A that is D A be the set of all attributes in D such that they belong to A that is they are actually a subset of A.

Now take away these attributes that is let D prime be defined as D minus D A that is shown in the slide here. Once we do this, we see that we have actually inferred a non-trivial functional dependency. That is A defines D prime that is we started by saying that because of the transitivity rule A defines D. However there is some triviality that is there in this definition and once we remove that triviality, we have actually encountered or we have actually inferred a new functional dependencies which is non-trivial that is which is of the form A defines D prime. So functional dependencies which are specified or which are given to begin with are called stated functional dependencies and FDs which are derived are called inferred functional dependencies.

(Refer Slide Time: 00:25:18)



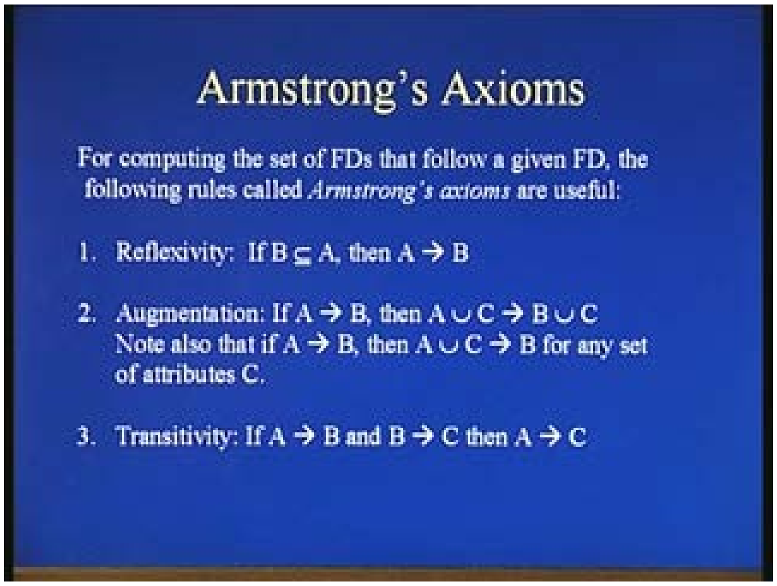
## Inferred FDs

A given set of FDs from which the set of all FDs for a relation can be inferred, is called the *basis* of the relation.

If the basis is such that no subset of the basis is also a basis, then it is said to be a *minimal basis* for the relation.

Now, given a set of relation, suppose we have a set of functional dependencies that are stated. This set of functional dependencies is called the bases of the relation. For example in the movie relation that we saw where a movie is defined by title, year, film type, star and so on. The functional dependency is the title and year define the length of a movie was given. Similarly the title and year define the film type was also given, so all these functional dependencies that are given are called the basis of the relation. Now if the basis of functional dependencies are such that no subset of this basis is also a basis. That is none of these functional dependencies that form the basis can be derived from one another then it is said to be a minimal basis for the relation.

(Refer Slide Time: 00:26:18)



## Armstrong's Axioms

For computing the set of FDs that follow a given FD, the following rules called *Armstrong's axioms* are useful:

1. Reflexivity: If $B \subseteq A$, then $A \rightarrow B$
2. Augmentation: If $A \rightarrow B$, then $A \cup C \rightarrow B \cup C$
   Note also that if $A \rightarrow B$, then $A \cup C \rightarrow B$ for any set of attributes C.
3. Transitivity: If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

So how do we go about inferring functional dependencies that is computing the closure or getting inferred functional dependencies and so on. How do we go about inferring functional dependencies, given a basis that is given a set of functional dependencies. For this, there are set of axioms or set of rules that define how functional dependencies behave. These are called Armstrong's axiom which are quiet useful when we are talking about properties of functional dependencies.

The first property is the notion of reflexivity which is like saying, which is easily obvious based on the triviality rule that is if set of attributes B is a subset of the attributes A then A functionally defines B. For example, theater and year or rather the title and year of a movie functionally defines title. That is if I have a set of attributes, these sets of attributes functionally define every possible subset of this attributes.

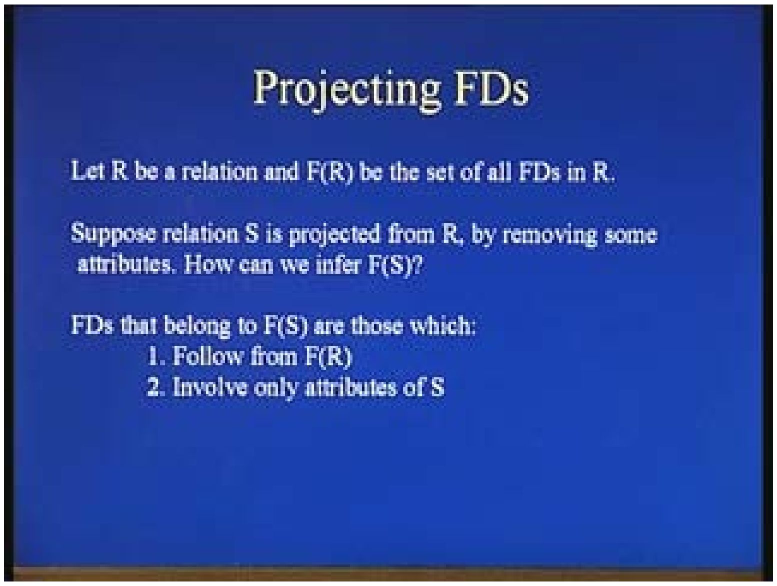
Similarly, the second role is that of argumentation that is if I know that there exists a functional dependency of the form A defines B that is theatre, year defines length. Then I can add a set of attributes to both sides of the functional dependencies without violating the dependency. For example I can say that if title and year of a movie determines the length of the movie then I can add an attribute called star or studio to both sides of the functional dependencies without altering the semantics. That is I can say title, year and studio of a movie uniquely determines the studio and the length of the movie.

Also note that I need not add studio to the right hand side because it forms a triviality rule that is there is some triviality that entails from adding the same attribute in both sides. I can as well add, I can just add a new attribute just to the left hand side and still the functional dependency holds. Take the same example again. Suppose I say title and year uniquely determines the length of a movie. Then I can as well say that the title of the movie, the year of the movie and the studio in which the movie was shot uniquely determines the length of the movie which is fine because adding new information to some set of attributes that uniquely determine something does not alter the dependency.

Similarly the last rule is that of transitivity which we have already seen that is if A uniquely defines B and B uniquely defines C then we can infer that A defines C. The next concept that we learn here is the notion of projecting functional dependencies. What happens if I compute a project operation on a relation. You know what a project operation is in standard relational algebra.

A project operation takes certain columns or certain attributes of a relation and produces a new relation out of these attributes that means it actually throws away certain attributes from the original relation. Now suppose let R be a relation and F of R be the set of all functional dependencies in R. Now suppose I project another relation S from this new relation R by throwing away certain attributes. Now what can we say about F of S? What kinds of functional dependencies exist in F of S?

(Refer Slide Time: 00:29:23)



## Projecting FDs

Let R be a relation and F(R) be the set of all FDs in R.

Suppose relation S is projected from R, by removing some attributes. How can we infer F(S)?

FDs that belong to F(S) are those which:

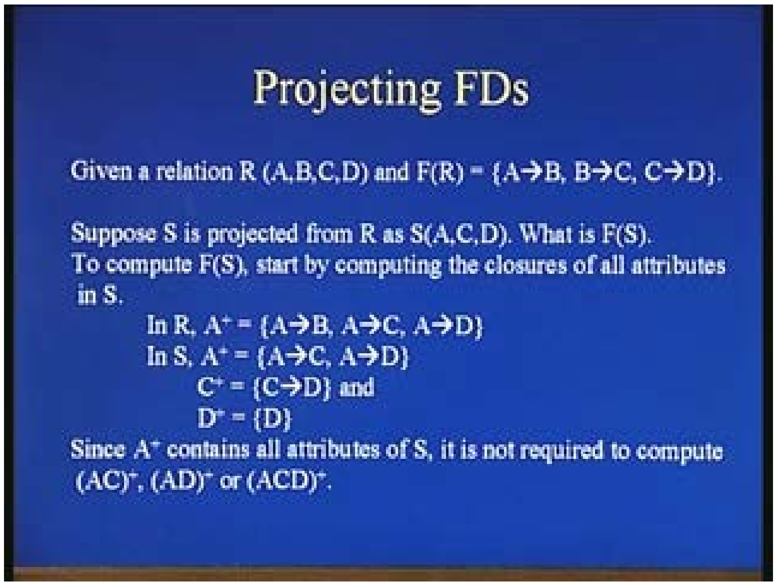
1. Follow from F(R)
2. Involve only attributes of S

Obviously for example if I say that from the movie relation, if I throw away the year attribute of this relation then obviously I cannot have the functional dependencies which says title, year uniquely determines length because the year doesn’t exist in the new relation at all. Therefore at this point, at the first step we can say that functional dependencies in the new relation S should satisfy the following properties. What are these properties?

Firstly they should follow from the functional dependencies in R that is we should be able to infer them from the functional dependencies that existed in the earlier relation. Secondly and, which is also quite obvious that they should involve only attributes of S. Obviously they cannot involve attributes of R which don't exist in S. A functional dependency only has to involve attributes that exist in the current relation.

So how do we compute the functional dependencies on a projection? Now consider this example rather than going through a formal set of rules. Let me explain it by an example. Now given a relation R containing the following attributes A B C and D and let us say the following functional dependencies that is A defines B, B defines C and C defines D.

(Refer Slide Time: 00:31:24)



## Projecting FDs

Given a relation R (A,B,C,D) and F(R) = {A→B, B→C, C→D}.

Suppose S is projected from R as S(A,C,D). What is F(S).
To compute F(S), start by computing the closures of all attributes in S.

In R, $A^+$ = {A→B, A→C, A→D}
In S, $A^+$ = {A→C, A→D}
$C^+$ = {C→D} and
$D^+$ = {D}

Since $A^+$ contains all attributes of S, it is not required to compute $(AC)^+$, $(AD)^+$ or $(ACD)^+$.

Now suppose S is projected from R and B is thrown away from R in getting S. So S has just three attributes A C and D. Now what should be F of S? That is what should be the set of functional dependencies that lie in S. Obviously we cannot have A defines B because B doesn’t exist at all in this relation and we also cannot have B define C because again B doesn’t exists in this relation. So but can we say anything else about these dependencies.

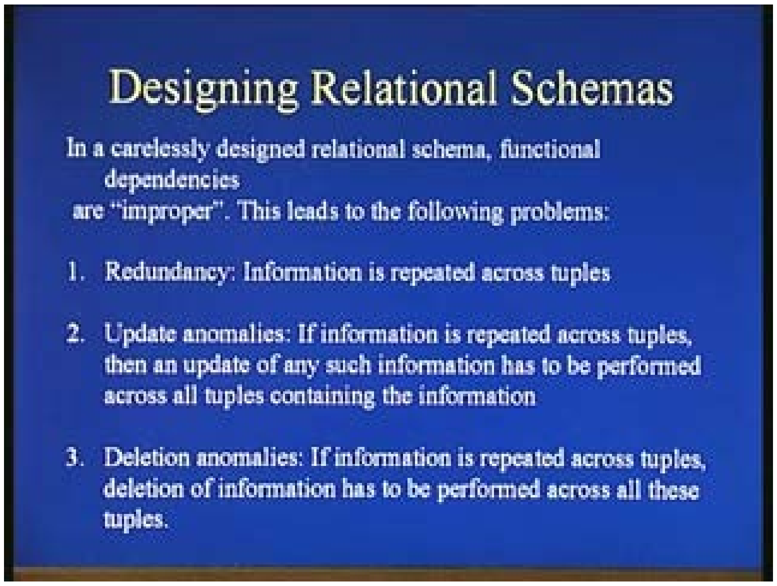
Now going back to R, let us see what happens when we compute the closure of A. Let us compute the closures of each attributes. So, I have not included the trivial functional dependencies in computing the closures that is firstly we start from A defines A obviously then we have a given rule which says A define B, a basis rule from which we can infer that A define C because there is another basis rule which says B define C and from which we can still infer A defines D because there is another basis rule called C defines D.

So in R, the closure of A is A defines B, A defines C and A defines D. In S, all we need to do is take away all these or throw away all this functional dependencies that contain attributes which do not exist in S. So here A defines B is one such attribute, is one such functional dependencies. It contains B which does not which is not part of S. Therefore the set of, the closure of A in S is essentially A defines C and A defines D or A C D.

Similarly the closure of C in S is C defines D that is C, D in this case and the closure of D is just D. Now since the closure of A contains all attributes of S that is A C and D, we don't need to compute anymore closures that is we don't need to compute the closures of A C or A D or A C D because we already have all attributes of A, we already know how to uniquely determine each attributes of A.

Therefore the functional dependencies that exist in S are A defines C, A defines D and C defines D and of course there are trivial functional dependencies like D defines D and C C defines C and A defines A which are not shown as part of this set here. So a simple way to compute the closure or compute functional dependencies in a projection is to first compute the closure of the set of all functional dependencies in the original relation and then throw away everything which contains attributes that do not belong to the new relation.

(Refer Slide Time: 00:35:10)



## Designing Relational Schemas

In a carelessly designed relational schema, functional dependencies are "improper". This leads to the following problems:

1. Redundancy: Information is repeated across tuples
2. Update anomalies: If information is repeated across tuples, then an update of any such information has to be performed across all tuples containing the information
3. Deletion anomalies: If information is repeated across tuples, deletion of information has to be performed across all these tuples.

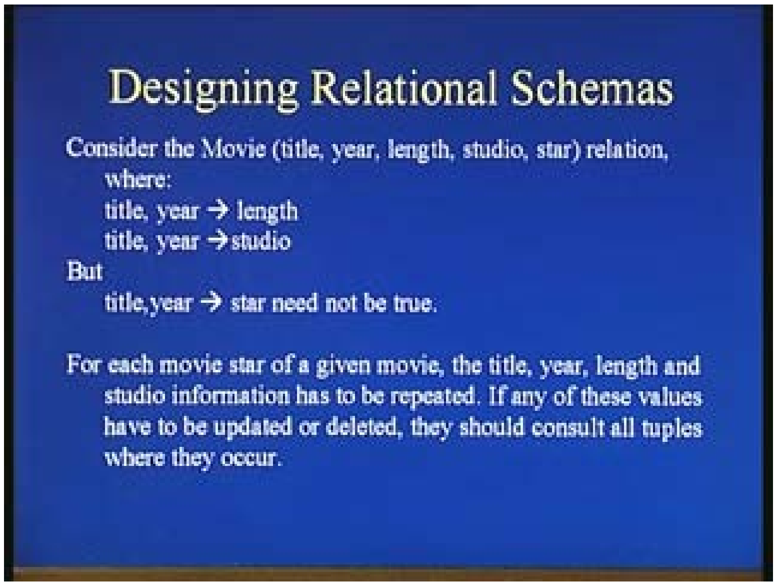
So what is a use of all this functional dependencies and closure and transitivity and axioms and so on. These sets of underpinnings are used for normalizing or optimizing relational schemas for better performance. What do we mean by optimizing or what are we optimizing against, what is the property that we are trying to remove, what is the undesirable property that we are trying to remove. This undesirable property is the property of redundancies in relational schemas. If a relational schema is badly designed then it is going to contain several redundant information which results in a number of anomalies.

What are these kinds of anomalies and what kinds of redundancies are these? Let us have a look at them. Redundancy of course means that some kind of information is repeated across different tuples. The same information saying the title is this one or the year is this one or the star is that one and so on.

So if the same information is repeated across several different tuples then we encounter two kinds of anomalies. What are these anomalies? The first anomaly is that of updation. Suppose I need to update a data element using let us say the update clause of SQL. I cannot update in just one tuple, especially if this particular information is repeated across several different tuples. If the title of a movie is repeated across several different tuples in the database and later on I see that the title of the movie is entered incorrectly, there is a spelling mistake. So I need to change the title of the movie in every tuple there in which it occurs.

Similarly, the notion of deletion anomalies. Suppose I delete a tuple which suppose I need to delete a tuple about a particular movie, given a particular title. I need to delete all tuples which contain this title as its attributes. So I need to essentially search the entire database and delete it in several different places, otherwise it again creates anomalies. So how do we design relational schema, so that we can remove redundancies and remove these kinds of anomalies.

(Refer Slide Time: 00:37:37)



## Designing Relational Schemas

Consider the Movie (title, year, length, studio, star) relation, where:

title, year → length

title, year → studio

But

title,year → star need not be true.

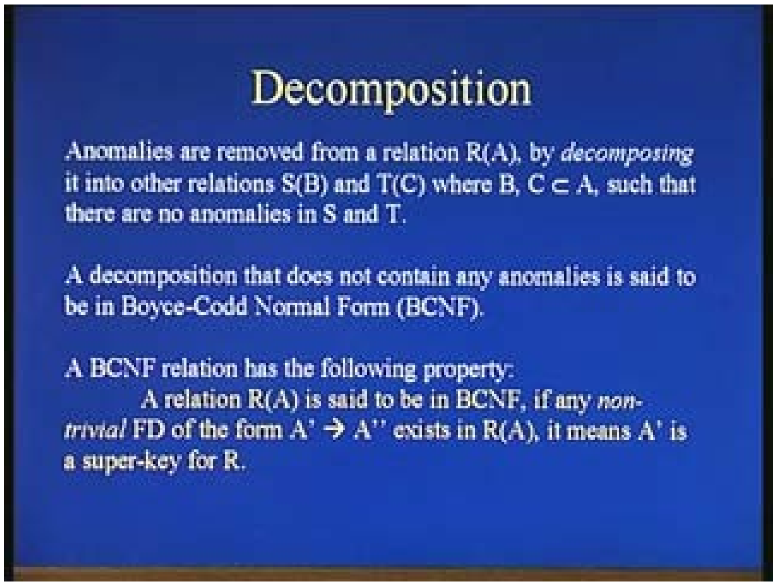
For each movie star of a given movie, the title, year, length and studio information has to be repeated. If any of these values have to be updated or deleted, they should consult all tuples where they occur.

Now have a look at the movie relation once again. The movie relation has the following attributes that is title, year, length, studio and star and we also have the functional dependencies which says title and year uniquely determine length and title and year of a movie uniquely determine the studio in which the movie was shot. However title and year does not uniquely determines star because there could be more than one actor who have stared in the movie.

Therefore for every actor who has stared in this particular movie, you have to repeat the information of length and studio across these tuples. Let us say some movie has two or three different actor say Shahrukh Khan, Hrithik Roshan or whatever. Now in a given movie let us say some movie X Y Z that is shot in the same year in the same studio and has the same length, it’s only the last field the star which changes from say Shahrukh Khan to Hrithik Roshan and whatever.

So for each actor who was stared in the movie, I need to repeat all the other information that forms part of this tuple which is what is the notion of redundancy. And suppose after doing all this, I find that I have entered the length of this movie incorrectly and I need to change the length of the movie, I need to change it in all of these different tuples where this is stored and similarly the problem of deletion.

(Refer Slide Time: 00:39:22)



## Decomposition

Anomalies are removed from a relation R(A), by *decomposing* it into other relations S(B) and T(C) where B, C $\subset$ A, such that there are no anomalies in S and T.

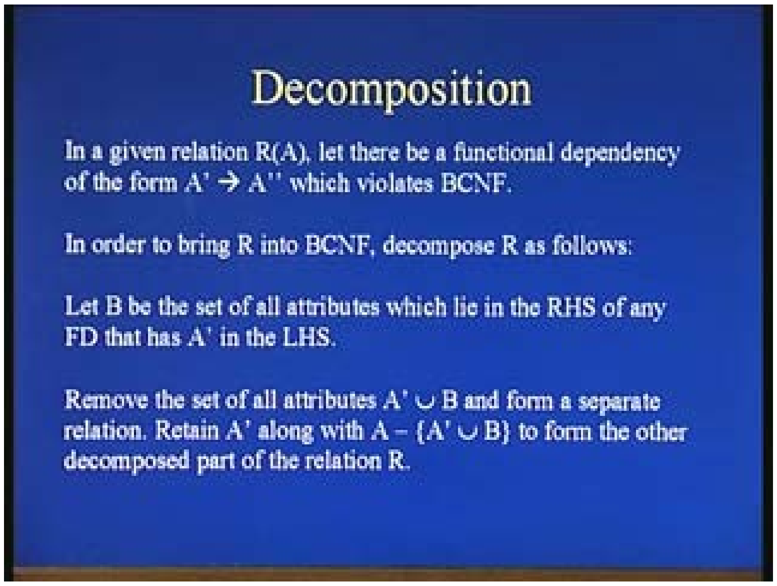
A decomposition that does not contain any anomalies is said to be in Boyce-Codd Normal Form (BCNF).

A BCNF relation has the following property:
A relation R(A) is said to be in BCNF, if any *non-trivial* FD of the form A' $\rightarrow$ A'' exists in R(A), it means A' is a super-key for R.

So anomalies are removed from a relation by the process of decomposition. Now what is meant by decomposition? Given a relation R containing a set of attributes A, you decompose the set of relations into two different relations S and T such that the sets of attributes in of S and T are a subsets of the attributes of A and there are no anomalies in S and T. Now a decomposition that does not contain any anomalies is said to be in what is called as Boyce-Codd Normal Form or it's also abbreviated as BCNF, so a BCNF has the following property.

Suppose given a relation R with the set of attributes A, it is said to be in BCNF if there is a non-trivial. Note the emphasis on the word non-trivial here, if there is any non-trivial functional dependency of the form A prime defines A double prime that means A double prime is not a subset of A prime. So if such a non-trivial functional dependency exists then it means that A prime is a super key of R. That is there is no functional dependency of the form A prime defines A double prime in which A prime is not the key. If that is the case then the relation is not said to be in BCNF. So we will take up examples of this once we complete the notion of decomposition into BCNF and when it becomes more clearer.

(Refer Slide Time: 00:40:51)



## Decomposition

In a given relation R(A), let there be a functional dependency of the form A' → A'' which violates BCNF.

In order to bring R into BCNF, decompose R as follows:

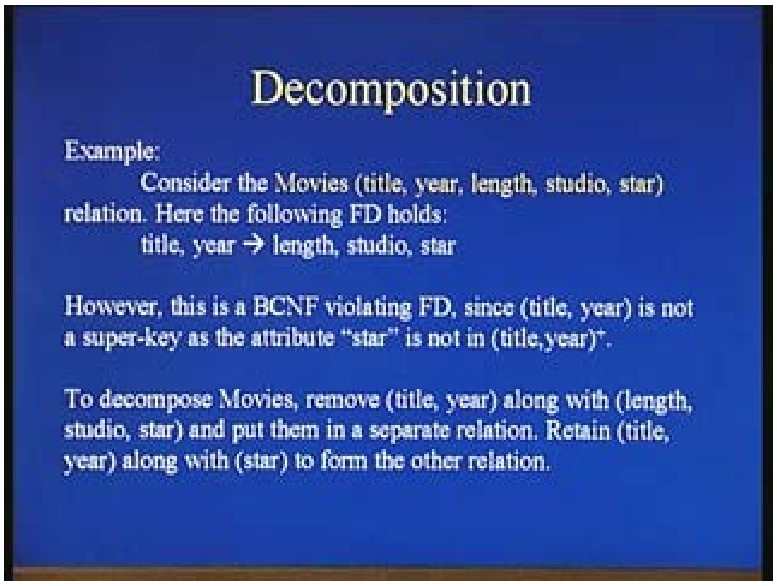
Let B be the set of all attributes which lie in the RHS of any FD that has A' in the LHS.

Remove the set of all attributes A' ∪ B and form a separate relation. Retain A' along with A – {A' ∪ B} to form the other decomposed part of the relation R.

Now how do we decompose a relation such that it forms or it becomes comply into BCNF. Now suppose in a given relation R, let there be a functional dependency of the form A prime defines A double prime which violates BCNF. What do we meant by violating BCNF? It means that A prime defines A double prime is firstly non-trivial that is A double prime is not a subset of A prime and A prime is not the super key of R that is it is some other sets of attribute.

In order to bring R into BCNF, we decompose R as follows. First take the set of all attributes that are defined by A prime. Now A prime defines A double prime, it may define something else and so on. now let B be the set of all attributes that lie in the right hand side of any functional dependencies that are defined by A prime. Now remove the set of all attributes A prime along with B and form a separate relation and retain the remaining set of attributes along with A prime to form the other part of the decomposed relation R. Let us have an example which makes this very clear.

(Refer Slide Time: 00:42:22)



## Decomposition

Example:

Consider the Movies (title, year, length, studio, star) relation. Here the following FD holds:

title, year → length, studio, star

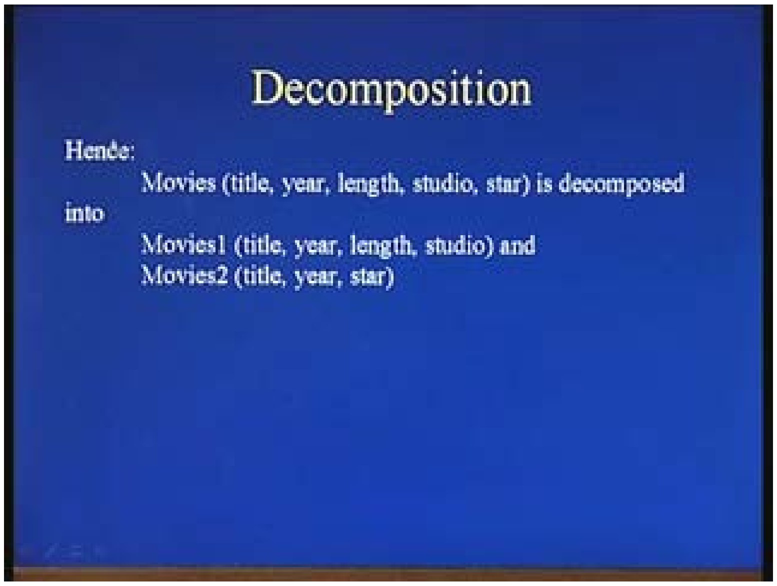
However, this is a BCNF violating FD, since (title, year) is not a super-key as the attribute "star" is not in (title,year)$^+$.

To decompose Movies, remove (title, year) along with (length, studio, star) and put them in a separate relation. Retain (title, year) along with (star) to form the other relation.

Consider the movies example once again. Let us have a look at this relation movies having the following attributes title, year, length, studio and star. Now here the following functional dependency holds that is title and year uniquely determines length. Title and year uniquely determines studio and title and year uniquely determines, it does not determine star actually. So this is actually a BCNF violating functional dependency because title and year obviously cannot be the key for this relation because it does not uniquely determine star. So title and year is not a super key as the star attribute is not in the closure of title and year.

So to decompose this relation, just remove title and year along with length and studio. So there is no star here, just remove title and year along with length and studio that it define and put them in a separate relation and retain title and year along with star to form the other relation. Therefore we get two kinds of relations that is movies is divided into title, year, length and studio because title and year define length and title and year define studio. So we have separated them from movies and whatever is left out that is star is combined with A prime which is title and year and retained as it is.

(Refer Slide Time: 00:43:33)



## Decomposition

Hence:

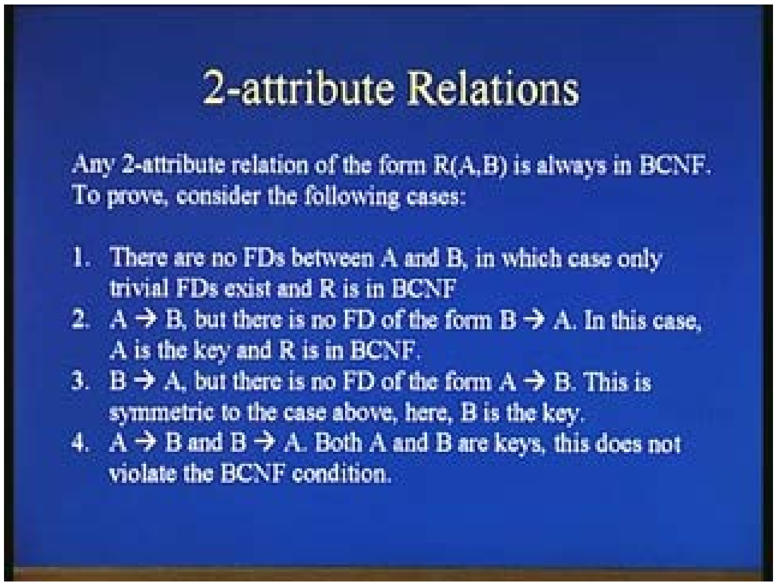
Movies (title, year, length, studio, star) is decomposed into

Movies1 (title, year, length, studio) and

Movies2 (title, year, star)

So movies one and movies two are decomposed forms of movies and it is also easy to verify that movies one and movies two are BCNF complaint.

(Refer Slide Time: 00:44:08)



## 2-attribute Relations

Any 2-attribute relation of the form R(A,B) is always in BCNF. To prove, consider the following cases:

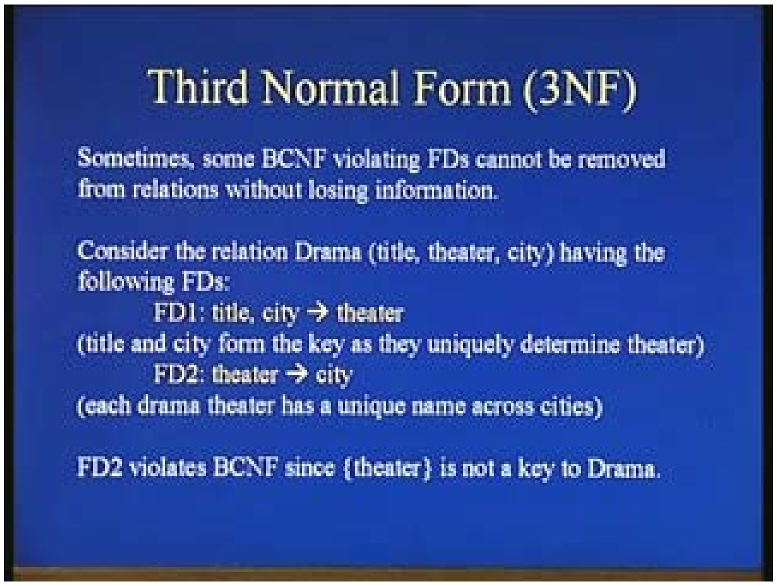
1. There are no FDs between A and B, in which case only trivial FDs exist and R is in BCNF
2. $A \rightarrow B$, but there is no FD of the form $B \rightarrow A$. In this case, A is the key and R is in BCNF.
3. $B \rightarrow A$, but there is no FD of the form $A \rightarrow B$. This is symmetric to the case above, here, B is the key.
4. $A \rightarrow B$ and $B \rightarrow A$. Both A and B are keys, this does not violate the BCNF condition.

There is one property of BCNF relations, the notion of two attribute relations. Suppose I have any relation that has just two attributes, we don't need to do anything, it is always BCNF complaint. How do we ensure or how do we prove that a two attribute relation is always BCNF complaint.

Consider these four cases, let us say there is a relation R which contains just two attributes A and B. Now there are four possible scenarios. The first one is there is no functional dependencies between A and B that is there exists no non-trivial functional dependencies. Only trivial functional dependencies are there, A defines A and B defines B in which case R is definitely in BCNF. Secondly there is a non-trivial functional dependency that is A defines B but there is no functional dependency of the form B defines A which is also no problem because in this case A becomes the key because there are just two attributes and A is defining B and B is not defining A. So A is the key of the relation and the relation is in BCNF.

Similarly if B defines A and A does not define B then B becomes the key and suppose A defines B and B defines A which is also no problem because both A and B are keys and both A and B are candidate keys and we can use one of them as super key or the primary keys which is fine and the relation is also in BCNF.

(Refer Slide Time: 00:45:46)



## Third Normal Form (3NF)

Sometimes, some BCNF violating FDs cannot be removed from relations without losing information.

Consider the relation Drama (title, theater, city) having the following FDs:

FD1: title, city $\rightarrow$ theater

(title and city form the key as they uniquely determine theater)

FD2: theater $\rightarrow$ city

(each drama theater has a unique name across cities)

FD2 violates BCNF since {theater} is not a key to Drama.

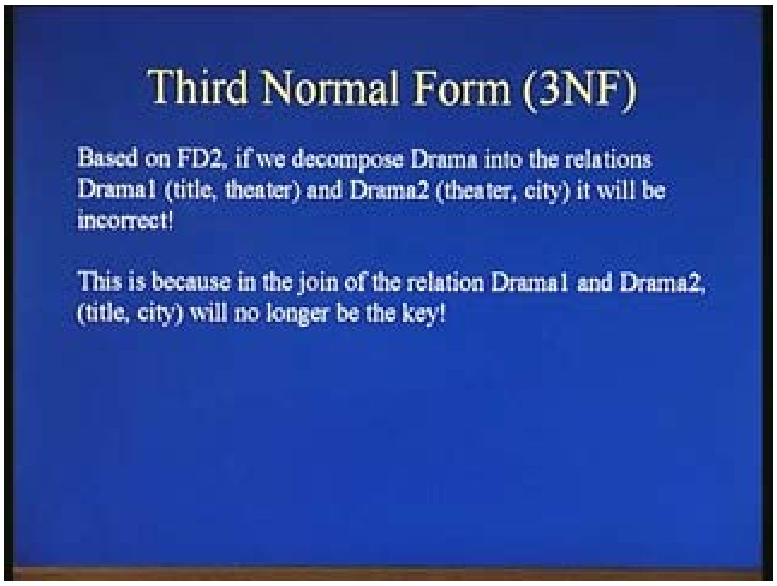
We now come to another form of normalization which is called the third normal form of a relation. The third normal form is useful because in some cases it is not possible to decompose the relation such that the decomposed relations are BCNF complaint, it is not possible to decompose without losing some information. Now let us understand this by an example. Now consider another relation of the form which is shown in the slide here. Let us say we have a relation called drama and it has the attributes title, theater and city. That is there is a particular drama troop that is performing a drama having a particular title in a particular city and in a particular theater in the city. So therefore we can identify the following functional dependencies, the first one is title and city.

The title of the drama and the city in which it is being played will probably determine the theater as well. So we can say that in this drama played in this city is being played in this theater only. Suppose there exist a functional dependencies of this form and let us also assume for the sake of argument in this case that given the name of a theater, we can uniquely identify where the theater is. Given the name of a drama theater, let's say something like kalamandira or guru nanak bhavan or whatever, some kind of theater name we immediately know which city contains the theater.

Now FD2 violates BCNF, as you can see here because theater is uniquely defining city but theater is not the key, in fact title and city form the key, title and city define theater. So given a title, drama title and a city we can identify theater therefore this is a BCNF violation in violating FD. However, based on the decomposition rule if we decompose drama into two relations title, theater and theater, city base based on this rule here because theater defines city.

Now based on this rule if we decompose drama like this into drama one and drama two, it will actually be incorrect. Why? Because once we perform the join, once when we decompose relations and we join back the relations, we should retain all the properties of the original relation. Now when we perform the join between drama one and drama two then the key constraint will no longer hold. Let us look at this by an example.

(Refer Slide Time: 00:48:03)

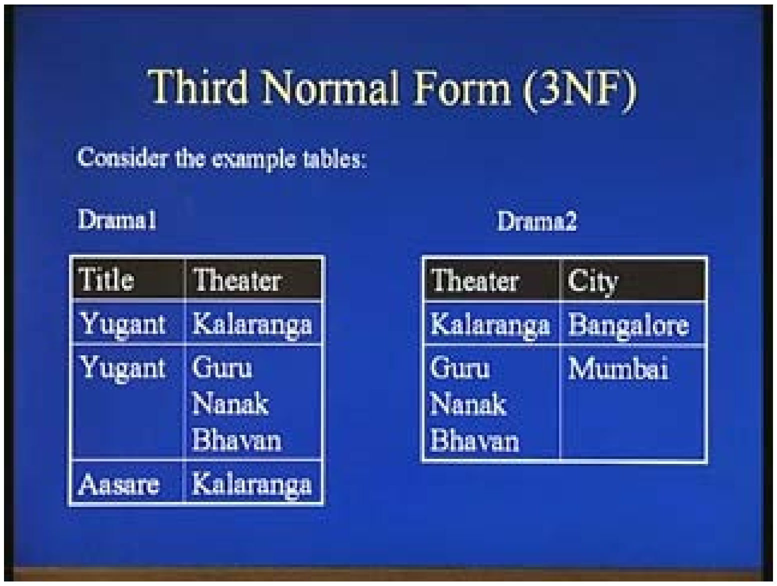


## Third Normal Form (3NF)

Based on FD2, if we decompose Drama into the relations Drama1 (title, theater) and Drama2 (theater, city) it will be incorrect!

This is because in the join of the relation Drama1 and Drama2, (title, city) will no longer be the key!

(Refer Slide Time: 00:48:46)



## Third Normal Form (3NF)

Consider the example tables:

Drama1

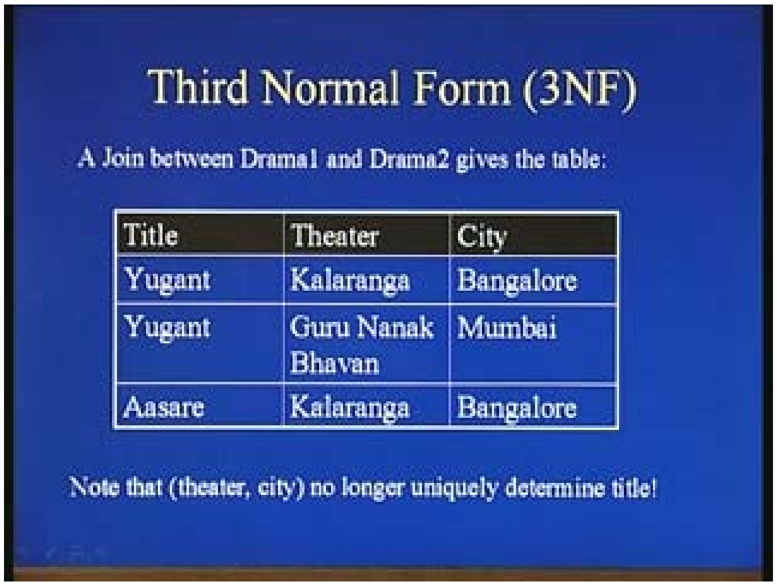
| Title | Theater |
|---|---|
| Yugant | Kalaranga |
| Yugant | Guru Nanak Bhavan |
| Aasare | Kalaranga |

Drama2

| Theater | City |
|---|---|
| Kalaranga | Bangalore |
| Guru Nanak Bhavan | Mumbai |

Let us say drama one contains title and theater entries like this, there is a particular drama with a title say Yugant and it is in a particular theater. Now there is another drama with a different title but in the same theater maybe at some other time. Now this theater uniquely identify city, let us say given a theater name I can uniquely identify city. Now if I join, if I perform a natural join between theater that is drama one and drama two based on the attribute theater what are we going to get.

(Refer Slide Time: 00:49:23)



## Third Normal Form (3NF)

A Join between Drama1 and Drama2 gives the table:

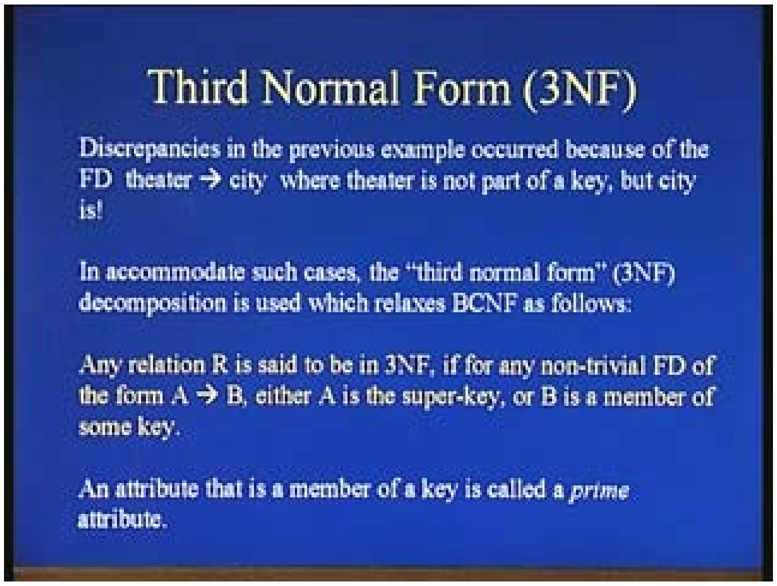
| Title | Theater | City |
|---|---|---|
| Yugant | Kalaranga | Bangalore |
| Yugant | Guru Nanak Bhavan | Mumbai |
| Aasare | Kalaranga | Bangalore |

Note that (theater, city) no longer uniquely determine title!

We will get a table like this that is a title, theater, city and title, theater, city. Now if you see given a title and a theater, there is no unique title and title and city is no longer the key that is we don't know we can have possibly different, let us say we can possibly have different sets of attributes that are defined by the same title and city combination. So it does not uniquely determine the title that is theater and city does not uniquely determine the title of the drama. Such dependencies or such discrepancies occur because of a particular property and which is what we are going to see in this slide here. So we are going to define the notion of third normal form which is essentially relaxation of the second normal of the rather the BCNF or the Boyce-Codd normal form assumption.

What is a relaxation that we are doing here? If you notice in the BCNF violating constraint that is theater defines city, the right hand side of the relation that is right hand side of the functional dependency called city is actually part of the primary key that is title and city was the primary key.

(Refer Slide Time: 00:50:05)



## Third Normal Form (3NF)

Discrepancies in the previous example occurred because of the FD theater → city where theater is not part of a key, but city is!

In accommodate such cases, the "third normal form" (3NF) decomposition is used which relaxes BCNF as follows:

Any relation R is said to be in 3NF, if for any non-trivial FD of the form A → B, either A is the super-key, or B is a member of some key.

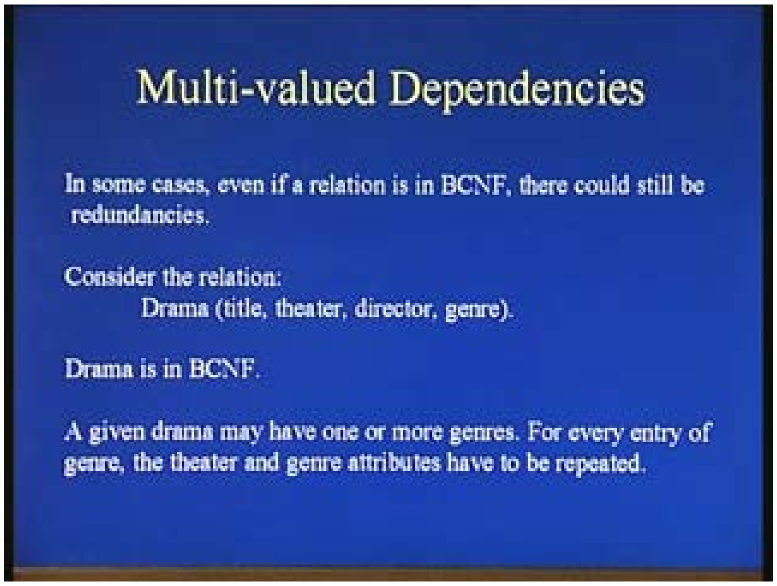
An attribute that is a member of a key is called a *prime* attribute.

So in order to accommodate such cases we use a third normal form assumption which says that any relation R is said to be in third normal form, if there exist any non-trivial FD of the form A defines B either A is a super key which was all the condition we need for BCNF. And here we have an extra condition which says or B is a member of some key. So an attribute that is a member of a key is called a prime attribute. So therefore either B should be a prime attribute or A should be a super key which is what makes the relation into a third normal form relation.

We shall not be going into details of how to prove properties of third normal form relations. However we will suffice it to note that it is a slightly general form of the BCNF or the Boyce-Codd normal form. Let us quickly visit the last kinds of dependencies and the next normal form that results from it, which is known as the multi-valued dependencies. Now even in BCNF, BCNF is a strict form of decomposition so even in BCNF we have not fully removed all possible redundancies.

Consider this following example here. The slide shows an example relation called drama which has the following attributes title, theater, director and genre of the drama. And we note that drama is in BCNF because let us say title is the key, let us say each drama is played in precisely one theater for the sake of argument. Therefore title uniquely determines theater and director and the genre of the drama.

(Refer Slide Time: 00:51:33)



## Multi-valued Dependencies

In some cases, even if a relation is in BCNF, there could still be redundancies.

Consider the relation:
Drama (title, theater, director, genre).

Drama is in BCNF.

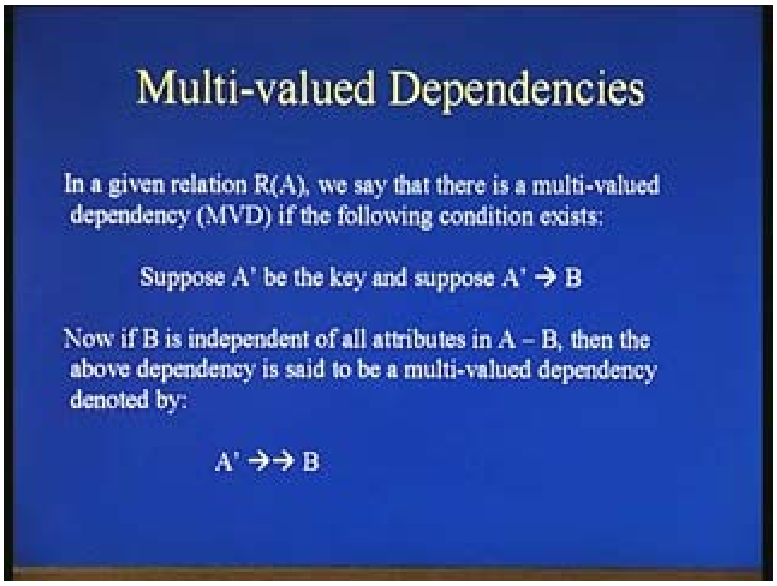
A given drama may have one or more genres. For every entry of genre, the theater and genre attributes have to be repeated.

However it may well be possible that a given title may be classified into two or more genres that is a given drama could be classified as comedy and it could also be classified as a social commentary. Now because it is classified as under two or more categories, there exists what is called as this one to many relationships between title and genre. So every time we identify the set of all comedy dramas, the theater and director has to be repeated.

Similarly every time we identify social commentaries, the theater and director has to be repeated because it is the same drama. So this is what is called as a multi-valued dependency. Now what is a multi-valued dependency in a more formal fashion? Now suppose in a given relation A, there is a non-trivial functional dependency of the form A prime defines B and suppose A prime is also a key for because it’s in BCNF.

Now suppose if B is completely independent of all other attributes of the relation then we say that there is a multi-valued dependency. In this case the attribute theater and director are completely independent of the category or the genre of the drama. It has no relationship between, a theater may play any kinds of drama and a director may direct any kinds of drama himself.

(Refer Slide Time: 00:53:37)



## Multi-valued Dependencies

In a given relation R(A), we say that there is a multi-valued dependency (MVD) if the following condition exists:

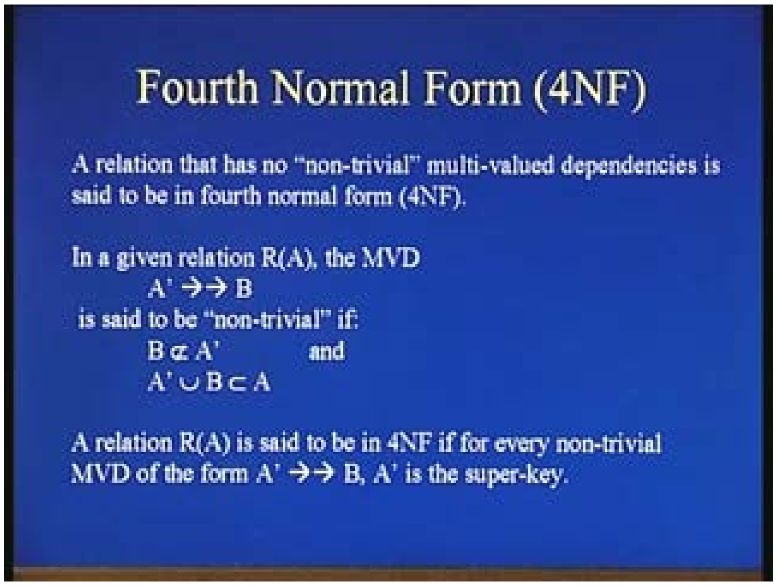
Suppose A' be the key and suppose $A' \rightarrow B$

Now if B is independent of all attributes in A – B, then the above dependency is said to be a multi-valued dependency denoted by:

$A' \rightarrow\rightarrow B$

So it’s completely independent of that, so we say that there is a multi-valued dependency.

(Refer Slide Time: 00:54:37)



## Fourth Normal Form (4NF)

A relation that has no “non-trivial” multi-valued dependencies is said to be in fourth normal form (4NF).

In a given relation R(A), the MVD

$A' \rightarrow\rightarrow B$

is said to be “non-trivial” if:

$B \not\subset A'$ and

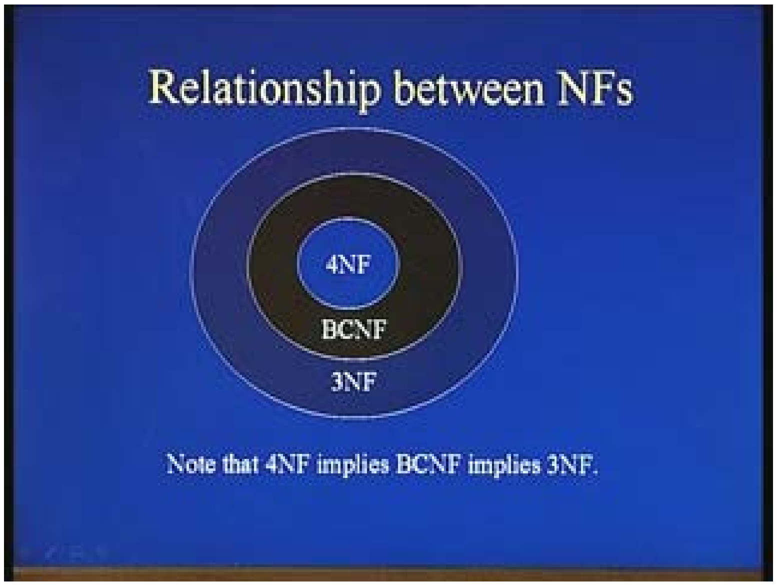
$A' \cup B \subset A$

A relation R(A) is said to be in 4NF if for every non-trivial MVD of the form $A' \rightarrow\rightarrow B$, A' is the super-key.

So we defined the notion of a non-trivial multi-valued dependency in order to remove them. So firstly what is the notion of a non-trivial multi-valued dependency? A multi-valued dependency of the form A, define A prime defines B is non-trivial if B is not a subset of A prime which is the non-trivial property for BCNF as well and there exist certain other attributes in addition to A prime and B. That is A prime union B is a proper subset of A, the set of all attributes that is there exist some more attributes in addition to B.

So a relation R of A is said to be in fourth normal form, if for every non-trivial functional dependency of this form that is A multi-valued dependency B, A prime is the super key.

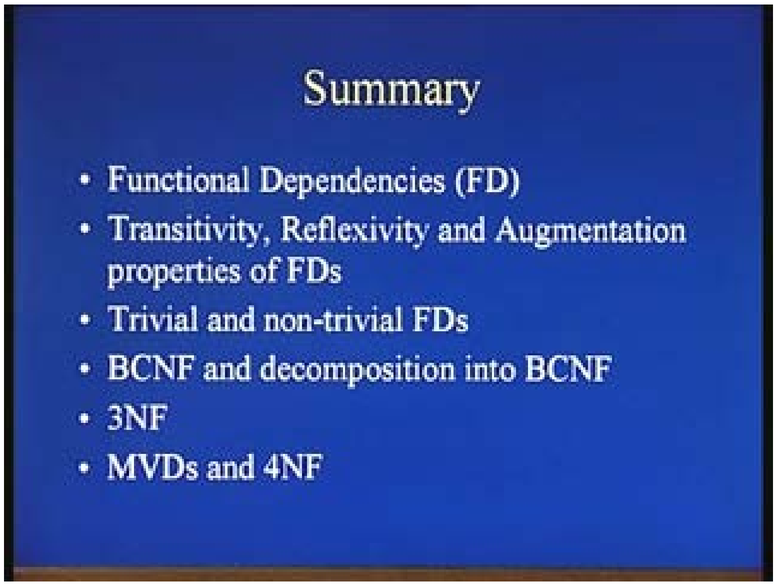
(Refer Slide Time: 00:55:31)



## Relationship between NFs

Note that 4NF implies BCNF implies 3NF.

So we shall not be going into more details of the fourth normal form and for this session we will suffice it to say that fourth normal form is an even most stringent criterion for removing duplicates or removing redundancy in relations. So essentially the normal forms can be categorized like this. Third normal form is the most lenient among the three and next is BCNF and finally the most stringent is the fourth normal form. So if any relation that is complaint to fourth normal form is automatically complaint to BCNF which is in turn automatically complaint to third normal form.
(Refer Slide Time: 00:56:15)



## Summary

- Functional Dependencies (FD)
- Transitivity, Reflexivity and Augmentation properties of FDs
- Trivial and non-trivial FDs
- BCNF and decomposition into BCNF
- 3NF
- MVDs and 4NF

So we now come to the end of this session and let us briefly summarize what all we have studied here. So we studied the notion of functional dependencies which is a generalization over keys and properties of functional dependencies like transitivity, reflexivity and augmentation, extra. We also saw the notions of trivial and non-trivial functional dependencies and how they affect BCNF and how we can decompose relations into BCNF so that we can remove redundancies.

However we see that not all relations can be decomposed into BCNF without losing information because of which we have also, we also need the concept of a third normal form. And we finally saw the notion of multi-valued dependencies which can remove redundancies that exist in BCNF and the fourth normal form which remove multi-valued dependencies. So that brings us to the end of this session.