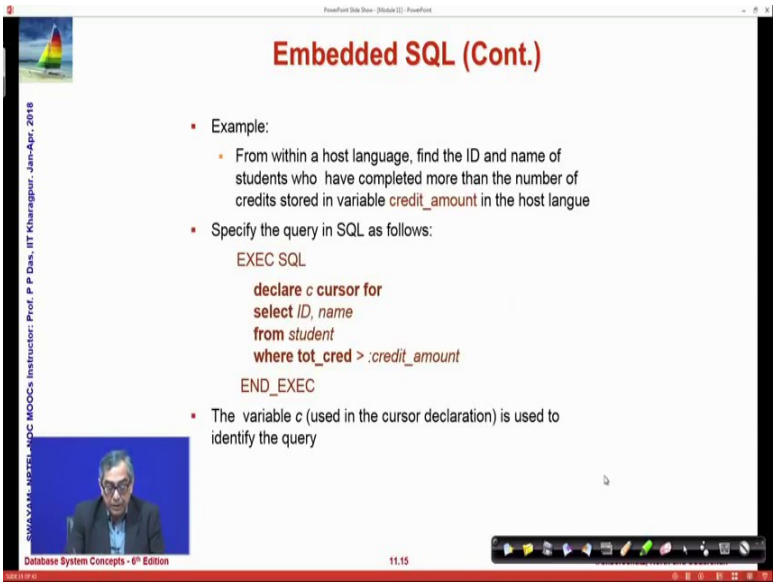
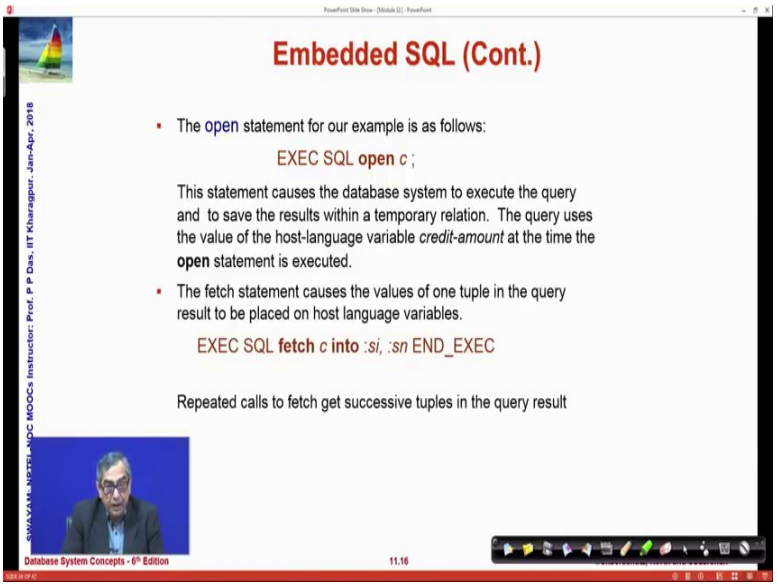
(Refer Slide Time: 17:14)



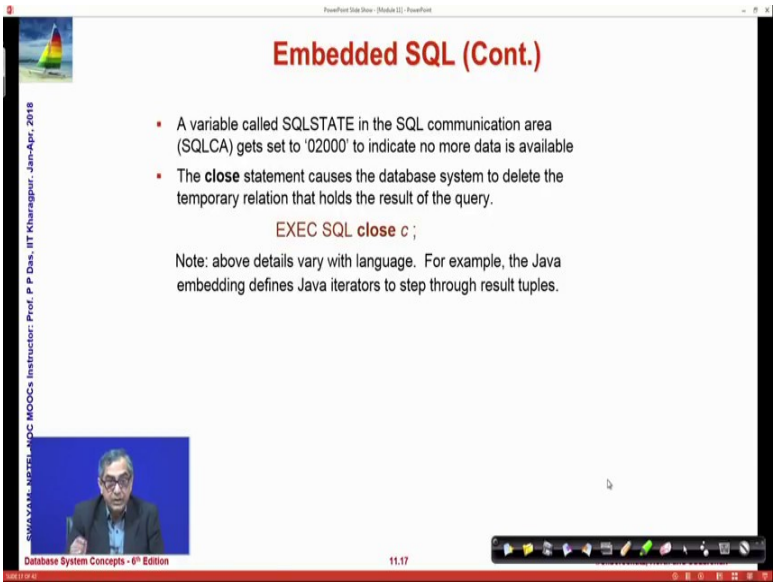
So, let us this is the example continued.

(Refer Slide Time: 17:18)



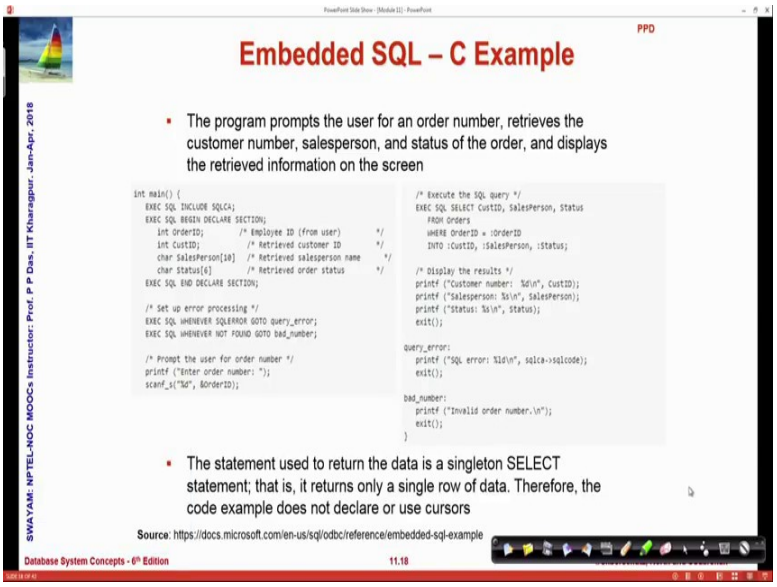
Let us look at other features. You can once you have set a query you can actually execute that by the open statement. So, you say EXEC SQL **open** c, the cursor. So, that will execute the query that you have associated with that cursor. And then once that has been done, then you can fetch the results into that cursor one by one; one tuple at a time.

(Refer Slide Time: 17:47)



Once you are done with all that then you simply close.

(Refer Slide Time: 17:52)



So, let us look at a example. This is a program which will which prompts the user for an order number, and retrieves the customer number I mean given an order number it will retrieve the customer number, salesperson, status of the order and it will display that as the retrieved information on the screen. So, here is a C program. It starts on here with the main. So, you can see that this says that EXEC-SQL **INCLUDE** SQLCA, SQLCA is the communication area. So, there is a there is a exchanges going on between the c program

and SQL program. So, the area that is used by both for this transaction is known as SQLCA.

Then you have the declare section. So, you are saying these are SQL exec declaration. So, these are, but within that what you have are pure c declaration, but all the declarations of c that are put within this can be used in SQL query with a colon at the beginning. So, that the value can be exchanged to the SQLCA then in the next you are specifying what will happen if you have an error.

So, you say SQL look at this EXEC-SQL whenever SQL error go to query error. So, it will go to this level. If it is not found that is no result is there it will go to this. So, you by making sure that if there is some error that happens in the SQL part, what will happen in your C program. And then subsequently you have simple C program which reads the order number, and after having read that you are doing this.

So, what does it do, EXEC-SQL is to say that this is embedded; this is the select query starts, the fields, the relation, the condition. And then there are three attributes. So, you are setting an association with three variables in the C program. So, if I want to the result of this select will be a table of three attributes three columns, so we are giving a name to each one of these three attributes in terms of our C program.

So, there is a cust id. So, I say the cust id attribute in SQL is colon cust id here which is basically cust id variable in the C program. Let me clean up again. So, this is cust id; this is in SQL; this is my C program, and this is my C program variable in SQL which corresponds to this its similarly order based. Similarly, this is in SQL attribute this is a array of character here. And this is a correspondence; this is a correspondence.

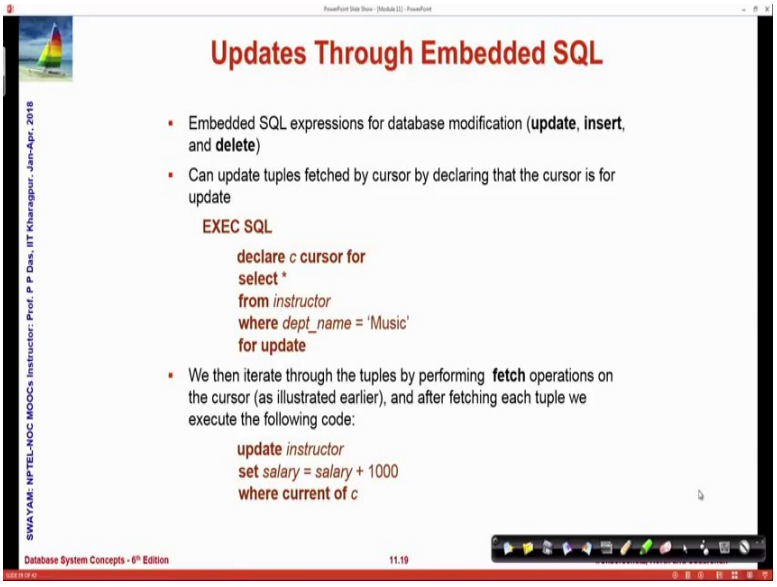
So, now, you can easily see that once this has been done I will be able to get all these values here, normally the program would be longer the you will have to iterate over the cursor as you did in case in the ODBC case. But in this case since we are using one order number we know that there will be only one record. So, we have not shown the iteration on the cursor.

So, once you have got this you are you are using those values to print out the result. And in case this query has got into some problem because of SQL, then it will automatically jump to SQL query this particular level. If it has got a bad number which means that no

such record was found, it was a null table then it will immediately take you to this. So, this is how the embedded SQL worked.

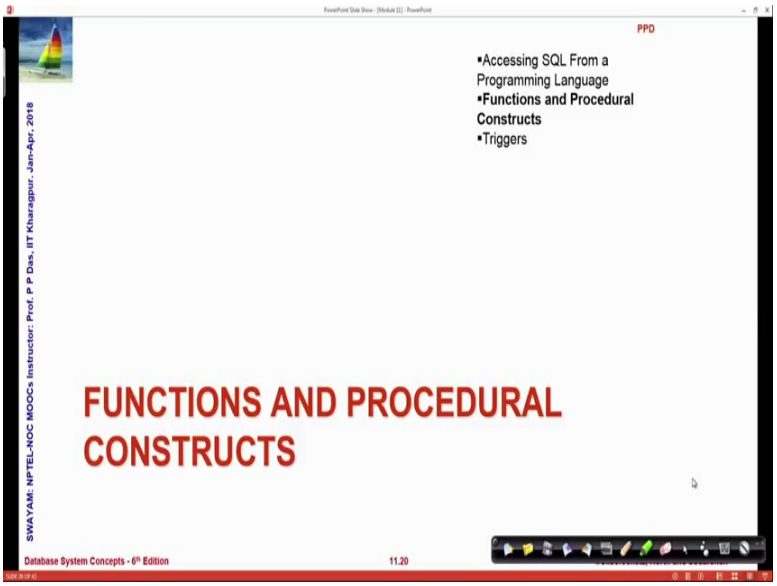
So, there are these are two different styles the ODBC style and the embedding style are two different styles. If you have if you depending on the preference you use that earlier days people used to use more of embedding, I believe that now the preference is more for the ODBC kind of connection oriented system ODBC JDBC kind of because they are certainly more programmer friendly this.

(Refer Slide Time: 22:05)



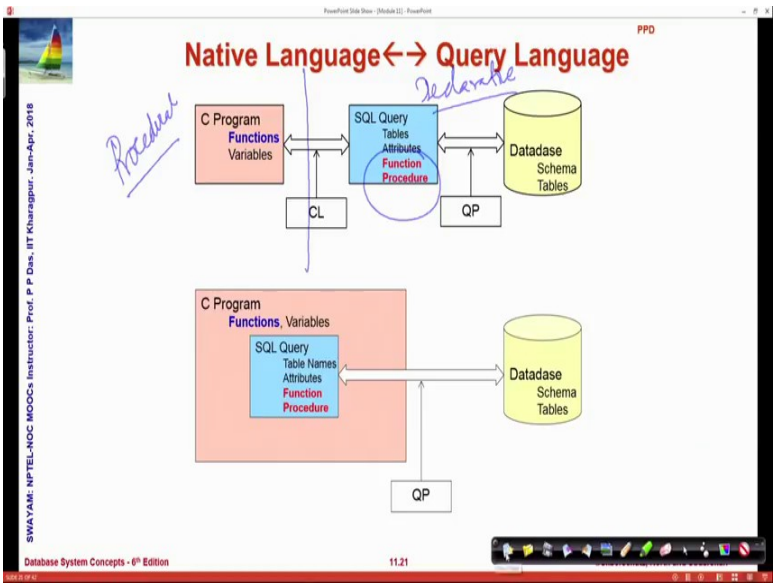
You can do updates through embedded SQL as well. So, I will not go through the details you can just go through this and understand that.

(Refer Slide Time: 22:15)



Let me move onto the next advanced part of SQL which is function and procedural construct.

(Refer Slide Time: 22:27)



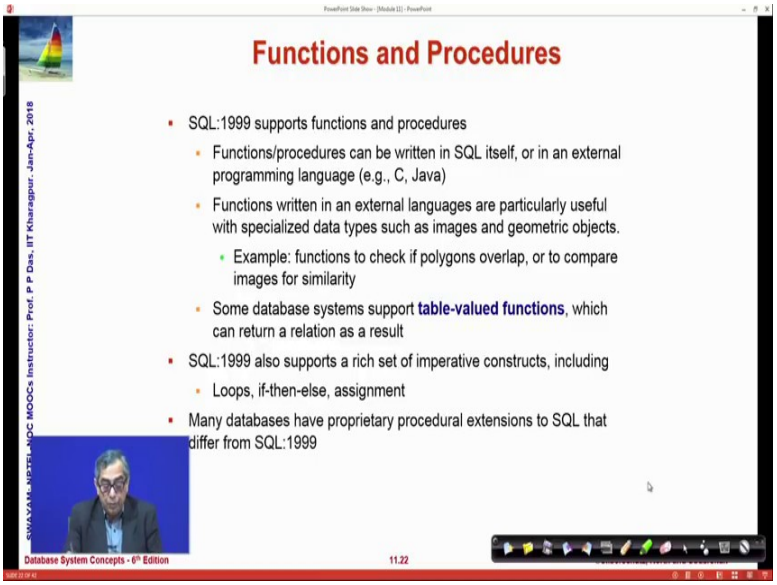
Now mind you again this is these are the two models that we have I have already shown you. The two models in which the application program the native language program and the query language can interact the ODBC mechanism and the Embedded mechanism. But what we are now empowering is so far our basic premise was that this site is a procedural I discussed this at the very beginning. And this side is declarative. So, in SQL

you do not say that how you find out a result, you say what you want as a result, these are more like predicates. And in C program, you cannot specify what you want as a result you would rather say that do step one, step two, step three and you will get this result.

So, C program is all full of functions again I am talking about C as a placeholder it is true for most of the programming languages we use otherwise. So, there are procedural languages. So, procedures in C are functions; whereas, in SQL you had select from where kind of conditional clause.

Now, what we are saying that SQL also in later version have allowed certain functions and procedures, which can be part of SQL. It also has allowed certain imperative constructs like case like loop like while, repeat those kind of to make certain kind of procedural programming easier in SQL. And naturally these again can be used in conjunction with the connection oriented applications with the native or embedded oriented mechanisms. So, we will just take a quick look into some of these function and procedural features.

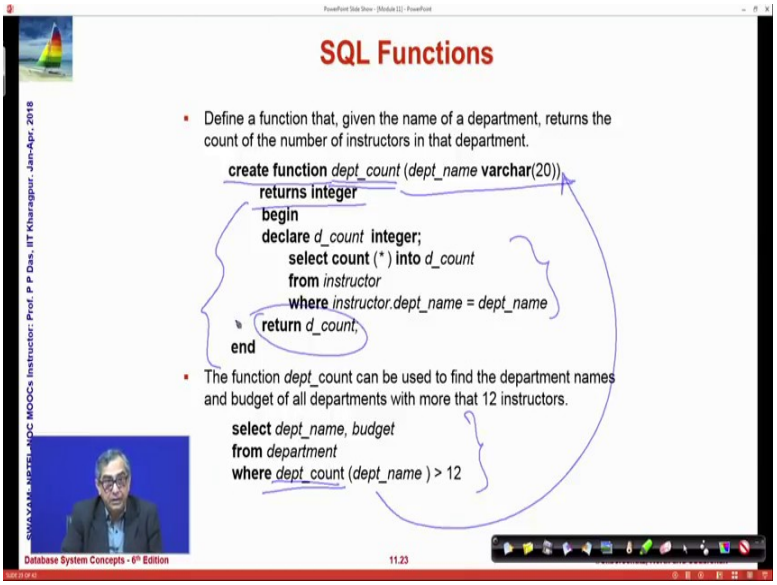
(Refer Slide Time: 24:15)



So, this started coming in from SQL 1999. And you can have functions and procedure written in SQL itself and function and procedures written external language or the host language also. So, both of them are available. What is interesting is some database systems support functions which are table valued. Functions, we always know functions

written objects only, but in SQL you can have functions which have table valued which written new functions. So, and certain imperative constructs have come in.

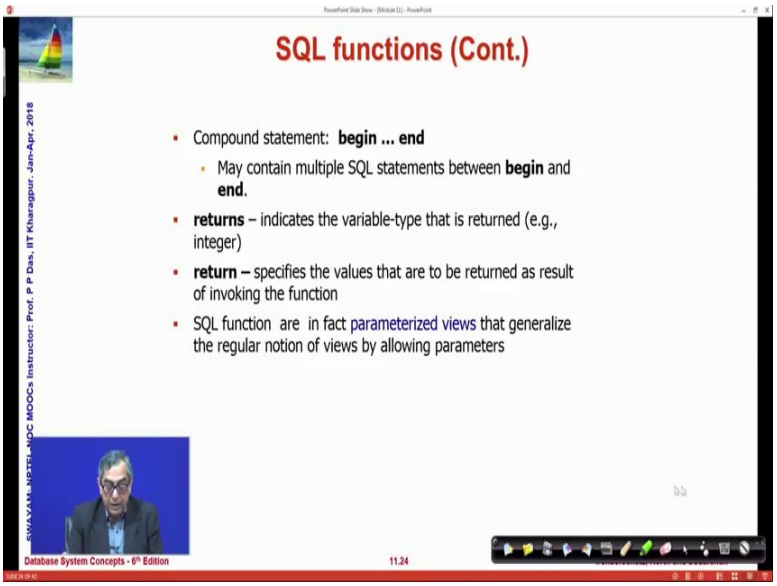
(Refer Slide Time: 24:56)



So, there are several databases have their proprietary constructs and all that also. So, at this level of the course since we are not focusing particularly on any specific database systems, we will not talk about those. We can do enough in terms of the standard SQL itself. So, this is how you define a function which looks very similar to the SQL definition, create function, give it a name certainly here are the parameters. And here is a return type. So, if you are familiar with C, C++, Java you I mean it is just that the syntax is different, but the elements are same.

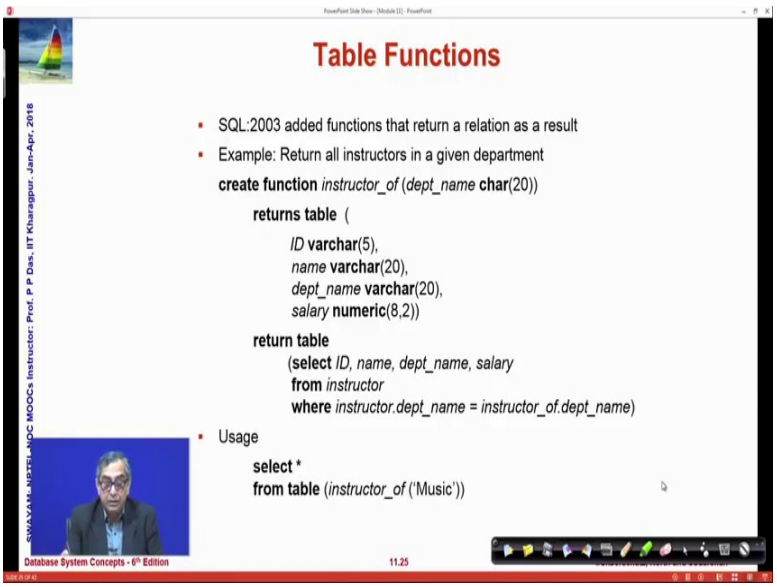
There is a begin end in the scope. And this is the pure SQL that you are writing here. So, this is a function which is not a function in C, this is a function in SQL. So, this you can write this function in SQL. And once you have written that function then you can actually use that. So, if you look at the function is department name, then separately I am writing a query, I am using this department name as function. So, whenever I whenever this query will get executed, this function will be called, and the corresponding values will get returned. So, this is the basic mechanism ok.

(Refer Slide Time: 26:16)



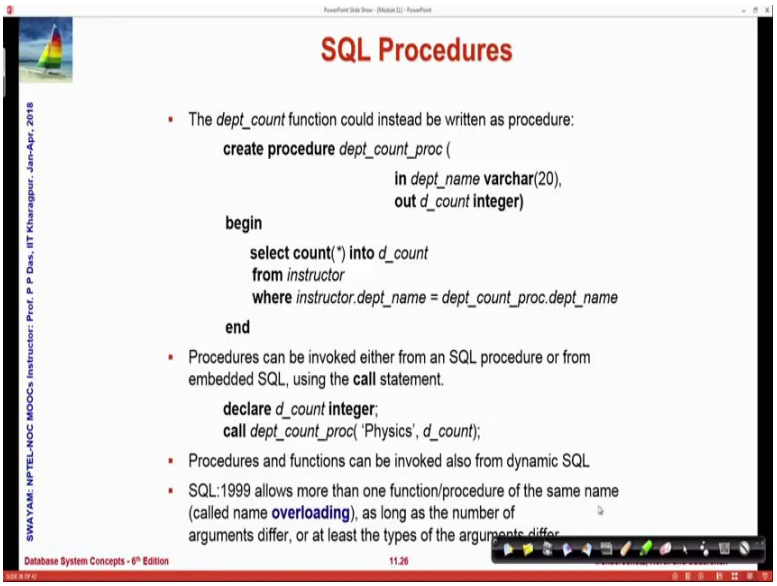
So, so there are SQL functions have several details which you can go through it has returned naturally.

(Refer Slide Time: 26:23)



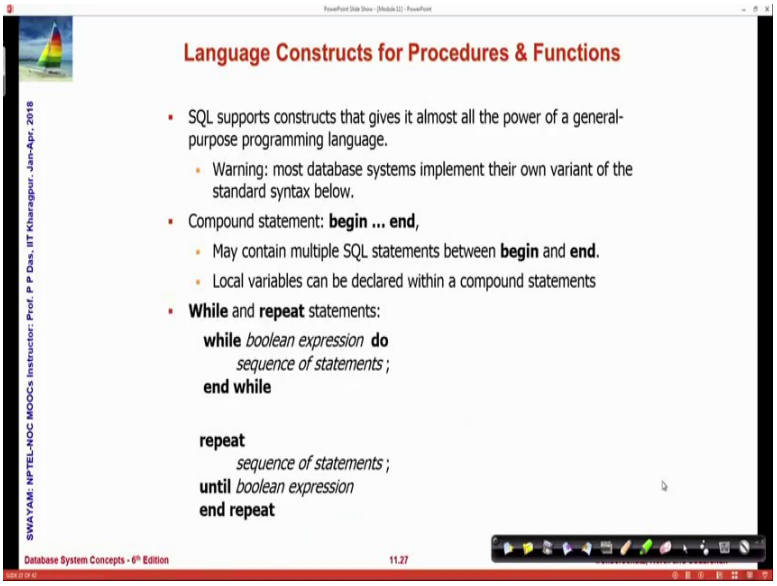
And you can have table functions where it can return table. So, the syntax is given again its clear to see what will happen if you return a table which is computed from the select from where query that you have within the function.

(Refer Slide Time: 26:37)



I can have SQL procedures which just performs some action, but does not have a have a return value so to say. And you can use them they can declare and call those procedures explicitly. Procedures can be functions and procedures can be overloaded as well if you are on SQL 99.

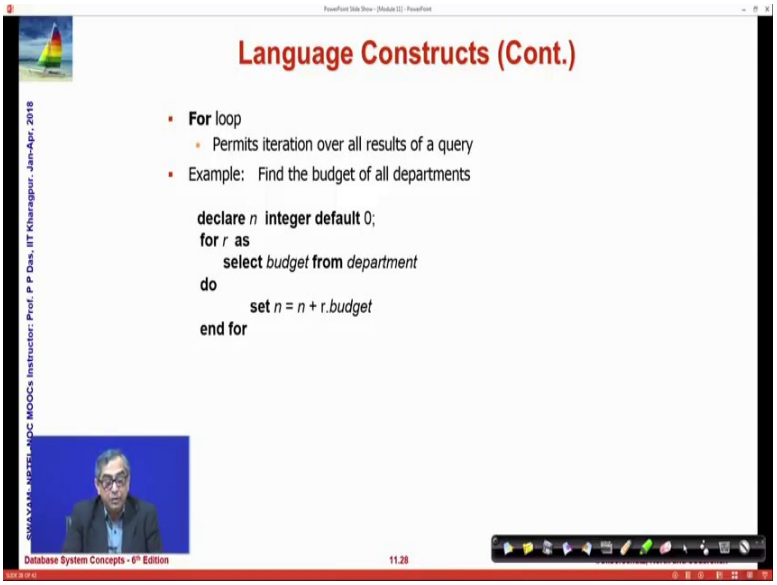
(Refer Slide Time: 27:08)



Now, there are several language constructs as I mentioned SQL does allow while repeat. So, I am I am just covering them in terms of completeness it is not that these are frequently used features or I recommend that you do lot of them right here. If you need

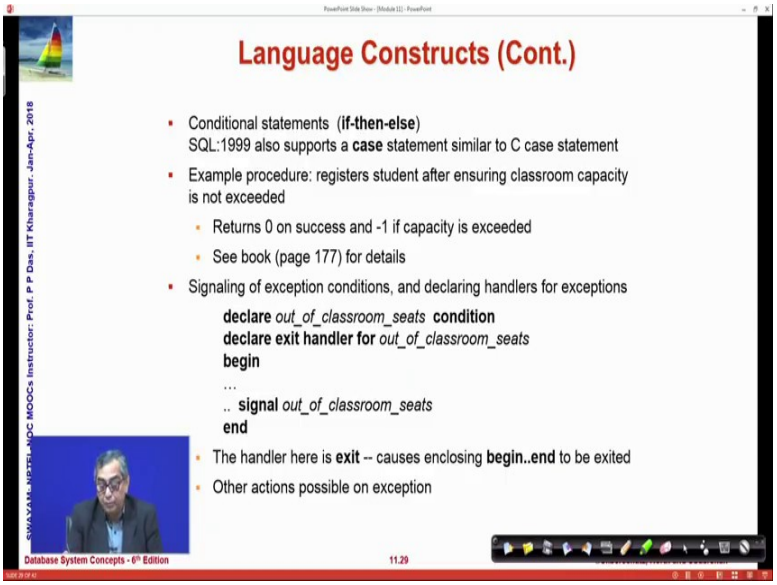
to do procedural thing its always better to do them outside, but in some cases it may be easier to code a query if you can write a while, repeat kind of loop.

(Refer Slide Time: 27:37)



You can write a for loop which iterates over the records of a table which is certainly very convenient. So, if you want to do something over the records of a table compute something that the for loop will become easier.

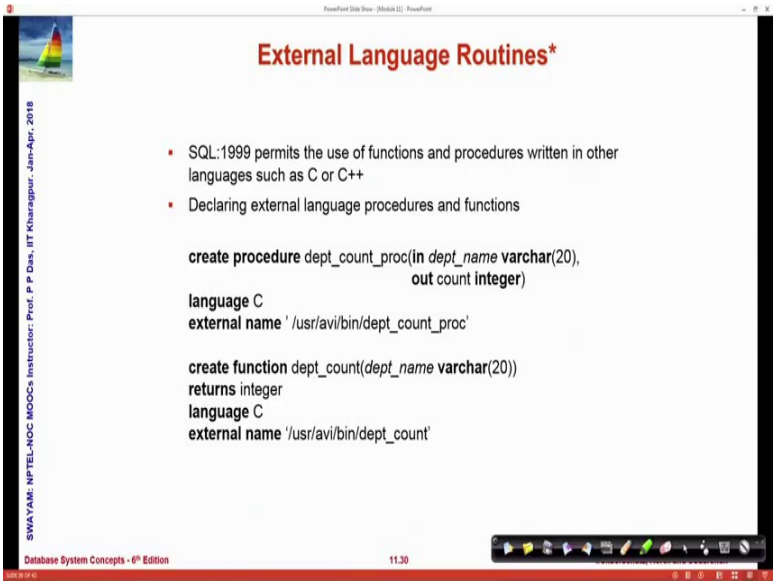
(Refer Slide Time: 27:50)



You have a conditional statement case actually we have seen the case already. So, which is very easy in terms of coding many of the features so, here are some of them then you

have exceptions. So, I am not going through these, these in depth, but this is just to making you aware that while SQL continues to be predominantly a declarative language, it does have quite a bit of procedural support which in an appropriate time can be used if required. So, you can look up the manual for that. And there are a whole lot of things you can do with the external language routines.

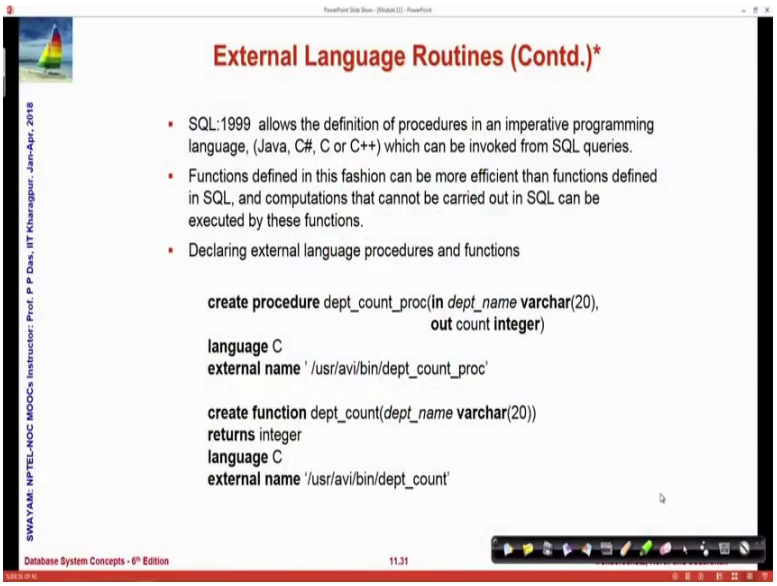
(Refer Slide Time: 28:30)



That is can write a function in C and actually call it from SQL, this is just doing the other way round. Earlier in terms of embedding or in terms of ODBC, a C function was executing a query. Now, you are doing the reverse you are saying that I can write a SQL query which uses a C function that already exist. So, there are external ways of binding, I will not go through these slides in depth, because again the you will have to come a certain way before you can actually start using such features.

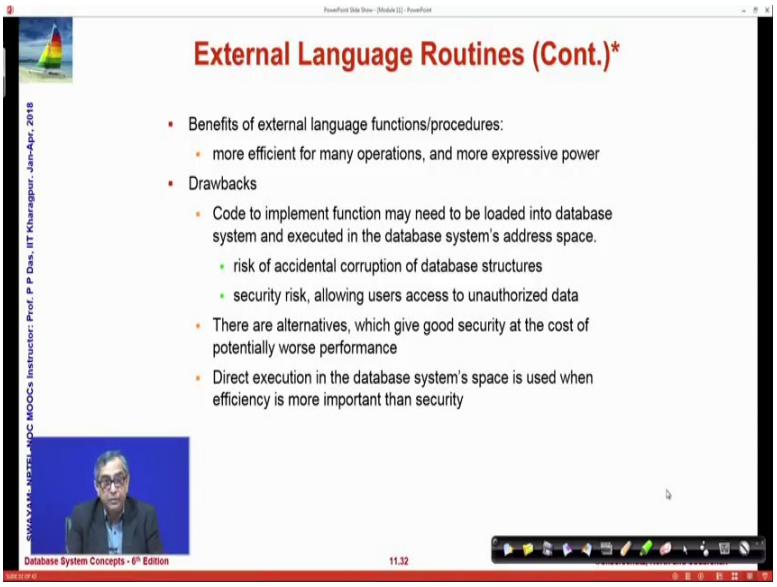
But get to know that there could be as from SQL you can use any external language library; and if some library is good for certain computation which is needed for your query which is a non database kind of computation then you can make use of this external language routines.

(Refer Slide Time: 29:23)



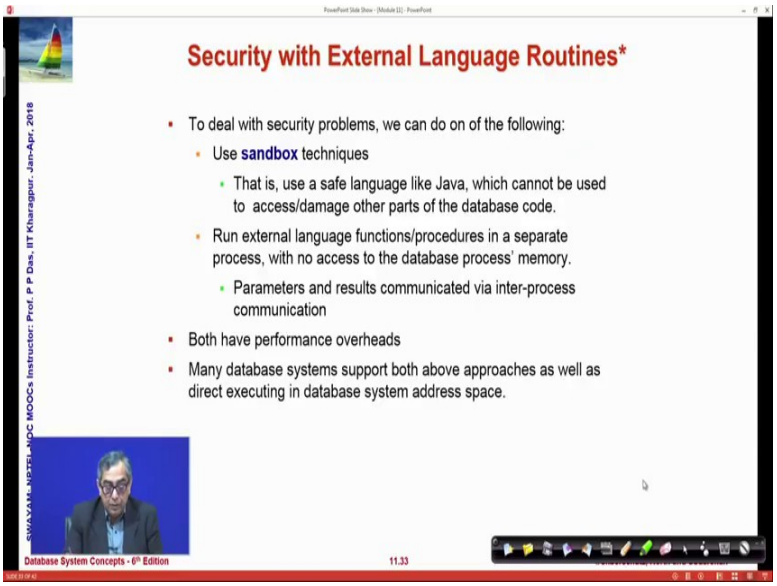
So, I have put together all the basic features that external language routines will need.

(Refer Slide Time: 29:30)



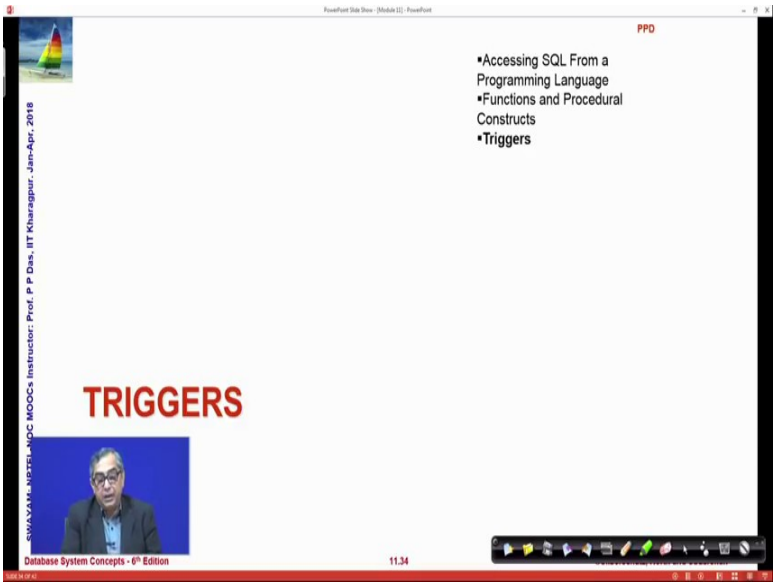
But again I would tell you that there are benefits, but there are lot of drawbacks in using them. And I would not recommend that you frequently use these features. You should primarily restrict to SQL programming, and then anything that you need to do in the native, you should go to choose your language and do that.

(Refer Slide Time: 29:48)



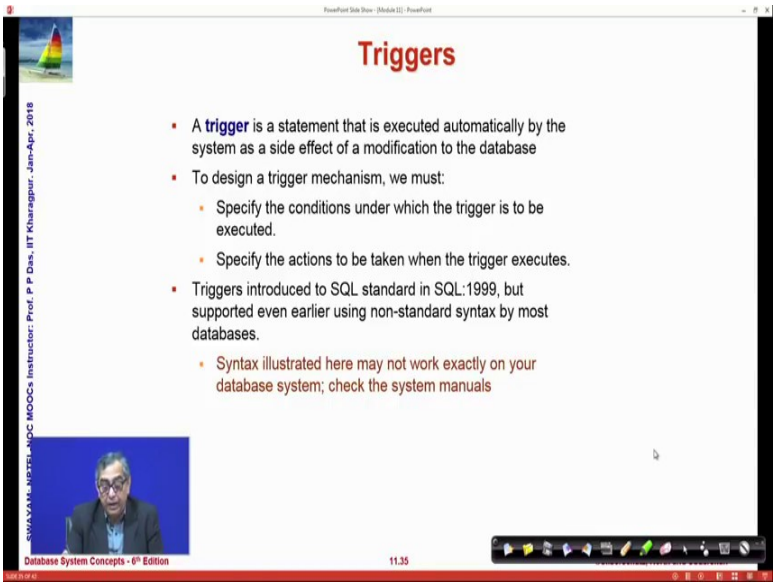
There are security issues also that you will need to understand here.

(Refer Slide Time: 29:58)



Finally, before we close I will just mention a another feature called triggers, triggers are very important.

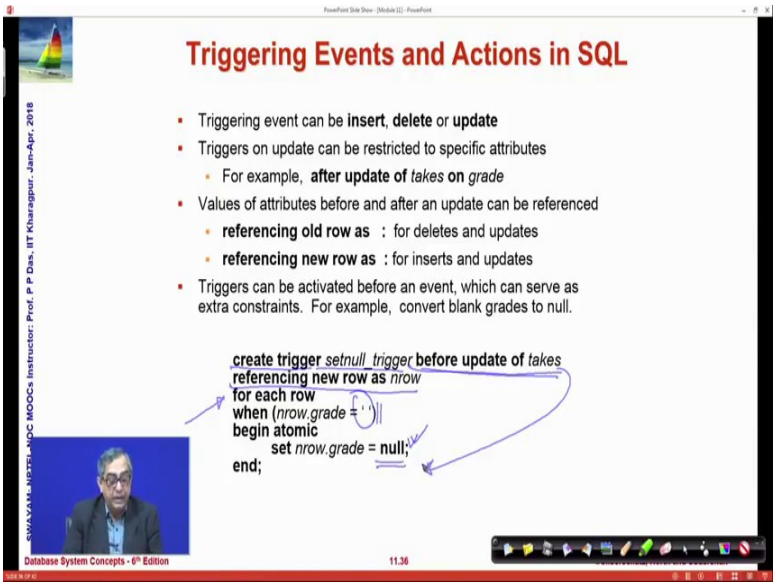
(Refer Slide Time: 30:04)



A trigger is a statement that is executed automatically when something happens in the database. So, you want that well things are happening. And well I want to know if a particular value has exceeded a certain level or if something has become null or some violatory things are happening and so on. So, how do you know that because a database is being accessed by hundreds and thousands of people and with hundreds of tables and millions of records.

So, triggers are a mechanism by which you can set that under this condition, I want a trigger, I want something specifically to happen. So, again they were introduced in 99, but earlier also triggers were there, but in 99 standard they became formal earlier they were somewhat you know differently structured. So, you might find that the system that you are using for practice the trigger in that may have a different format and semantics then the what we are discussing here.

(Refer Slide Time: 31:12)



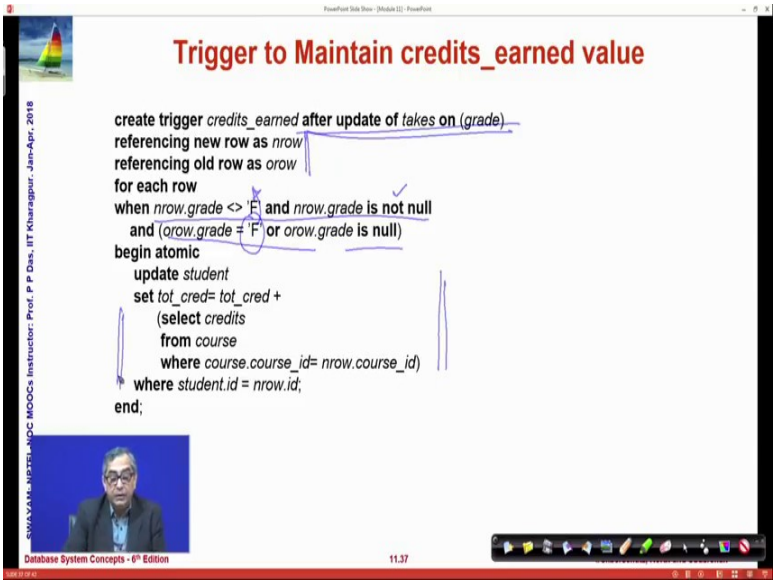
So, the most common triggering events are insert, delete, update. So, if something some update is happening, so I can say that after this update, I want such and such things to happen. Or I can during the update I can one that well I am doing an update. So, there is an old value which is typically referred to as old row, the row that is getting updated. And there is a new row the new set of values that are getting created. So, I might want between the old row and the new row that certain things happen.

So, I can say that well just look into this. So, again the syntax is all similar. Create trigger, trigger there is a name of the trigger and this is the condition. Before update of takes, takes is a relation **referencing new row as nrow**. So, this is the new value that we set. Now, what are you saying, saying that for each row what you do when n grid is blank n row dot grade is blank that is if this is if you are updating and you have got a got you are going to update, a grade value which is blank then you simply set it to null.

So, it is possible that the grade that has come in and grades are characters and what has come in from the input and is going to get updated is a show a certain grade to be now because it may not have been decided. Now, you do not want those blank values to be present. You want because blank cannot be checked, we have we have checkers for null and so on.

So, you want to set that to null. So, trigger can make this thing happened because otherwise how will you know what value is actually getting changed. So, there could be several ways trigger can be used.

(Refer Slide Time: 33:23)

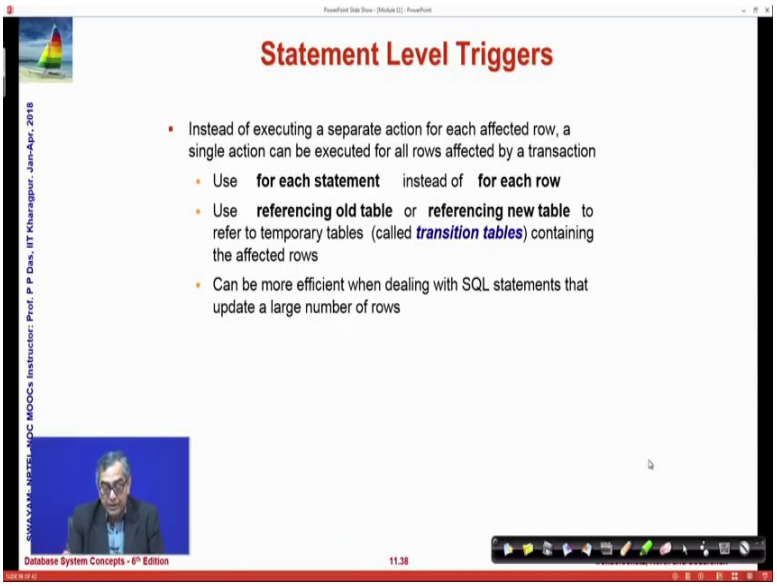


For example, this is showing one that as you change the grades then certainly based on the grades credit earned value is computed. So, as a greatest change if it is now once grades have been entered say for 200 students. Now, after reviews grades for three of them are getting changed. So, how do you know that for those students, the change of the grade may impact the computation of the on credits.

So, you would like to update that on credits or it may or may not be required. So, the trigger will tell you that after update. So, whenever the update happens you take the old and the new value n row and o row, and then you are putting some conditions that if that new row is grade is f, and is not null; old row is grade or it is not null, then you try to do this.

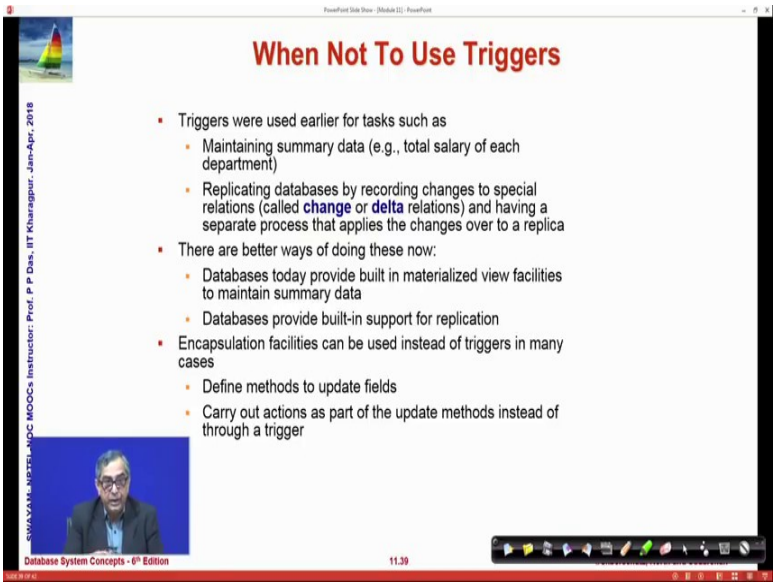
So, if it was failure, and if it continues to be failure, new grade is not f; if it is f then you do not have to do anything; if it is null it do not have to do anything. But if it is if it was f or it was null, and now it has become a different grade then certainly the computation to update the credits earned is required. So, and the trigger gives you the right point when you can do this because otherwise you will not know in terms of millions of updates happening when this particular thing is going on.

(Refer Slide Time: 35:08)



Triggers can be on statements as well you can decide leave for your reading.

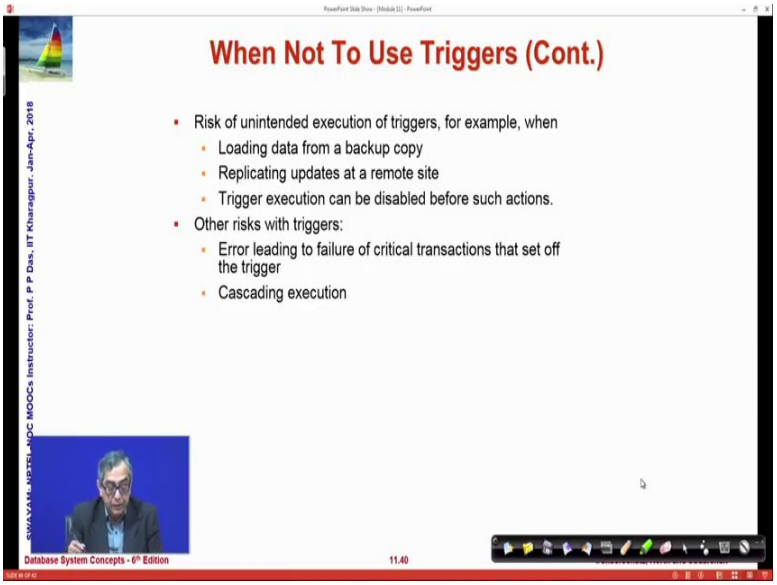
(Refer Slide Time: 35:12)



But you have to be careful that triggers sounds very interesting and what happens particularly with the early stage of programming people get overboard with triggers and start using them severely, but triggers do have a lot of overhead. So, you should not many of the things that triggers can do, can be done through other means for example, by materialized, views and so on. So, as we go along we will mention that these are the problems that need to solve get solved by triggers; otherwise normally you should think

twice before you actually use a triggers. So, there could be different other ways of solving the same problem.

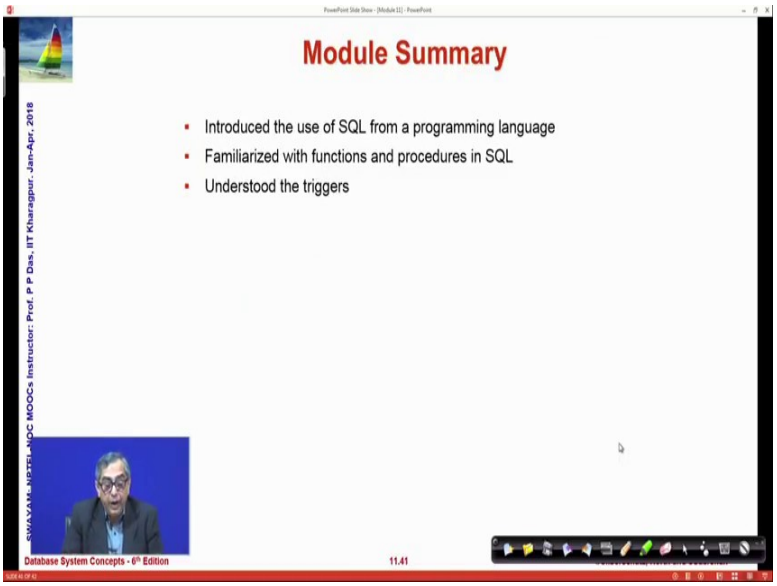
(Refer Slide Time: 35:55)



And because you have to keep in mind that triggers are expensive because once you have triggers and actually internally database for every update. If you have an update trigger on a field or a relation, then with every transaction with every change the database has to check if your trigger is true or not and so therefore, there is a cost to that. The other thing is there are triggers are for the live execution.

So, if you have offline for example, you are loading the data from a backup copy or you are replicating your database at a remote site and so on, then you have to put off the trigger; otherwise you know falsely the triggers will start happening and that may have a catastrophic effect that might trigger of different alarms and all that. So, you have to be careful with triggers in that manner so, cascading executions and all those.

(Refer Slide Time: 36:46)



So, to summarize we have and this kind of closes our direct discussion on SQL. So, we have introduced the use of the very important aspect the use of SQL from a programming language, the interface between the native language and query language boundary. And that is something which will be extremely useful for application programming. And we are familiarized with you know the imperative extensions of SQL, the procedural extensions of SQL in terms of functions and procedures that you can directly write in SQL. And we have just introduced concept of triggers, so that you can sniff what is going on in your database.

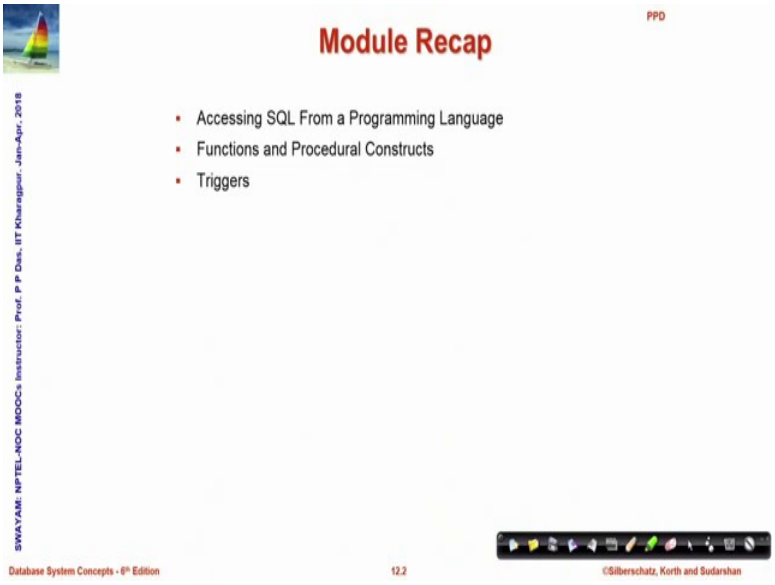
So, there is the quite a few other features of SQL as well majority of them are advanced features dealing with olap and several others, which I chose to skip at this level of the course. So, this will close our discussion on SQL. And next we will move onto the design of the database looking into the algebra and the modelling.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture - 12
Formal Relational Query Languages

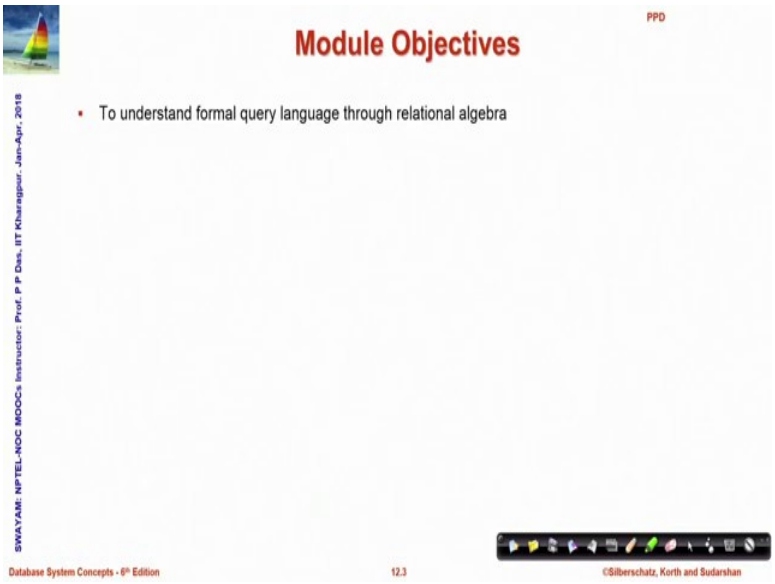
Welcome to module twelve of database management systems, in this module we will talk about the formal relational query languages. In the last couple of modules we have discussed about SQL at length introducing it dealing with the intermediate level of SQL features, and then exposing to some of the advanced features as well. The foundational mathematical model of SQL the query languages are to be discussed in this present module.

(Refer Slide Time: 01:04)



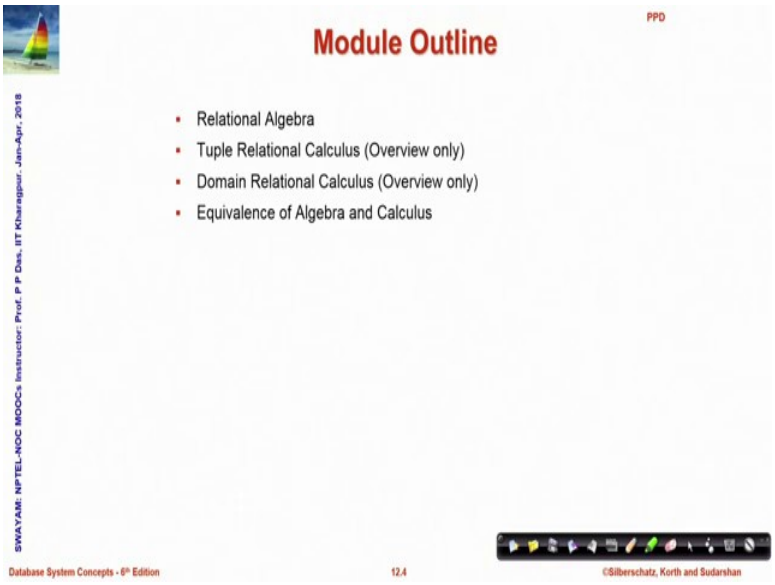
So, this is what we had done in the last module.

(Refer Slide Time: 01:10)



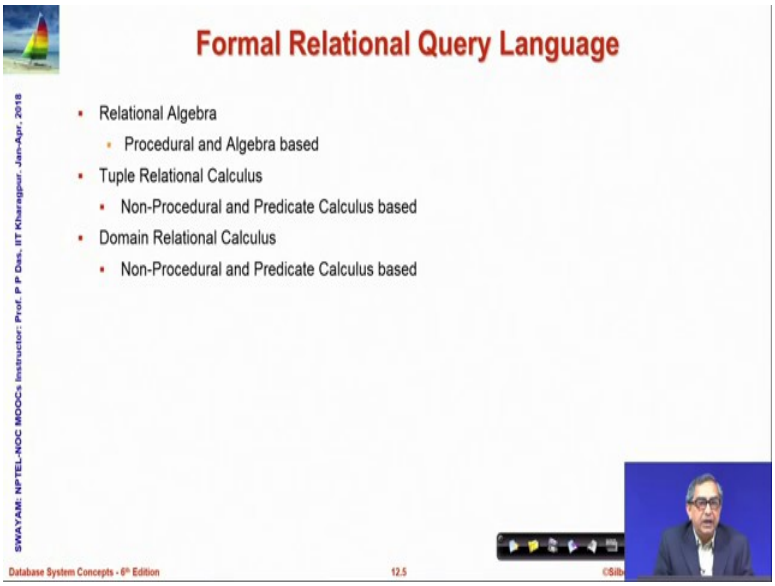
In the current 1 we will work to understand the formal query languages.

(Refer Slide Time: 01:19)



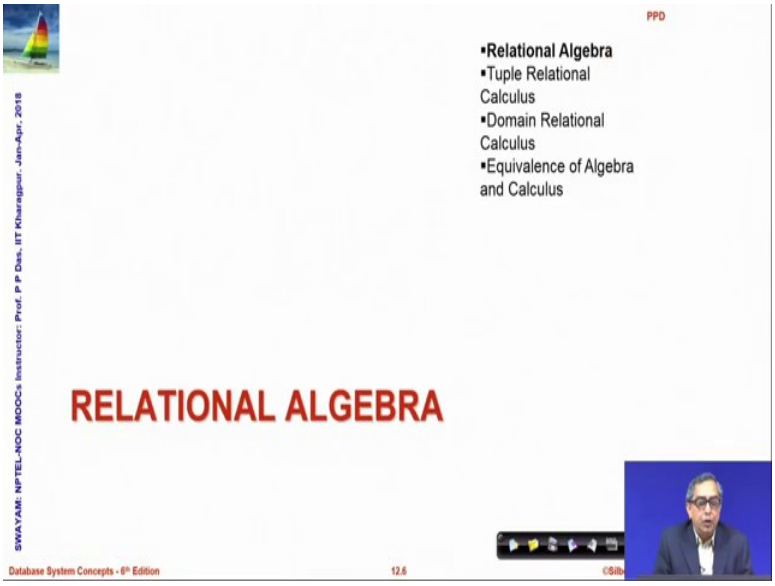
Primarily through relational algebra, and then we will also take a look into some of the calculus aspects tuple relational calculus, and domain relational calculus and we will show by example the equivalence between the algebra and the two calculus.

(Refer Slide Time: 01:40)



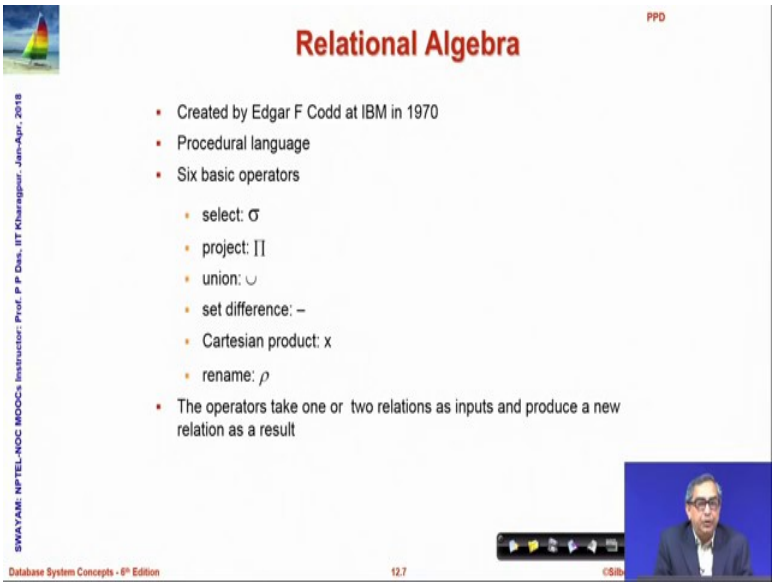
So, formal relational query languages are of 3 types 1 is known as relational algebra this is procedural in nature. So, we specify what operations need to be done to achieve the result and the whole formulation is based on set algebra. The second formal query language is tuple relational calculus which is non procedural and is based on predicate calculus. The third one the domain relational calculus is a minor variant of the tuple relational calculus is and is also non procedural and predicate calculus based.

(Refer Slide Time: 02:35)



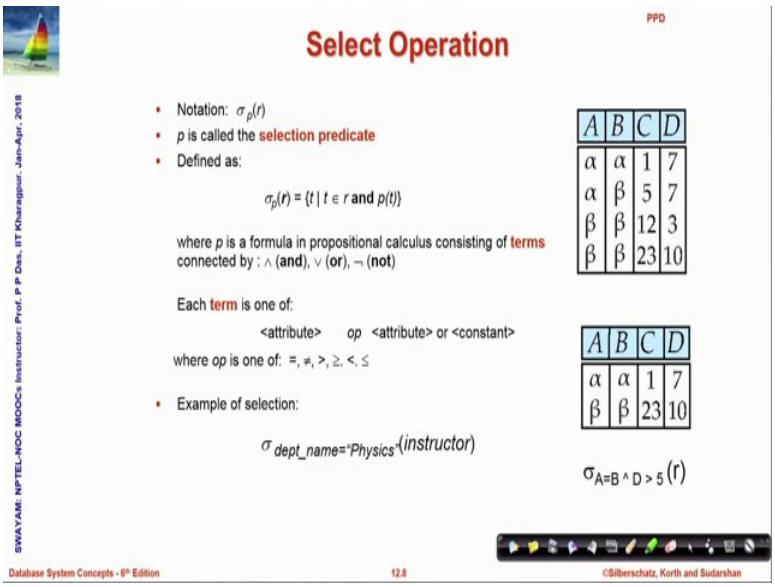
So, we start with the relational algebra in the relational algebra.

(Refer Slide Time: 02:40)



It was created by Edgar F Codd at IBM in 1970. So, you can see that it is quite an old formulation it is a procedural language it has six operators we have taken a quick view of these earlier in this module we will look at them at length. The select(**σ**) project(**Π**) union(**U**) set difference(-) Cartesian product(×) and rename(**P**), we will also look at few derived operations like intersection and division which can be expressed in terms of these basic operators. And each one of these operators can take one or two relations as input and they produce one relation as a result.

(Refer Slide Time: 03:35)



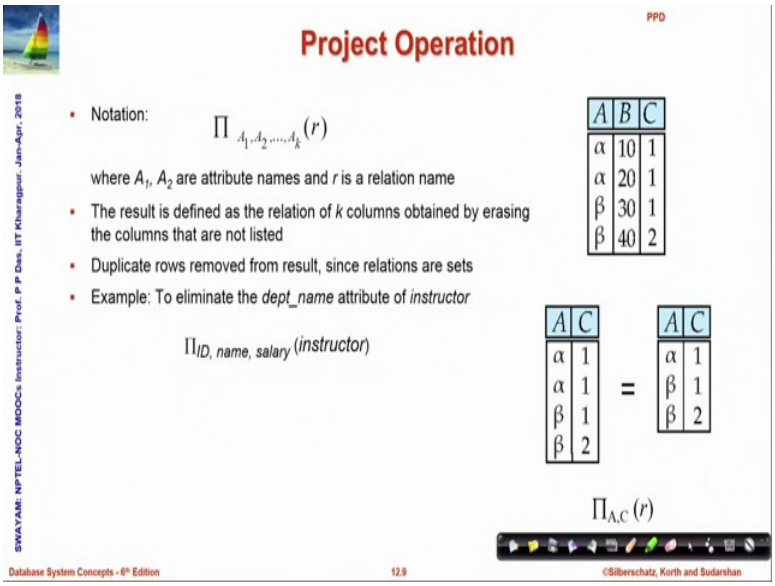
So, we start with the select operation which you know has a notation σ_p where p is a predicate it is called the selection predicate and within parentheses we have a relation r (r) on which this predicate applies. So, it is defined as a set where you collect all the tuples all the rows designated by t and you specify that **t € r**. So, it already exists in that relation and it satisfies the particular selection predicate.

So, any tuple that satisfies this predicate is included in the result any that does not satisfy is excluded from the result, p here particularly is a propositional calculus formula or expression. Where we have different terms that are connected by conjunction (^)or and disjunction (**v)** or negation (¬) or not, and each term by itself could be something like this it is an attribute operator and an attribute where operators are different comparisons operators one of the any six or a term could be an attribute operator a constant.

So, given that we can write any expression, which is a predicate and applying that we can select the tuples from the relation r which satisfy this predicate. So, here we show a simple example instructor is a relation, department_name is an attribute within, course physics is a constant or literal. So, this selection show will select all the tuples where the attribute department name is equal to physics and all the others will be eliminated for reference I have also quoted here the example that we had shown at the time of introducing relational algebra.

So, you can see that here we have a more complex propositional term propositional formula where there are two terms, the **a =b ^ d > 5**. So, in the selection result both of these conditions must be satisfied by all the tuples which feature here. So, this is the first operation that relational algebra has the select operation.

(Refer Slide Time: 06:40)

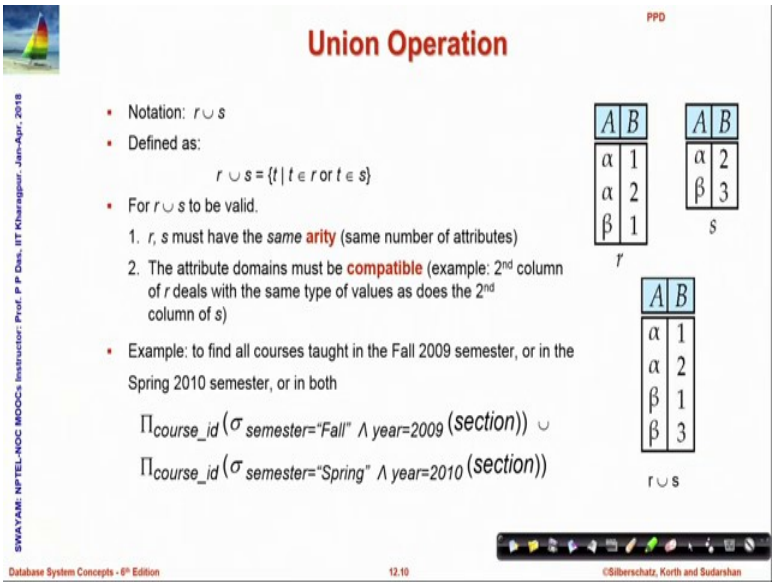


Next we move on to the second operation which is a project operation where a relation can be projected in terms of a number of attributes. So, pi(**Π**) is the notation r again is the relation and the subscript at A_1, A_2, A_k, are key attribute names k has to be at least one and these attributes will be retained in the relation. $\mathbf{\Pi}_{A1, A2, Ak}$

So, the result is defined as the relation of k columns by erasing all the columns of r which are not listed amongst this A_1 to A_k. Naturally, if you erase some columns it is possible that two rows that were distinct in those columns, but are identical in A_1 to A_k feature in the relation since every relation is a set no distinct no two copies of the same people are allowed. So, the duplicate rows will be removed from the result remind you this is in contrast to what SQL does by default where duplicates or multi sets are allowed by default here we are talking about the formal relational algebra where it is purely set theoretic.

So, duplicate rows on projection will be removed from the result. So, we have an example from the instructor relation we had seen earlier we are projecting id name and salary($\Pi_{id\_name, salary}$). So, we are removing the department name, which also exists in the same relation and as a reference you can see the example that we had seen earlier while introducing the relational algebra where projection is done from three columns A, B, C into two columns A and C and this results in at least results in two rows which are identical and therefore, in the final result one of those the duplicate one is removed.

(Refer Slide Time: 08:57)

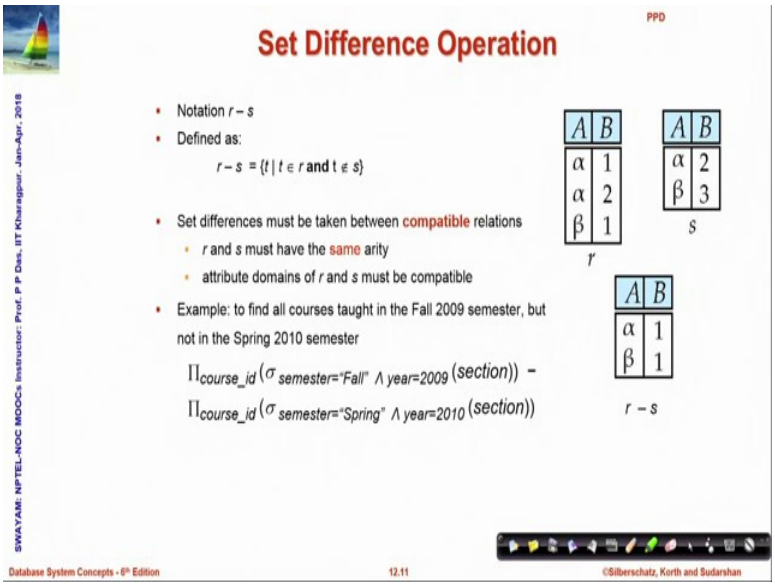


Moving on that the third operation is quite simple it is set theoretic union. So, **r U s** where r and s are two relations our set of peoples which either belong to r or belongs to s, or belongs to both, the condition is you can take union of both these relations have the same arity and the order of the attributes must satisfy that every corresponding attribute must have compatible domains. So, if we I talked about the second column of r, and if we talk about the second column of s they must be of the same type and this must hold for all columns that the union forms that is all columns of r as well as s , otherwise this operation is not defined.

So, as an example we show that to find all courses taught in fall 2009 semester, this is a this is the query where we do a selection to find all tuples which are taught, which represent courses taught in fall 2009 semester, from the section relation we do a projection to get the ids of those courses only $\Pi_{course\_id}(\sigma_{semester="Fall"\wedge\ year=2009}(section))$. And the second row tells you the courses that are taught in the spring 2010 semester $\Pi_{course\_id}(\sigma_{semester="Spring"\wedge\ year=2010}(section))$, and we do a union to get courses that are taught either in fall 2009 semester, or in spring 2010 semester, or both. This is how the union is performed and this is the earlier example repeated here.

$\Pi_{course\_id}(\sigma_{semester="Fall"\wedge\ year=2009}(section)) \cup \Pi_{course\_id}(\sigma_{semester="Spring"\wedge\ year=2010}(section))$

(Refer Slide Time: 10:54)

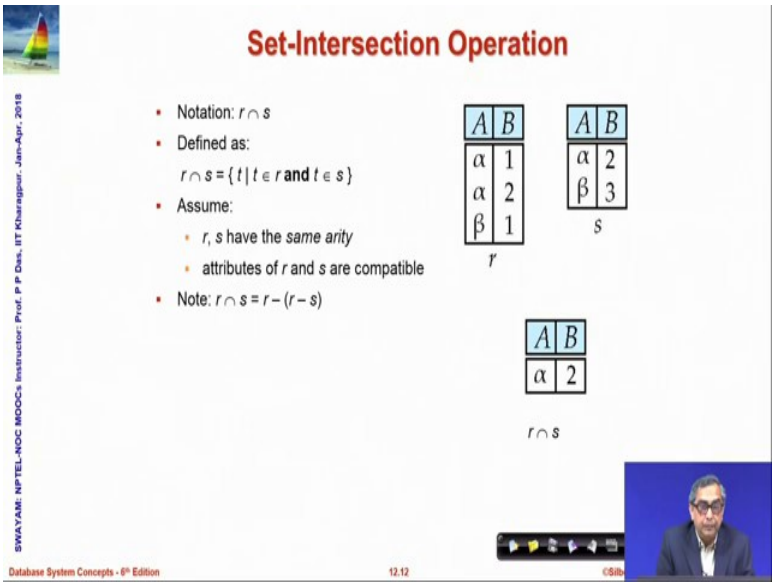


set difference is again just simple difference of sets r minus s where a tuple belongs to r and it does not belong to s. So, you remove all the tuples belonging to s that exist in r to get r - s again they must have the compatibility of having the same arity and attribute corresponding attribute domains must be compatible , this is an example to show to find all courses taught in fall 2009, but not in spring 2010.

$\Pi_{course\_id}(\sigma_{semester="Fall"\wedge year=2009}(section)) - \Pi_{course\_id}(\sigma_{semester="Spring"\wedge year=2010}(section))$

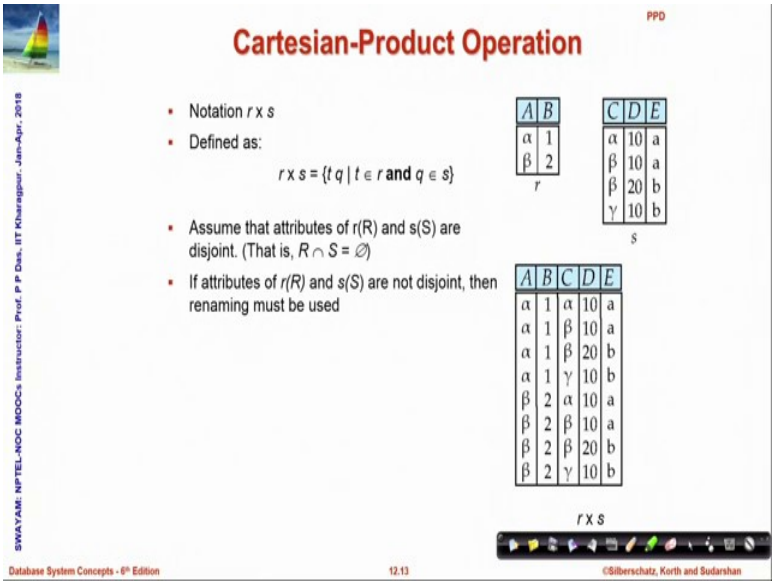
So, as opposed to union in the last slide you do a set difference to get this result. So, this is how you can use set theoretic operation to get different relational results this is also the result from the earlier example.

(Refer Slide Time: 11:49)



Set intersection can be supported is supported, but it is not a basic operation because as it is defined by all tuples which belong to both r and s. It can actually be computed by applying set difference twice **r-(r-s)** and certainly for set intersection also the same assumption about arity and compatibility of types hold and this is the earlier example.

(Refer Slide Time: 12:20)

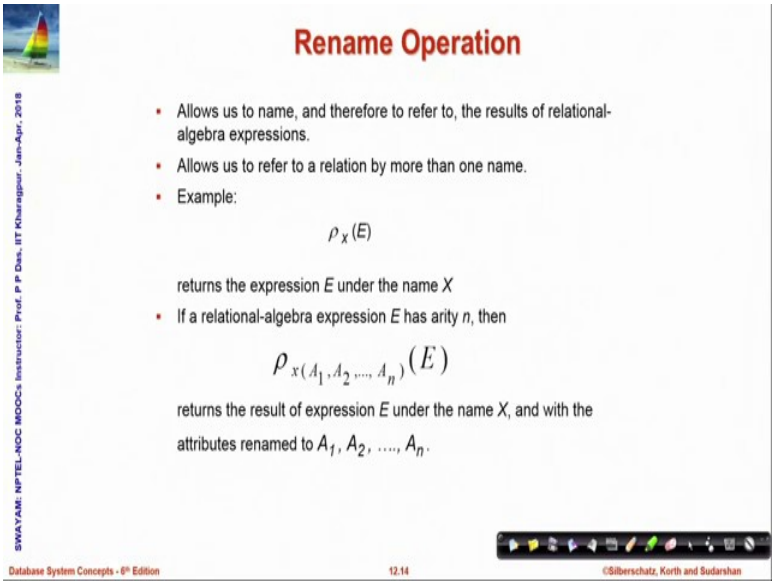


Next is Cartesian product, where we take two relations and for the Cartesian product we make we juxtapose one relation with the other. So, t is a relation from is a tuple from r(**t ∈ r**), q is a tuple from s (**q € s**) and we put them side by side to get a tq in the Cartesian

product **r** × **s**, which basically means that you compute all possible combinations of pupils from r and of s. It is assumed that the attributes of r and s are disjoint that is a schema of r intersection schema of s is null. **R ∩ S = Φ**

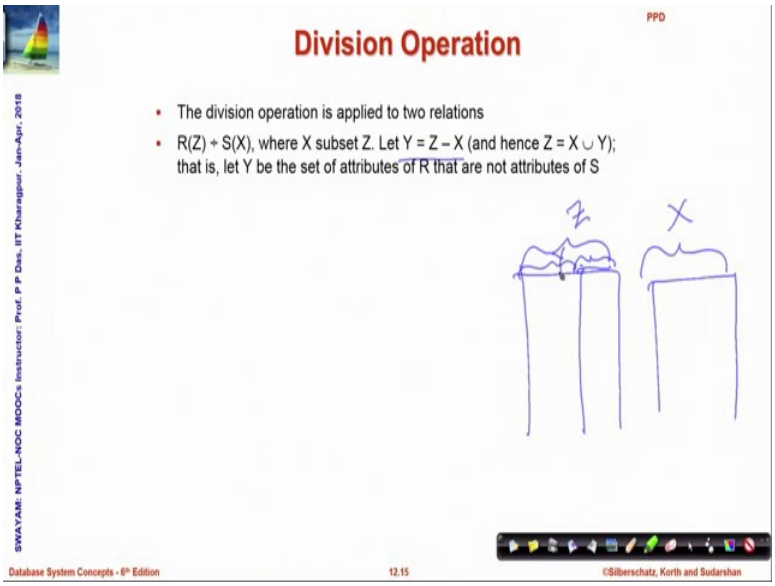
If the attributes are not disjoint then we must use renaming which we will soon see, and here is an example that we had shown earlier of r and s computing r process, Cartesian product is a very useful operation particularly for computing join as we have seen in sql already.

(Refer Slide Time: 13:36)



Rename operation basically allows you to rename some expression attribute into another. So, the operator is **P** and you have an expression to which you give the name x and that is how the renaming of you can have multiple attributes of x as well.

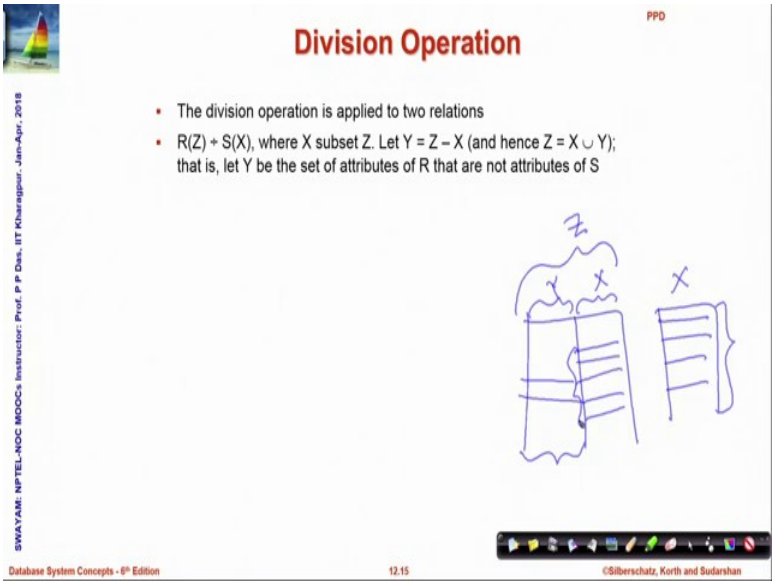
(Refer Slide Time: 13:58)



Division is a is another operator in relational algebra that can be applied between two relations, but it is a derived operation. So, it says that if I have, so let me just show you by, by a little bit of sketch that if I have two relations which are which has a set of attributes z and s which is a set of attribute x, such that actually the set z is a superset of x. So, z has more attribute the relation r has more attributes.

So, if you take the difference of attributes between z and x and call it y, Y=Z-X then we are interested what happens on these remaining set of attributes y..

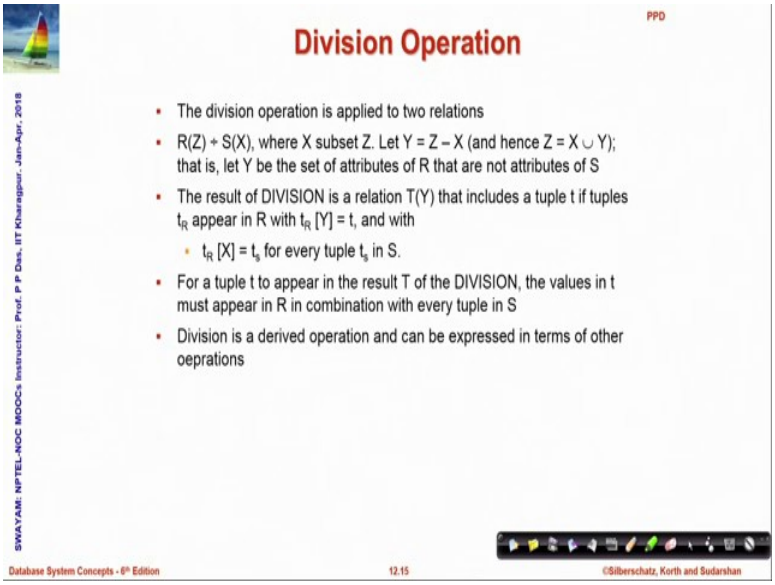
(Refer Slide Time: 15:02)



So, this is what, we have this is my X set of attributes this is where X occurs this difference is Y this whole set is Z. Now in this what we want is we want to in the output we want a relation having only the y attribute such that for every tuple in that relation if I consider all the tuples of s then their cross product must be a part of r.

So, for every tuple here if there are say four tuples here, a tuple here must have all these four tuples along with it in the result. If it does not have any one or more of them then that tuple will not feature in the final result.

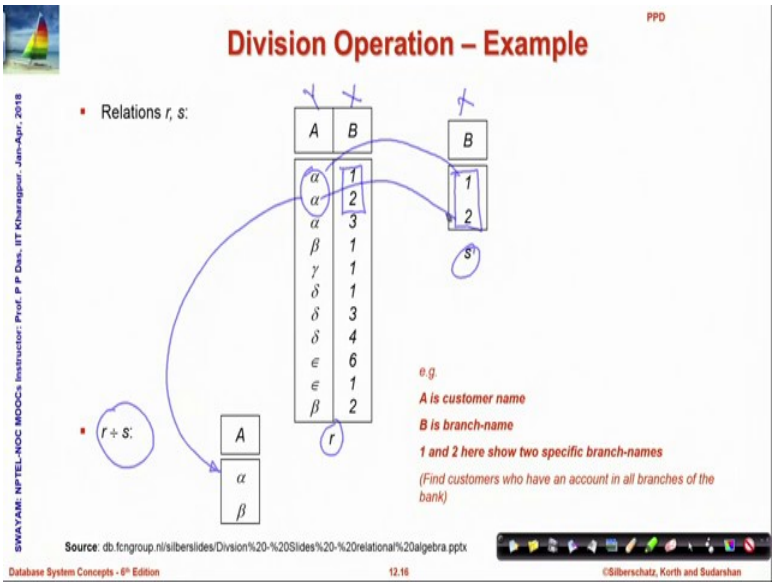
(Refer Slide Time: 15:58)



So, the result of a division is a relation T(Y)that include tuple t if tuples tr that is the part of the tuple the tuple that appear in r and on that on the y part the difference part it matches. So, that you have that t_r [**X**] = t_s where t_s actually exist in s.

This must happen for all tuples in s, so division is a very good interesting operator which is often required for coding different queries. So, it is a derived operation.

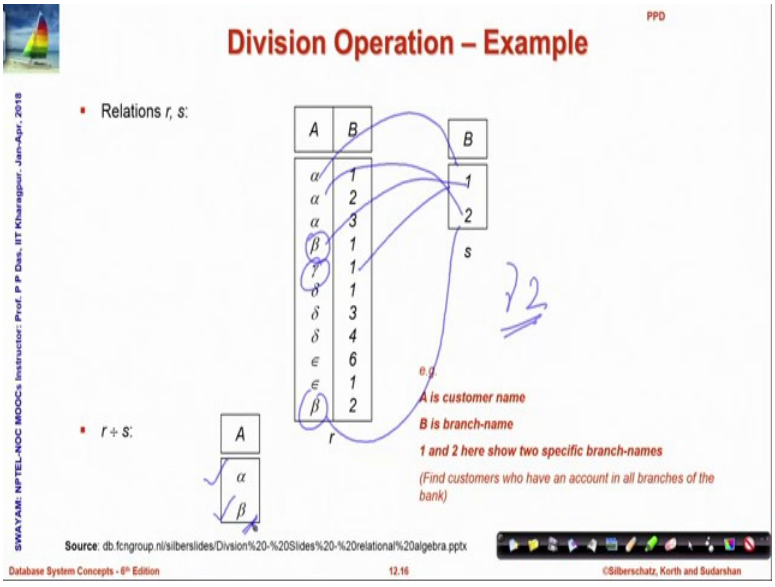
(Refer Slide Time: 16:44)



Let us take an example, let us say this is this is the relation r and this is a relation s and I am trying to compute r ÷ s.

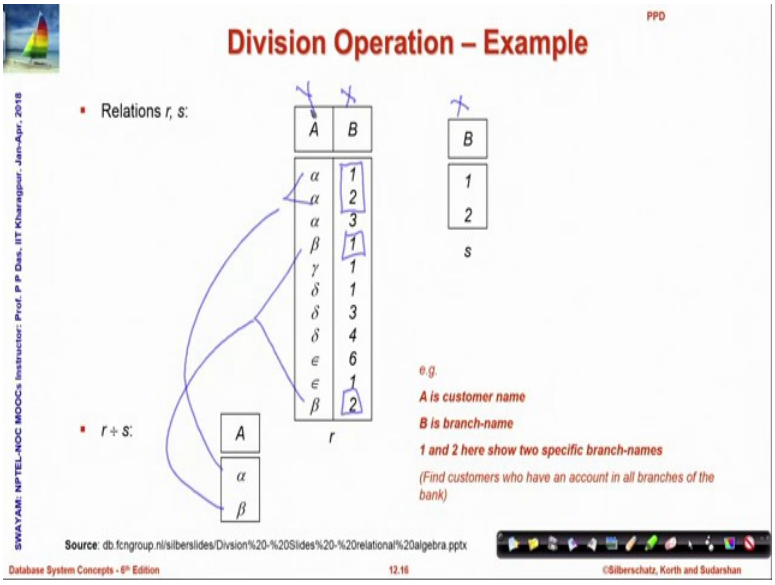
So, what I want is over the attributes of this is therefore, X this is x this attribute y attribute set Y. So, all over X all the values that I have, I must have those values in the relation r, if I do then the attribute the particular tuple matching on the attribute Y goes onto the result. So, **α** goes onto the result because you have both α a, **α 1** as well as **α 2** in the set r, in the relation r.

(Refer Slide Time: 17:44)



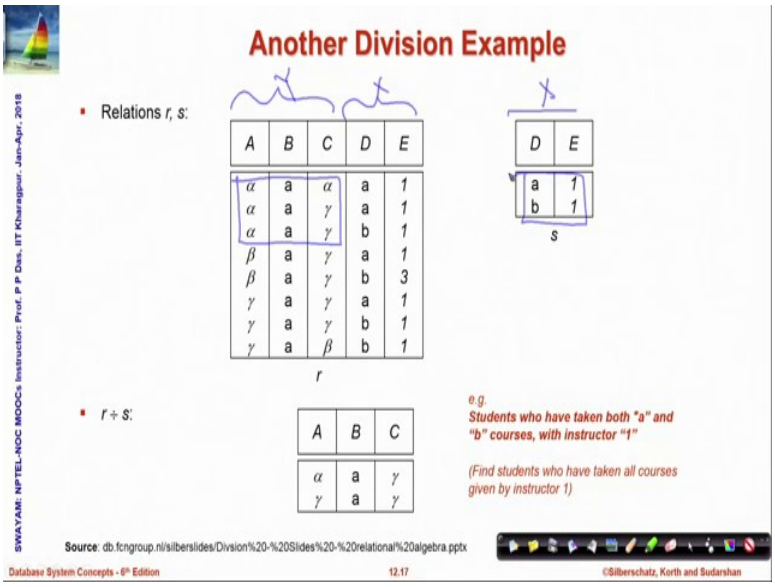
β1 is there and β2 is also there, so β also goes into the relation α goes in because 1 is there 2 is also there, but γ will not go in because I have γ 1, but I do not have a tuple γ 2 in r if I had γ 2 in r that will go in the result.

(Refer Slide Time: 18:13)



So, if I can say it again the whole of the relation s must happen over the X attributes of r, consider these two together. If that happens then the attributes on y the tuples would be chosen and that is how we get the result having α and β.

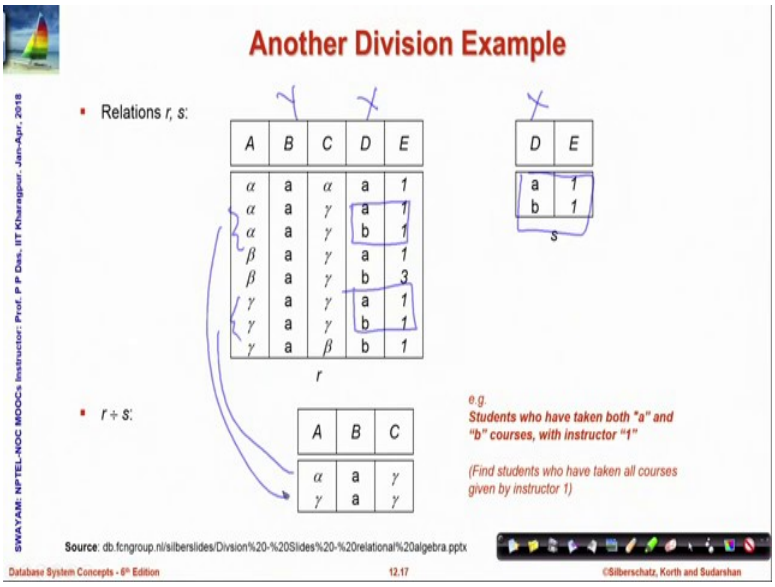
(Refer Slide Time: 16:47)



Let us look at one more example, so this is got r has five attributes this has X as two. So, this is this is two attributes X, these are three attributes Y and what I have to look for is those tuples in r where the values over Y would be same and I should be able to get the whole table of X over whole table of the relation S over the X attributes. So, if we look at here this is a 1, b 1, a 1, b 1, here these are identical.

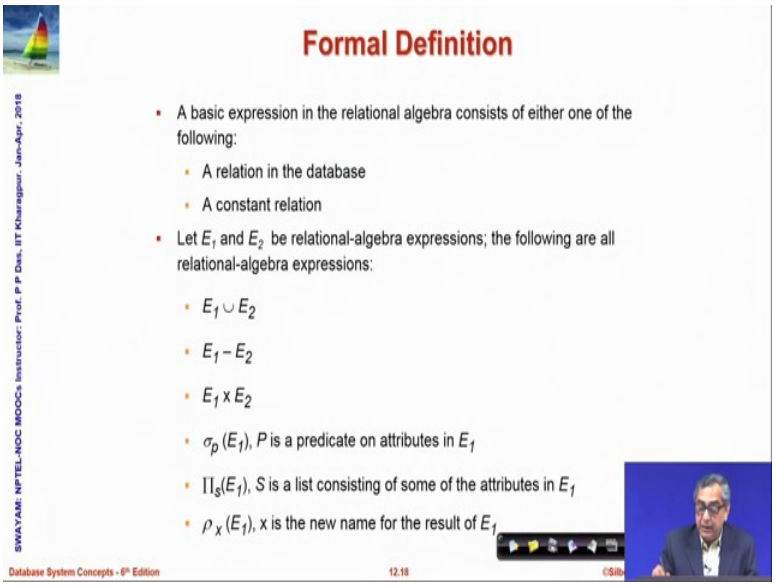
So, this particular tuple will go into the result if I look in here this tuple will go into the result, but if I consider this tuple β a γ which has a 1 over d, but β a γ does not have b 1 it has b 3, so it will not go into the result. So, if you conceptually look at that is the reason this is called a division. So, you get this here you get this here. So, this is like the way we divide that this is the whole and wherever it goes in if the tuples that are identical on the wide set of attributes we will collect them into the final result.

(Refer Slide Time: 20:30)



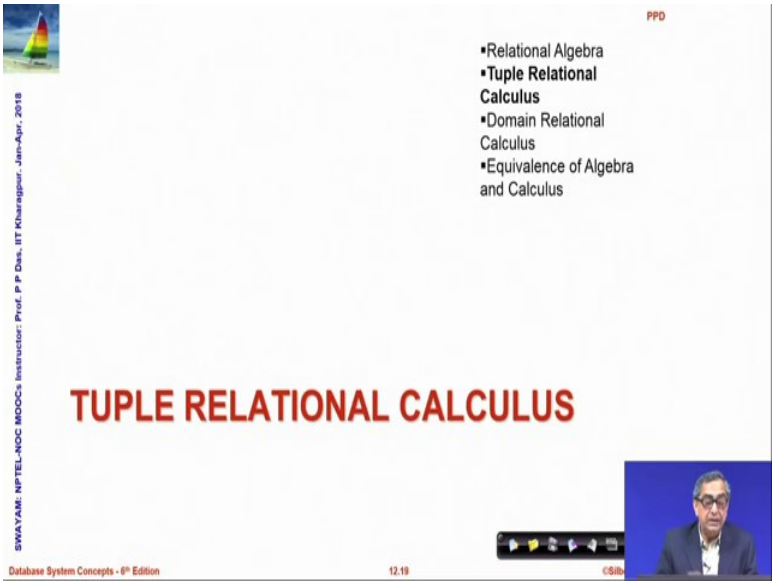
So, this is the division operation which by which we can compute the students who have taken both a and b courses instructor 1 will be found out from this division operation.

(Refer Slide Time: 20:51)



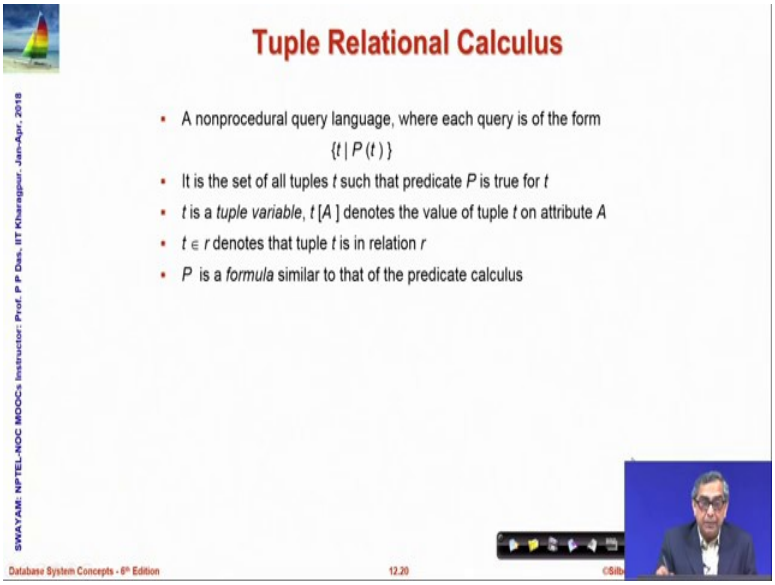
so formally speaking a basic expression in relational algebra consists either of a relation in the database, which is a instance or a constant relation which does not change which is given. And then we have six operations of union set difference cross product, selection projection, and renaming that can give us all sorts of different relational algebra formula and also the derived operations and whatever we have seen of sql can be expressed in terms of this relational algebra formula.

(Refer Slide Time: 21:30)



Now, relational algebra is not something totally unique the same thing can be done in terms of other formulations also.

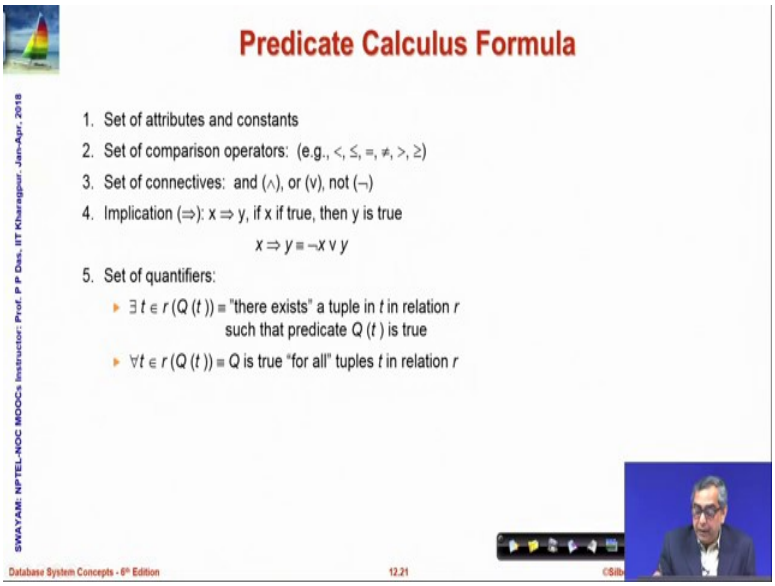
(Refer Slide Time: 21:41)



A second formulation which is also used is known as tuple relational calculus, which is non-procedural relational algebra was procedural because you are actually doing the explaining what the operations or you are detailing out what the operations are in tuple relational calculus you are specify what the condition is you are specifying what this condition is.

So, those tuples which satisfy this condition form the relation, so p is a predicate. So, whatever t satisfies the predicate are included and if a is an attribute then t [A] will denote the value of the tuple on attribute A, A could be a single attribute it could be A set of attributes also and t € r, P is as I said it is a its a predicate calculus formula.

(Refer Slide Time: 22:43)

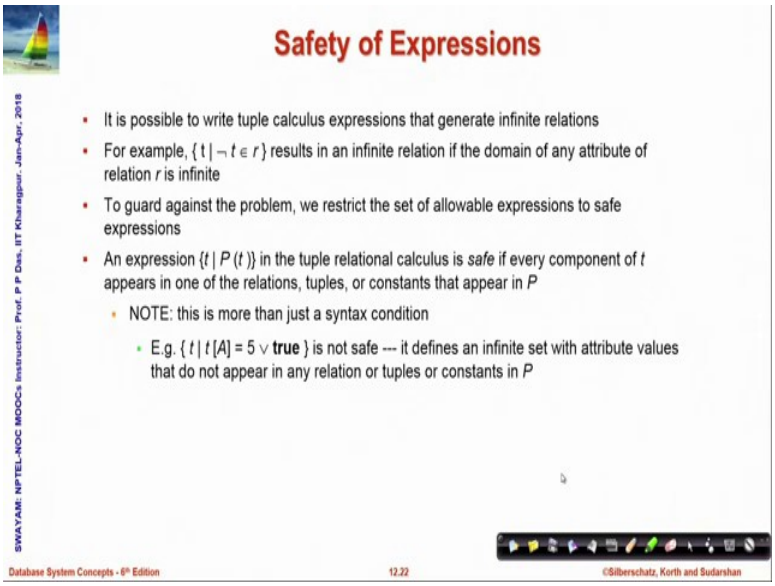


So, it could be a set of attributes or constants this I am just included for your help if in case you have become rusted with predicate calculus you can refer to a predicate calculus as a set of attributes and constant. It has set of comparison operators the six of them, there are a set of connectives these are all same as the propositional calculus there is ⇒ which says if x is true then y is true if x is false then the whole thing is true vacuously.

And what makes it primarily predicate calculus is a fact that it has existential quantifier, which says that the formula there exists t ∈ r Q(t) holds if I can find at least one tuple t which satisfies Q(t).

Similarly, there is a universal quantifier where I will say that for all t € r, Q(t) is true if for all tuples of r, t satisfies Q(t). So, this in tuple relational calculus all conditions all predicates are formula of this kind and with that we can represent any.

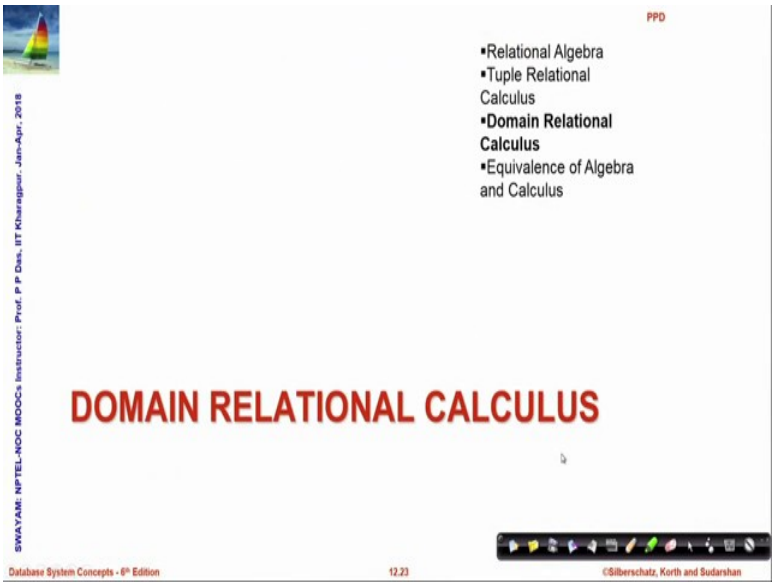
(Refer Slide Time: 24:07)



Relational set in full there is a word of caution because it is possible to write tuple relational calculus expression that can potentially generate infinite relations. Now, infinite relations are naturally not representable for example, if I write simply this that r is a relation and I write this predicate that ¬ t € r, which is basically complement set of r now a complement set of r potentially may be infinite if the domain is infinite..

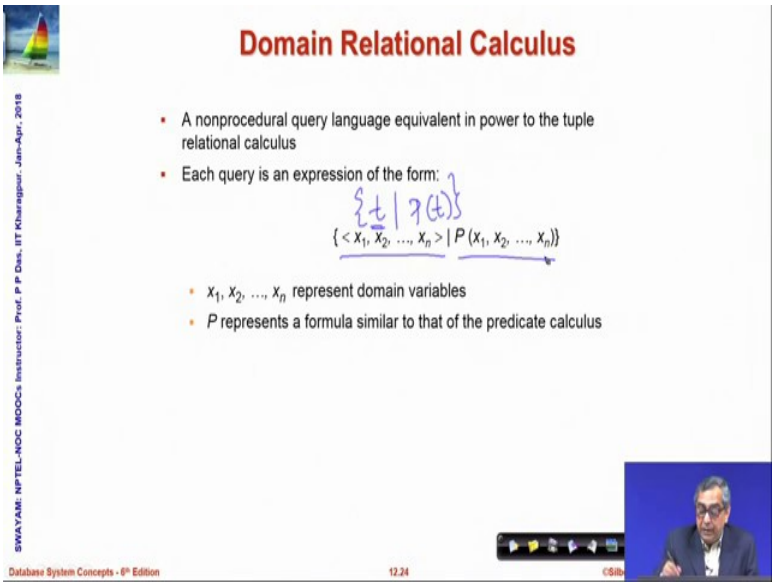
So, such expressions tuple relational expressions are not acceptable as a part of the design. So, whenever we want to do this we would like to guard this by putting some additional condition and we have to make sure that any expression that we have in tuple relational calculus is a safe expression, in the sense that it does give me finite number of tuples in the relation.

(Refer Slide Time: 25:14)



A third formalism that exists that is used is known as domain relational calculus.

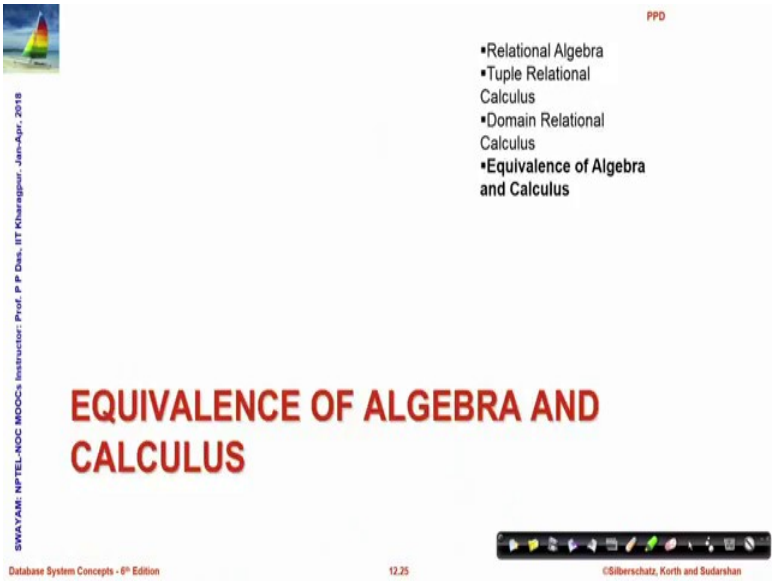
(Refer Slide Time: 25:25)



Which is also non procedural and equivalent in power to tuple relational calculus again, which is very similar to tuple relational calculus the only difference being if you just recall tuple relational calculus. We are writing collection of tuples t such that P(t) that is the predicate P is satisfied by t here, instead of writing a tuple variable t we write expand it out in terms of all its components..

So, we write the values of the different components of t over different n attributes and write it as a n tuple and so here instead of having one variable t we have n variable $<x_1, x_2, x_3 \ldots. x_n>$ and therefore, the predicate is formed of this n variables where x_1 to x_n are represent the different domain values, domain variables and that leads to the reason for the name domain relational calculus.

(Refer Slide Time: 26:28)

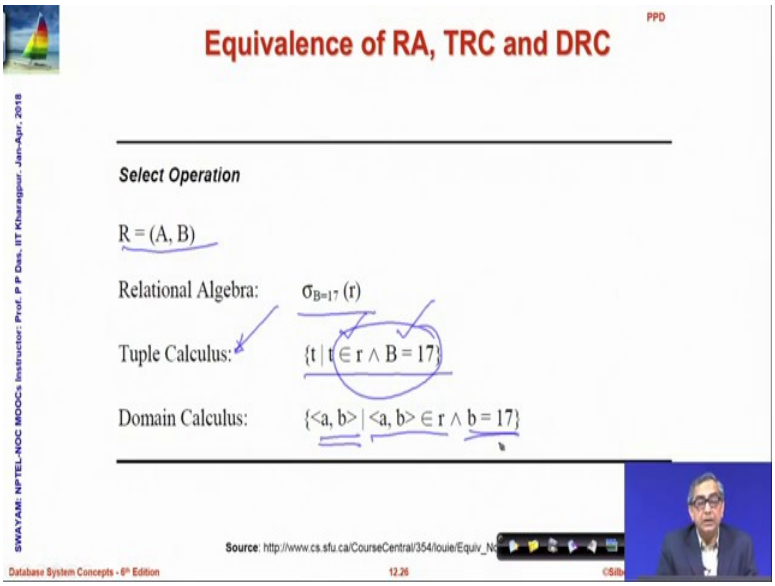


Now, of the three formalisms that we have seen we will not go into direct mathematical proofs, but in the next couple of slides, I just show that they are equivalent in nature. What means that if I can write an expression in relational algebra then it is possible to write an equivalent expression in tuple relational calculus and in domain relational calculus and vice versa..

So, if I can write an expression in any one of these formalisms then there are equivalent expressions in the other two formalisms as well which is probably very easy to see between, tuple relational calculus and domain relational calculus because 1 is just representing the whole tuple as a single variable whereas, the other is representing it in terms of n domain variables.

So, their equivalence is pretty much very similar the fact that your predicate calculus formula has to change, but it is not, so obvious for the equivalence between relational algebra and the calculi.

(Refer Slide Time: 27:41)

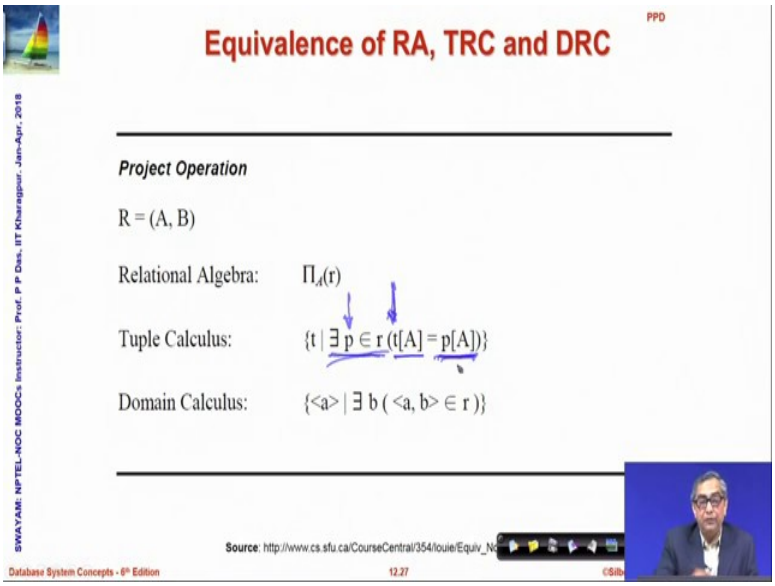


So, we just show a few examples of the basic operations for example, say select operation. So, I am just not showing the proof, I am just giving you some example cases to show through a relation r has two attributes A, B this is what you wanted to write in relational algebra that you want to collect all tuples where b =17.

Naturally in tuple relational calculus you can easily write the first condition is you are doing it on r. So, t must belong to r and your condition is B should be 17 {**t | t € r ^ B = 17**}. So, this predicate will represent the same set or the same relation as in tuple calculus in domain calculus there are two components a and b. So, you have to say component taken together must belong to r and the component b must be equal to 17. **{<a , b>| <a , b>€r ^ b=17 }**

So, you can see that it is pretty straightforward to see the equivalence between a relational algebra expression involving select and the corresponding tuple calculus or domain calculus expressions this is through an example, but you can certainly easily generalize this as a proof.

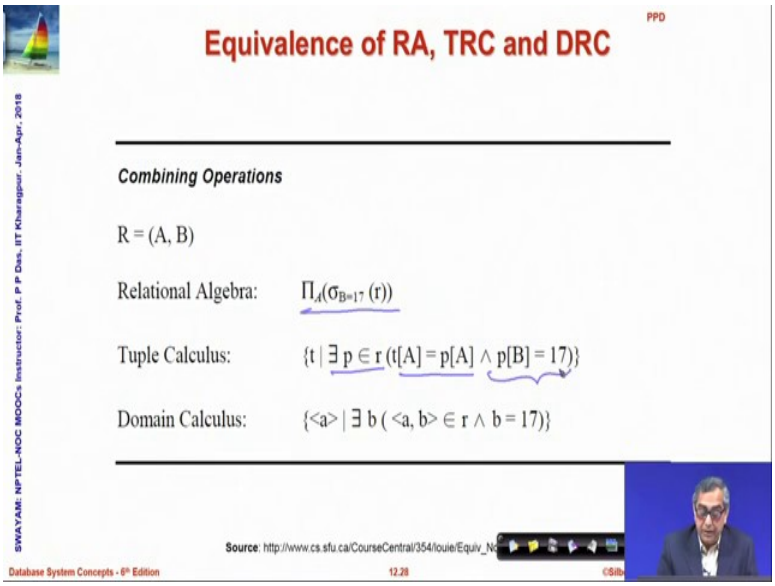
(Refer Slide Time: 28:49)



Similarly, for projection if we do a projection on a then all that we are trying to do is we are trying to create a new relation where only the a attribute exists. So, in the tuple t the a attribute exists and if I have projected and got this tuple t then in my original relation r there must be some tuple p such that on the attribute a they match they are same.

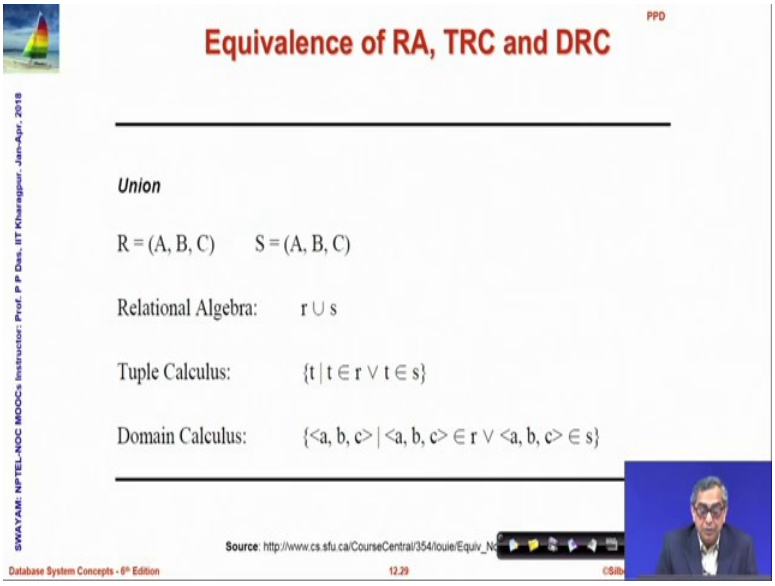
So, it is the same thing in relational algebra we said that keep a and erase everything else here we are saying that if we have been able to get a tuple t which has a value t a then there must be a tuple p in r, which has the same value over the same attribute. So, this condition is equivalent representative of the projection, and the same thing can be written in domain calculus you can go through it carefully and convince yourself.

(Refer Slide Time: 29:59)



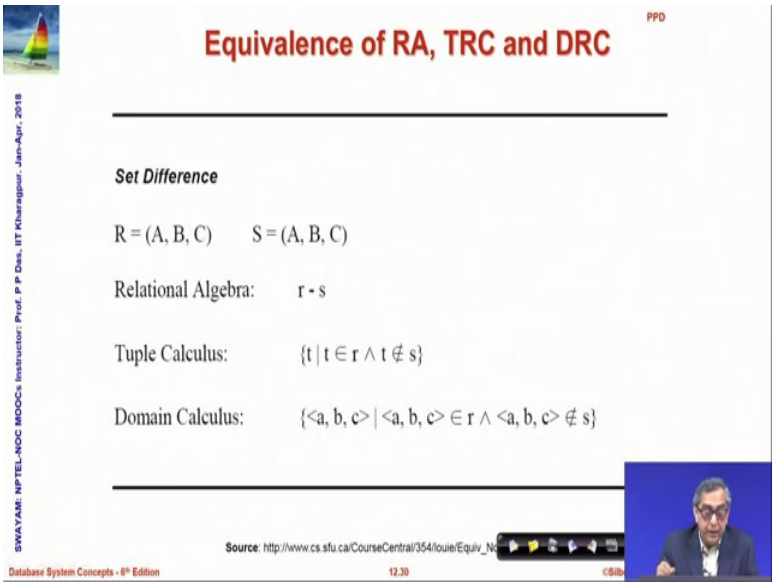
You can combine this as in the relational algebra as well as in tuple calculus. So, here you apply one relation one operation and then the other one selection then projection here you are combining this is part of projection this is also part of projection, but this condition has come from the selection and get a total predicate calculus predicate which will give you the tuple calculus expression for this combined expression of relational algebra , domain calculus will certainly happen in a similar manner.

(Refer Slide Time: 30:34)



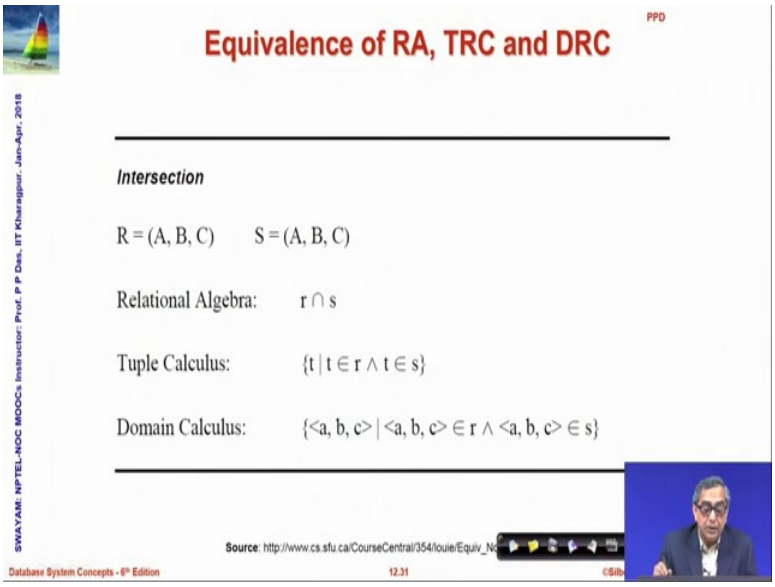
Union certainly straightforward, so you can do it yourself.

(Refer Slide Time: 30:40)



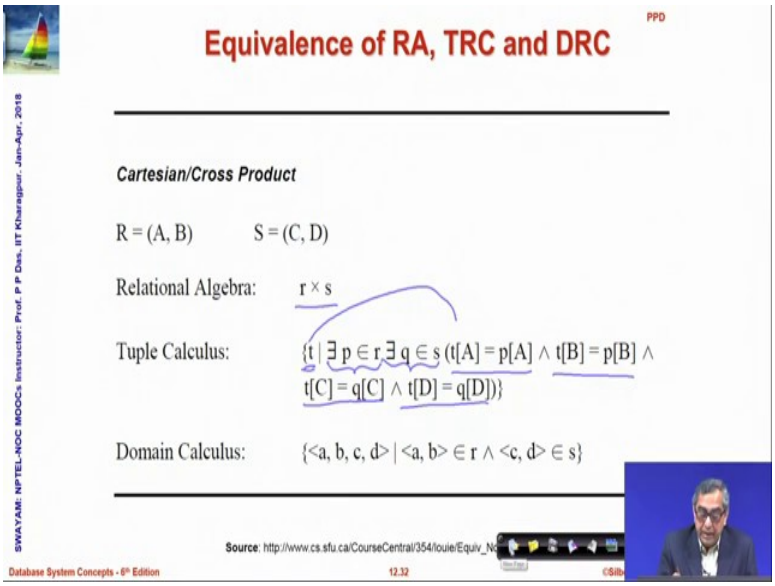
Set difference is again very straight forward because that is. In fact, in set theoretically whatever operations we have their relational algebraic definition itself is a tuple calculus formula you can expand them out and write in the domain calculus as well.

(Refer Slide Time: 30:57)



Intersection plays out in the same way tuples that belong to both r and s.

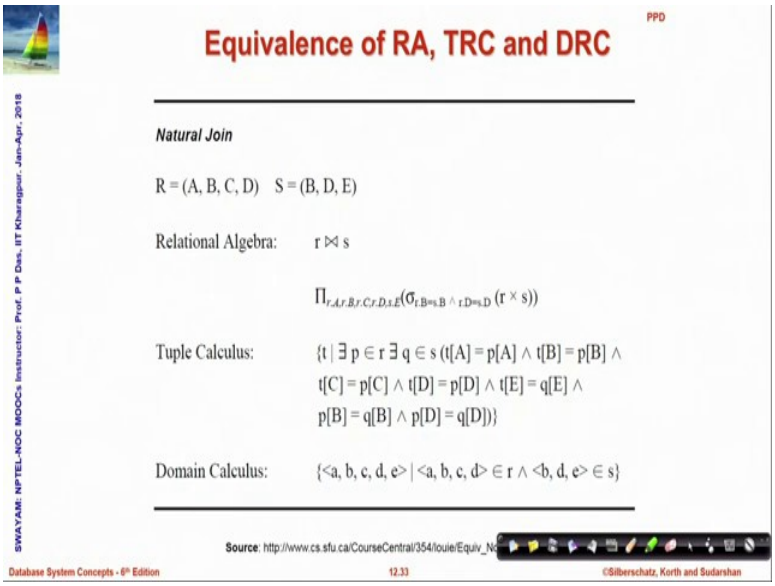
(Refer Slide Time: 31:04)



Cartesian product is a little bit more involved because all that you are saying here is if I have a Cartesian product then if that product has a tuple t. Then there must be a tuple p in the relation r (p € r)the left relation there must be a tuple q in the in s (q € s) the right relation. So, that the final tuple t matches p on the a attributes the b attributes that is attributes of relation r and the components of t matches the tuple q in the attributes of s.

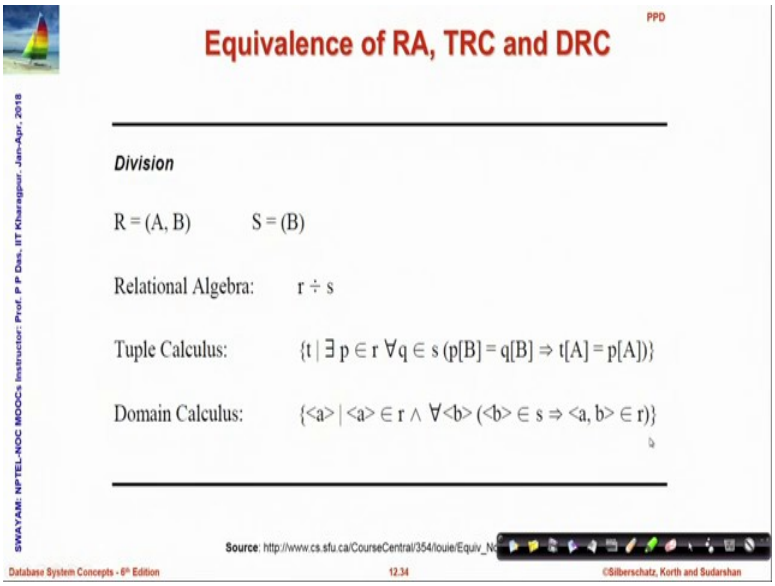
If all these conditions happen together then naturally this tuple t is a valid tuple for the Cartesian product. So, you could take examples and work this out and convince yourself that these are really equivalent.

(Refer Slide Time: 31:58)



We can define a natural join in a similar way, which I will leave it as an exercise for you to convince yourself that this relational algebra expression for natural join indeed has similar equivalents in tuple and domain calculi.

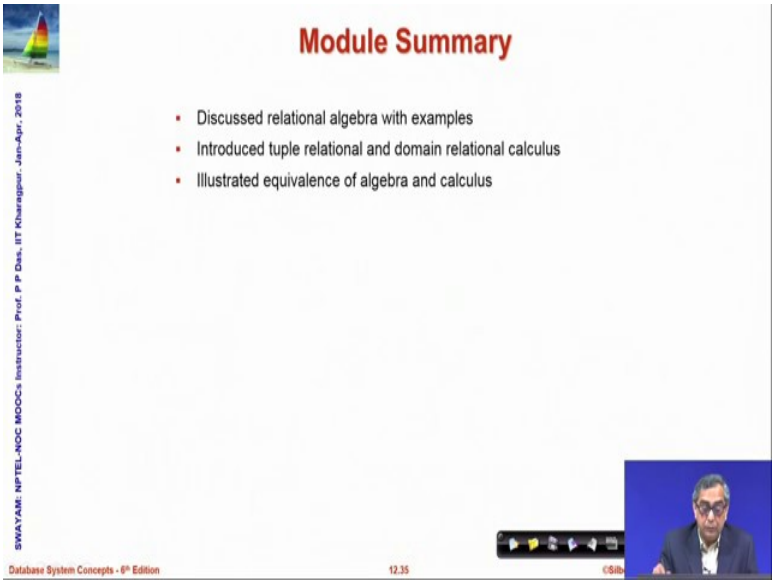
(Refer Slide Time: 32:17)



Division we just showed as a derived operation, we have not showed how in relational algebra you can write division using the other operations. I will leave that as an exercise as well, but here what I show is in tuple calculus how you can write division using the quantifiers.

Here you can see that here for the first time we do need to use the universal quantifier to make sure that while I divide that the whole of the table of s must be available against the part of the tuple part of the y attributes as we said that will be collected in the result.

(Refer Slide Time: 33:02)



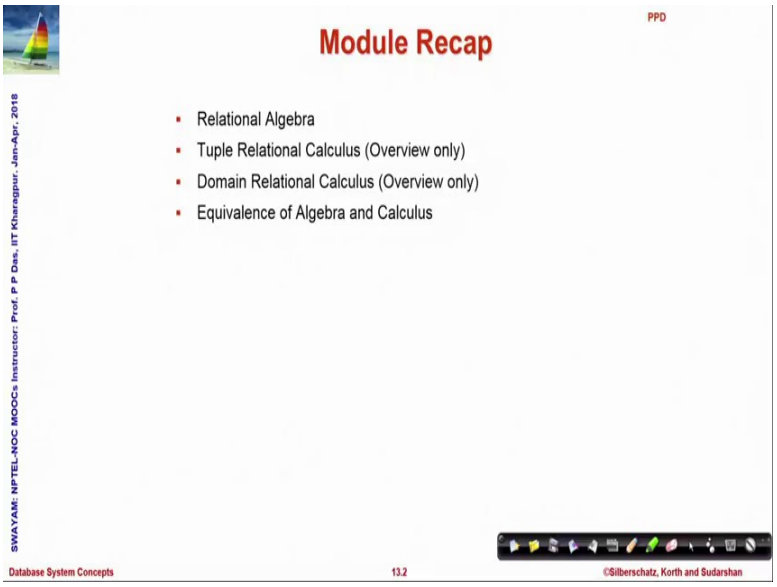
So, to summarize we have discussed primarily the relational algebra with some examples, we have introduced the tuple relational and domain relational calculus and through a set of examples. We have shown that we have illustrated that the algebra and the calculi are equivalent and I would request you to work out more examples to understand the equivalence, or if you are really enthused please try out proving their equivalence formally as well.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture – 13
Entity-Relationship Model/1

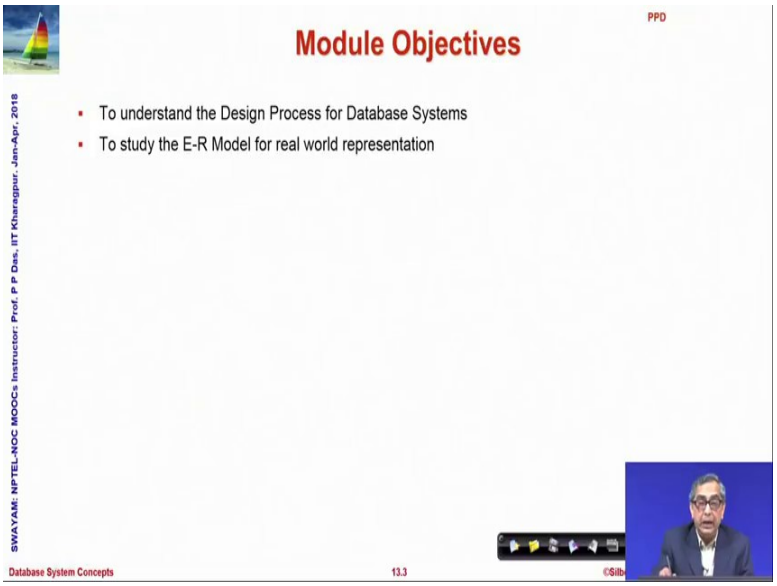
Welcome to module 13 of Database Management Systems. In this module and the next 2 we will discuss about Entity Relationship Model.

(Refer Slide Time: 00:34)



So, far we have had a good look into the SQL language, the query language and it is formal basis in terms of relational algebra and calculi.

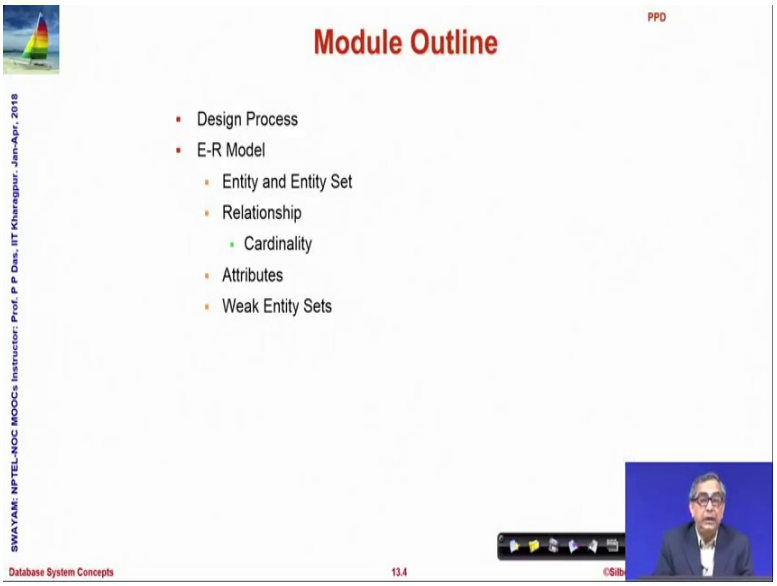
(Refer Slide Time: 00:44)



In this module we will un try to understand the design process for database systems, because so far whatever we have done we have assumed that the schema is known to us, that some instance is given to us and then we have tried to extract different query information from the relation; but now we will look into how do we model the real world and actually get into the design process.

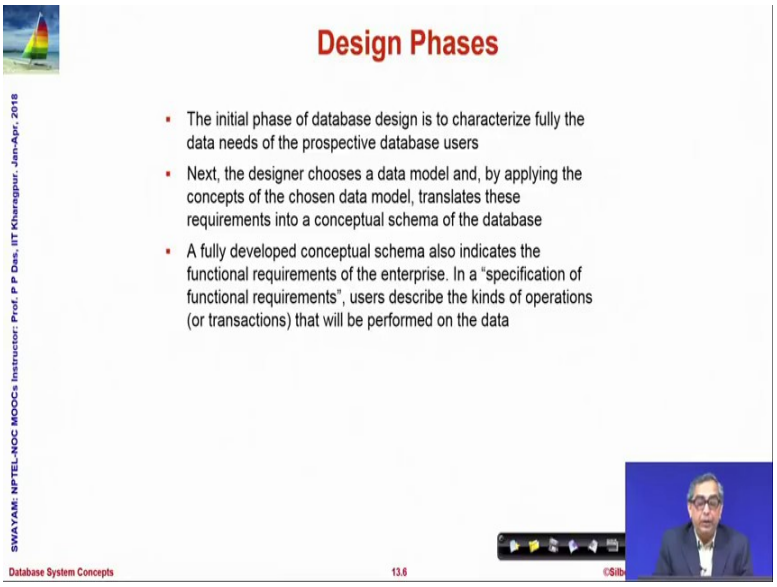
So, after an overview of the design process we would study entity relationship model, which is used to represent the real world whatever exists in the real world that will have to be represented for our use and final representation in terms of different relations.

(Refer Slide Time: 01:32)



The design process at an abstract level.

(Refer Slide Time: 01:43)



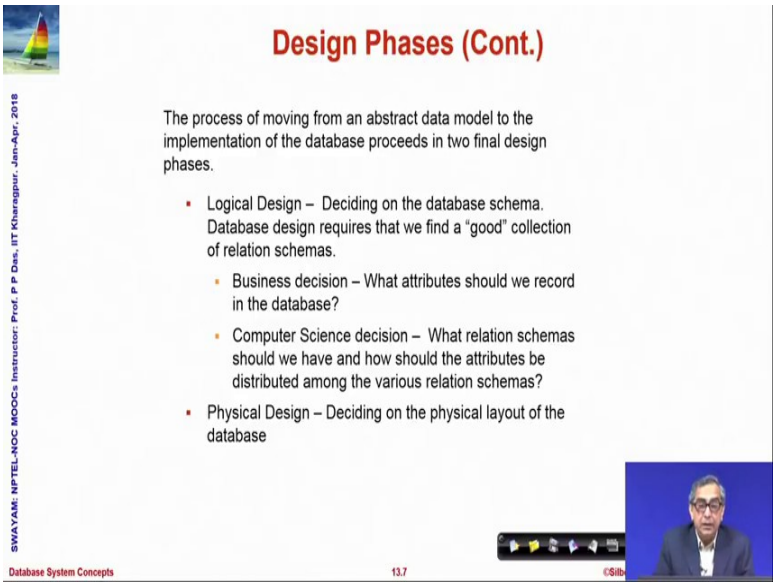
The initial phase of database design certainly has to characterize what data is required to be maintained for an enterprise. So, whether I am doing, if I am doing an university database naturally we will need to identify that what are the data needs the students need to be described, the instructors need to be described, the courses sections time slots grades examinations etc; but if I am trying to deal with an world which is say railway

reservation, then I will need to deal with stations trains date (Refer Time: 02:26) the different classes of coach that the train has and so on.

So, the initial phase is to characterize the data requirement next the designer has to choose a data model because, unless we can we cannot deal with a natural language or English kind of description and work towards getting a particular schema.

So, we will need to use a data model and apply the concepts of the data model that we choose and translate the requirements into what is known as a conceptual schema of the database which is not a not a very concrete 1. But, a conceptual one this is what grossly what I want to do and a fully developed conceptual schema will indicate my functional requirements, in terms of what usually is called a specification of functional requirements system requirements. If it will specify what kind of users will be involved what kinds of operations transactions will be performed and so on.

(Refer Slide Time: 03:43)

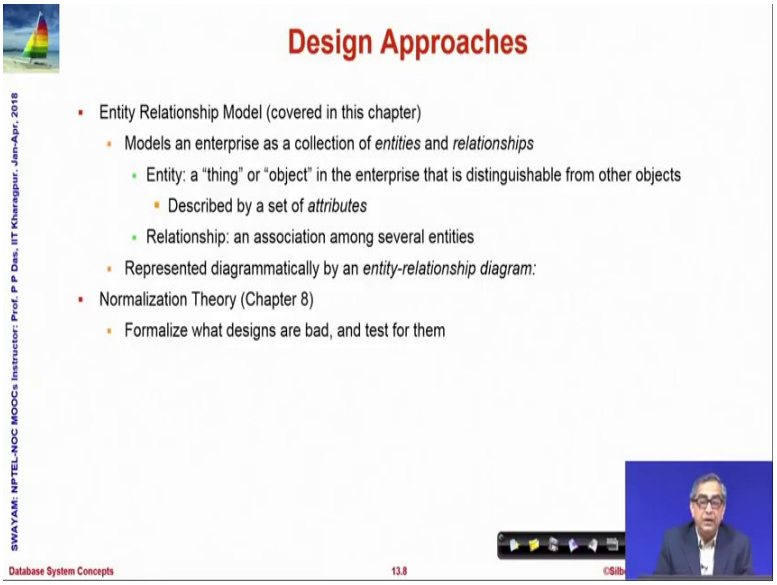


Now, once we have that kind of a conceptual model that abstract that is a conceptual more abstract data model, we will go to the next phase of the design which is finding out what is the more concrete design through a process of logical design. In the process of logical design we will first decide on the database schema, we need to decide on what is a good schema.

So, there are principles to say that what is good and what is not so good, we need to make business decisions to find out which attributes we record in the database; we need to make computer science decision as to how the relational schemas will be interrelated between themselves, how the attributes will be distributed and at a last phase we need to also decide on the physical design which will tell us what is the physical layout of the data.

So, conceptual design refined into logical design finalized with physical design is our gross process of design.

(Refer Slide Time: 05:06)



Now, in this for the conceptual design we primarily follow a model called entity relationship model; that tries to identify the collection of entities and relationships. An entity is nothing, but it is an object is a thing that is distinguishable from other objects. So, if I say that student is an entity then the student is distinguishable from another entity course both of them are distinguishable from a third entity instructor and so on.

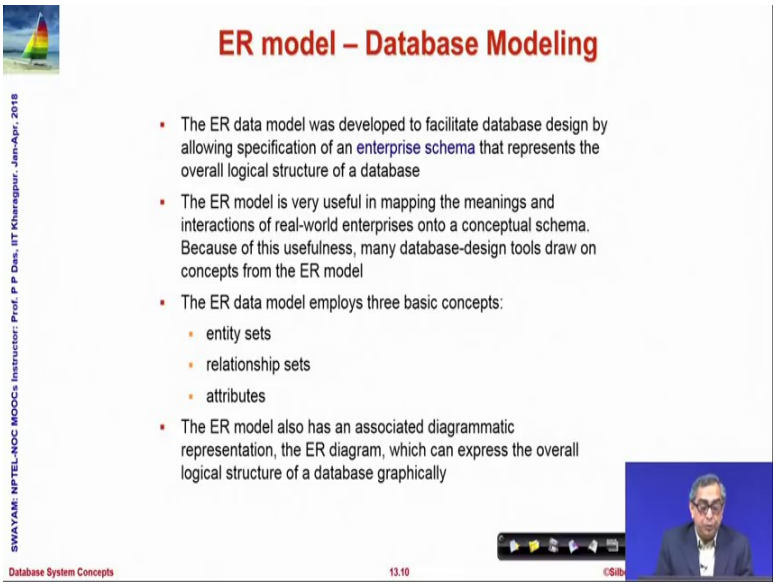
So, every entity for the purpose of distinction is described by a set of attributes or properties and these entities will have relations between them. For example, you can say that a course will be attended by students; students will be advised by instructors. So, this attended by advised by these are relationships or association between several entities and the model which represents initially diagrammatically and then in textual form this kind

of relationship is known as the entity relationship model or the entity relationship diagram.

We will then use it to get a relational set of relational schema which subsequently we normalize; the normalization is nothing but refinement of the design which improves a design to make it better in terms of correctness, in terms of ease of manipulation performance and so on. So, it basically removes bad designs from the database and converts them into good designs; we will talk about this normalization theory later in the course. Right now we are interested only in the entity relationship model, which will be used for conceptual design and then will give us the basis for the logical design in terms of the schemas.

So, let us take a deeper look into the entity relationship model and entity relationship model as I said is developed to facilitate the database design.

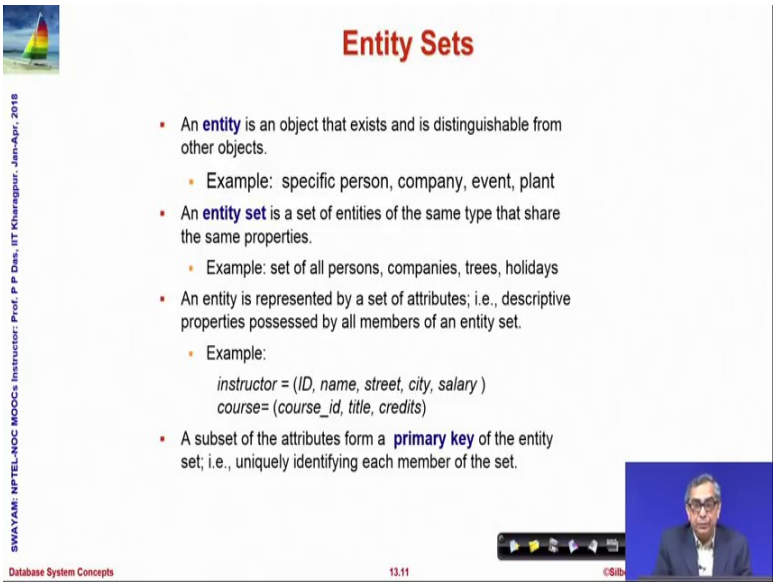
(Refer Slide Time: 07:45)



Get the overall logical structure it is useful in mapping the meaning and interactions of the real world in terms of certain diagrammatic schemas and it employs 3 basic concepts entities or entity sets we talked about entities, all entities that share the same set of properties like if student is an entity, then the collection of student is an entity set a instructor is an entity collection of instructors is an entity set.

So, all entities in an entity set will share the same set of attributes, will have relationship sets which define relationship between multiple entity sets and certainly in the process will use make use of attributes. These are the 3 key components of an ER model it also has a ER diagram as we will show soon.

(Refer Slide Time: 08:51)

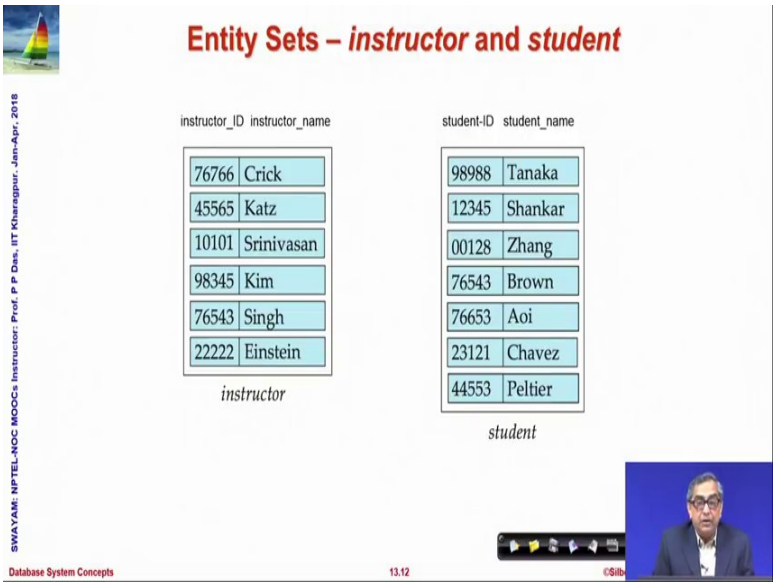


So, as already defined entity is an object that exist and is distinguishable from other objects, entity set is a set of entities of the same type that share the same properties and an entities is represented by the set of attributes or properties that describe it.

So, when we say instructive for example, if we say here these are my attributes you have already learned this in terms of studying SQL. So, it has there is 5 attributes and these 5 attributes together or the values of these 5 attributes for a particular instructor defines my entity set instructor, collection of these attributes define my entity set courses. So, these are my different entity sets that exist that can be defined.

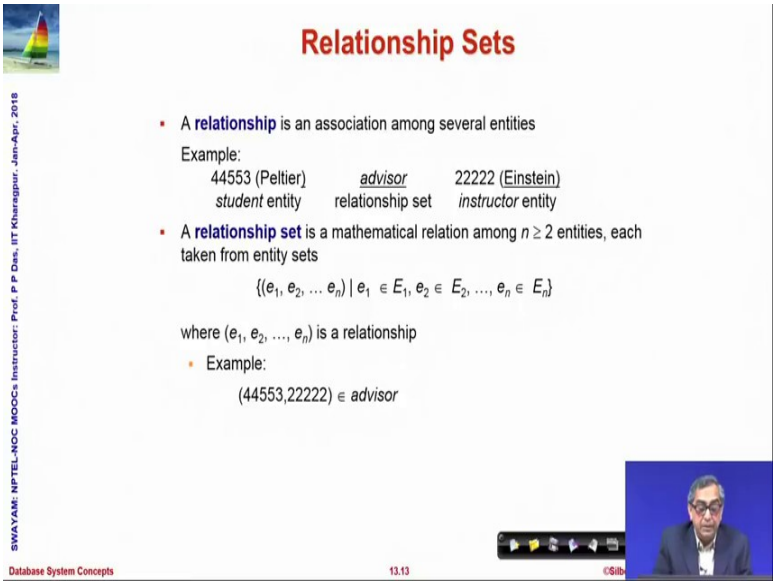
So, a subset of attributes in the entity set forms a key called the primary key, which can uniquely identify every entity in that entity set we have already been familiar with this concept of primary key the same concept continues.

(Refer Slide Time: 10:00)



So, these are examples of entity sets instructor with 2 attributes and student with 2 attributes as well.

(Refer Slide Time: 10:14)

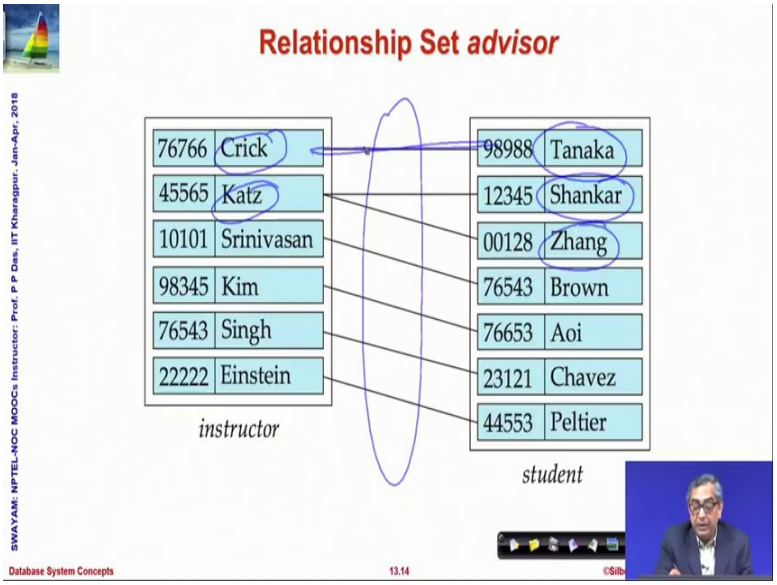


A relationship is an association among two or more entities, so here we have an entity here shown as a student this is a student entity, identified by the student id which is a primary key in the student entity set. We have an instance of an instructor entity identified by the id of the instructor Einstein, which identifies any instructor uniquely and then adviser is a relationship set which relates these 2.

So, what I we want to mean is if I say adviser relates 44553 to 2222, what I want to mean is peltier the student peltier is advised by the instructor Einstein. So, whenever we relate two or more entity sets like this we get relationships. So, a relationship is a mathematical relation Emma more than two or more entities, each taken from the entity set.

So, you can see that it can have components e 1 e 2 e n, n entity sets and each entity e 1 should belong to entity set capital E1, e2 should belong to entity set capital E2 and so on and is called a relationship we have already seen the advisor relationship as above. {(e1,e2,e3,e4….en)| e1€E1, e2€E2, e3€E3…..}

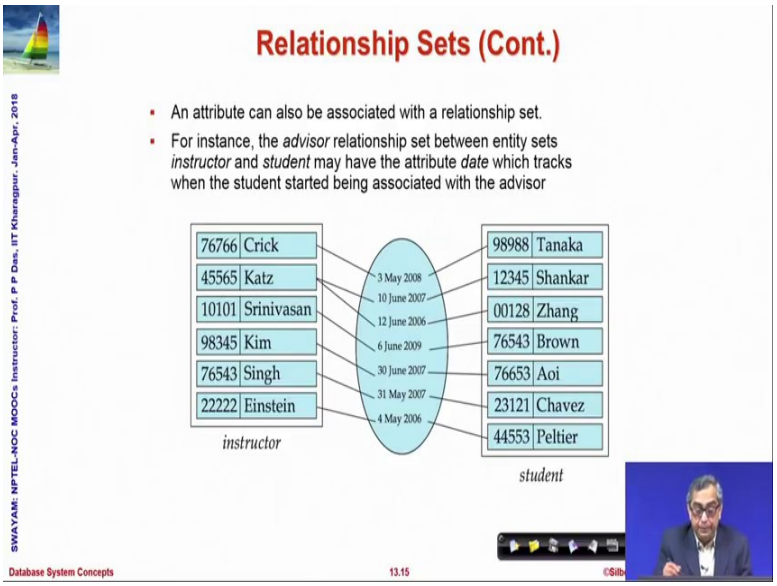
(Refer Slide Time: 11:58)



So, here what we show is a relationship advisor by these arrows these lines. So, what we are showing is this connection between these two, show that this student is advised by this instructor where as you can see. So, crick advises Tanaka where as Shankar and Zhang both are advised by Katz.

So, this group of associations between instructor and student is the gives me the relationship adviser as to who advises whom.

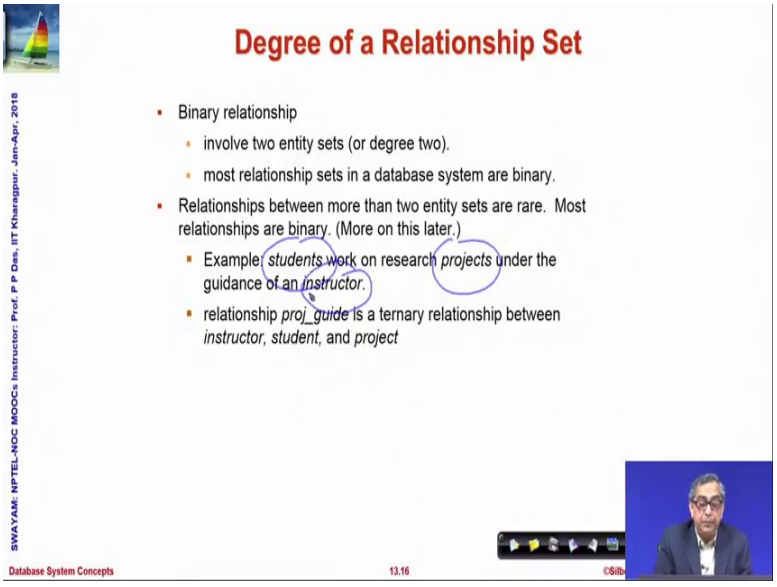
(Refer Slide Time: 12:39)



A relationship also like the entity sets the relationship also can have some additional attribute. For example, when I say that crick advises Tanaka I may associate an attribute date type attribute set third May 2018, to mean that when did this process of crick advising Tanaka started, we can it can be some other attribute also.

So, all that I am trying to highlight is attributes can be assigned to relationships as well.

(Refer Slide Time: 13:15)

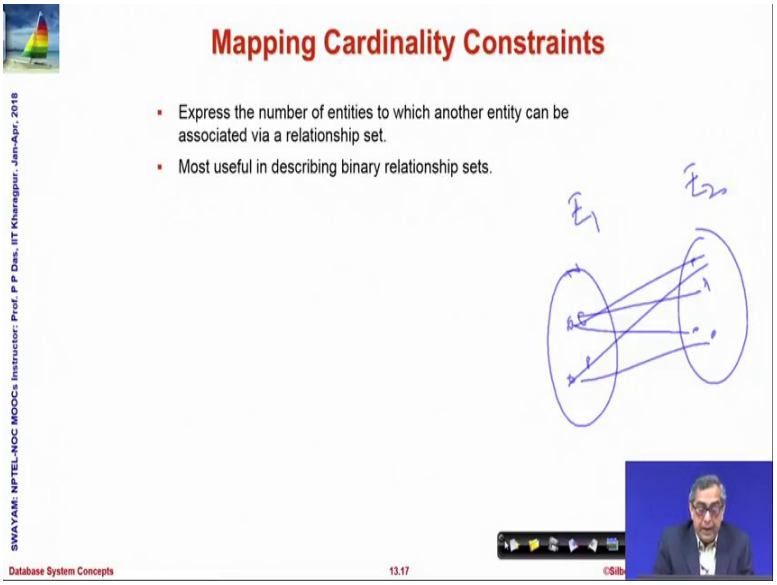


Now, how will a relationship span out, we have said that a relationship must involve two entity sets. So, primarily relationships are binary it involves two and most relationships

in most databases are binary in nature, but it could be that there are we will see later that there are possibilities of having relationships which are more than binary ternary and higher.

So, here are example students works on research projects under the guidance of an instructor. So, here we have as you can see student's research projects and instructors, so there are 3 entity sets. So, if I want to maintain a relationship of say project guidance between them then that turns out to be a ternary relationship we will talk about this more later.

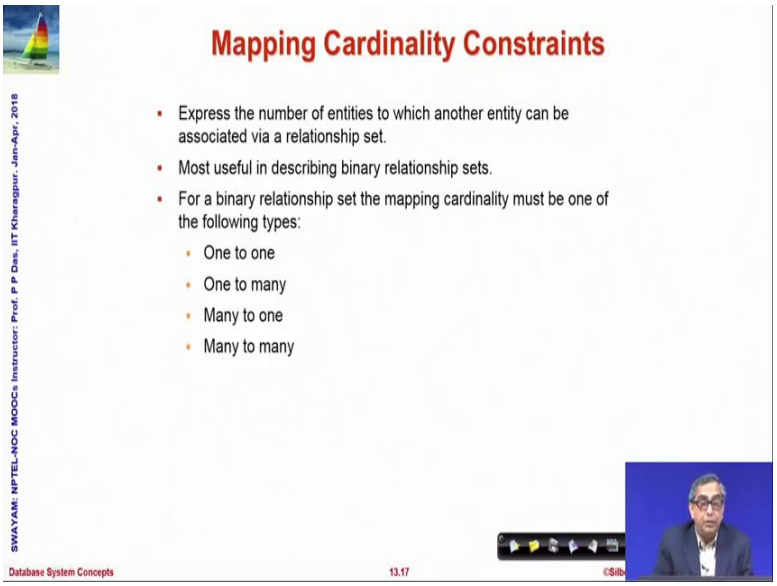
(Refer Slide Time: 14:16)



There are constraints in terms of the cardinality of the relationship, the cardinality basically talks of that when we have when I have a relation entity set E1 and identity set E2.

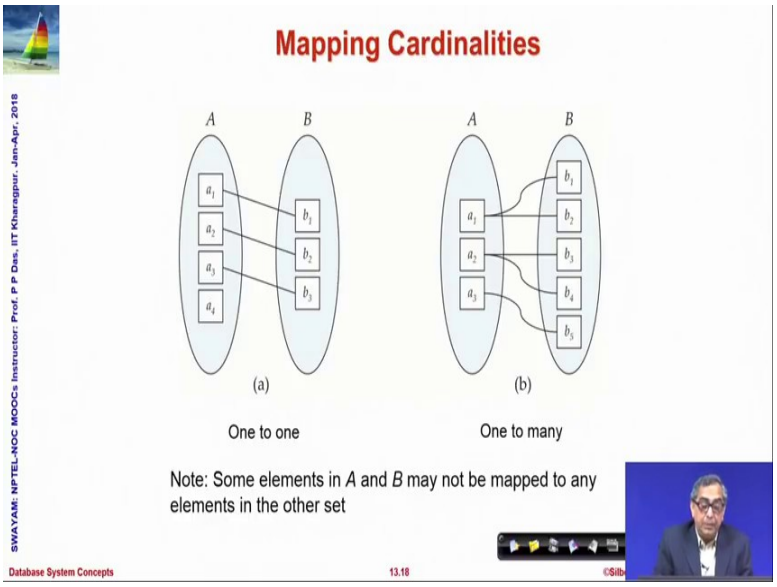
So, there are different entities in them and I have different associations between them, then the question is how many of the entity of one entity set is related to how many of the entities of the other entity set and certain types of cardinality measures are very important.

(Refer Slide Time: 14:58)



To track and we say it is whether it is 1 to 1 to many many to one or many to many.

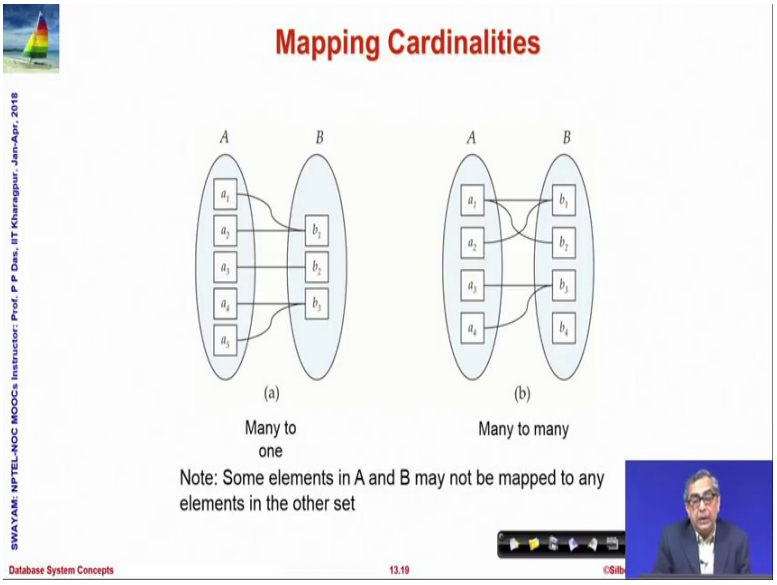
(Refer Slide Time: 15:08)



So, here the examples or the schematics, so in the first one in the diagram A you see that every entity from the entity set A relates to exactly one entity in the entity set B or you can say at most one entity in then they decide B similarly every entity in entity set B relates to exactly one entity in entity set A or at most 1 entity and entity set a if this holds then we say this relationship is one to one whereas, in diagram B you see that a 1 relates to b 1 as well as b 2 a 2 relates to b 3 as well as b4.

So, 1 entity in A relates to more than 1 entity may relate to more than 1 entity in B, but if you look from B side every entity in B is related to at most 1 entity in A then we say from A to B it is 1 to many.

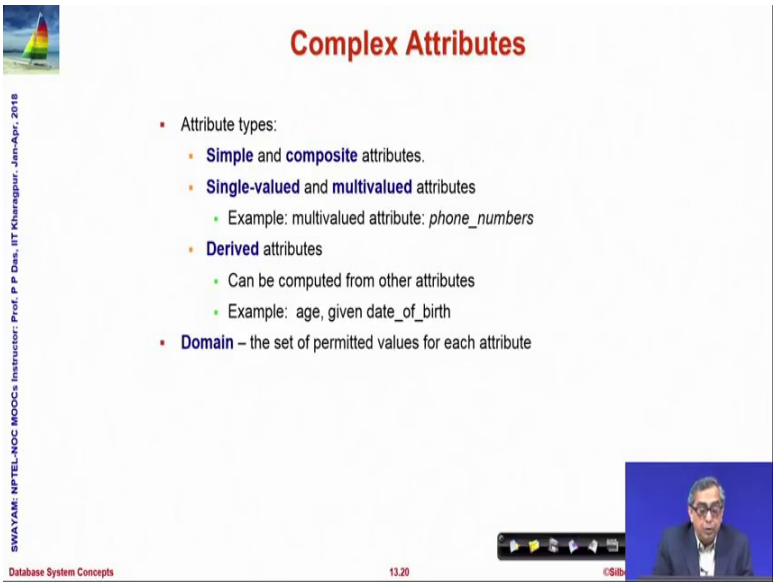
(Refer Slide Time: 16:04)



Now naturally since I can put the relations in any order as we have one to many if you look in the other direction it becomes many to one. So, many to one is from A to B many to one is where more than one entity in set A may relate to one entity inside B, but all entities in set B relates to at most one entity inside A and when there is no restriction at all that is any number of entities in set A may relate to any number of entities inside B and any number of entities in set B may relate to any number of entities in set A we say it is a many to many relation.

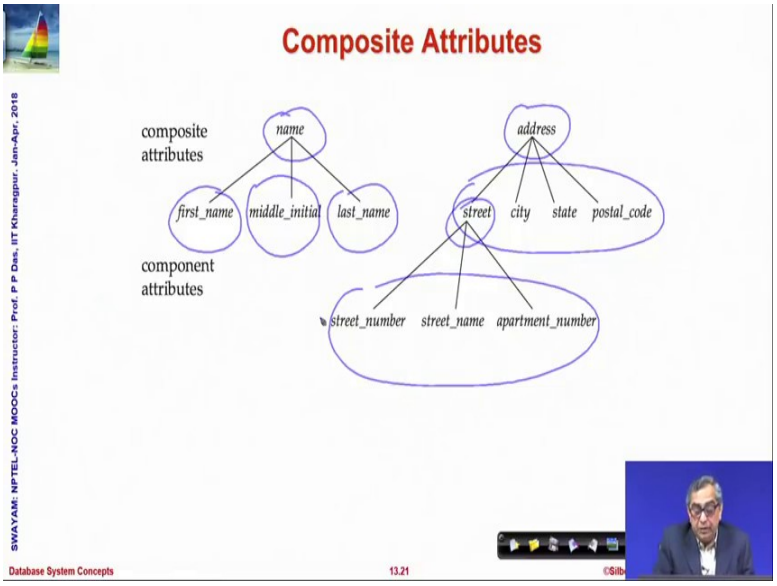
So, we have one too many one to one, we have one to many and many to one and we have many to many and it often helps in the design to be able to characterize which type of relationship we do have.

(Refer Slide Time: 16:51)



Coming to the attributes we can note that attributes are of different types, one is they could be simple or composite a simple attribute is just one single domain value like a salary number like an id like a name string and so on; whereas, a composite attribute may comprise of multiple parts.

(Refer Slide Time: 17:16)

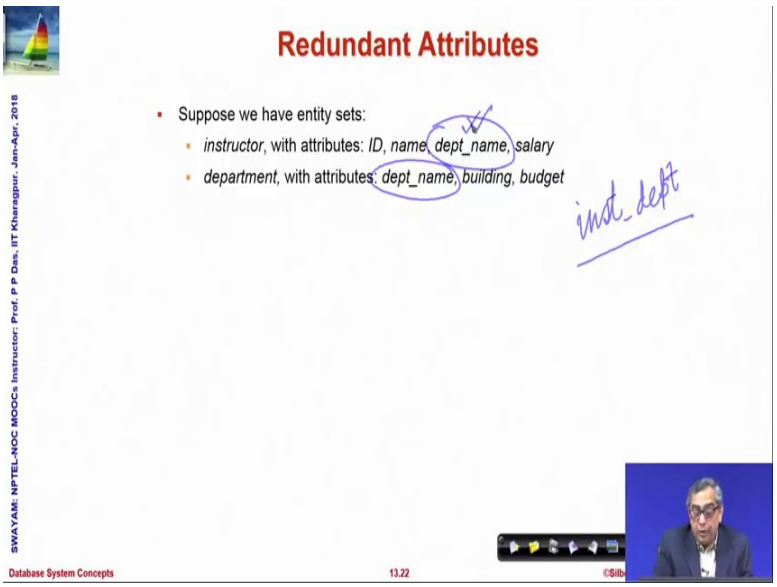


So consider this, this is a composite attribute, so name is an attribute if I think of then it has different parts, it is a first name middle name last name if I think of address it has. So, many different parts then street itself has so many different parts. So, whenever an

attribute is comprised some more of the components, when it is not a simple value then it is called a composite attribute we will see how to handle them; then some attributes may be single valued, for example a person has a has 1 name let us say, but has one address, but may have 2 or more phone numbers, the attributes which can take more than 1 value is known to be multi valued attribute.

So, we also need to specify whether certain specifying in the design whether certain attributes are single valued or multiple valued multi valued; of course, single value attributes are easy to deal with if it is multi valued we need to do some design changes. Certain attributes can be derived for example age, now I cannot keep the age of some a person in the database because, with every day the age changes. So, what we typically keep is a date of birth and the age is computed on the day when the particular query is made to find out what the age is so it is called a derived attribute and each 1 of them will have corresponding set of domains.

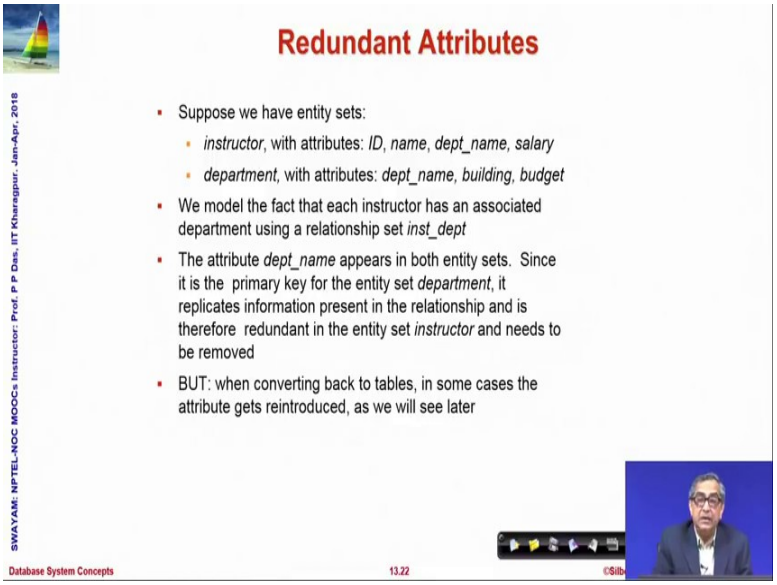
(Refer Slide Time: 18:53)



Some attributes in the design may turn out to be redundant also consider this you have already seen this this is an instructor, which has a department name along with the different attributes and certainly I have a department table, so which department relation which gives the details of the department. Now since every instructor belongs to a department, so naturally we might want to have a relation inst_department which could give the instructor and his or her department name. So, if we maintain that then this

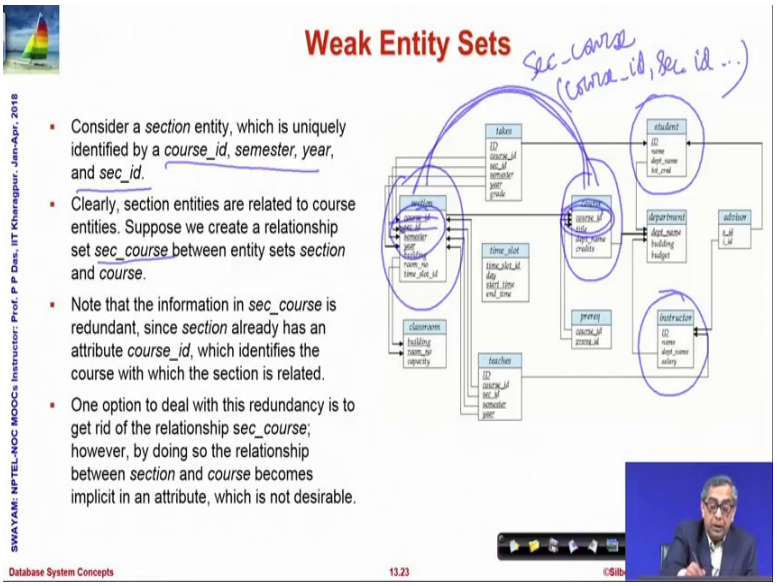
becomes a redundant attribute this is not required, because if that information is already there in this relation.

(Refer Slide Time: 19:49)



So, in several cases there is a question of whether we maintain some information in terms of a relation or we can make that directly include that directly in the entity set and get rid of that relation. So, if I have the inst_dept relation and then the attribute department name appears on both these sets, inst_dept as well as on the instructor and there is duplication replication of the data which we want to avoid. But we will see the different cases when which style of design, whether we would be better to maintain the department name as a part of the instructor relation or it would be better not to have it there and have a separate relation which maps instructor id against the department name.

(Refer Slide Time: 20:48)



Finally comes a concept of weak entity sets you need to understand this a little bit, consider the university database example. So, we have courses we have students we have instructors and we have section, a section is if a course is large then it needs to be taught in multiple sections. So, for the same course at the same semester in the same year I may have different sections, in which the students are divided and naturally there could be multiple instructors each teaching 1 section of that course and students will be distributed on the sections not on the course.

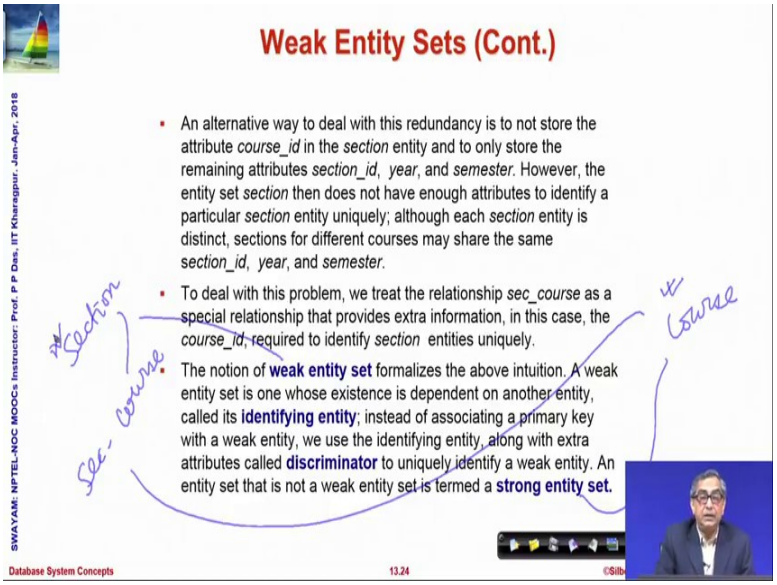
Now, consider this section entity, if you look into this then this is how what we we maintained we did a course id semester year and section id, but if you look into specifically and if you want to now know you know that there is a section and there is a course. So, you may want to relate these two section with the course and set up an entity between them. So, what will it relate it will relate the course id of the course with all of these, but the course id is already there as a part of the section right. So, you would say that well it is not required to have the course id, since it already has that and it it identifies it.

So, we can can we remove this course id from here, well if we remove the course id now we have a different problem. If you remove the course id then you have section id semester and year. But this does not uniquely represent the tuples of this relation

because, there could be 2 section as in the same semester in the same year for 2 different courses how do you distinguish them.

So, you get into a situation where the course the section gets identified uniquely provided, either you know the relationship between the section and the course in terms of the sec_course relationship or you include the primary key of course, into the relation section which we did in the design and this is not a coincidence this is something which happens regularly and is is the characteristics that specify the existence of weak entity sets.

(Refer Slide Time: 24:11)

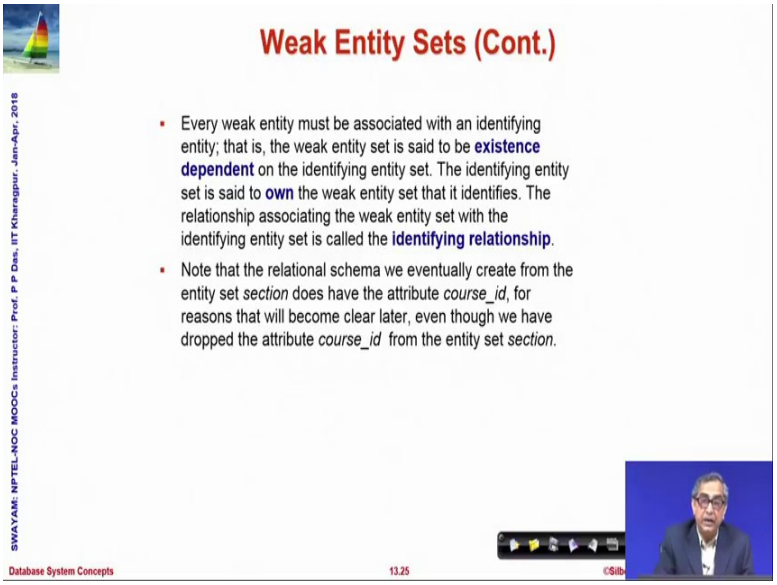


So, the weak entity set is one whose existence depends on another entity set. So, if I just say section having section id year and semester then it is not uniquely specified, until I have a relation section course which relates the section to the particular course id. When such relationships are used to identify entities of a particular entity set, then the unique side the course side which is unique is known as the identifying entity and the other attributes in this case section id year semester are known as the discriminators.

So, we have a relationship between a weak entity set which is section, we have a strong entity set which is course why is it strong because course is identified by course id itself. Section is not unless you have a set course kind of relationship set between the course and the section which specifies that well, for this course this is the section this is the year this is the semester. So, this is the identifying entity through which the entities of this set

gets specified and whenever that situation happens then we say that we have a weak entity set.

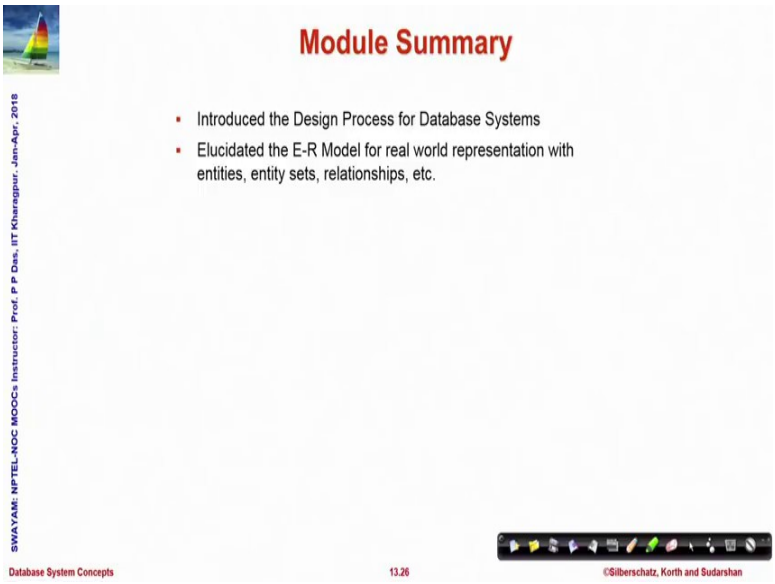
(Refer Slide Time: 26:21)



So, weak entity sets naturally cannot happen by themselves their existence dependent on identifying entity set and the identifying entity set owns the weak entity set. So, the courses in that way own the section and the identifying relationship between them is necessary to uniquely identify every entity of this weak entity set or section in our case.

So, this notion is very important for the design as we will see that the relational schema that we eventually created, in this case from the entity set section we did include course id as a part of the primary key not using the sec course kind of relationship and we will show how this design style for dealing with entity weak entity sets influences the different database designs. So, weak entity sets are critical notions that you need to be aware of need to be confident of.

(Refer Slide Time: 27:34)



So, in summary we have introduced the design process for database systems I will just quickly recap, the first stage is identifying the data items which is leading to the conceptual design, which will primarily do in terms of the entity relationship model identifying the entities the entity sets, the attributes that define the entity set, describe the entity set, the subset of attributes forming primary key that uniquely specifies every entity set, every entity in the entity set and the relationships typically binary may be non binary also, relationships that hold between the different entity sets.

So, this is the conceptual design that will lead to more detail logical design of how the relationship should be organized, what is the cardinality of that, what kind of attributes do I have, whether it is sample whether it is composite whether certain attributes are derived or not. So, all those are different aspects will have to be detailed out and we need to identify what are the weak entity sets and what are the strong entity sets, what are the identifying entities and with that we could complete the logical design and then we will need to make it in terms of it express it in terms of a relational schema.

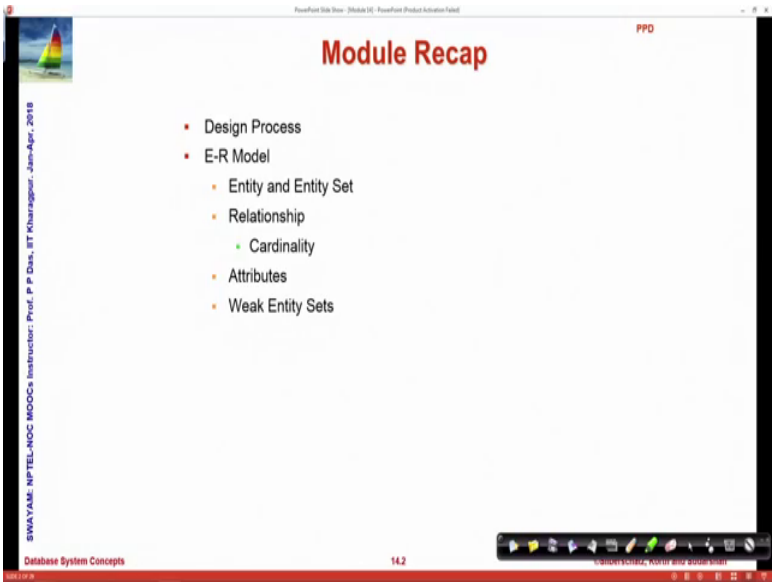
So, in this module we have just taken a look in the first part the entity relationship model and the very basic of how the conceptual design. We will go forward, in the model we have seen all the different primitives required to represent the reality represent what holds in the real world.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture - 14
Entity-Relationship Model/2

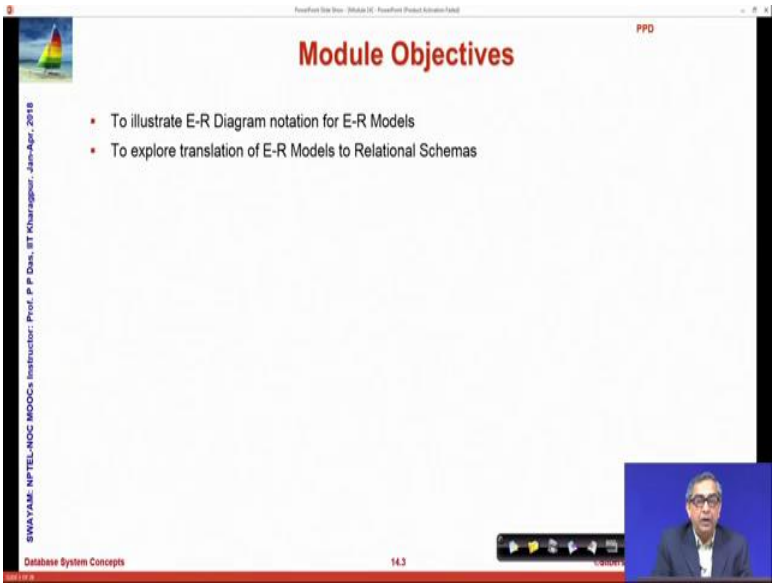
Welcome to module 14 of database management systems. In the last module we will started our discussions on the entity relationship model.

(Refer Slide Time: 00:29)



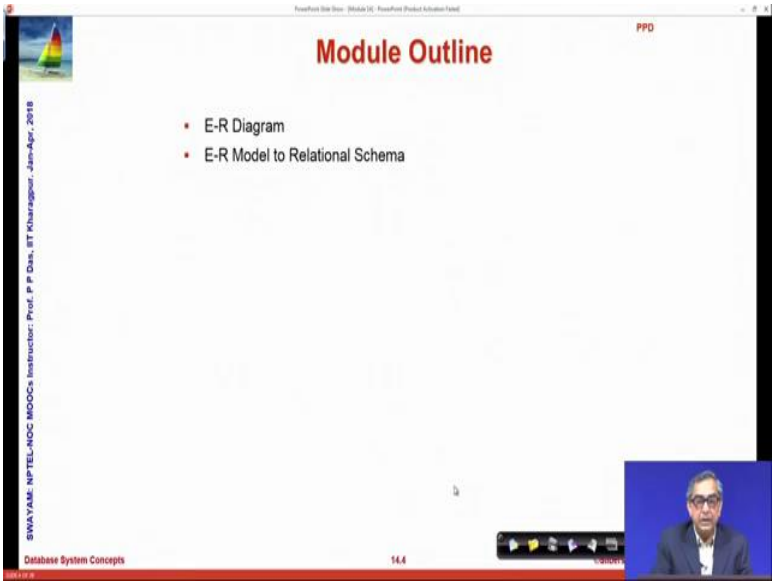
We will continue that in this module as well, and actually conclude it in the next module. So, these are the items that we had discussed in the last module.

(Refer Slide Time: 00:39)



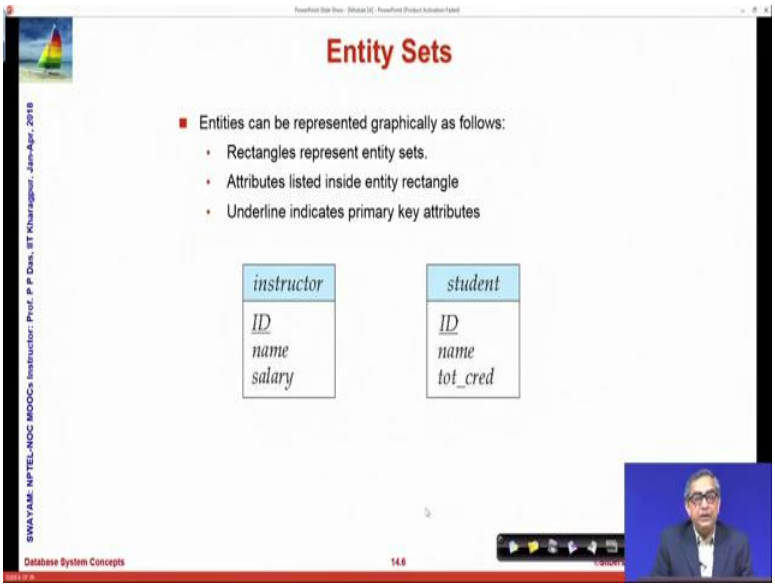
In the present one, we will first illustrate the entity relationship diagram notation; that graphical notation for E-R model, how nicely this can be shown in terms of certain diagrams. And then we will explore how E-R models can be translated to relational schemas, which is a basic step of the logical design.

(Refer Slide Time: 01:11)



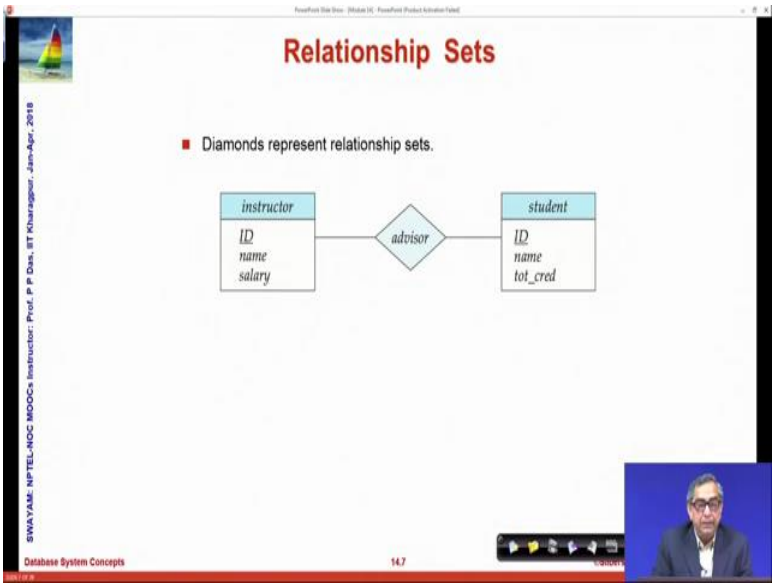
So, these are the topics. So, we start with the E-R diagram.

(Refer Slide Time: 01:16)



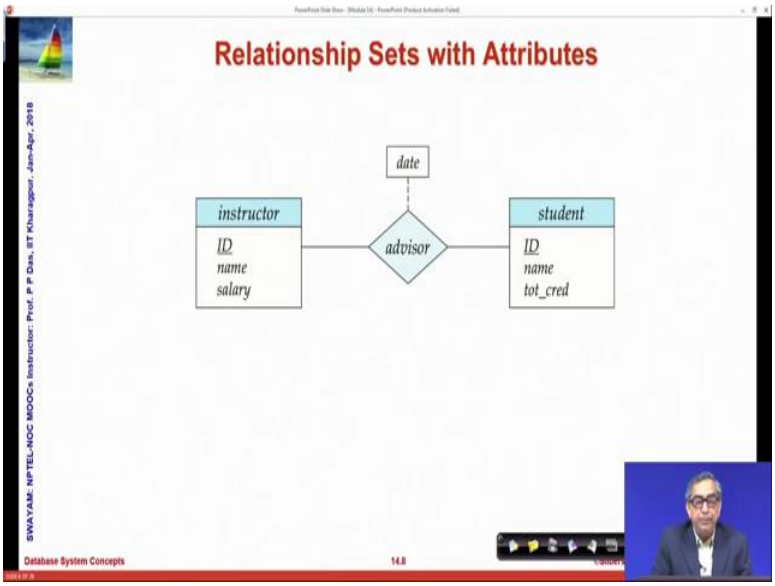
Naturally the first thing to represent in an E-R model is the entity set. Every entity set is represented by a rectangle. On the top, we write the name of that entity set ss you can see examples here the instructor. And student are the two entity sets and below that we write the names of the attributes that are involved. And we underline the attribute or attributes that form the primary key of that entity set.

(Refer Slide Time: 01:49)



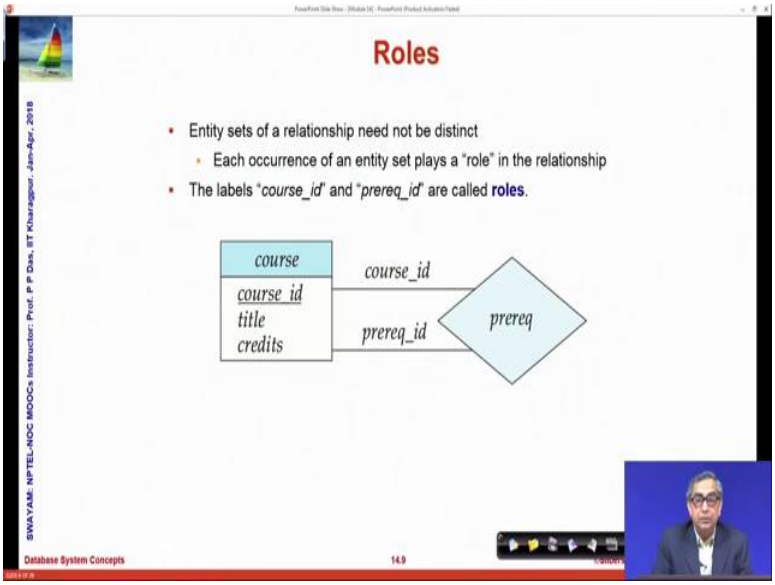
A relationship between two entity sets is represented by a diamond, and 2 connecting lines to the 2 entity sets. So, here it says that advisor is a relationship between entity set instructor, and entity set student. Trying to convey the real-world situation that students are advised by the instructors or students have instructors and so on.

(Refer Slide Time: 02:23)



As we had mentioned that relationships could also have attributes. So, if the advisor relationship has an attribute date then it will be tagged to the adviser relationship with the attribute coming as a within a rectangle and attached to the name of the relationship by a dotted line. So, this shows that advisor is the relationship between instructor. And student and the adviser relationship has an attribute date.

(Refer Slide Time: 02:58)



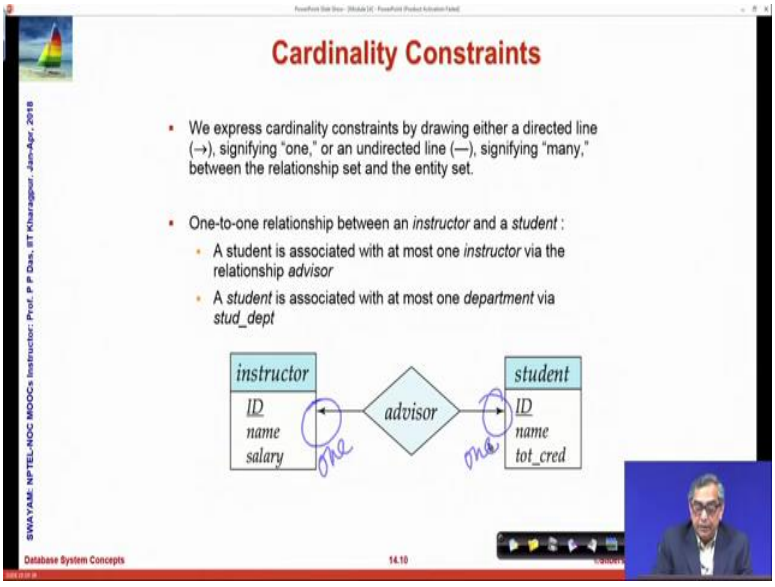
It is possible that the relationship that hold between 2 entity sets can be can use entity sets which are same. That is, it is possible that a set is related to itself.

So, as an example we show, the entity set course which has a relationship prereq, prerequisite which takes a course id, and relates it to another course id called the prerequisite id. Because obviously, if a course has a prerequisite. Then that prerequisite itself is another course id which must occur in this table itself.

So, in this case, if you can see that unlike the earlier case the prereq id is not actually a field of this relation course. So, we say these are rules. So, we say the rule that prereq relate from the course relation to itself are course id and prereq id; where in the actual table both of them relate to course id, but prereq will pair them to show which course has what prerequisite. And we often need this kind of; we have seen similar instances of this while we treated dealt with the recursive queries in databases.

The source destination of aligns problem that we discussed has similar kind of relationship structure. So, you can think about a relationship flies from the set of source to this set of destination, and basically this; these 2 sets the places source places in the destination place are necessarily the same set the same relation. There could be a constraint on the cardinality.

(Refer Slide Time: 05:07)



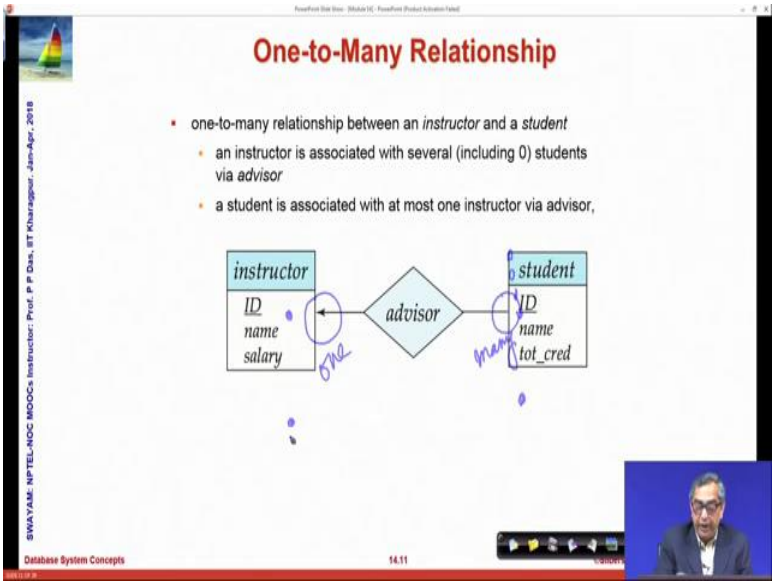
So, the line that links the relationship diamond with the rectangles of the relations rectangles of the entity sets. Those lines could have an arrow at the end or may not have an arrow at the end. So, if it has got an arrow, then it means that suppose it has an arrow it means 1. And if it does not have a arrow if it is simple then it means many. So, using

this rotation we can designate one to one to many these all different kinds of cardinalities that we had discussed.

So, for example, if we are showing this arrow on both hands, both ends of this relationship advisor then, it means that it is a one to one relationship because there is an arrow here. So, this is 1. There is an arrow here. So, this is 1. There is an arrow here. So, this is 1; which means that a student is associated at most with at most one instructor. And it also means that and an instructor is associated with at most student.

This may not be a reality this usually is not the reality, but this we are just showing this as an example. So, if the student instructor relationship advisor relationship is one to one, then this is how we will denote it.

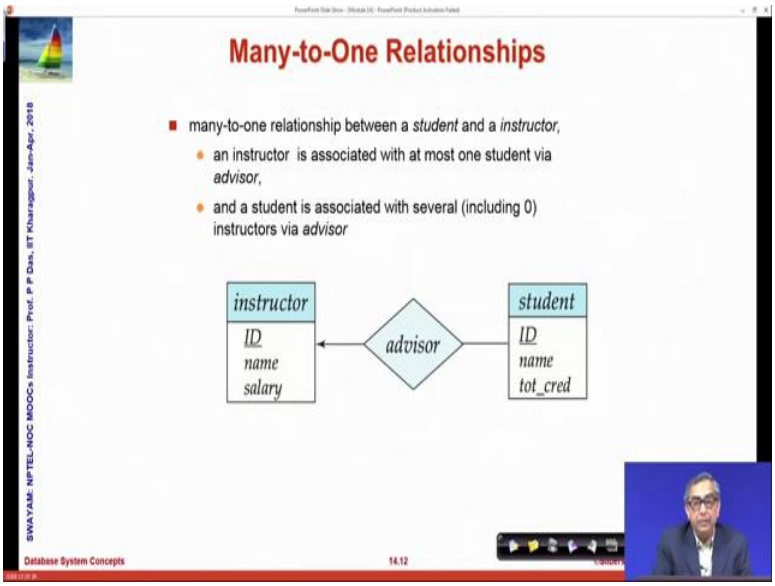
(Refer Slide Time: 06:54)



If it is one to many; so, which side is 1, this side is 1 and this side is many. So, it is one to many from instructor to student; which says that every student has at most one instructor. And an instructor may have several students. It could be null also it could be none also.

So, for an instructor here there are many students, but for a student here there are only at most one instructor. So, it is one to many and this is how we designate.

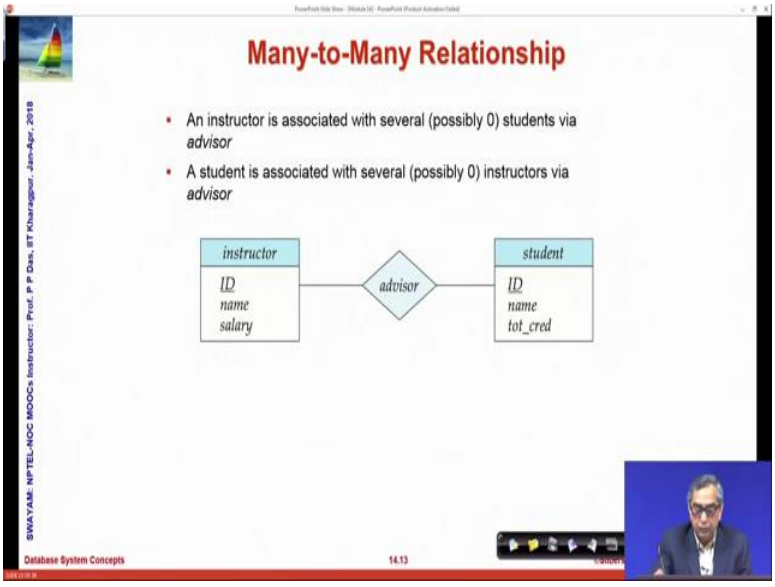
(Refer Slide Time: 07:35)



A similar thing will happen, if I read the same relation in the other direction. So, instructor to student was one to many. So, student instructor also drawn in the same way, because this is this is the one side, this is the one side and this is the many side.

So, if the situation is the same. So, if we read from the student instructor, then it is also designates the many to one relationship.

(Refer Slide Time: 08:05)

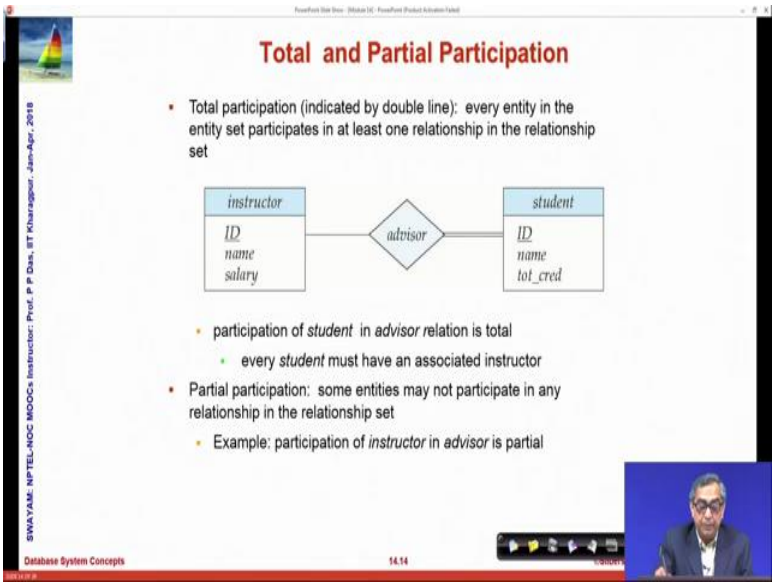


And finally, we can have a many to many relationship, where there is no arrow at either end which means that an instructor is associated with; several possibly none no student

why the advisor relation and the student may also have several instructors by the advisor relationship.

Now, we can you can certainly figure out that in the particular case of student instructor scenario of providing advice, one to one as well as many to many are not the usual real world scenarios, but these are we have just using to show you how to model this usual scenario would be from instructor to student it is one to many relationship.

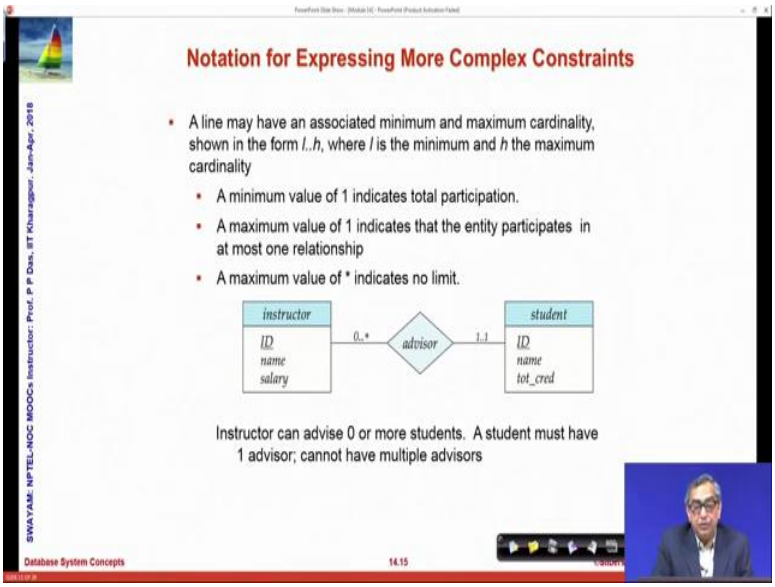
(Refer Slide Time: 08:50)



A relationship could be total or it could be partial. If you if one side of the relationship, or whichever side of the relationship is total, then we draw a double line. So, you can you can see here in the diagram we are drawing a double line, which means that in the advisor relationship the involvement of the student is total, which means that every student must feature in the advisor relationship. Or in other words every student must have an advisor. But it is partial on the instructor side because every instructor may not have a student.

So, this double life shows that reality. Some entities may not participate in any relationship is a partial.

(Refer Slide Time: 09:59)



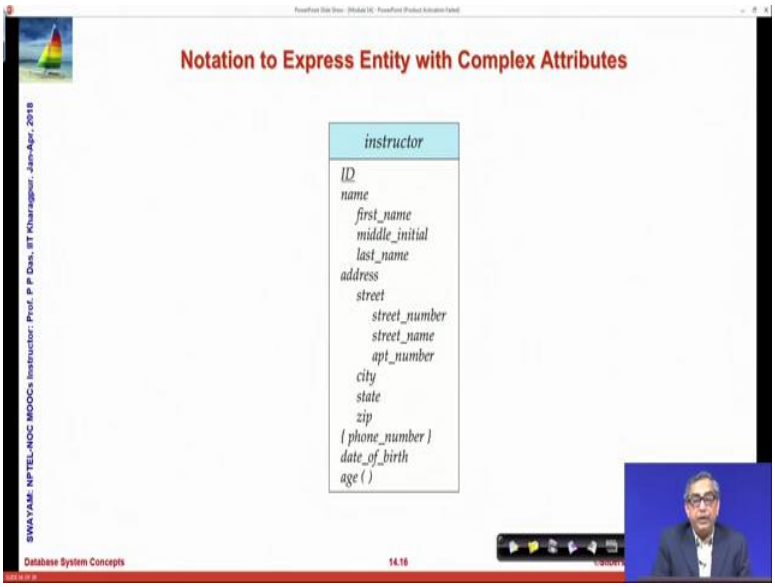
Now, these constraints the cardinality constraints can be made more precise by actually using numbers. You can actually say on the 2 sides of the relationship, that at the minimum how many entities should relate, and at the maximum how many entities can relate. For example, if we are saying that we are on the right-hand side here, if you see we are saying that it is maximum minimum is one maximum is one which what does it say it says that every student the minimum is one. So, every student must feature in the advisor relationship.

So, in real world, every student must have a an advisor, must have an instructor. It says maximum is one; which says that every student can have at most one instructor. So, this one to one one, dot dot one says that every student must have at least one instructor, every student must have at most one instructor. So, together it says that every student must have exactly one instructor.

Whereas if I if we see on this side, it says that 0 dot, dot, star, star stands for no limit. It can be anything any number. So, the minimum is 0 which means that an instructor may not have a student. And star says that if the instructor can have any number of student. Naturally 0, 1, 2, 3, 4, 2, or 200. So, any instructor can advise any number of students.

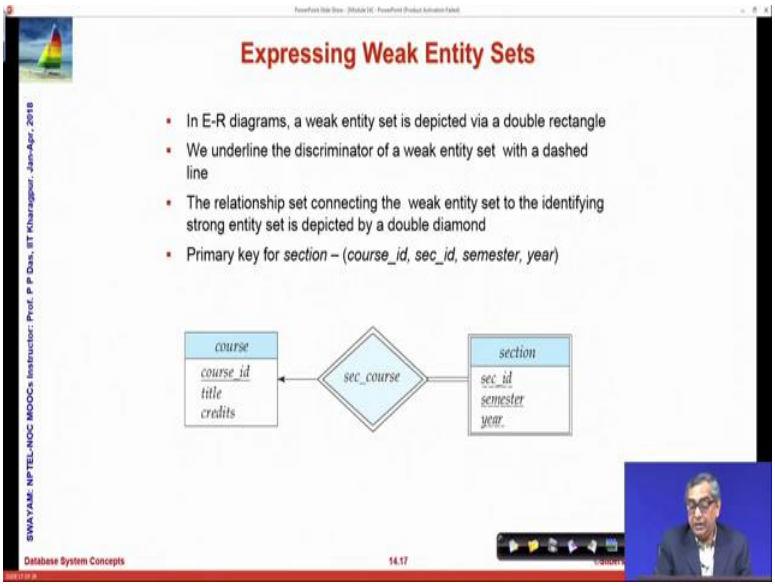
So, these kind of precise number constraints can be put in addition to the one to many, or one to one, many to many kind of notations in the diagram. So, when we do that we have the precise cardinality of the complex relations that exist.

(Refer Slide Time: 12:10)



Next, we take a look into the handling of the complex attributes. The first you remember that, the first kind of complex attribute is one which is composite. Say, name which has first name middle name, initial middle initials and last name.

(Refer Slide Time: 12:26)



So, when we have that, then the way we represent is at the actual name of the attribute is at the outermost level. And it is composites are written with certain shift on the left. So, these all say that these are composites of name. So, if this says that street city state zip are composites of address, and further indentation say, that these are composites of street. So, this is how graphically we show that how complex attributes feature.

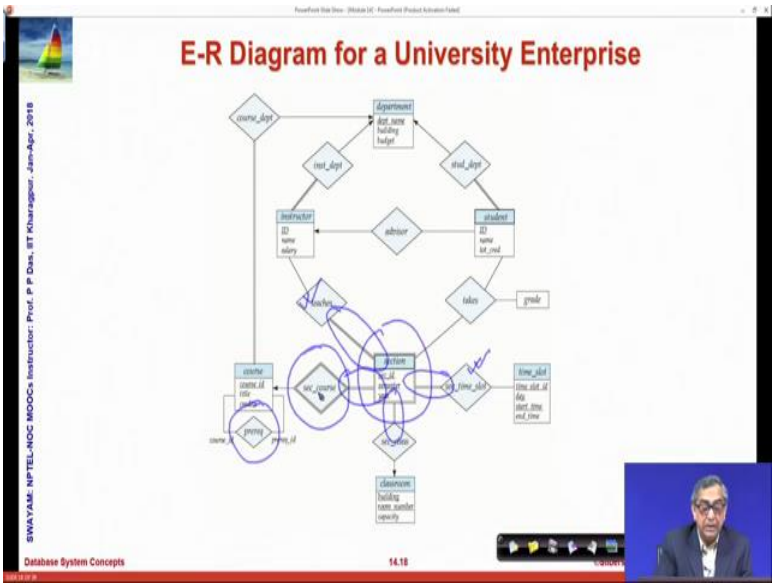
Now, let us go back to discussing the weak entity sets. In the E-R diagram, a weak entity set is represented by a double rectangle. You remember the section is a weak entity set. And why is it so? Because the same course may have 2 different, I am sorry, 2 different courses may have the same section id semester and year that is 2 courses 2 or more courses may run sections by the same name in the same semester and the year. So, a section cannot be uniquely identified by these 3 attributes. It needs a relationship with the identifying entity set course to be specific the course id so that the entities here in can be uniquely specified.

So, since this has happened. So, we designate that by putting this double rectangle around the weak entity set section. We underlined the discriminator of a weak entity set with dashed line. So, you remember these are the discriminators. Because given the identifying attribute in the identifying set. These are the attributes which distinguish different tuples of section. So, they are not shown with solid underline, they are shown as dotted underline, dashed underlined. So, that you can make out that this is a weak entity set and these are the discriminators.

The relationship set connecting the weak entity set to the identifying strong entity set, is also so, this the moment you have weak entity set. You know that there has to be a relationship to the strong entity set which identifies it. So, that relationship set course which say, course id against this minds that is designated with a double diamond. So, that you know that this is the identifying relationship between a weak entity set and the corresponding strong entity set. And once that happens in the primary key becomes the discriminators of section the weak entity set, and the primary key of the identifying a strong entity set the course. So, that forms our final primary key for this entity set section.

My new course id is not a part of this relation, but it actually plays the role through this section id as a key for the section relation without which the section entities in the section cannot be uniquely identified.

(Refer Slide Time: 16:28)



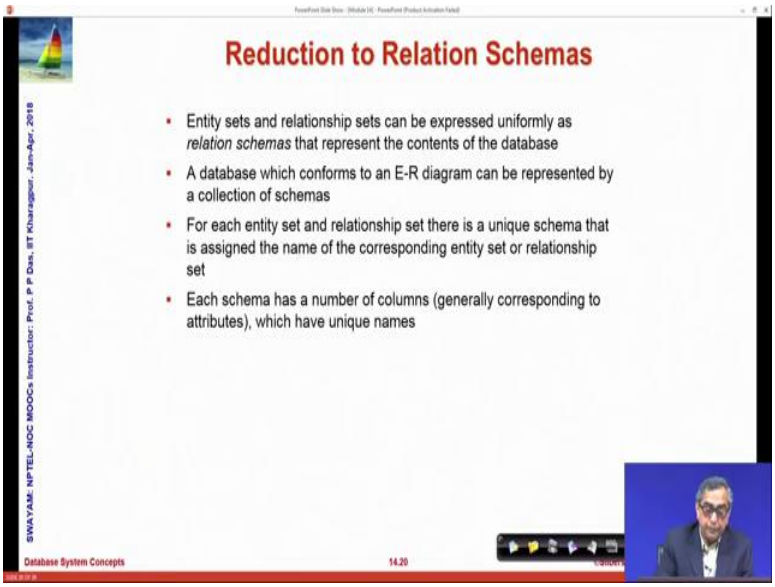
So, having said that this is a the E-R diagram of the university enterprise. Some of the points that you could take a look at; this is the weak entity set we have just seen. This is the relationship to the identifying strong entity set. This is a prerequisite multi role relationship. this we can see is a is a total involvement. So, worry why is it a total involvement, because every section must have at least one teacher. So, there cannot be a section which does not feature in the teacher's relationship. Similarly, every section must get a timeslot where the classes for that section is held.

So, every section must feature in the sec timeslot. So, these are the similarly, it must get a classroom. So, these are all different total involvements. That we total roles that we can see we can see some of that elsewhere as well. For example, you can see it here, we can see it here, because in between instructor and the department the inst department relationship. Certainly, every instructor must have a department. So, it is total, but it is not the same for the department, every department will not have instructors.

So, these is how if we can you can go through carefully. And for example, this is another which is total, which means that every course need a department you cannot run a course which does not have a department. So, this is how we can see that how the E-R diagram that the first conceptual level diagram of a very simple university enterprise is being designed following the notions and symbols of E-R model that we have already developed.

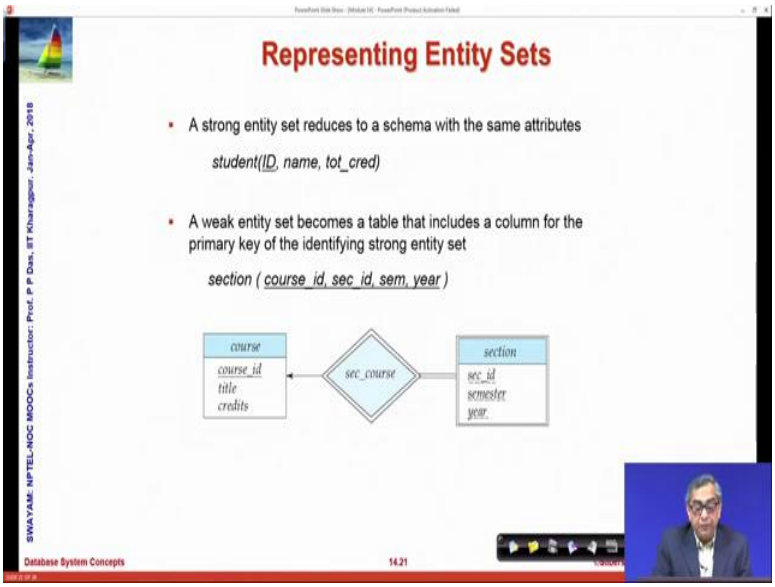
Next comes the part where, from this model which is primarily diagram based we have to really go to the relational schema, which is names of relations and attributes which is pretty much a straightforward job.

(Refer Slide Time: 18:41)



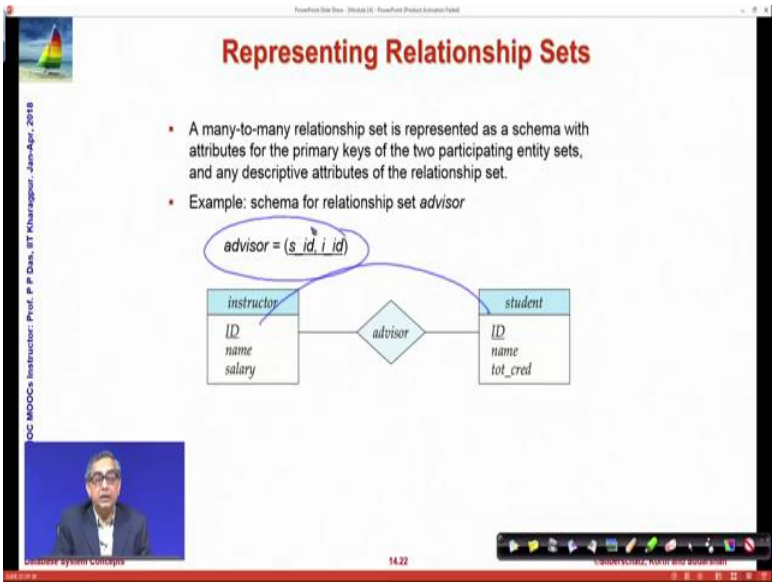
So, entity sets and relationship sets have to be represented in terms of relational schema. What is the beauty of the E-R model? And the relational schema is that that, when you reduce the entity relationship model to relational schema both entity sets and relationships sets. Both of them turn out to be relational schemas. And so, that the database finally, can be represented simply as a set of schemas each one of which must have a set of identifying primary key.

(Refer Slide Time: 19:20)



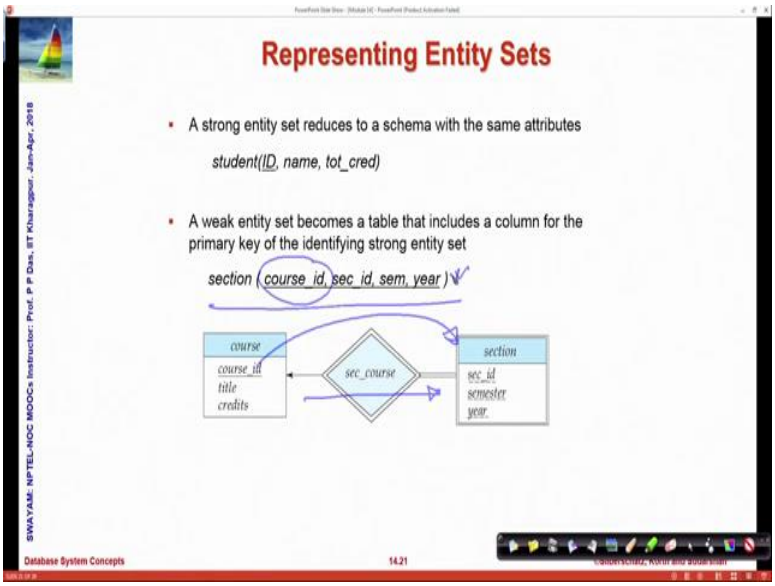
So, let us look into that. So, on the strong entity set, that reduces to schema with the same attribute as student. So, student has id name and total credit so, which we saw earlier. Now this gets converted to a schema with id being the primary key. The other case of weak entity set, section which had the three discriminators, and through set course relationship was identified from the strong entity set. Course borrows the primary key of the course to be defined in terms of this relational schema.

(Refer Slide Time: 20:05)



One moment; this borrows the primary key from here and becomes.

(Refer Slide Time: 20:10)

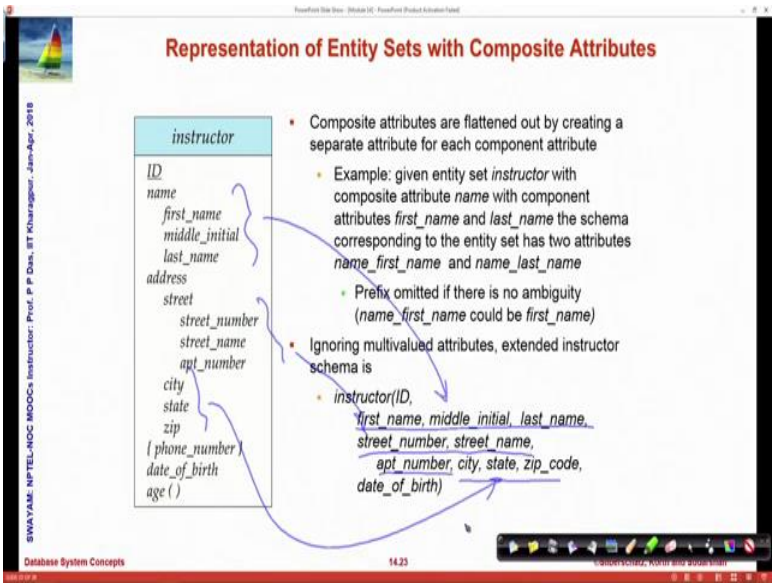


So, you can see that in the E-R model, the section did not have course id as an attribute, but while we reduce this to the relational schema through this sec course relationship, we have borrowed this primary key from course. The primary key of course, the course id and added that to section to make it a complete relational schema.

Next comes the representation of relationships. So, we are showing a relationship advisor so, which relates instructors to students. So, naturally every instructor is identified by id every student is identified by id. Since both of the attributes have the same name id. We are calling them as s underscore id and for the student and I underscore id for the instructor. So, the advisor relation is basically a pairing of these two ids which gives rise to a relationship which looks like this relationship schema which looks like this. So, it can in general say that if we have a relationship in the E-R model, which we want to represent in the schema then we will take, we will create a schema which has the primary key of both the sets, and put them together. And if the names clash we will just change the name with the name of the relation, and that will give us the schema for the relationship in this case the advisor.

So, you have seen how to represent entity sets, weak entity sets and relationships.

(Refer Slide Time: 22:03)



Ah let us look at how do we deal with composite attributes. Because so far, we had assumed that the relational schema has attributes, and every attribute has a domain. The type from where it is values come. So, if I have a composite attribute, where every attribute has a set of components. Then the easiest way to handle this is to what is known as flatten the composite attribute.

So, flattening basically is for example, if I take a name it has 3 components. So, each one of them I can call by given new name, name underscore first name, name underscore middle initial name underscore last name. By prefixing with the attribute name, I make the names of these components necessarily unique. Now after I do the prefixing I might figure out that actually prefixing is not required first name itself is a unique. Because it does not occur anywhere else if it is then I can I may drop the prefix name. But in general, I can take the attribute name prefix on the component, and just flatten them out make them all attributes each separate attribute.

So, here when we flatten out we will have first name, middle, initial, last name as you can see these flattened out from here. Then we have street number, street name, apartment number, flattened out from the street. Subsequently, we have city state zip flattened out from here. So, all of them flattened out has become separate attributes. And flattening is a very straightforward mechanism by which you can convert complex composite attributes into the regular schema design getting to little bit of issue if you have a multi valued attribute.

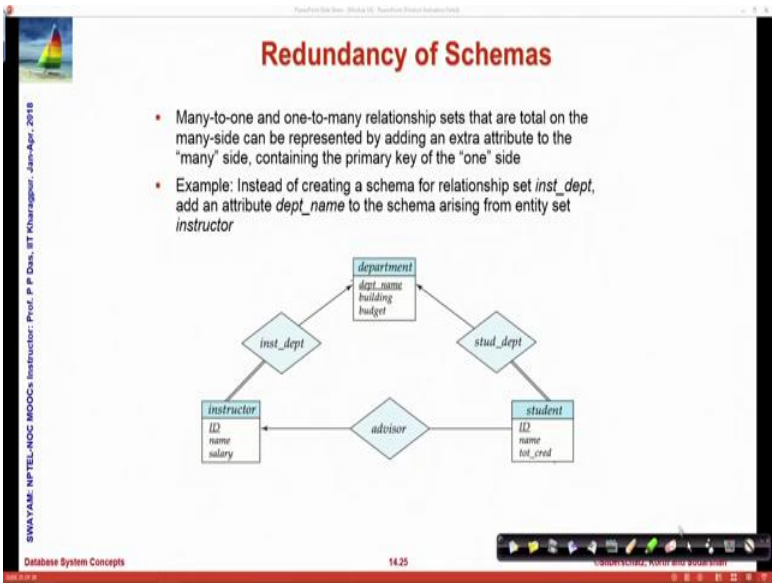
(Refer Slide Time: 24:02)

Multivalued attribute is one where one attribute may have multiple values at the same time. And the example we talked about is a phone number.

I may have multiple phone numbers. So, certainly against an attribute I can keep only one value. So, if I if my attribute is multiple value. Then the basic idea is to use a separate schema to maintain this multiple values. For example, if I have to maintain multiple phone numbers of an instructor. I may have a separate I may decide to have a separate relation which relates the key of the instructor relation, and the attribute that I want to maintain multiply.

So, in this relationship in this relation inst underscore phone against the same id I can have different phone numbers. So, there will be different records which match on the id, but do not match on the phone number which gives me the different values that the phone number can take. And then this inst phone in conjunction with the instructor relation will actually denote the multi valued phone number attribute. So, this is just an example showing that for one primary key of an instructor 1 to 2 2 2 2 2 and there are two phone numbers. So, this will basically mean you have two tuples in the new relation.

(Refer Slide Time: 25:37)

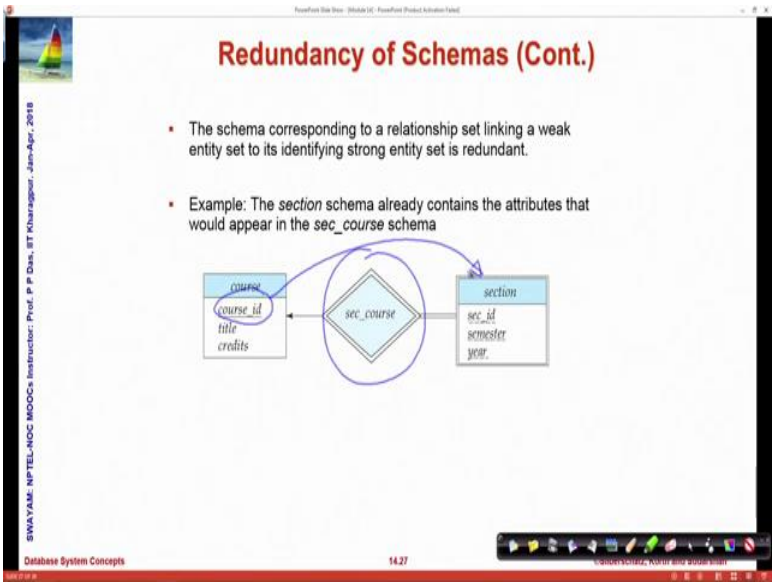


So, with that we can handle multiple multivalued attributes also. Some of the relationships that we may have modeled, which we have done in doing the database E-R schema could have redundancy. For example, take a case here we have the instructor. And we have the student advisor relation is incidental here, and we have department. So, we want to say that the instructors belong to certain departments. Every instructor belongs to one department which is the totality of the relationship here.

Similarly, every student belongs to a department, totality of the relationship on this side. And inst dept inst_dept in that context is a relationship which is between instructor and department. Similarly , so, we can there is certain redundancy in this, because we can get make this simpler if we just take the primary key of this relation and put in here. If we do that then basically this become redundant these are no more required. So, all that you are saying is the instructor has a dept name field which says which department does it belong to.

So, if there is a choice between whether you will keep such relationships or you will actually reduce the redundancy in the schema, and involve the primary key of the other relation into your primary table which is instructor or the student here. So, instead of creating a schema for relationship set inst department. You will simply add a department name. So, mind you this is at the at the E-R model level you did have a separate relationship, but while you reduce it to your relationship relational schema you are reducing that relationship by including the dept name as an attribute in instructor.

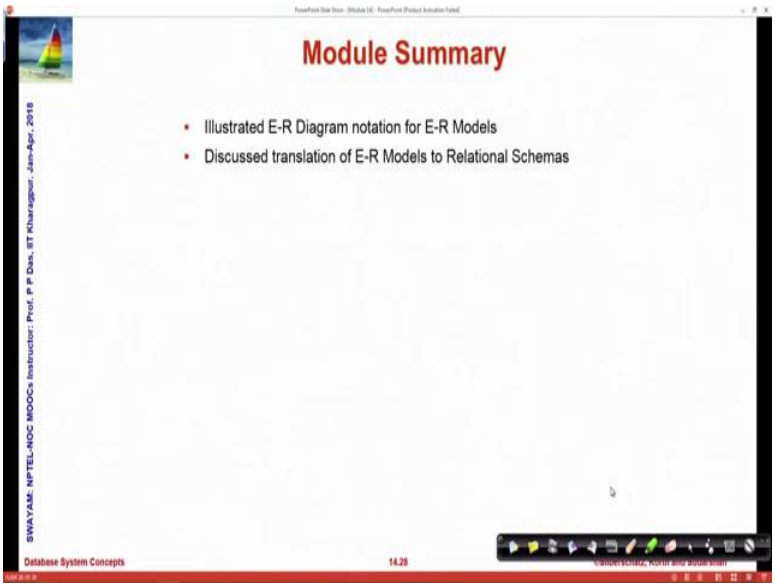
(Refer Slide Time: 27:53)



So, that is called the reduction of schema which is often used. for one to one relationship. So, this is this, was this is good if you have many to one relation. Because this was possible, because every instructor has one department. So, if you just include the department name with the instructor. Or every student has one department ah. So, it is possible that way, but if you have a one to one relationship, then naturally you can do the similar reduction by I by choosing the either side as the many. You know, because because the the unique side has to come on the many side. The unique side here. This is the unique side here. Because every instructor has a unique department and this is the many side here.

So, the unique side has to attribute has to come in here. The unique side primary key has to come in here. So, instead of so, you can apply the same principle to a one to one relationship by treating any one of them as a many side, and add the extra attribute on the other side to get rid of this additional schema. The schema corresponding to a relationship set linking a weak entity set to it is underlying strong entity set is certainly redundant, we have already seen this. So, this (Refer Time: 29:13) is made redundant by including the primary key of the identifying relation, identifying entity set in the weak entity set. So, that is another reduction of schema that can be done.

(Refer Slide Time: 29:31)



So, to summarize we have in this module illustrated the entity relationship diagram, which are very nice ways of graphically representing what we see in the real world. So, it has graphical representation of entity sets, attributes the key attributes primary key attributes the weak entity sets, and the relationships along with the cardinality information.

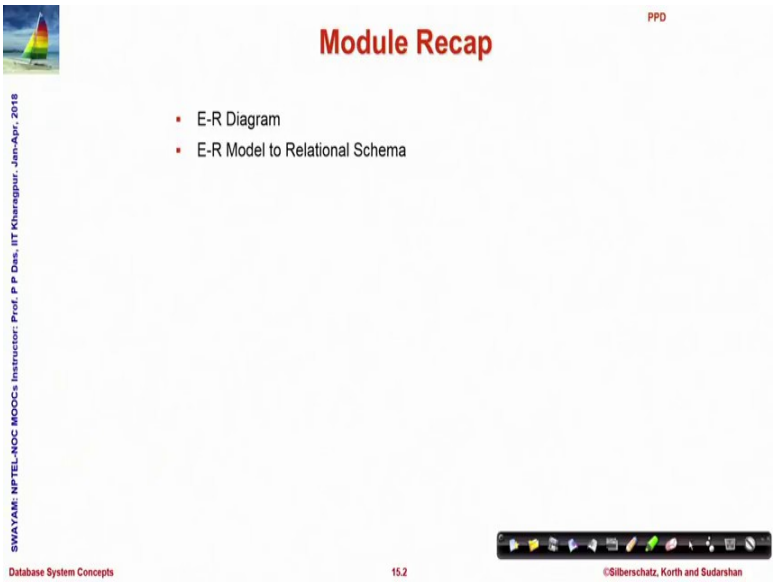
And then we have shown that using certain reduction rules how we can easily reduce this entity relationship diagram or entity relationship model into the traditional relational schema. And we have seen that both the entity sets as well as the relationship sets become relational schemas.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture – 15
Entity-Relationship Model/3

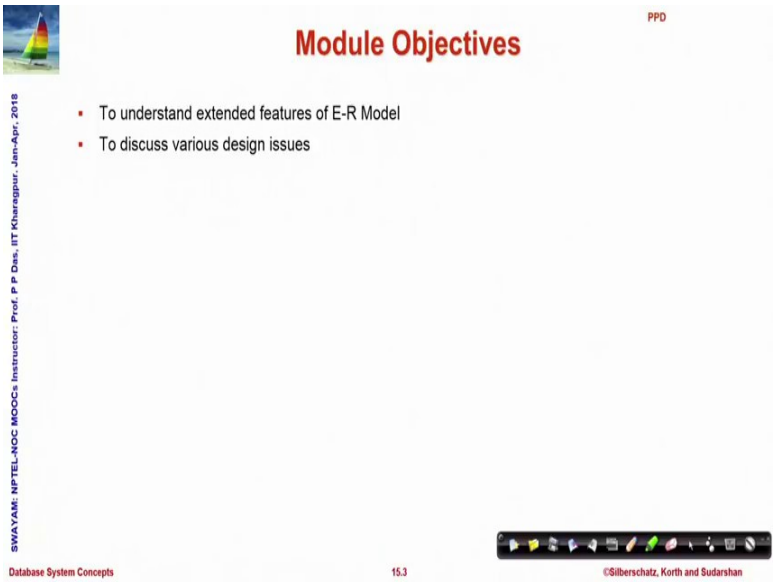
Welcome to module 15 of Database Management Systems. We have been discussing about entity relationship model and this is the third and the closing module on this topic.

(Refer Slide Time: 00:37)



So, in the last module we have discussed about E-R diagram and we have also seen how E-R model can be converted to a relational schema.

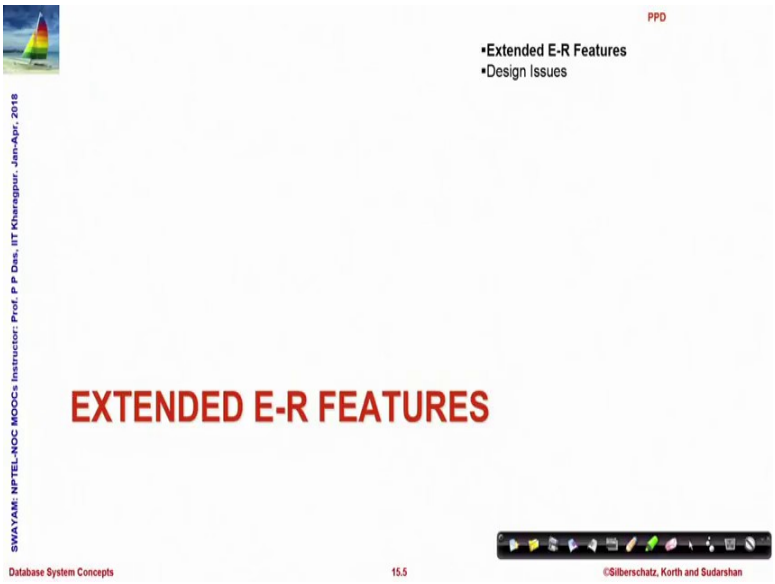
(Refer Slide Time: 00:47)



In this module we will try to go through a few extended features of E-R model, try to show some of the more complicated situations how they can be modeled in the E-R model and along with that we will discuss a variety of design issues that will follow.

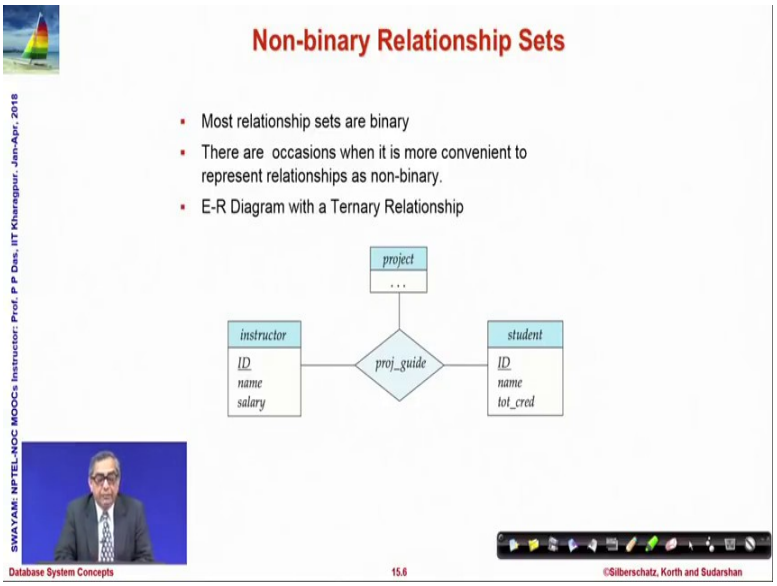
So, these are the outline.

(Refer Slide Time: 01:19)



So, we start with extended entity relationship features.

(Refer Slide Time: 01:23)

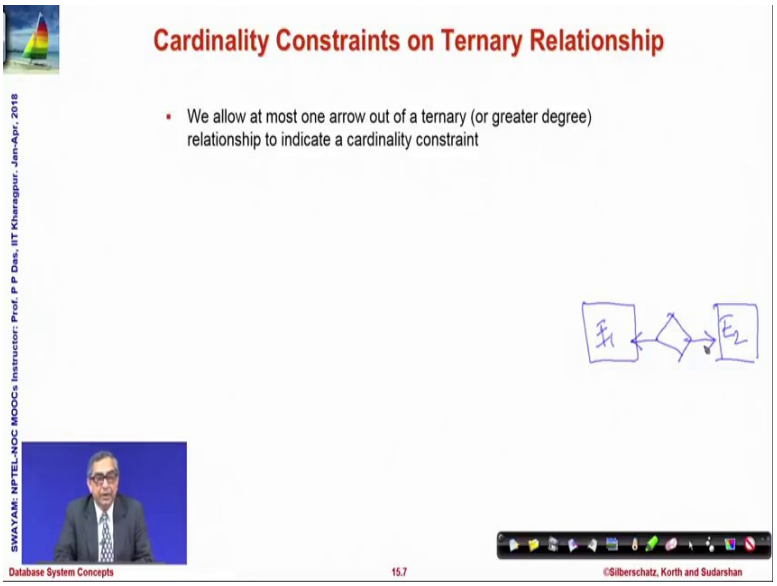


The first that we note is so, far in the entity relationship model we have talked about relationships between two entity sets. So, we have talked about the student attending courses or instructors advising students and so, on. So, such relationships are naturally called binary, but it is possible that more than two entity sets, let us say three entity sets could be involved in the same relation and we show an example here where we have three entity sets instructor, student and project.

So, the project entity set is a list of projects being done by the students or to be done by the students. So, the relationship project guide is a relationship between the guide who is an instructor, the student who will do the project and the project that has to be performed. So, all three together define this relationship; so, in such cases it is possible in E-R model that we can represent it conveniently as a non binary relationship.

Now this is an E-R diagram this is called a ternary relationship R.

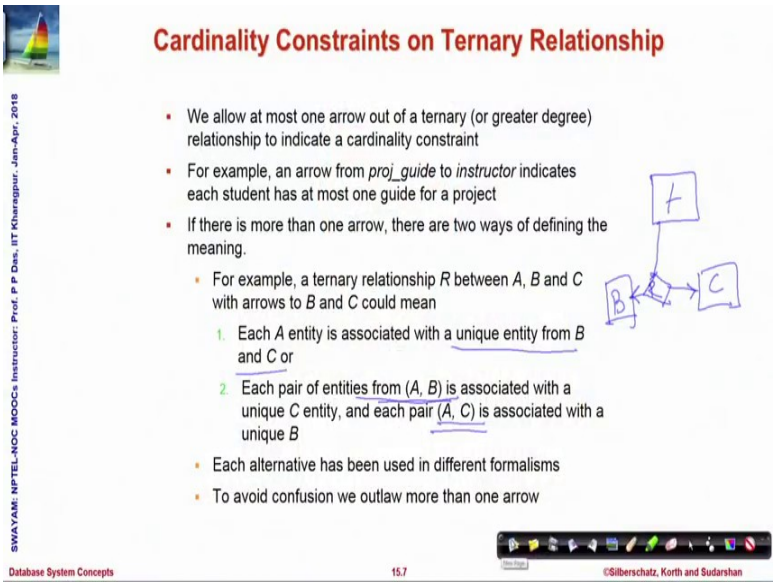
(Refer Slide Time: 03:02)



Now we have talked about cardinality constraints on binary relationship one to one, many to one one to many and many to many. And we specified that if we have a binary relationship say this is one entity set and this is another entity set and we have a relationship. So, if we just connect them it means a many to many relation, but if we have an arrow on one side entity set then it means on the arrow side its one.

So, this is from entity set E 1 to E 2 this is one to many; we could have arrow at both ends and; that means, one to one. Now the question is how will that work out what will be the meaning of arrow in terms of a ternary relationship.

(Refer Slide Time: 03:55)

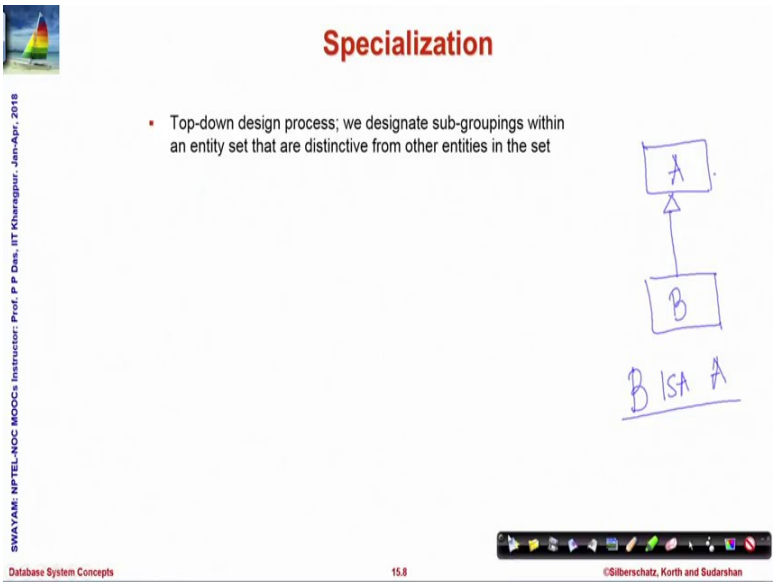


Now, in the case of ternary relationship or this would generalize to relationships of higher degree whether where more than three entity sets may be involved. We put a restriction that we will allow at most one arrow out of the ternary relationship. So, we could have only if we if we look at if we look at say this relationship then we could have an arrow only at this end.

But multiple arrows are not allowed and the reason is certainly to keep the semantics of the cardinality meaningful. For example, if we have a ternary relationship between A B and C let us say this is a A this is B and this is C and we have a ternary relationship between them. And let us say if we have a more than one arrow; there for example, suppose then we have say an arrow to B and an arrow to C the question is how should we interpret that?

Should we interpret that an entity of entity set A is associated with unique entity from B and C together or should we associate, should we interpret that the entities formed by the pair a B and the entity formed by the pair A C are uniquely related. So, there is multiplicity of interpretation; if we allow more than one arrow in case of eternity or higher degree relationship. So, what will follow for simplicity in this course and that is what is followed often in practice; is in case of a ternary or higher order relationship only one arrow will be allowed in that in it in that relationship.

(Refer Slide Time: 06:20)



Now, let us talk about specialization those of you who have some background of object oriented systems would be familiar with the notion of specialization and generalization in object oriented system. So, we say that if we have a certain concept say we have a concept called person and then we say that a student is a person; what we mean is a student is a specialization of person. And person is a generalization of student and in that process student inherits all the attributes of person, but in addition the student may have some specialized attributes.

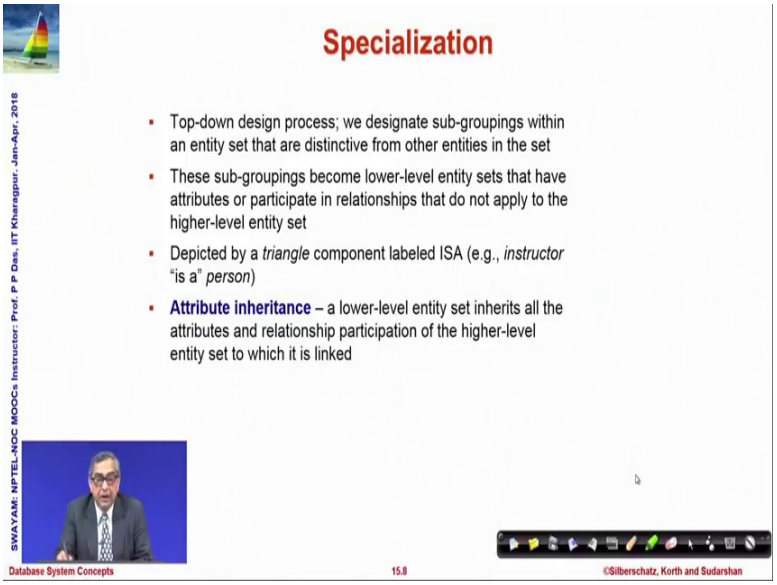
So, what it means that if you look from the perspective of such specialization; say if I draw like this; this is an entity set A and this entity set B and to mark the specialization we show an arrowhead with a blank triangle at that end. So, if we by this what we mean is B is a A. So, B inherits all the properties of A, but can have some more properties. So, if you look at all the entities that A may have you will find that a subgroup of the entities in the entity set A have some additional common properties.

So, if A is set of persons and B is a set of students then A may have entities who which represent people who are not students; who are employees, who could be retired and so, on, but there will be a number of entities. So, who have the commonality of being student they are enrolled in certain course of certain university and so on.

So, in terms of the E-R diagram E-R model what we do is we try to look at the entity A and move top down. So, that whenever we find a group of entities which have certain

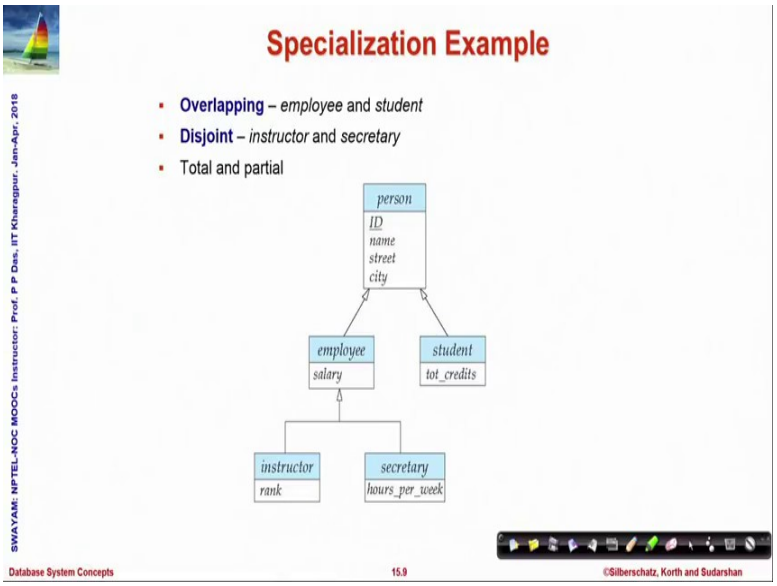
commonality; we move them into a lower separate, specialized entity set and relate these two entity sets to the specialization relation.

(Refer Slide Time: 08:54)



So, these sub groups form lower level entity sets and as I said that it is designated in a certain way. And as in the object oriented system the lower level entity set inherits all the attributes and relationships of participation of the higher level entity set. So, here is an example.

(Refer Slide Time: 09:16)



Can say the person at the so, called root of this hierarchy of specialization; it has a set of properties ID, name, street, city and employee is another relationship; another entity set which is a specialization of person relation person entity set.

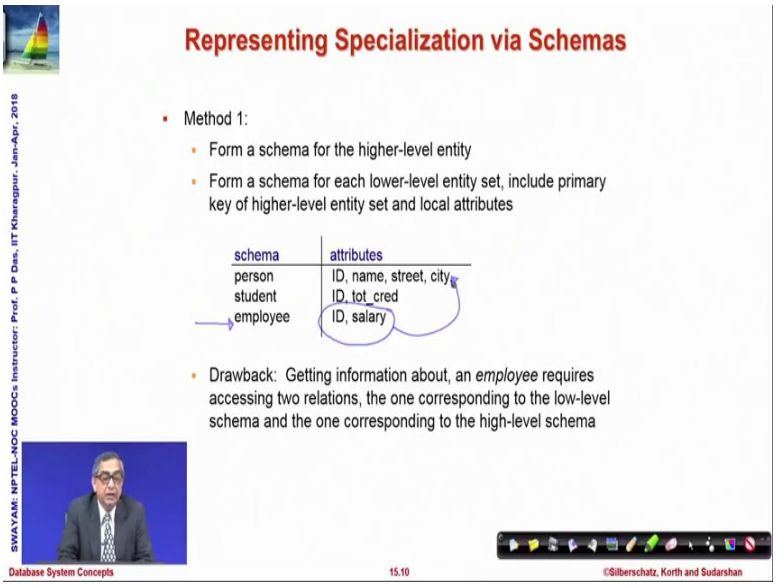
So, employee inherits all attributes ID name street and city, but in addition the commonality of the entities in the employee entity set is a fact that all of them have a salary attribute. A similar entity set student is a specialization of person where the again all attributes are inherited, but there is a common attribute called total credits which is common for all the students, but is not available or common for the persons in general.

And as you can see that it could be hierarchical it could go further down employee could be specialized into instructor and secretary. Again by the rule of specialization instructor will inherit all attributes of employee which means that it will inherit attributes of person; which imply has inherited plus the employee specific attributes salary. So, it will inherit all five of those attributes and then it adds another attribute which is specific for the instructor which says a rank which is another specific attribute that you have.

Now, when we specialize a certain entity set into two or more entity sets like we specialized person in employee and student; then there could be different situations that could exist for example, certain entity may be a member of both employee as well as student. If that happens then we say that their overlapping entity sets or they could be disjoint where no member of instructor would be a member of the secretary and vice versa.

So, we say that this disjointness tell us that no instructor can be a secretary and no secretary can be an instructor. Whereas, overlapping specialized sets denote that well an employee may or may not be a student, but it is possible that some employee is also a student and vice versa and that is how we will represent this. And we will see that when we specialized the specialized entity could be total or they could be partial.

(Refer Slide Time: 12:06)



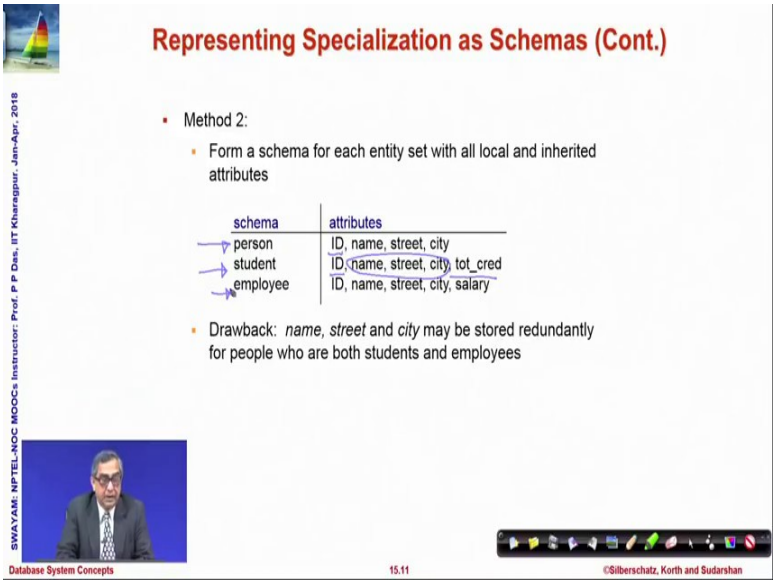
We will talk about that totality and partiality little later; let us look at how do we represent this information in the relational schema because as we have seen that whenever we have a E-R diagram; it is important to find out what is the relational schema that will be corresponding to that E-R diagram or E-R model. So, here we could do this in two ways one that we are showing here is form a schema for the higher level entity. So, form a schema for the person as you can see here that person is described in terms of four attributes and this form a schema for each of the lower level entity set where you include the primary key of the higher level entity set.

So, when you are forming the schema of person of student which is a specialization of person you include the ID which is the key of the higher level entity set person. And along with that you include the so, called local attributes or attributes which are specific to this lower level entity set in this case total credit similar thing happens with employee.

Now this representation is in a way optimized because you are representing the information only once when it is needed, but the drawback is if you have to find out information about say employee; then you will not only have to access the employee entity set or the corresponding relation in the relational schema, but you will also have to access the parent or higher level entity set to get the attribute values which are inherited. And if you have a multi level hierarchy as we have shown this could involve accessing multiple relations to find information about a single entity in an entity set.

So, this is an in terms of data representation this is an optimized representation, but it has the overhead of having to access multiple entity sets to get information about certain entities.

(Refer Slide Time: 14:22)

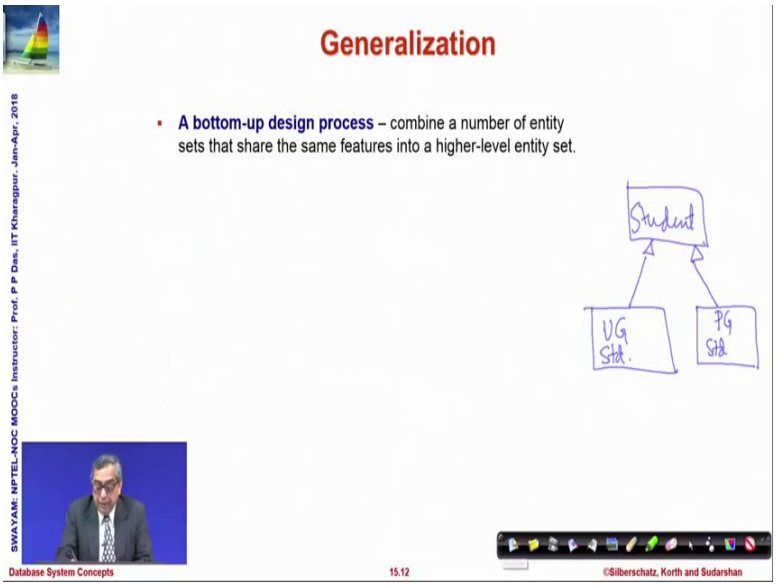


An alternate scheme would be that based on the hierarchy of specialization, you assume all attributes as they are inherited and then represent every entity set in full. So, when you the representation of person does not change, but when you represent student now earlier you are just having ID and total credit; now in you include all entities that are inherited.

Similarly, you do the same thing; so, you have the all entities of the parent to all attributes of the parent entity set as well as the local attribute of that specific entity set. Now this naturally makes it easy to extract information from a for a single entity set, but at the same time, you are storing the same data redundantly for people who are having overlapped representation. So, if we have as we know student and employee overlapped. So, the same entity will happen in student as well as in employee; so, it will the information of the common attributes name street city etcetera they will occur in both these tables in the design.

So, these are two methods and now we have just given you the relative advantages and its advantages of the same and based on a particular situation you will have to choose what is a good method to represent.

(Refer Slide Time: 15:51)

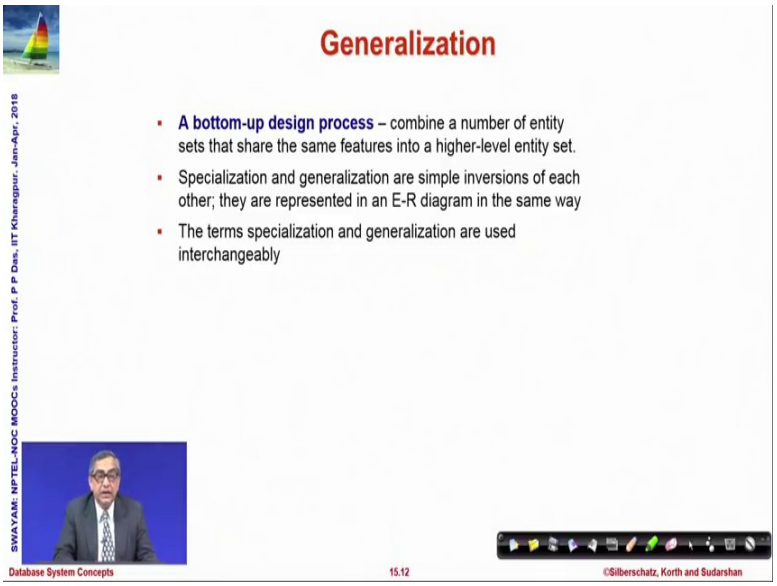


You can look at now you know from the object based system that if you have a specialization hierarchy you can think of it as a generalization hierarchy also.

The generalization hierarchy goes in a bottom up manner. So, instead of actually starting with an entity set and finding out subsets of entities which have greater commonality between them and putting them as specialized, you could actually group them in terms of finding out what they share and create a higher level entities. For example, the way I am saying is let us say that I have one entity set which say UG student and have another entity set in the same university which say PG student.

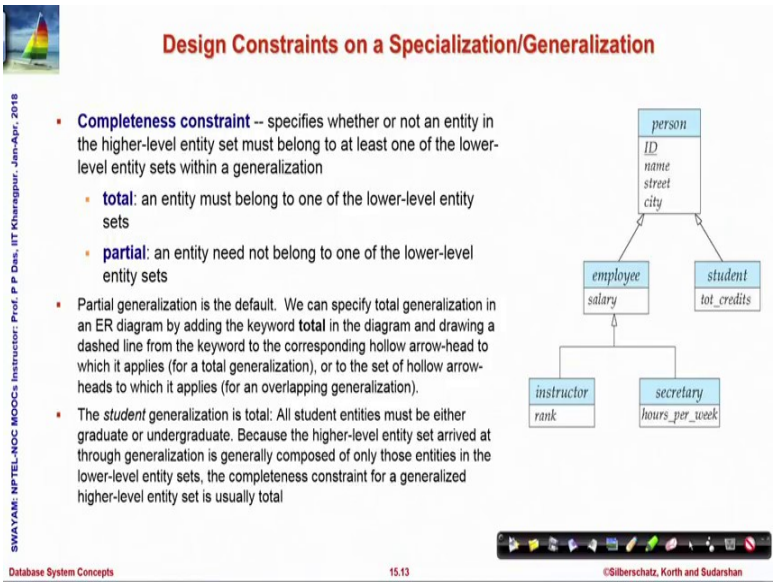
So, there are both of them are students naturally they are disjoint a person cannot be UG as well as PG student at the same time. And once you represent that you find that well there are a lot of information which are common between these two entity sets like the student roll number, name and so on so, forth. So, you could choose that well you instead of having them as two separate entity sets, you could extract out the common attributes and put them at a higher level entity. So, all that you are doing is instead of going top down you are going bottom up in the whole approach.

(Refer Slide Time: 17:21)



So, if you do that then there you can easily see that specialization and generalization is are inverse of each other and they are used interchangeably in terms of the relational entity relationship design.

(Refer Slide Time: 17:35)



The other constraint that you can identify, you should identify is the constraint of completeness which say that if I have a entity set say a person. And then we have specializations of employee and student; the question is for a higher level entity set is it

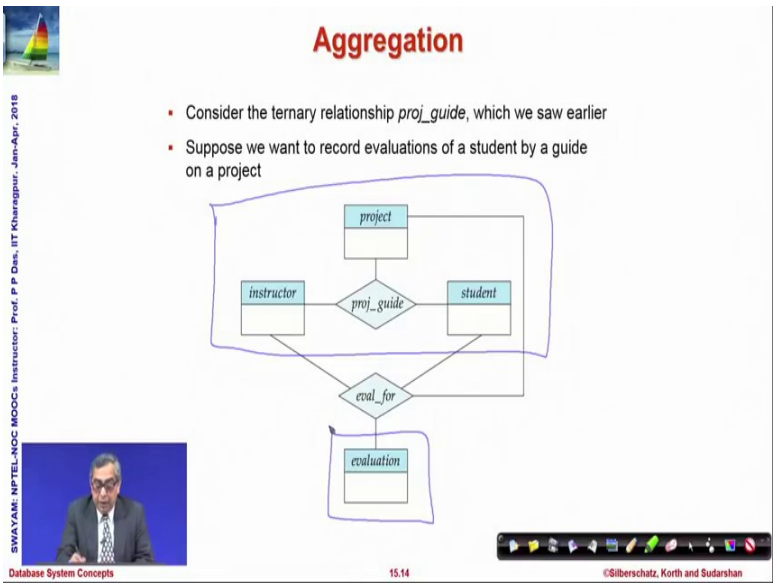
necessarily that every entity will be represented in one of the or more than one of the lower level entity sets.

If that is guaranteed that an entity must belong to one of the lower level entity set we say that it is a complete specialization. But if it is that a high level entity is may or may not be featuring in a entity set, which is at a lower level then we will say it is a partially specialized hierarchy. So, the both of these are possible depending on different situation that we have. So, by default we assume partial specialization and. So, if we want to say a certain specialization is total we will have to write the keyword total by the side of the arrow head that is representing the specialization hierarchy.

You can say that the example I talked of in uniting unite undergraduate and graduate or postgraduate students into the entity set of students, gives you a hierarchy which is total because every entity in the entity set student must be either a UG student or a PG student; it is not possible that I have a student who is neither a UG student nor a PG student. So, every high entity at the higher level entity set student must feature in one of these two specializations.

So, therefore they are necessarily total in the relationship. So, this is the completeness constraint that you can think of.

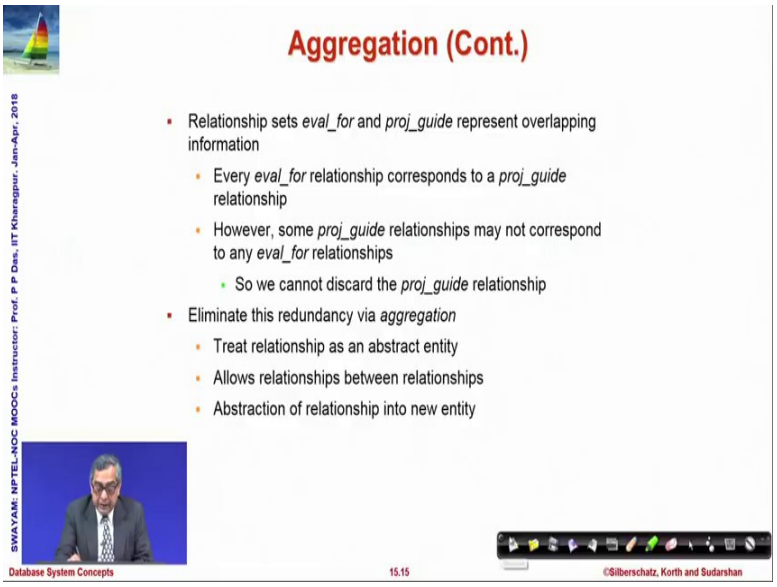
(Refer Slide Time: 19:52)



Moving on let us talk about another feature which is known as aggregation; the situation is like this we have already talked about this part of the diagram which is a ternary relationship which relates project instructor and student.

Now, let us say once the project progresses you would need to add evaluation to that. So, there is another entity set which represents evaluation and how we are grading or putting marks and so, on. So, naturally the evaluation of a student will be dependent on the project, the student and the supervisor and that will relate to the evaluation. So, evaluation eval for the relationship is necessarily a relationship between four entities or four entity sets so, to say.

(Refer Slide Time: 20:47)



Now, the question is how do we represent this information? The relationship sets eval for project guide the two that we saw if we just want to recall once more. The project guide involves three of the relation entity sets and the eval for relates to four of the entity sets. Now every eval for relationship corresponds to a project guide relationship; that is if I have an entity in eval for relationship, I will have a corresponding entity in the project guide relationship which specifies the student project and the instructor.

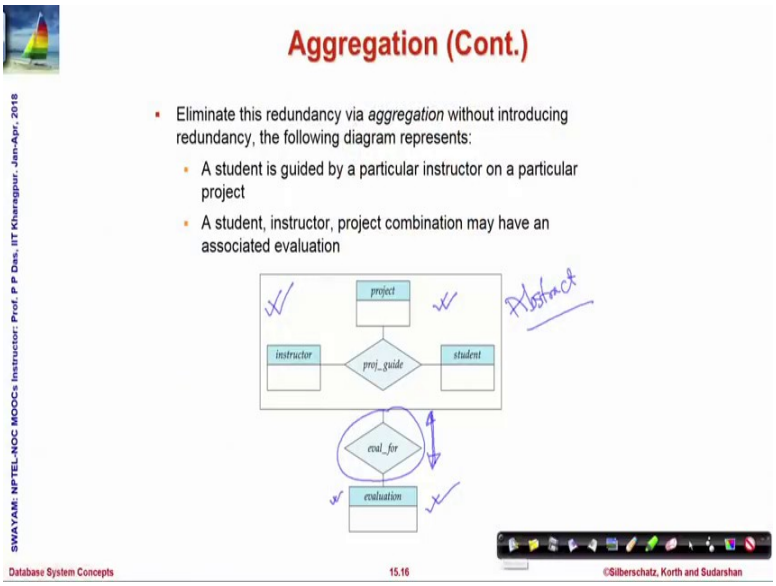
So, but it is other way it is possible that some project guide relationship may not correspond to any eval relationship; that is it is possible that there is a allotted project by a student with a particular instructor which has yet not been evaluated; the evaluation process is not complete or the time has not come.

So, if we have to represent the information only for the eval for relationship; we will get partial information because it is possible that some entities in eval for does not have the eval for information do not feature there, but need to be preserved because I need to remember the project guide, instructor, the student and the project. So, we need to keep both duplicating the information.

So, we can use aggregation to eliminate this duplication of information or redundancy of information. So, what we do is we treat the first relationship; the project get relationship as if it is an abstract entity and then you allow relationship between two relationships; this is something we did not do before relationship so, far has always been between entity sets.

So, what you can see that project guide relationship itself as if it is an virtual entity; it is an abstract entity and then you allow the relationship between the project guide and the eval for relationship which relates to the evaluation entity set this really this shows I mean I will just show you in the diagram.

(Refer Slide Time: 23:08)

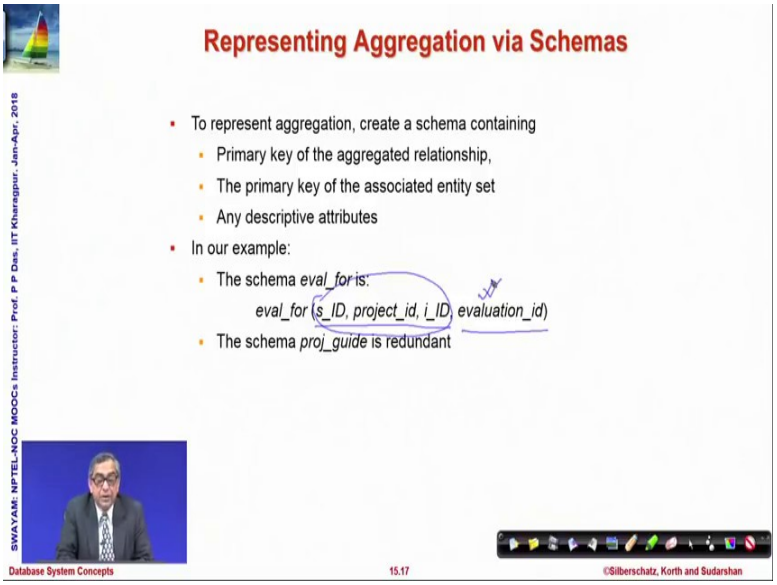


So, this is how it will now work out to be. So, this is the abstract project guide entity set which is an abstract entity set because it is actually relationship. And that relates to eval for which on the other site has the evaluation. So, what will happen is a student is guided by a particular instructor in a particular project will feature in this abstract entity set; which relates the three entity sets project student and instructor. And if it has an

evaluation then the; this through this relationship it will be represented and the evaluation value will exist.

So, we know that if a project is evaluated then it certainly have a corresponding entity in the abstract entity set project guide, but the reverse may not be true I may have an entity in the project guide entity set which does not have an evaluation. So, by using this aggregation model; I can represent the information of this situation model this situation more accurately than I could do otherwise.

(Refer Slide Time: 24:30)

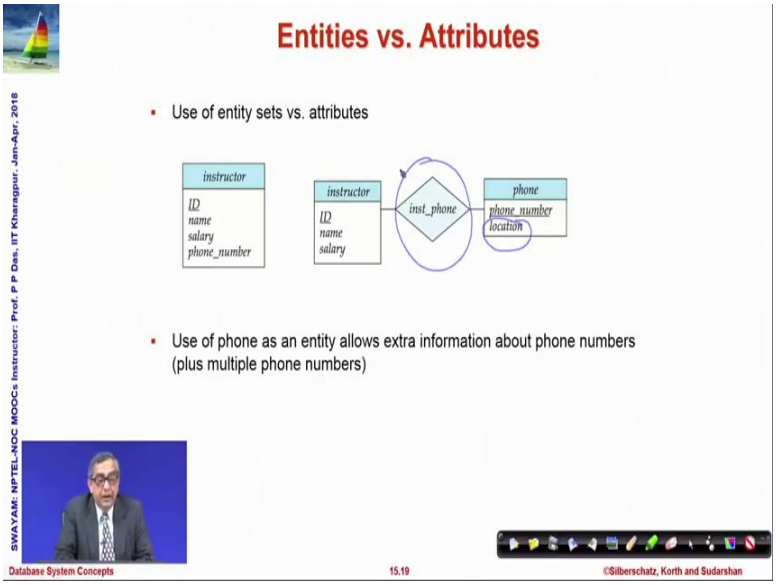


So, this can be represented again how to represent this in terms of the schema? So, what we do we represent the aggregation we create a schema containing the primary key of the aggregated relationship. The primary key of the associated entity set and all the other descriptive attributes and put them together.

So, in our example the schema would be eval four and that schema will have these are entities these are attributes of the aggregated or abstract entity set which is coming from the student project and the instructor entities and this is for the evaluation ID. So, we put this together; so, now, you can see that all of these are related representing who is the guide of which student in what project and if this exists then this gives you the evaluation.

So, naturally once this has been represented; the project guide schema by itself becomes redundant and therefore, it can be removed. So, this is a process through which we come to the decision of actually having this schema to represent all the required information. Naturally if the evaluation is not done then the evaluation ID for in eval for will not exist and that will be a null showing that it is not present right now ok.

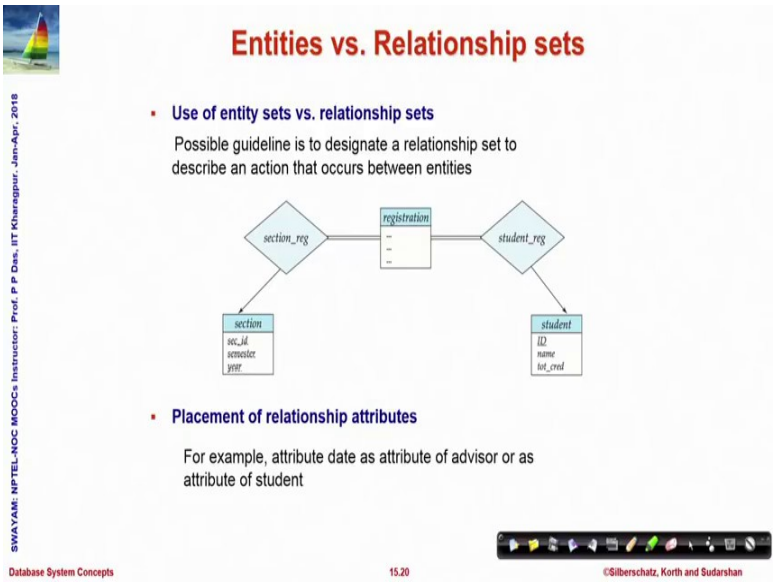
(Refer Slide Time: 26:25)



Now, given these basic features as well as the extended features let me talk about a few design issues which will be required to see what kind of information that that the different challenges that we have seen so, far. For example, we have seen the case of multivalued attributes.

So, and the way we can represent that is using that multivalued attribute as a separate entity set like the phone number which also has the advantage of having its own added information. For example, once we do this then not only I can have against the same instructor ID; I can have multiple phone numbers, but I can have location for each one of these phone numbers. And I make use of this relation relationship that I create which allow me to represent this multi valued attribute.

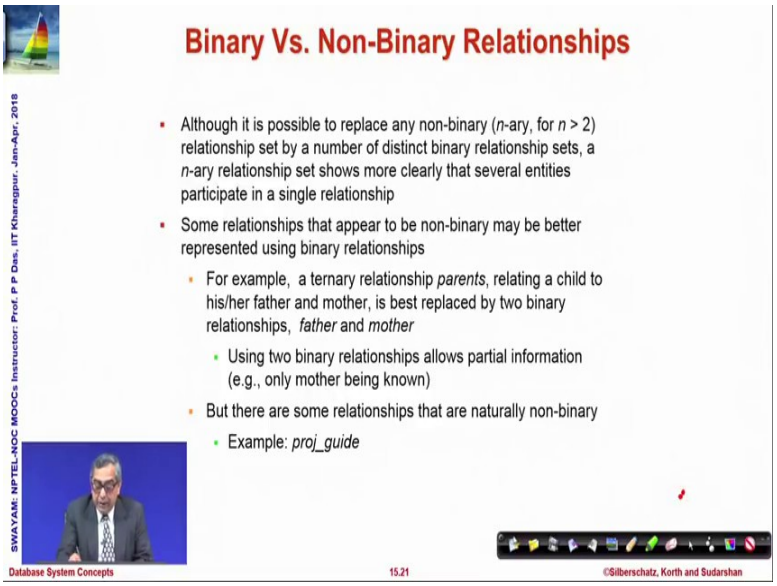
(Refer Slide Time: 27:30)



So, this is a common technique that will be used frequently in such cases you can have entities versus relationship for example, if we have inform we need to keep information about registration how students register to different sections; then we could represent registration as an entity set and have different relationships of section registration which specify how registration is related to section and student reg; which specify how do the station is related to students to represent that kind of information.

We can have placement of relationship attributes also attribute date we have talked about as an attribute of adviser to designate as when that particular instructor became adviser of a student is a common situation that we have already seen.

(Refer Slide Time: 28:25)



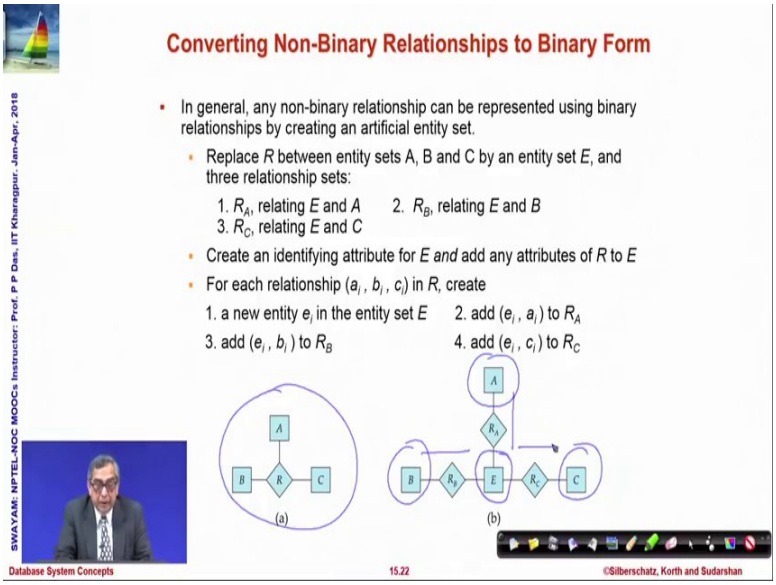
There is also question of the choice being made between the binary and non binary relationship ternary or higher degree. Now as it turns out that it is possible that you could represent ternary relationships directly or you can decompose that. For example, a ternary relationship can be decomposed in terms of two binary relationship; for example, let us say if we talk about persons then person every person has parents; so, he or she has a father and a mother.

Now, if we represent this as a ternary relationship then the one difficulty that we have that a person must have both the father and mother to be represented there. For example, if we can come to a situation where only the mother is known, the father is not known I will not be able to represent that because it will always have to come as a triplet of three persons thel the person under consideration her father and her mother, but if I represent the person and the father in one relationship; the person and the mother in another relationship, then I can take care of the situation where when one of the parents are known; I can still represent this.

So, there are certain tradeoffs which can be done between the choice of binary and non binary relationships, but; obviously, there are certain relationships which are inherently non binary for example, the project guide example we have seen. The project guide information cannot be decomposed without certain loss of information to be represented

by say the instructor and the project and another relationship between the student and the project it really that does not represent the same information.

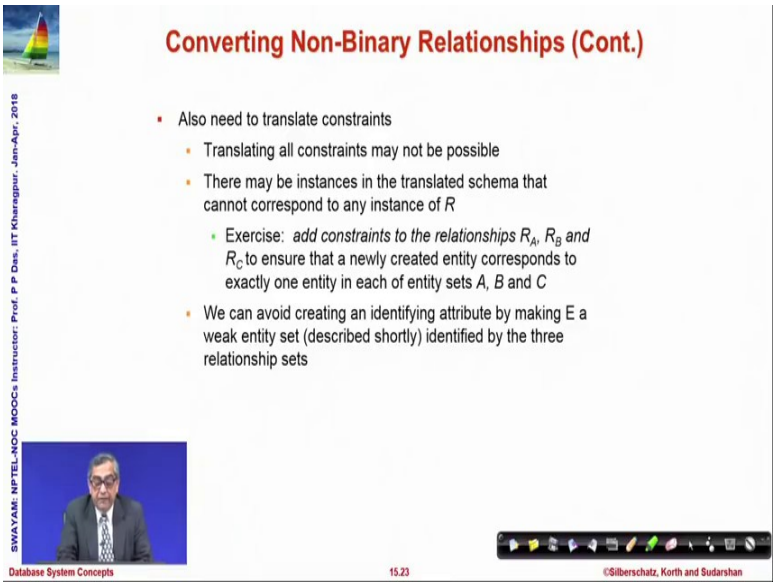
(Refer Slide Time: 30:20)



So, in general you can convert a non binary relationship by in the binary form by doing this. So, this is a ternary relationship being shown and for doing that these are the three entity sets involving the ternary relationship and to make decomposite into a ternary relationship; what we do is into binary relationships we inject a new entity artificial entity set E and then we define three different relations between them.

So, which individually relates to the entity sets A B and C. So, this is a standard decomposition and you can easily understand that A B and C in our earlier example could all be persons and R A could mean that father of R B could mean mother of and so, on. So, I can do it in decompose it in this manner and represent that.

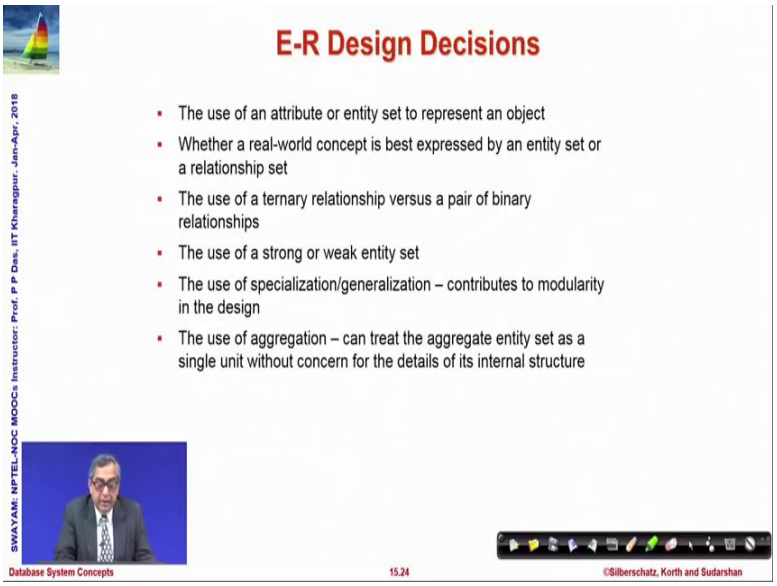
(Refer Slide Time: 31:24)



Now while we do this decomposition we will also have to remember in that; we need to translate all constraints that are present for the ternary relationship. And often times it may become difficult to translate all constraints, it may not be possible and there may be instances in the translated schema that cannot correspond to an instance of the original relationship.

So, we will have to avoid we can we will have to take care of this situation by identifying attributes. And making use of the weak entity sets which we have already seen in our earlier discussions.

(Refer Slide Time: 32:09)



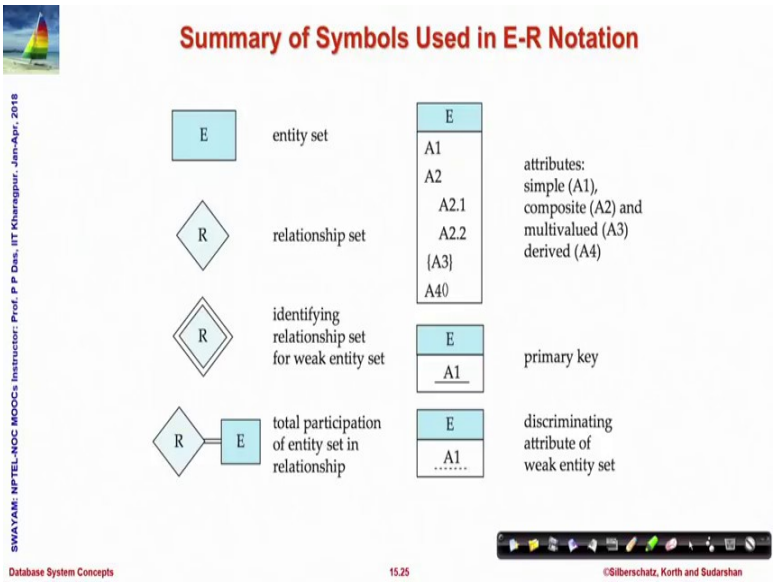
So, if we summarize the discussions on the E-R design decisions; we see that the first decision that we need to take in case of design is the use of an attribute or entity set to represent the object.

So, that is the first modeling that what is the concept and what is what are the attributes or what is the representing entity set for the object that we are trying to deal with instructor, student project and so, on. And we will also have to see whether in the real world this actually is an entity set or it is a relationship set that it is not and concept by itself, but is a concept which relates two or more entity sets and thereby becomes a set of representation.

The use of ternary relationship versus a pair of binary relationship; this trade off will have to be weighed as a design consideration; we have to look into the use of strong or weak entity set. So, we will have to identify the weak entity sets and see if they should be represented through the identifying relation as against a strong entity set. We have to identify the specialization generalization situation where so, that we can get more specific information and create appropriate modularity in the design.

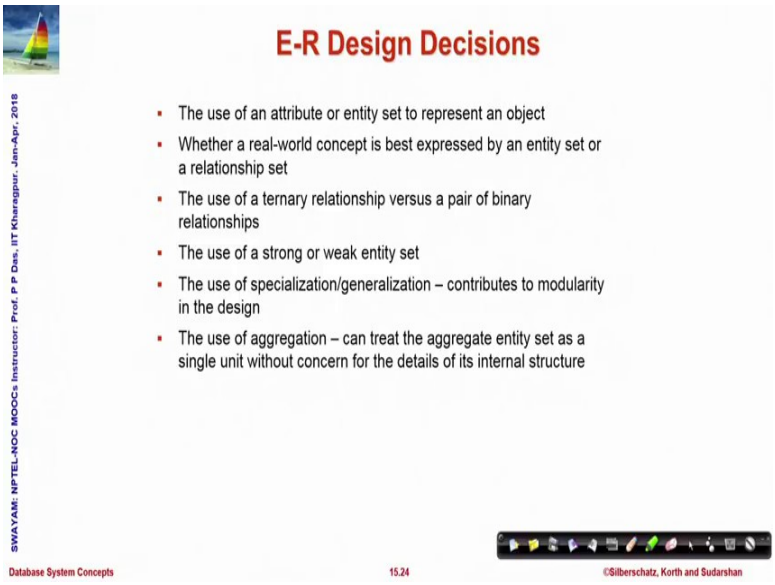
We have to look at aggregation which where we can aggregate entity sets bound by a relationship and create an abstract single unit which can play a role of an independent entity set in the whole design.

(Refer Slide Time: 34:00)



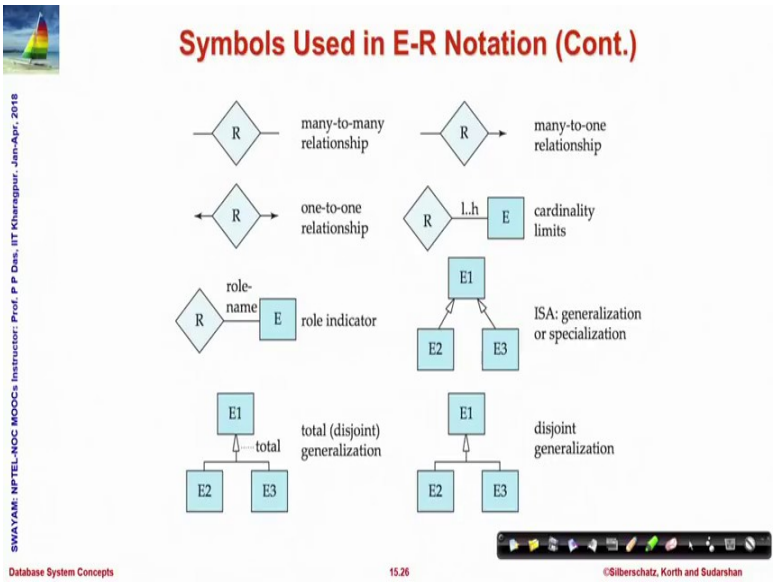
So, this is the basic. So, these are the basic design decisions that you need to make.

(Refer Slide Time: 34:02)



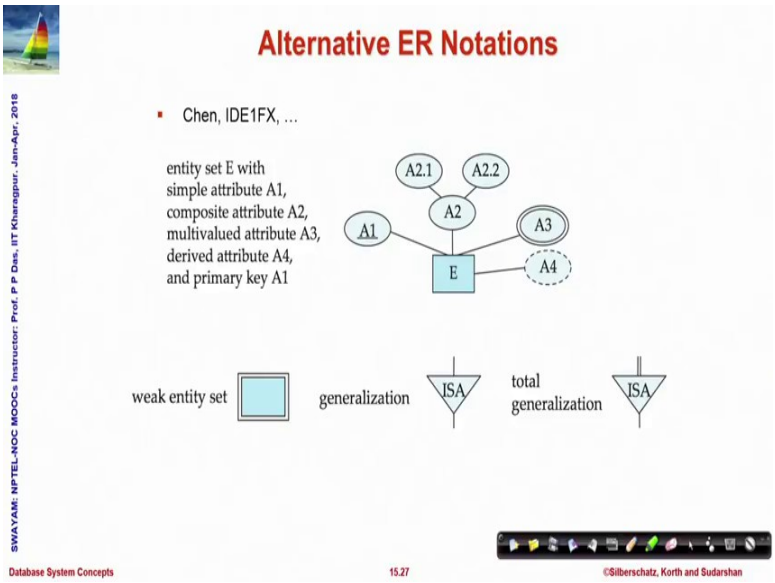
And we will certainly come up with lot more of design decisions as we go along. And before I close in the presentation I have summarized the different symbols that are used in the E-R notation. So, I will not these have already been discussed in depth. So, I will not go through them one by one, but I have put them as a list in the couple of slides.

(Refer Slide Time: 34:30)



This is a next slide in that which will be a quick reference for you while you are initially doing the E-R diagram so, that you know exactly which symbol to pick up for what situation.

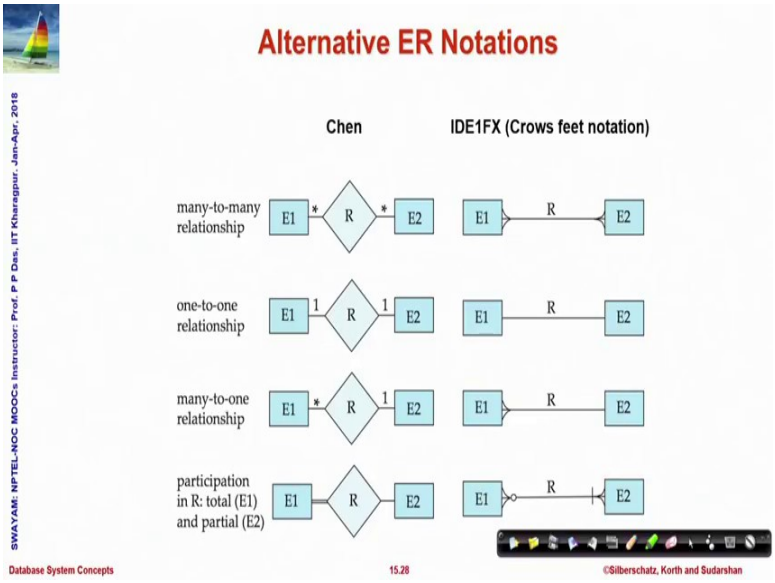
(Refer Slide Time: 34:42)



And at the end also there are few slides which show you that the E-R notation itself is not a unique one.

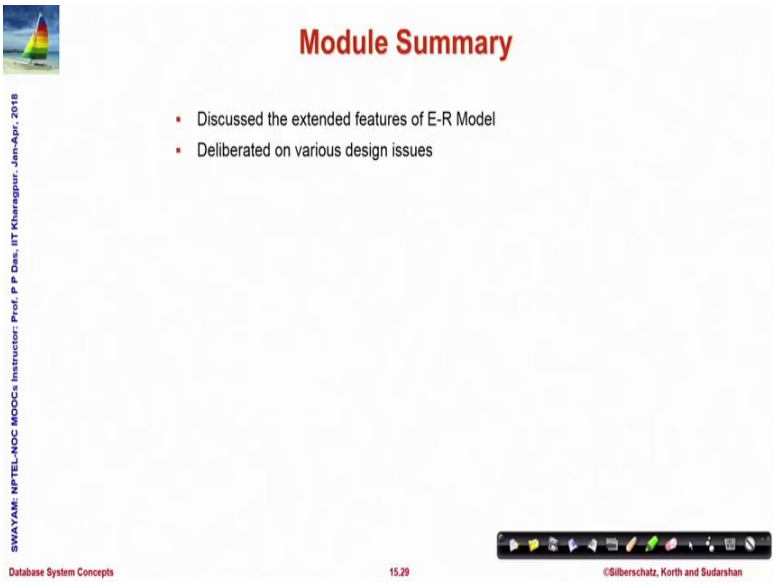
There are multiple ways to represent similar things for example, this is one which is showing you different composite attributes, the generalization, relationship is shown differently.

(Refer Slide Time: 35:04)



So, there are; these are all different styles of showing the constraints that that apply to a particular relationship. And we will I mean we have included this not because we will use these alternate notations, but I have put them because it is possible that you come across some E-R diagram where these notations are used and if you come across and you are not able to identify then please refer to this slides.

(Refer Slide Time: 35:36)



And you will be able to recognize what is what is corresponding symbol that you already know.

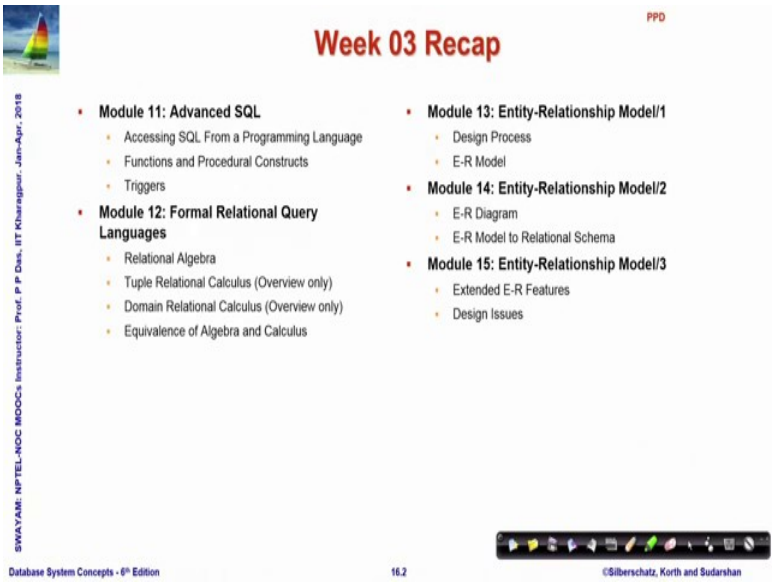
So, in this module we have discussed the extended features of E-R model and we have deliberated on certain design issues. And we will close our discussion on the entity relationship model here and move on to discuss the actual relational design.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture - 16
Relational Database Design

Welcome to Module 16 of Database Management Systems till the last module which closed with the third week.

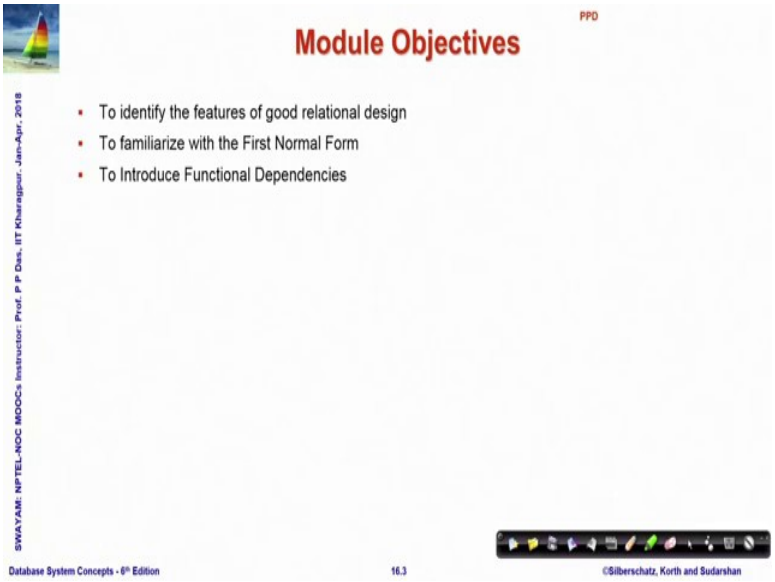
(Refer Slide Time: 00:31)



Specifically in the third week, we talked about certain advanced features of SQL and the formal query language in terms of relational and algebra and calculi and then, we talked in a depth in terms of the entity relationship model, the first basic conceptual level representation of the real world that we can do in terms of designing a system.

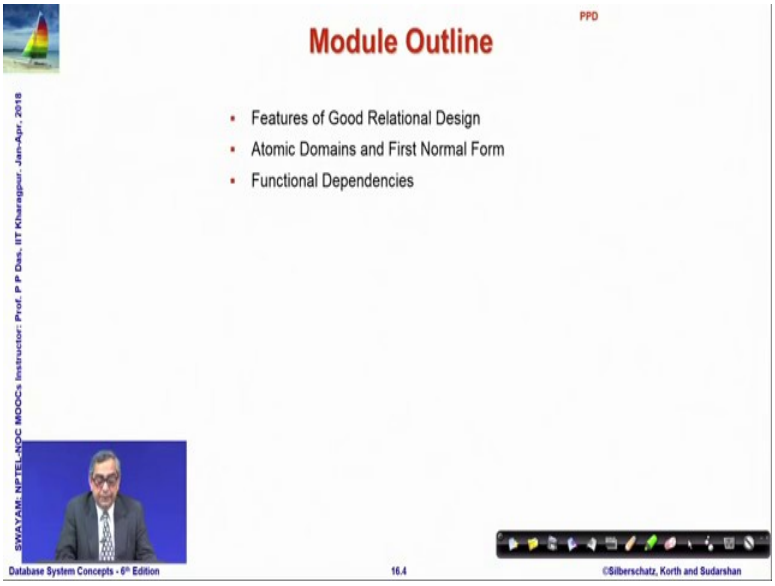
Now, our next task would be to take it to more proper complete relational database design and this will have a lot of theory at different levels that we need to understand. We will slowly develop that and this discussion will span five modules that is we will take the whole week to complete.

(Refer Slide Time: 01:25)



So, the objective of the current module, the first of the Relational Design Module is to identify features of good relational design having done the ER model. We have ER model we do the ER model, we have the entity sets relationships, we convert them to schema. We have seen how to do that and immediately we have some design, but the question is, is it a good design. So, we will discuss about what are the features of a good design and then, we will introduce the formal definition of what is First Normal Form and we will introduce a very critical concept of relational database design, the Functional Dependencies.

(Refer Slide Time: 02:07)

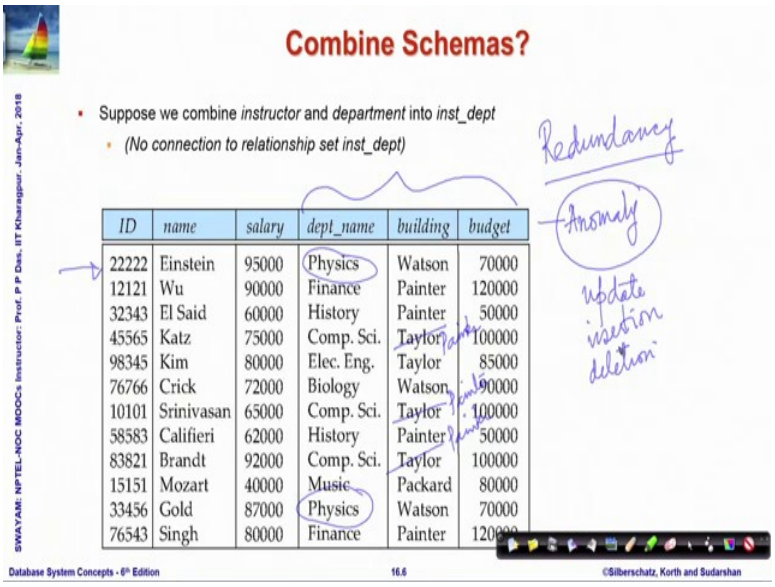


These are the module outline for that.

(Refer Slide Time: 02:10)

So, to start with the features of good relational design, let us take an example.

(Refer Slide Time: 02:13)



Suppose we have seen the instructor relation, instructor entity set as a relation. You have seen the department relation. Now, let us consider that if these two were not two separate relations, if they were all kept in a common relation that is all the attributes are kept in the common relation, so earlier if you recall that your instructor relation was this and your department relation was this much. So, if we keep everything together, of course we

are calling it inst_dept, but please keep in mind this is not the same inst_dept that we discussed in terms of the ER model. This is just putting these two together.

Now, the question is if you look into this data carefully, for example if you look into this particular row, if you look into this particular row and if you look into this particular row, these are rows of instructors who all belong to computer science. Now, earlier we were representing the information of instructor only in this part. So, we just knew that it is computer science and we represent the information of department in this part. So, given a department name say computer science, we knew, where is it located, the building and what budget it has. Now, when we are combined, we will see that naturally since computer science is located in the Taylor building, we know that it has a budget of say 100,000. So, all of these records will have this information repeated.

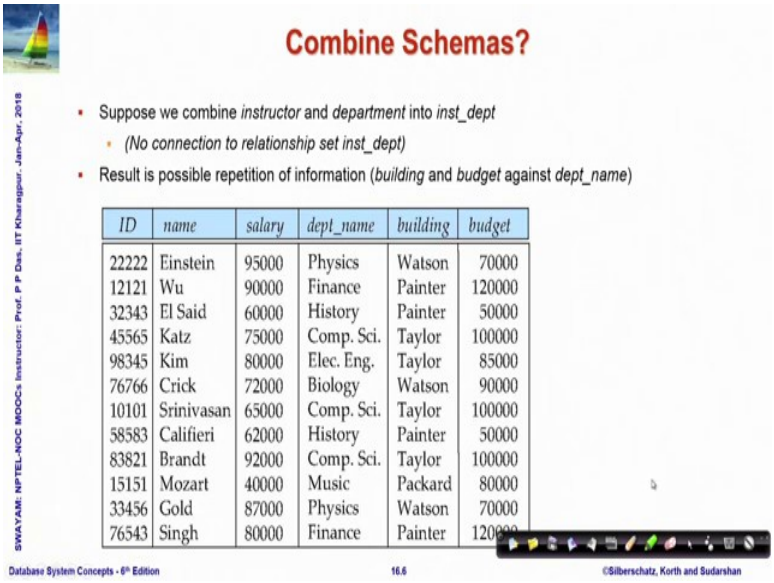
So, this is not a very good situation. This is not a good situation because this kind of situation is typically in database is known as redundancy, that is you have the same data in multiple places. So, what is the consequence of redundancy? For example, there could be different kinds of anomaly when you have redundancy. What is an anomaly? An anomaly is the possibility of certain data getting inconsistent. For example, let us say Computer Science department moves from Taylor building to Painter building. Now, what will have to happen if it moves to painter building? Then, I will need to remove this, make it a painter, make this value painter. I have to also do this, make this painter. I have to also do this, make this painter. So, if I have a change, then I will have to make the change at multiple entries. Think about the earlier situation where I just had these three in my department relation, then naturally computer science had only one row and therefore, this change, this update could be done at only one place.

So, it is not only that if while doing this in case of this redundancy, I have to do this multiple times. It also has the difficulty that if I forget to update any one of them or more of them, then I have inconsistent data. Similarly, if I want to insert a new value, I will have to do that for all this redundant information. If I have to delete say for some reason let say the university decides to wind up the Physics department, then I have to delete all these rows which have physics as an entry and the consequence of that is the department is deleted, but as a consequence of that I will delete the whole row and therefore, I will not only remove the department, but I will also remove the corresponding instructor who was enrolled for that department.

So, this kind of redundancy can lead to different kinds of anomalies in a database design. On the other hand, if you look at, well why am I complicating the whole situation? We have already had a good design in terms of where these anomalies were, not their departments were separate instructor was separate. In that case, the situation is that to answer some of the queries, I may have to do a very expensive join operation. For example, if I want to know if Einstein wants to know what is the budget of his department that cannot be found out from the earlier instructor database, instructor relation which had only these fields.

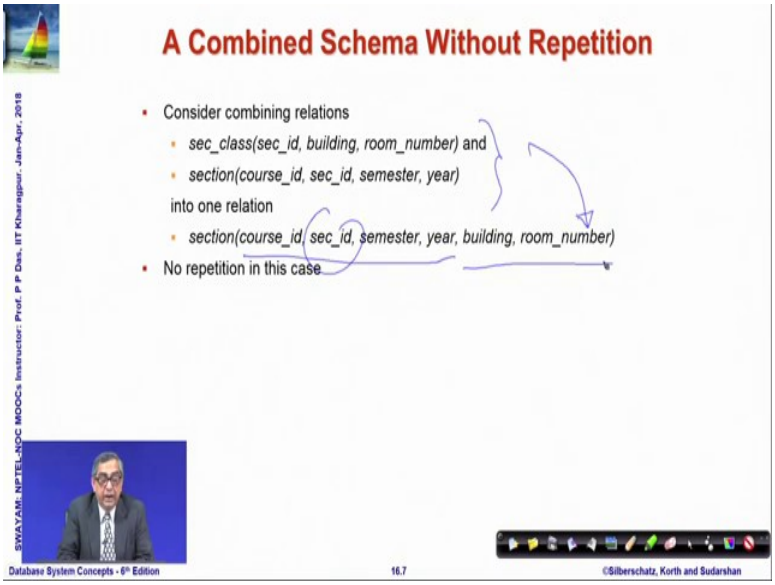
So, I have to pick up Einstein from here, do a join based on the department name, dept_ name with the department table department relation and then only, I will be able to find out that an Einstein belongs to Physics. Physics has a budget of 70000. So, Einstein's department has a budget 70000. So, there is a tradeoff between how much data information if you make redundant and lead to different anomalous situations or how much data you optimize in the representation, but get into the possible situation of having a higher cost in terms of answering your queries.

(Refer Slide Time: 07:49)



So, this is one of the core design issues that we will start with. So, let us look into some more.

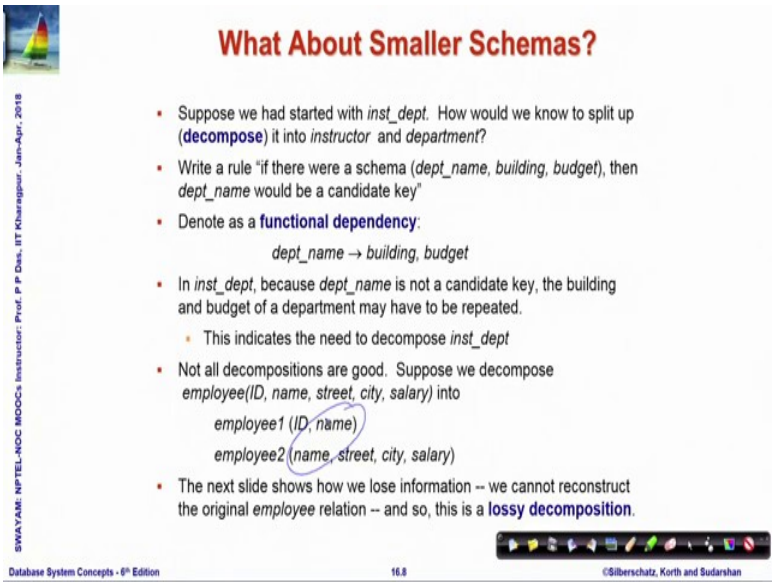
(Refer Slide Time: 07:54)



Of these examples, let us say we look into another combined combination of schema. Suppose section is a relation which have the sections of a course which give the section id, semester, year and say section class is another relation which tell me for a section id, what is the building and room number where it is located. So, if we have this kind of relations combined into a common relation, then I have all of these coming from the section and this and these coming from the section class, but we can see that there is no repetition or redundant information in this case.

So, it is note that the combining schemas is necessarily always bad in terms of repetition or in terms of redundancy. So, different situations will have to be assessed.

(Refer Slide Time: 08:57)



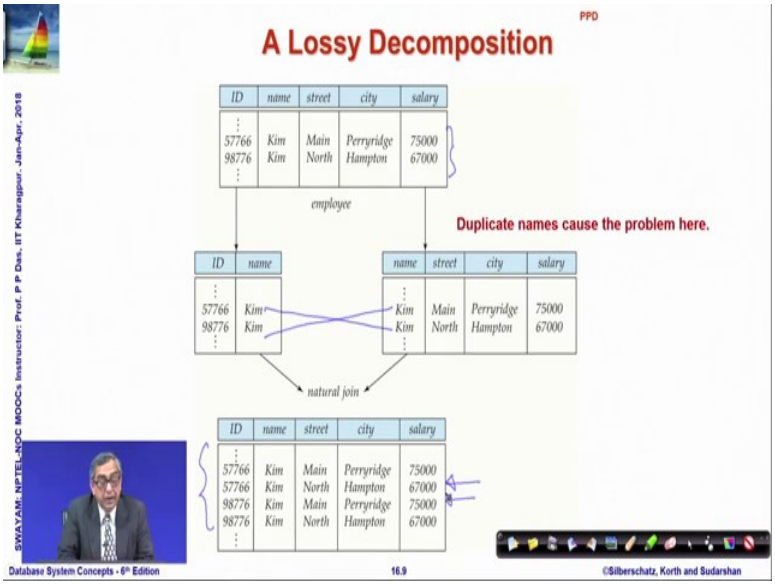
So, if we want to look at the other side that if we just as I said that if we make the schema smaller, so that we avoid redundancy and then, what we see that 12 from the combined inst_dept relationship that we saw. So, let me just show you once more. So, this is if we look at the inst_dept, then in this we can we know that from the earlier information about the department relationship that department name is a key, is a primary key of the relation which has department name, building and budget. What is the consequence of being a primary key? If it is a primary key, then no two records can match on the department name and be different in terms of the building and the budget.

If two records are there which have the same department name, they must be identical. So, they are distinguishable completely by that. So, let us see what is the consequence of this. So, we are saying that we write it as a rule that if there is a schema department, name, building, budget, then department name would be a candidate key and we write this observation that if two records match on the department name, they must match on the building and budget and very loosely, we will come to the formal definition. Very loosely we call this the functional dependency. We say that the building and budget is functionally dependent on the department name and that is a situation where we can split this inst_dept and create a smaller relationship because department name is not a candidate key in the inst_dept. It does not decide the records of inst_dept uniquely.

So, since it does not, so when the values of this key, this attribute department name is duplicated or triplicated, the values of the building and budget are repeated and we have the redundancy. So, this is a situation, very common situation which is indicative of the fact that we need a decomposition into smaller, but at the same time we can also observe, I mean let us take a different example. If we are thinking that decomposition is the panacea of solving these kind of redundancy and related problems, then let us try to see a different relationship employee which has id, name, street, city, salary and we want to make it smaller and want to make two relations id and name and name, city street, salary.

So, if we do that, then how do we get the salary for a particular id? We will naturally have to join these two relations in terms of the common attribute name. We have seen that in the query and the question is when I do this join, do I get back the original information or I lose some information.

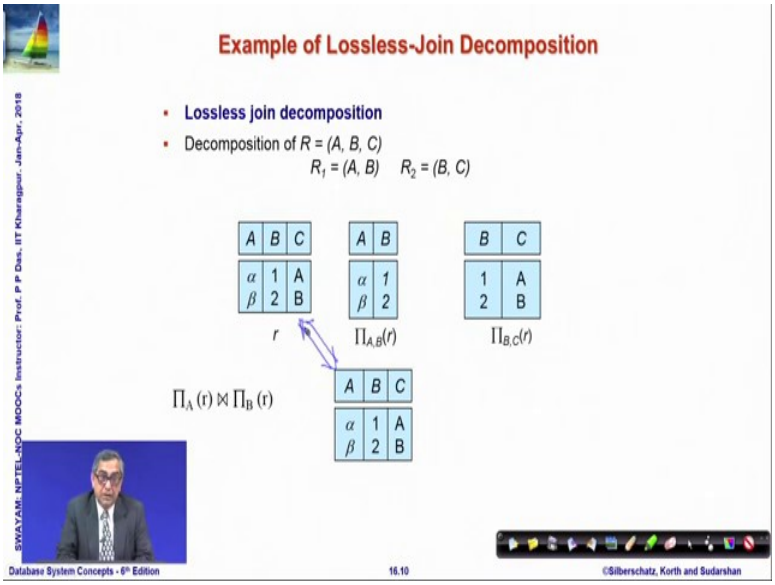
(Refer Slide Time: 12:42)



Look at an example. So, here is an example of the combined instance and I have two different ids, but incidentally the names are same. The names of these two distinct employees are same. So, when I decompose, I get this relation which shows id and name. I get this relation which against the name shows this, but when I try to join them by national join, I not only get the combination of this with this which is what I need, but I also get this combination. So, if I say this is what I get as well in terms of natural join, this is what I get as well in terms of the natural join which are really not there in the original relation.

So, you can see that in the natural join, I get four records, I get four rows whereas, in the original one I had only two rows. So, I get some entries which are actually erroneous. These are not there in the database. So, this is when this happens. We say that we have loss of information and such joins are said to be lossy joins. So, when we decompose, we need to make sure that our joins are lossless in nature; otherwise that is not a good design.

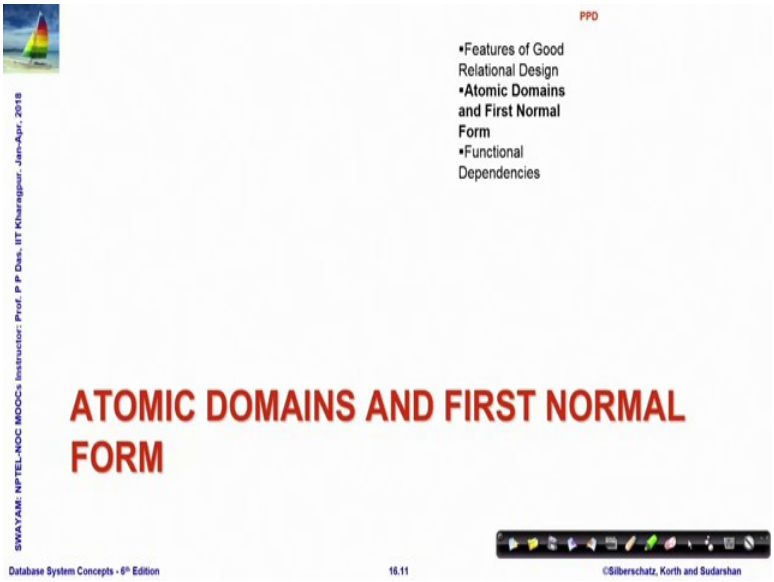
(Refer Slide Time: 14:08)



So, you can see this is again a hypothetical example which shows three attributes in relation having three attributes. You have decomposed it into two relations having two attributes each and we have shown an instance and in this case, it shows that when I take the join, the original information I am sorry, wait.

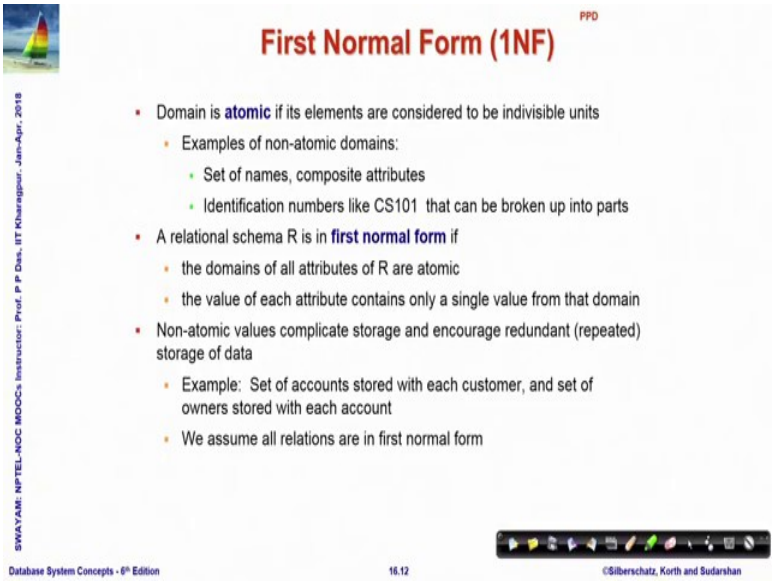
When I take the join, the original information is completely retrieved. I get back the same table and when that happens, I say that the join is lossless. So, what we need to understand is on one side there is a need to decompose relations into smaller relations to reduce redundancy and while we do that, we will also have to keep this in mind that the smaller relations must be composable through certain natural join procedure to the original relation, and I must get back that original relation, otherwise I have a lossy join which is not acceptable. Also, the decomposition will have the costs of doing natural join every time I want to answer those queries.

(Refer Slide Time: 15:35)



The next that we look at is the way the relationships are categorized as First Normal Form.

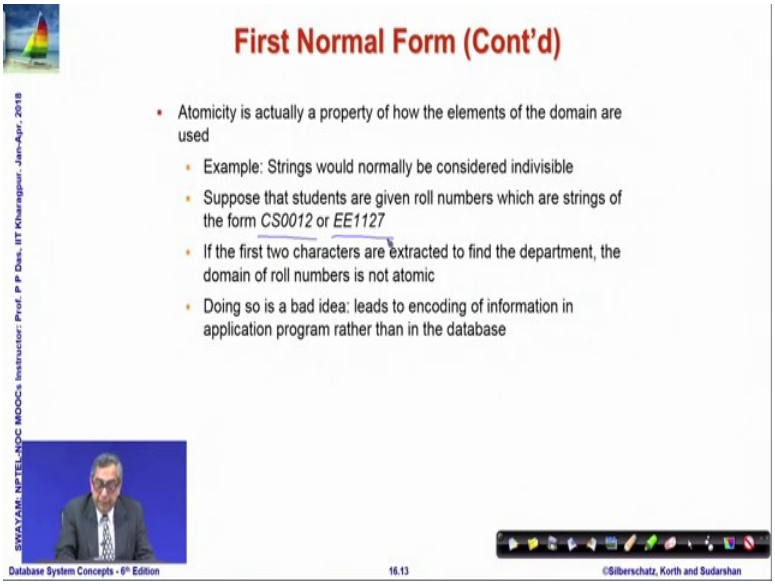
(Refer Slide Time: 15:45)



We consider that the domains of attributes are atomic if they are indivisible. So, anything that is a number, string and so on is considered to be atomic and we say a relational schema is in its First Normal Form if the domains of all attributes are atomic and all attributes single valued, there is no multi valued attribute. If these conditions are satisfied, then we will say that every relate that relational schema is in its First Normal Form. So, we will slowly understand the purpose of defining such normal forms, but let us initially understand the definition. So, if we have attributes which are composite in

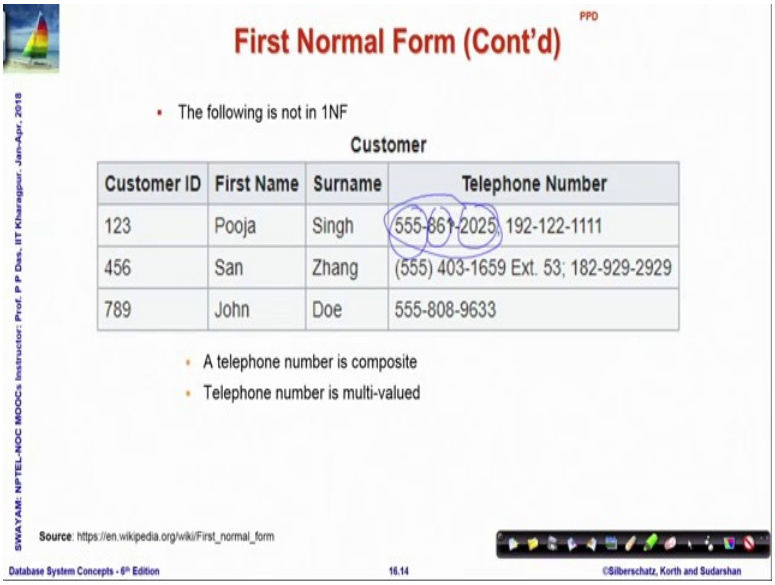
nature, naturally my relationship, my relational schema is not in First Normal Form if we have attributes which are multiple valued, it is not so.

(Refer Slide Time: 16:44)



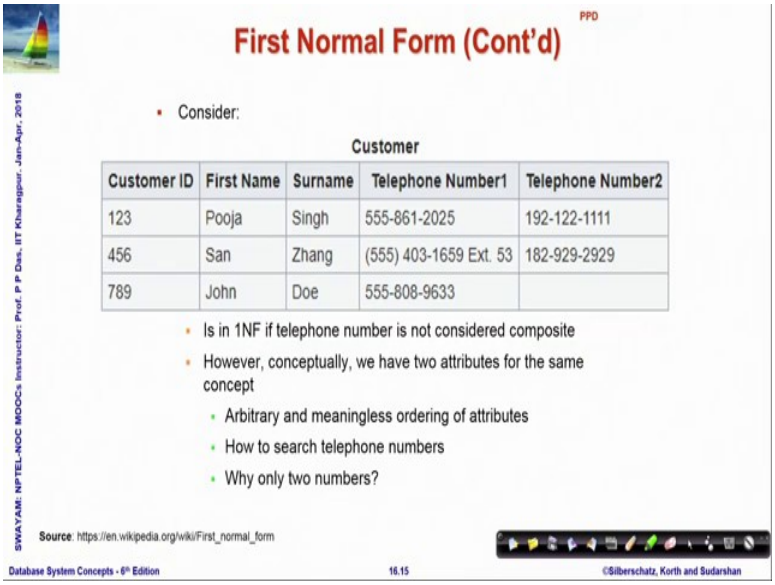
So, if we say that we have possible values are like this, then if we just treat them as strings, then the corresponding relational schema is in First Normal Form, but if we say that from this string we can extract the first two characters which is CS which tells me what is a department. The next four characters gives me a number, the serial number of the particular student in the role. Then I am not actually using an atomic domain because my domain needs to be interpreted separately than just being a value. So, these are not parts of what can be a First Normal Form.

(Refer Slide Time: 17:28)



So, I have given some examples of what is not and what is First Normal Form. So, this is an example where at the telephone number field exists and there can be multiple telephone numbers. So, this is not in First Normal Form because the telephone number itself is composite because it has different components and also, you can have multiple telephone number. So, this relation is not in the First Normal Form.

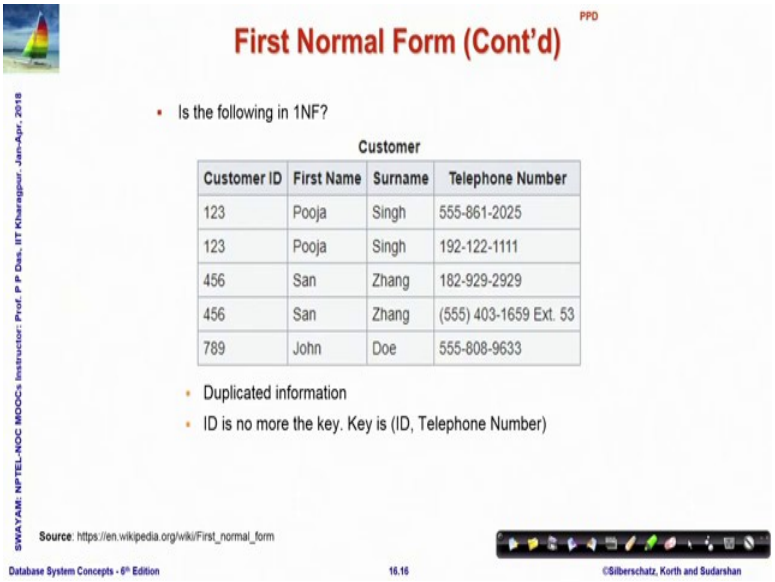
(Refer Slide Time: 17:54)



What you can do? You can separate out these phone numbers into two different attributes; Telephone number 1 and 2. Even then it is not exactly in First Normal Form because you do not know in which order they should be handled. If you have to search for a telephone number, then you will have to search multiple attributes which are

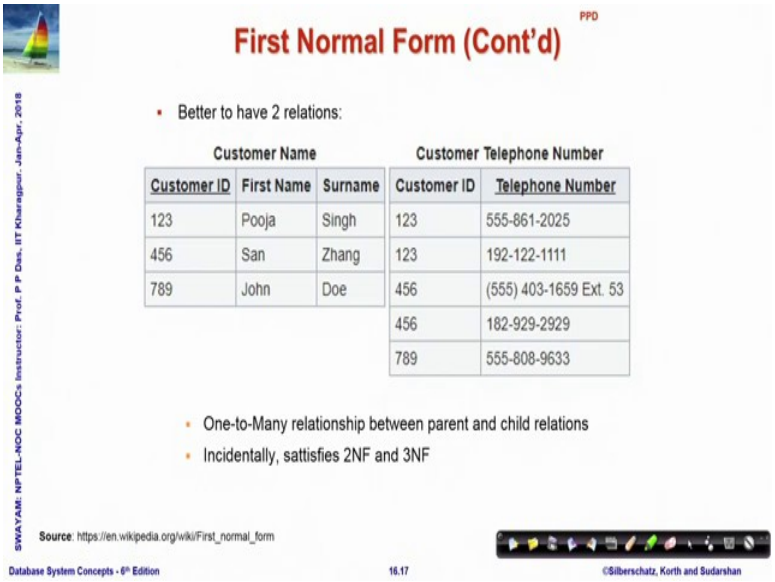
conceptually same and then, the question is why only two attributes. Cannot anybody have 3 phone numbers, 7 phone numbers and so on. So, this is really not a good option.

(Refer Slide Time: 18:26)



So, the other way could be that for every telephone number, you introduce a separate row. Once you do that you already know you have redundancy and you have possibilities of varied kinds of anomalies that could happen.

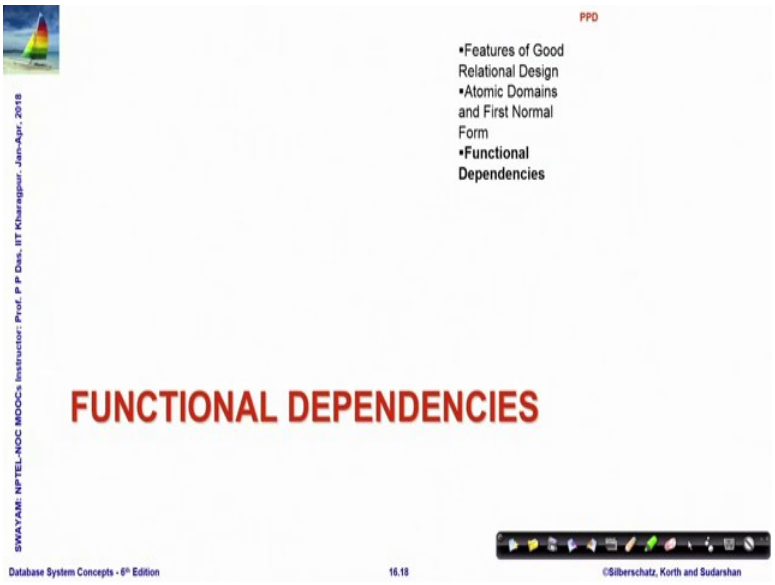
(Refer Slide Time: 18:40)



So, one way it could be achieved is we follow the principle that we had seen in ER modelling that this multivalued dependency can be represented in terms of a separate

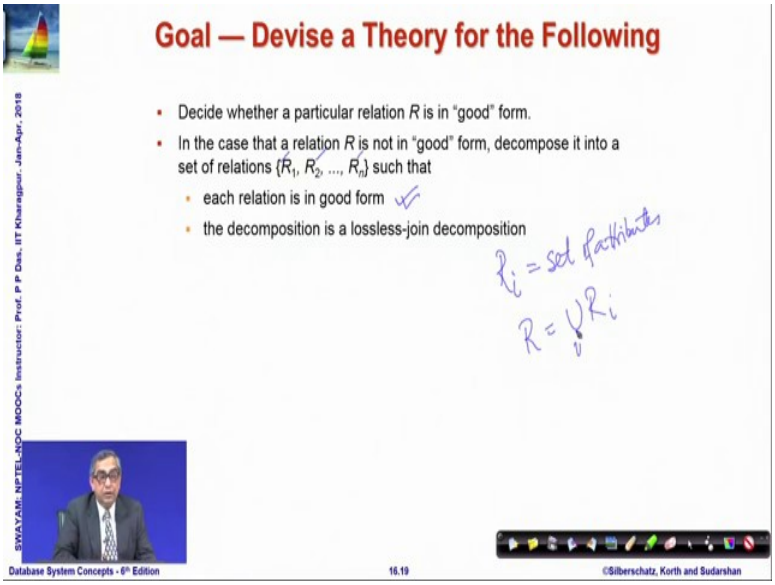
relation where against the customer id we just keep the telephone number. So, we can keep multiple of them and we take that out from the customer name. So, one to many relationship between the parent and the child, between the customer name and telephone number, every customer may have more than one telephone number is possible and that makes it 1 NF relation, First Normal Form relation and we will later on see that it also is 2 NF and 3 NF, but that is a future story.

(Refer Slide Time: 19:29)



Now, finally we come to the core of what the mathematical formulation which dictates much of the data base, relational database design is known as functional dependencies.

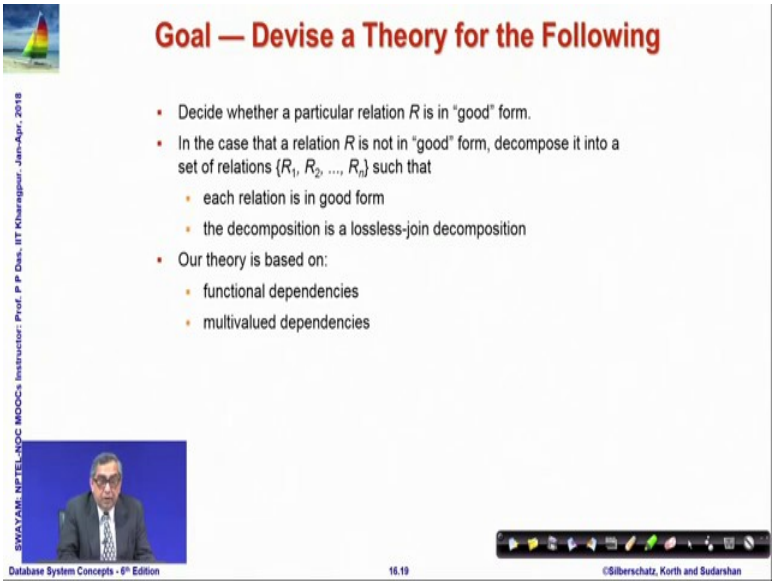
(Refer Slide Time: 19:47)



I just talked about little bit of that while talking about department name building and budget. Now, to decide whether a particular relation is good or rather a particular relational scheme is good, we need to check against certain measures and if it is not good, we need to decompose it into a set of relations such that these conditions satisfy that every, each one of these, {R1, R2, …., Rn}. So, I mean if you have you now got rusted, then it is basically Ri is a set of attributes because it is a relational schema. A relational schema is a set of attributes.

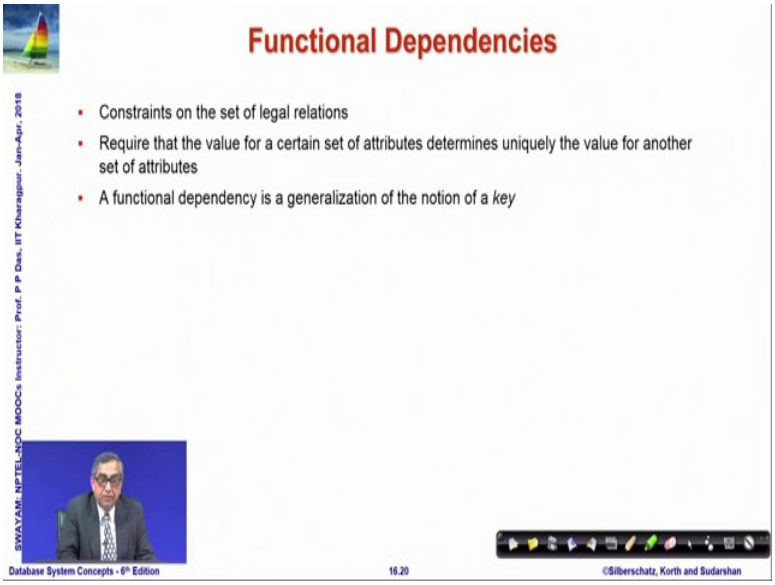
So, naturally R will be the union of all of these, Ri the total set of attributes. So, instead of keeping all the information into one relation in one table, we are basically decomposing it into n different schemas.

(Refer Slide Time: 20:42)



So, what we need to guarantee is each one of these relation R1, R2,….. Rn is in good form. How do I get back the original relation? Original relation that was represented by all that attributes enough is to take a lossless join. This would take a join and that this decomposition must give me a lossless join. So, to ensure that; we make use of two key ideas more foundationally; functional dependencies and then, multivalued dependencies.

(Refer Slide Time: 21:20)

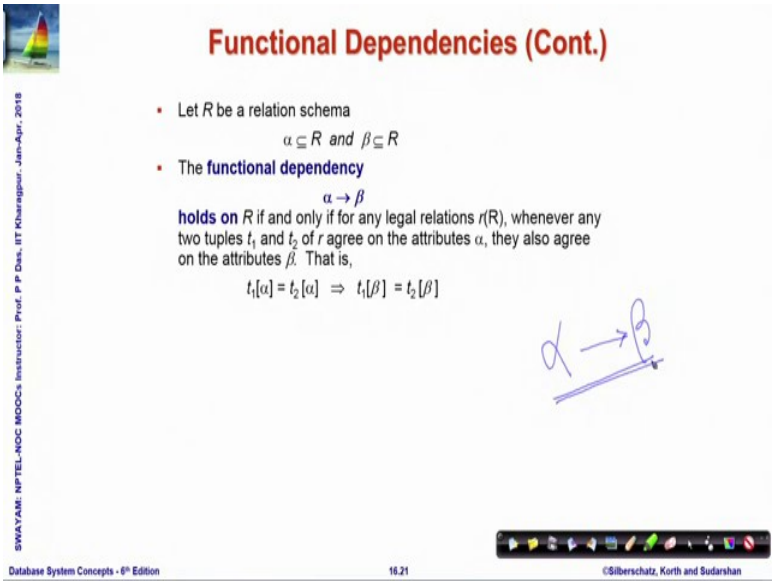


A functional dependency is a constraint on the set of legal relation. So, mind you it is a constraint on the schema and once that constraint is defined, it must hold for all relations that the schema satisfied. So, here we need that the value of certain set of attributes uniquely determine the value of another set of attributes. So, I know the value of three

attributes, I should be able to say that the values of the other four attributes would be fixed. So, you have already seen this notion in terms of key or super key. You have seen that similar type of concept exists where we said a key is a set of attributes, so that if the values of two rows are identical over these set of attributes, then the two peoples, the two rows must be totally identical.

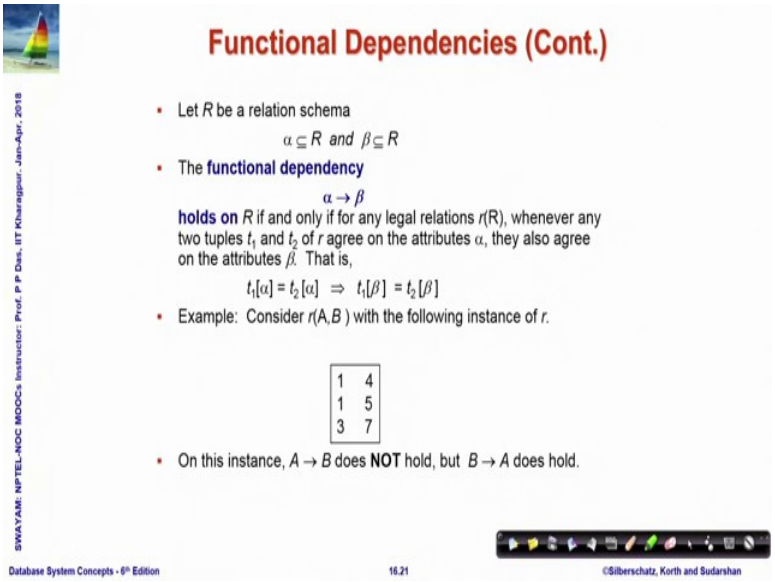
So, key is something which does a similar thing as a functional dependency, but is more specific. Functional dependencies are generalization.

(Refer Slide Time: 22:30)



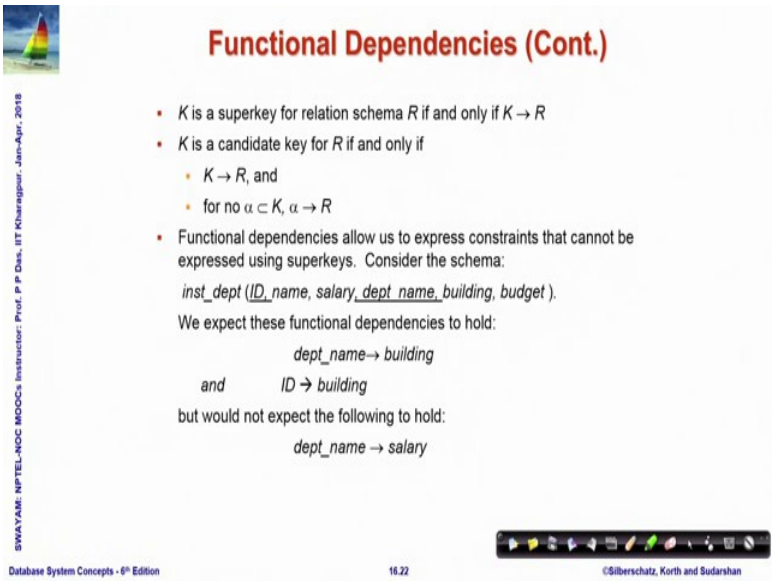
So, let us formally define that let R be a relational schema which means that it is a set of attributes and let us say **α, β** $\subseteq$ **R**, then we write this and note this notation. **α** is a set of attributes; **β** is another set of attributes. Both are subset of the same R and we say **α** $\rightarrow$**β** that is if I know the value of a tuple over the attributes of **α**, then the values of that tuple over the attributes of **β** would be fixed or in other words, they say that if I have two tuples t1 and t2 and their values over the set of α attributes are same, then necessarily their values over the set of **β** attributes must be same and mind you this is something which is a design constraint. It is not just an incidental property. It is not just the fact that a particular instance of a schema satisfies this, but when you say this is a functional dependency, we need all possible past, present and future instances of the schema must satisfy this.

(Refer Slide Time: 24:03)



So, consider this if you take a relation, a schema with an instance as given here between two attributes **A** and **B**, then we can say at least given this instance not we still do not know what happens in the whole schema for all instances, but on this instance we can say that **A** → **B** does not hold because between the first and the second record, the value of **A** is same one, but the value of we are different 4 and 5, but we can certainly say that on this instance at least **B** → **A** holds because whenever the value if we take any two tuples, their value over **B** does not at all match. If they does not match, then naturally there is no question of what happens to the value of the tuple over the set of attributes **A**. So, we will say that **B**→**A** holds in this instance.

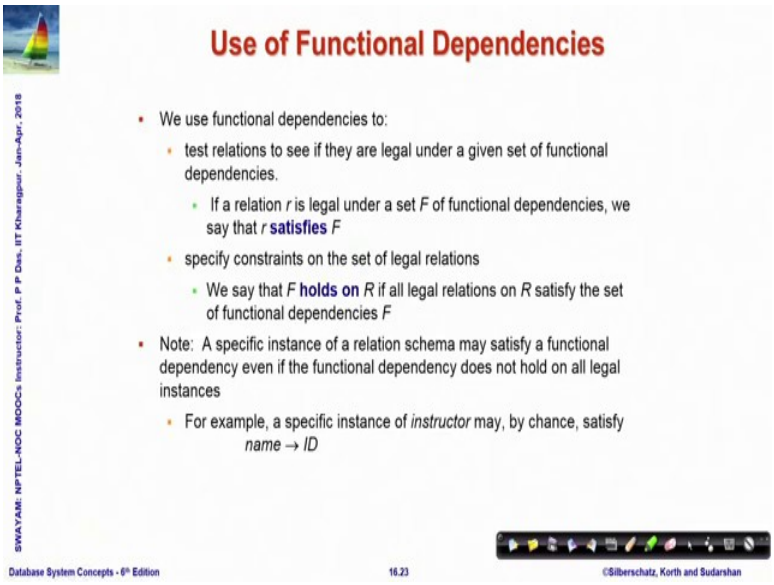
(Refer Slide Time: 25:01)



So, given this definition of functional dependency, now we can have a formal definition of what the super key is. Super key is naturally a subset of attributes which → the whole set and a candidate key is a super key which is minimal which means that k is a candidate key. If the two conditions have to satisfy this condition say there is a super key that it → all the attributes and the other condition says minimality that there is no subset **α** ⊂ **k**, such that **α** → **R** if there exists a subset **α** ⊂ **k**, the proper subset **α** ⊂ **k**. So, **that α** → **R**, then k would not be a candidate key. We will have to check for α. So, these two; what we had stated earlier in qualitative terms and now mathematically established. So, we can say that there are different functional dependencies. For example, inst_dept combined relation if we look at, then we know that **department name** → **building** functionally.

So, these are functional dependencies that must hold, but certainly we would not expect department name to functionally determine salary. That would be too much, right. So, functional dependencies are facts about the real world that we try to understand from the real world and then, represent in terms of the functional dependency formulation in the database.

(Refer Slide Time: 26:41)

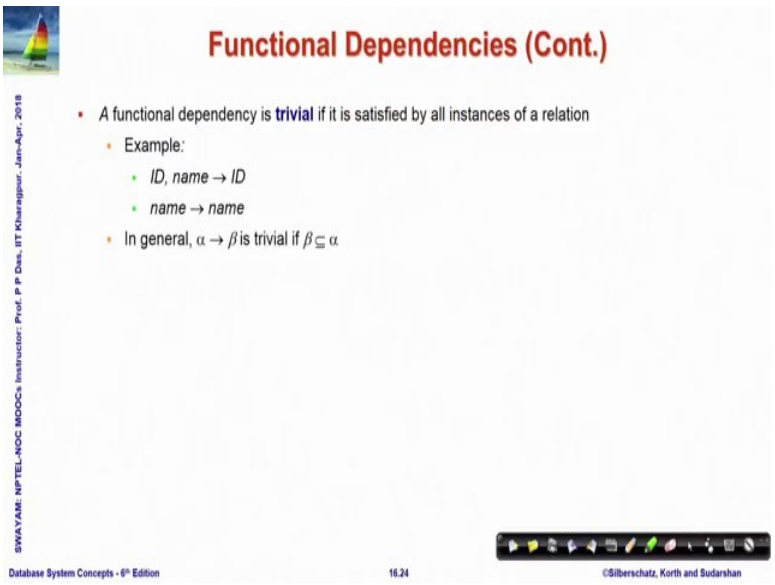


So, we can use functional dependencies to test relations if they are valid under the set of functional dependencies. So, there could be multiple functional dependencies in the set and if a relation we are using r here just to remind you that a relation means that a

particular instance is legal under a set of functional dependencies. We will say that r satisfies that and if we have that it holds F will be satisfied by all possible instances of a relational schema R, then we say F holds on R.

So, a relation satisfies a functional set of functional dependencies and a relational schema for a relational schema, the functional depend set of functional dependencies holds on that schema which means that for all possible past, present and future instances relations, the relations will satisfy the functional dependencies. So, we have for example id. We know **id → name** that if the id is distinct, then the name has to be distinct, but we may find that instance where **name → id**. So, we can say that **names → id** is satisfied by a particular instance where it so happens that there is no two rows where the name is identical, but we cannot, may not be able to infer that as this dependency holding on the relational scheme as a whole because tomorrow we can get another entry, so that two rows might match on the name, but could still be distinct entries not matching on id.

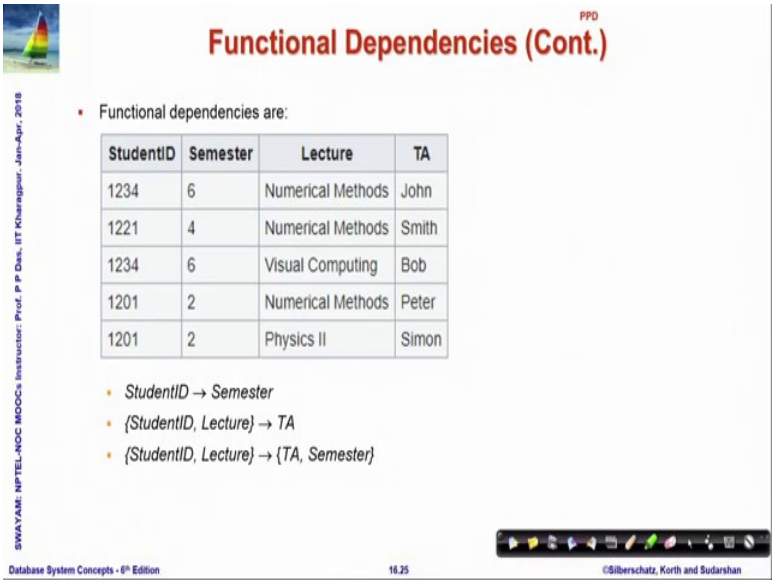
(Refer Slide Time: 28:46)



So, that is how this will have to be looked at in specificity. We say that a functional dependency is trivial if the left hand side is a superset of the right hand side. So, if I have a bigger set of attributes on the left hand side ID and name, then obviously **ID, name →ID**, **ID, name → name**, **name →name**. So, if you just think about because in a functional dependency the left hand side attributes that tuples have to match on the left hand side attribute and if they do, then they must match on the right hand side attribute.

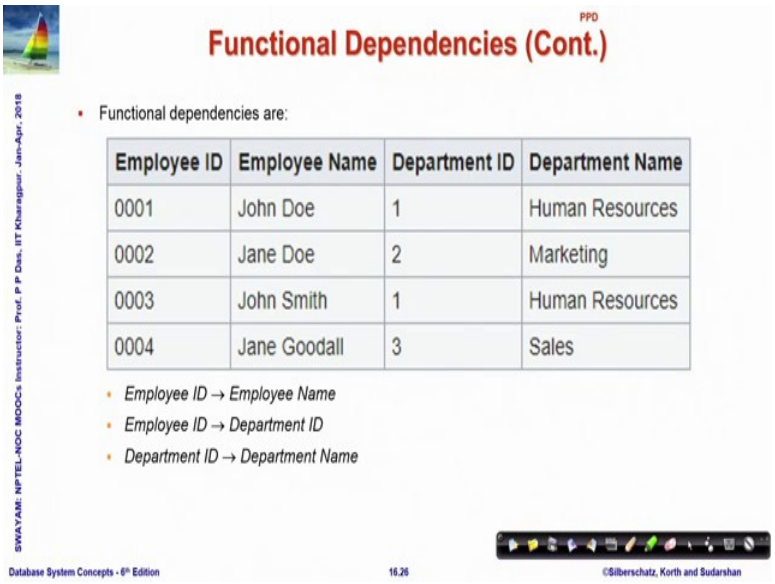
So, if the right hand side set of attributes is a subset of the left hand side, then obviously the functional dependency will be vacuously true and these are called trivial dependencies.

(Refer Slide Time: 29:33)



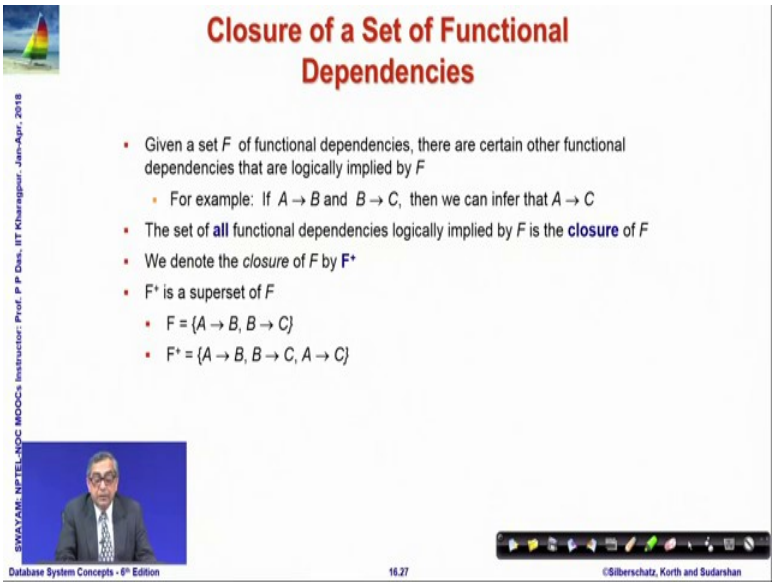
So, in the next couple of slides, I have shown few examples of functional dependencies of different tables. Here **StudentID → Semester** which mean that we are trying to model that a student cannot be at the same time in two semesters, then **(Student ID, lecture) → TA** and so on and you can see for this particular relation **(Student ID, lecture)** also happens to be the candidate key.

(Refer Slide Time: 30:05)



These are another example. So, these are just go through them, try to convince yourself that these functional dependencies are very genuinely real world situations that can be modeled in this way.

(Refer Slide Time: 30:19)

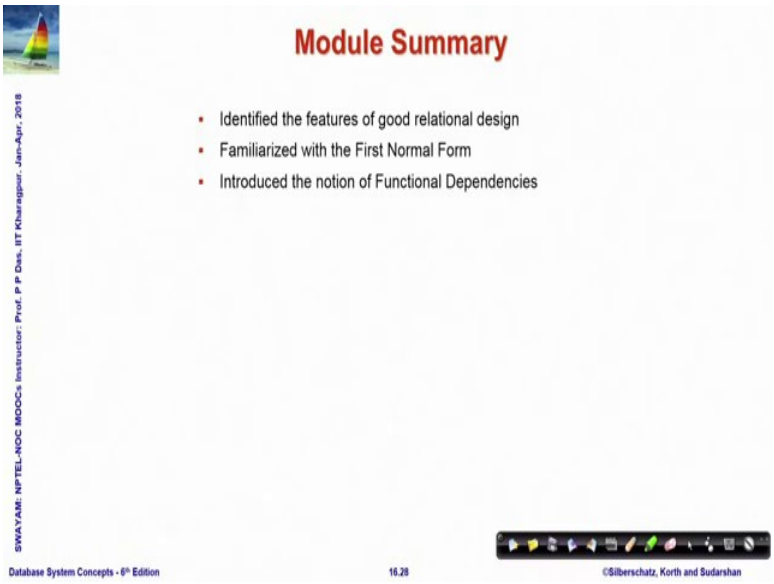


Given a set of functional dependencies, we can actually compute a closure. For example, if **A** → **B** and **B** → **C**, then we can infer that A functionally determined because if two peoples match on, **A** → **B** says that they match on B. Now, if **B** → **C** also holds, then if they match on B, they match on C. So, if the match on A, then necessarily they may have

to match on C. So, this is called the logical implication of a set of functional dependencies and we will see more of this later, but if we take all functional dependencies of a given set F, that are logically implied from this set F. We said that is a closure set and we represent that by **F** $^{+}$**.**

So, **F** $^{+}$ necessarily is a superset of F. So, here in that above example, this is F and this is **F** $^{+}$.

(Refer Slide Time: 31:00)



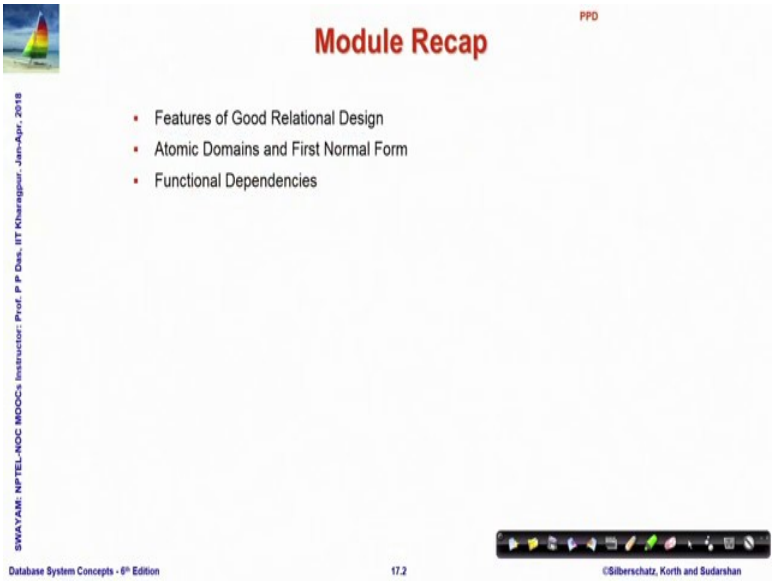
So, we will continue more on the theory of functional dependencies, but let us conclude this module by summarizing that we have identified the features of good relational designs tradeoff between decomposition and lossless join properties that we need. We are familiarized with the First Normal Form and atomic domains and we have introduced the notion of functional dependencies on which we will build up more and try to get zeroing very concrete strategies for good results.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture - 17
Relational Database Design (Contd.)

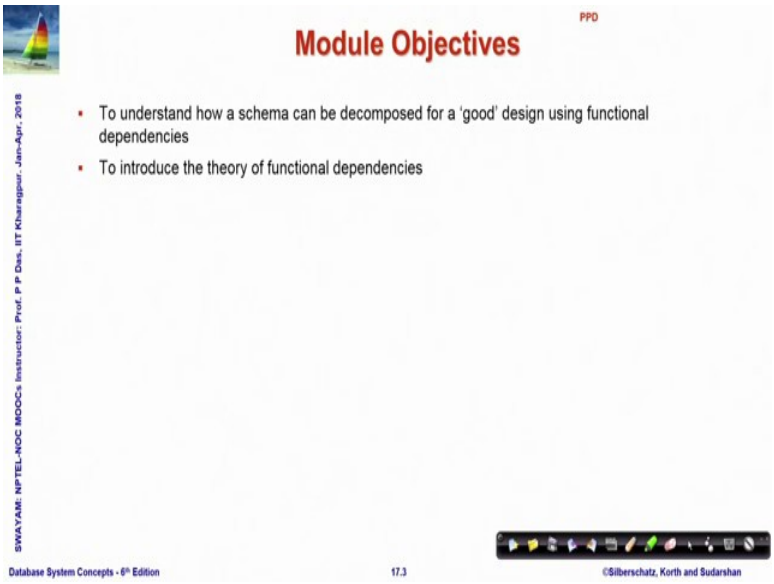
Welcome to the Module 17 of Database Management Systems. From the last module, we are discussing Relational Database Design. So, this is second in the series of five modules which we will discuss this.

(Refer Slide Time: 00:31)



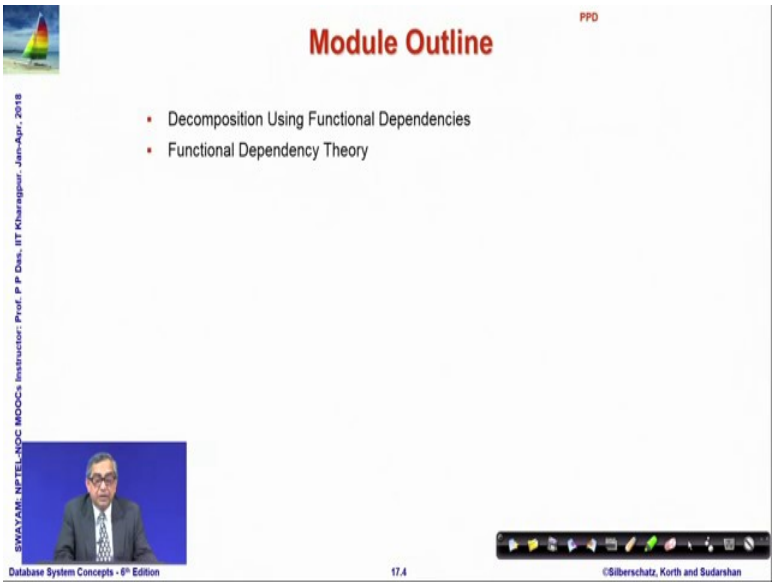
We have already seen basic features of good relational design. We have studied about first normal form, atomic domains and got introduced to functional dependencies.

(Refer Slide Time: 00:43)



So, we will develop further on that to see how decompositions into good design can be done by making use of the notion of functional dependencies and we will more formally introduce the theory of functional dependencies.

(Refer Slide Time: 00:57)

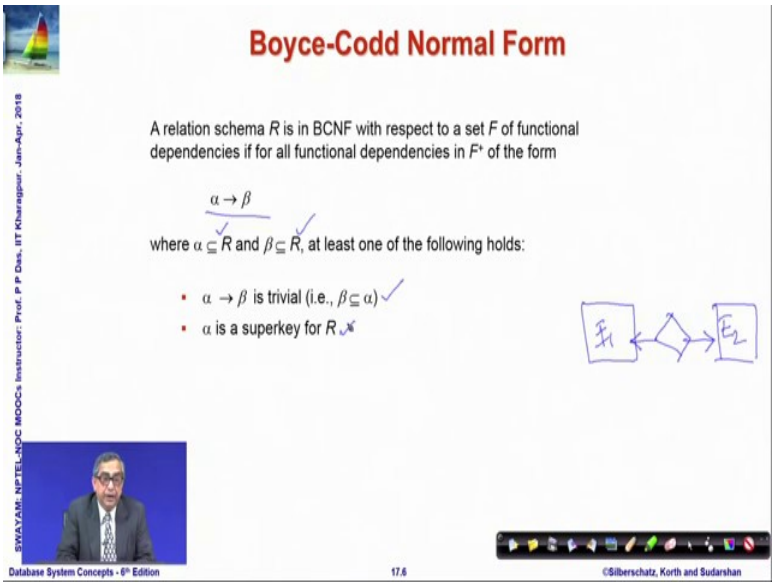


So, that is all that we discuss in this.

(Refer Slide Time: 01:00)

So, decomposition using functional dependencies is the first thing that we look at.

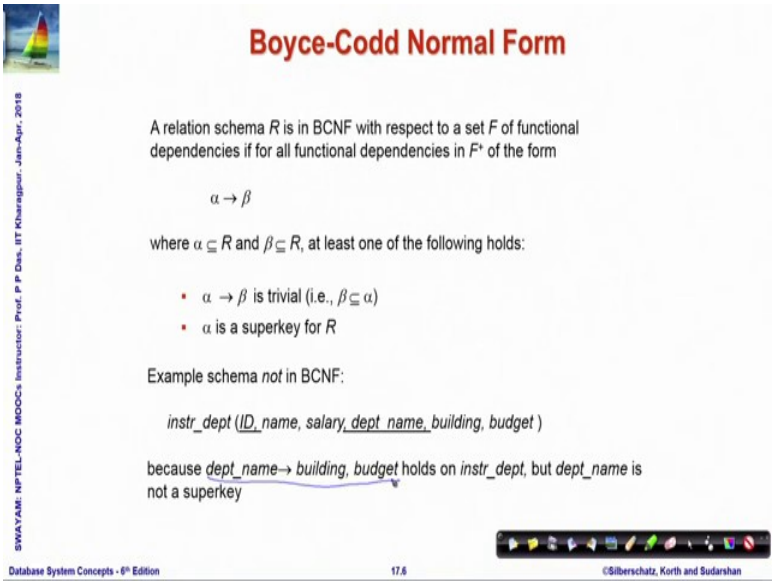
(Refer Slide Time: 01:04)



The first normal form of relations which were studied, we look at its Boyce-Codd Normal Form. So, normal forms are kind of set of properties which is satisfied by a relational schema and if they are satisfied, then we have certain guarantees in terms of what can or cannot happen in that relational schema design. So, Boyce-Codd is a simplest kind of beyond 1NF is a simplest kind of normal form and a relational schema is said to be in Boyce-Codd normal form if with respect to a set of functional

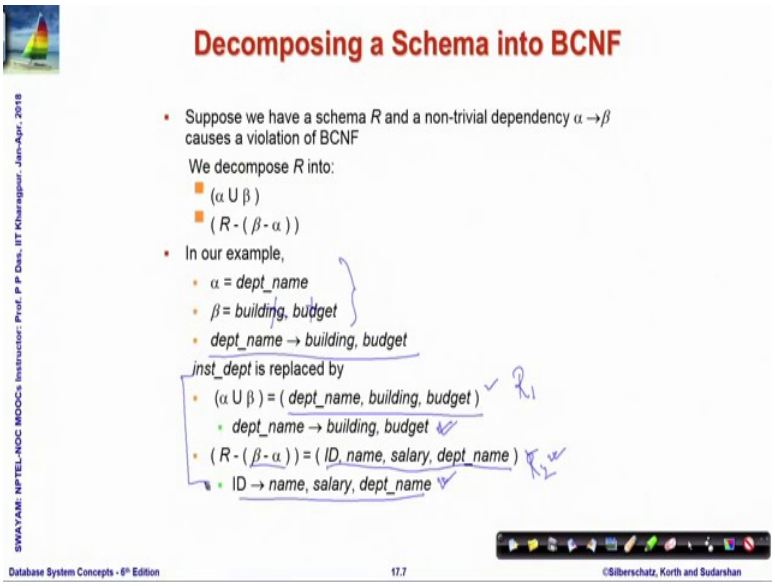
dependencies, all functional dependencies in the closure. So, in respect of F, we compute $\mathbf{F}^+$ which is a closure and if I have a dependency $\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta}$, then naturally $\boldsymbol{\alpha} \subseteq \mathbf{R}$, $\boldsymbol{\beta} \subseteq \mathbf{R}$, but what is important is every functional dependency in the closure set must either be trivial that is right hand side $\subseteq$ the left hand side or the left hand side set $\boldsymbol{\alpha}$ must be super key. So, only those kind of functional dependencies are possible. No other functional dependencies are possible. If that is satisfied by the relational schema R, then it is said to be in the Boyce-Codd normal form.

(Refer Slide Time: 02:39)



So, if we look at inst_dept schema of the combined relations we saw last time, then we will know that certainly this is not in Boyce-Codd Normal Form because this functional dependency holds in this schema where it is neither a trivial dependency and nor department name is a super key. So, this is not in BCNF.

(Refer Slide Time: 03:10)



So, if a relational scheme is not in BCNF, then the question naturally is can I make it into BCNF so then that process is the process of decomposition. So, what you do? You divide the set of attributes into two or more at sets of attributes. So, here let us say that we have a relational schema which has a non-trivial dependency, **α → β**, where **α** is not a super key. So, with respect to this functional dependency, the relational schema is not in BCNF, then we can decompose R by two sets. One is **α U β** take the **U** of these two attribute sets and remove **β - α** from R, take the difference of **β - α** and remove that from R. The resulting pair of relations, relational schemas will be in Boyce-Codd Normal Form with respect to this particular functional dependency.

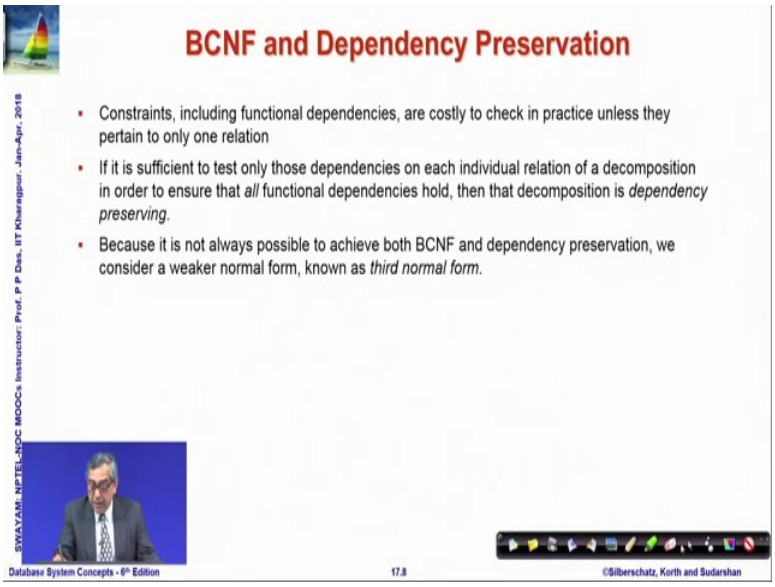
So, let us see an example. So, if **α** is department name **β** is **(building, budget)**, we have department name→s building budget. So, **α → β** and we have already seen that it does not hold. It is not satisfied by the **inst_dept**. So, you replace it by taking **α U β**. So, **α U β** is this set of relational, this relational schema and you do **R-(β -α)**. Naturally if this is **β** and **α**, then **β -α** is necessary building budget because if the department does not occur in **β**.

So, this set is building budget and if I remove it from R which means that id, name, salary and department name are retained, but building and budget gets removed. So, I get another relational schema which has these four names and it holds the functional dependency id determinant. So, even now if I look into this schema R1 and this schema

R2, there are different dependencies that hold on R1 and with respect to that dependency R1 is in BCNF because department name is the super key, is the primary key and with respect to this dependency, R2 is in BCNF because id is a key.

So, I can see that the original combined relational schema was not in BCNF with respect to this functional dependency, but when I do this decomposition, I get two schemas which are each in BCNF normal form. So, this is the basic process and we will see depending on the normal form and different notions of functional dependencies, we will see how these conversions can be done, but this is a basic approach of converting a schema into a normal form.

(Refer Slide Time: 06:24)

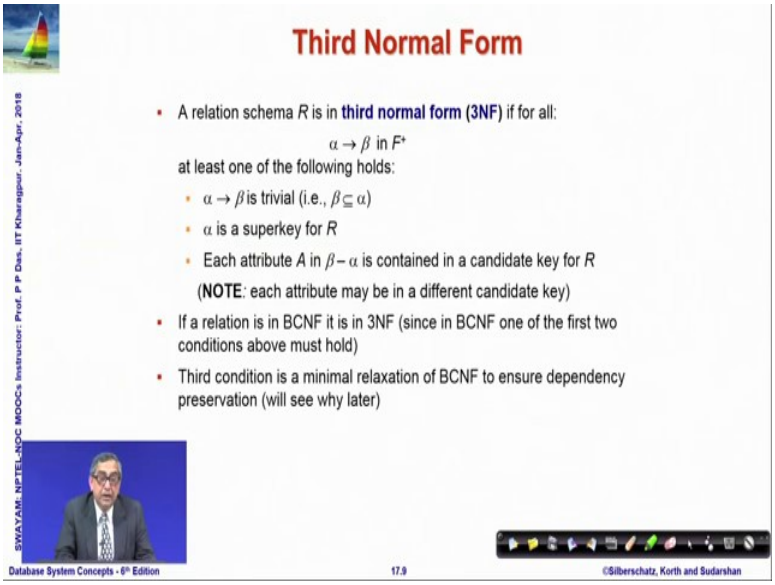


Now, the question is if the constraints including the functional dependencies if we look at, then functional dependencies will have to be checked on different instance. Now, in general it is difficult to check a functional dependency **α** determining **β** if the attributes **α** and the attributes of **β** or the attributes of **β** are distributed between multiple relations because naturally how do I check if they are true, how do I check that two tuples which match on **α** is indeed matching on **β** unless I perform a costly join operation. So, there objective is to be able to come to designs where it is sufficient to test only those dependencies on individual relations of the decomposition and with that I must be able to ensure that all functional dependencies hold. So, it is a very interesting situation.

So, we are saying that we will decompose, get into a number of relational schema. Every schema will have a number of dependencies, functional dependencies and those functional dependencies if they involve only the attributes of that relational schema, they can be tested very easily and if these functional dependencies together mean ensure that all functional dependencies hold that is if the closure of this set of functional dependencies is same as the closure of the earlier set, the original set, then we say that the decomposition that we have achieved is dependency preserving because I can actually effectively compute.

This is dependency preserving because I can effectively compute whether every dependency is satisfied by checking on every individual relation, but the unfortunate part of the reality is that it is not always possible to achieve a Boyce-Codd Normal Form Decomposition which also preserves the dependencies. See if there are in some cases will be able to do like the example we saw just now the instructor and department, but it is not always possible. So, we usually need another weaker form, normal form which is known as a third normal form and we will subsequently look into those.

(Refer Slide Time: 09:09)

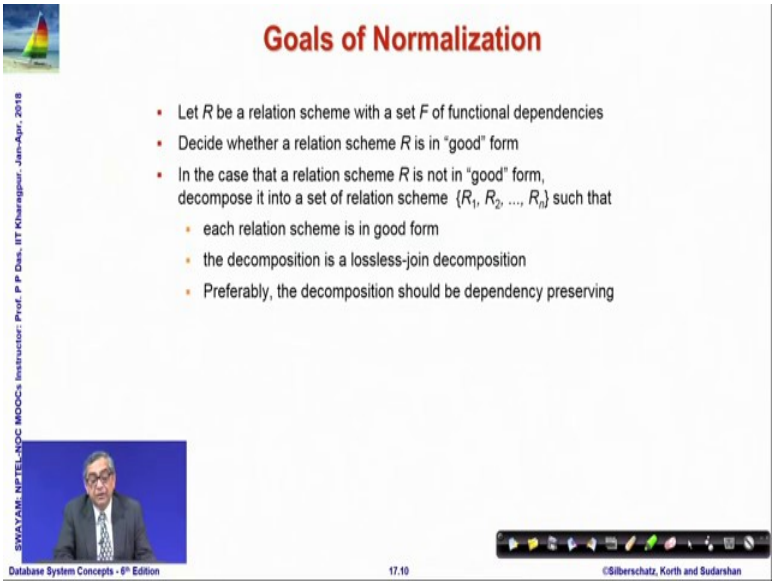


A third normal form is again a relational schema is there and for all attribute, for all dependencies that belong to the closure of the functional dependencies, this following conditions must hold either **α** → **β** is trivial which is a condition which BCNF or **α** is a super key of R which is also a condition that we say in BCNF or each attribute in **β - α**

that is right hand side difference the left hand side is contained in a candidate key for R. It is not very obvious as to why we need that. That will unfold slowly. This is the condition we did not have in BCNF. So, naturally you can see that based on the first two conditions, you can always say that if a relational schema is in BCNF, it necessarily is in 3NF, but not the reverse.

There could be some schema which is in 3 NF because of the third condition where there exists a functional dependency. So, that **β - α** is contained in a candidate key for R, but it is not in the BCNF form and also you can note that the attributes of that are contained in **β - α** must be in some candidate key, not necessarily in the same candidate key. If they exist in some candidate key, then itself 3NF condition will get satisfied. So, if a relation is in BCNF, it is in 3NF. We have already seen that. So, third condition minimally relaxes BCNF to ensure that we have a dependency preservation. We will see this more later. So, I am just introducing the concept of a relaxed normal form here.

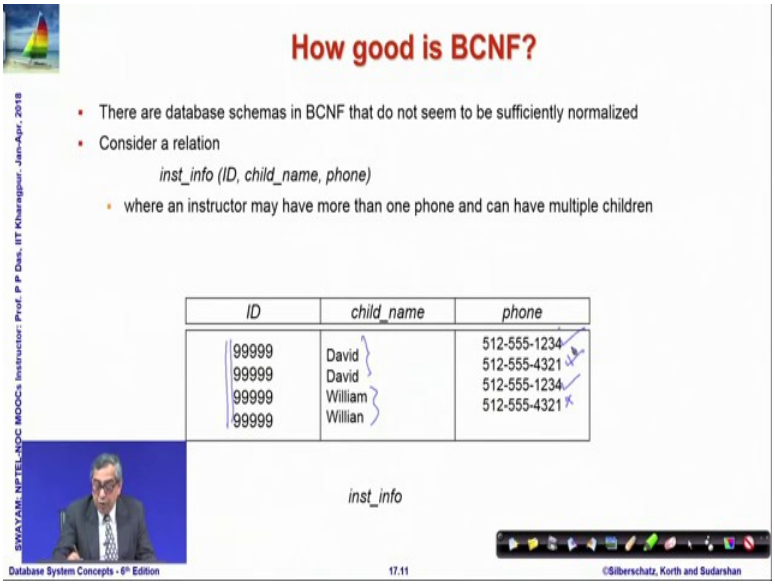
(Refer Slide Time: 11:11)



So, what is a goal of this normalization is if to summarize let R be a relational scheme, F is a set of functional dependencies, we need to decide whether the relational scheme R is in a good form which means that it should not have unnecessary redundancy. It should be impossible to acquire information by doing lossless join. So, in case it is not in good form. We can convert it by decomposition into N relational schema, such that each schema is in good form. The decomposition has a lossless join, so that I can get back the

original relation from this and preferably the decomposition should preserve the dependencies. So, that is what we will target henceforth.

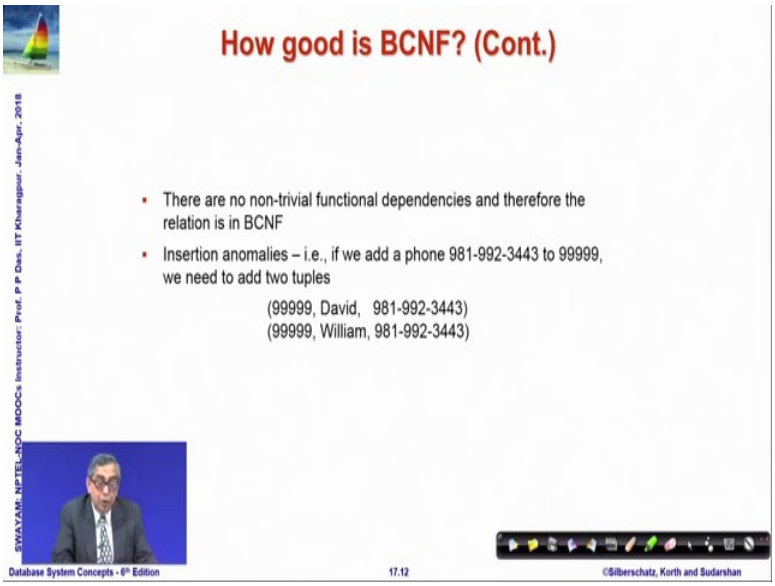
(Refer Slide Time: 12:08)



So, when we do that let us quickly evaluate as to we have seen BCNF. So, how good really BCNF is. So, if I have something in BCNF should I really be very happy always. So, let us look at a relational schema. This is an information relating the idea of a person, the name of the child and the phone number and naturally the person, the instructor may have more than one phone and may have multiple children. So, this is a possible instance that you can see though all of these belong to the same instructor. He has naturally you can see that two children and there are this is here, this is here.

So, this is here and this is here. So, there are two different phone numbers. So, naturally you have four possible combinations that you need to look at.

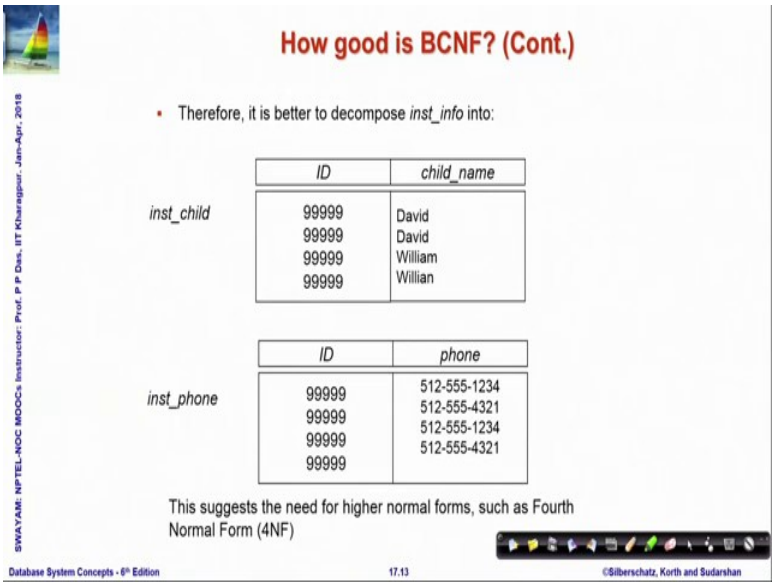
(Refer Slide Time: 13:08)



So, now there is no non-trivial functional dependency in this relation. So, since there is no non-trivial functional dependency, this relation naturally is in BCNF form because that is the existence of non trivial dependency is what makes a schema not conform to the BCNF form. So, there is no such. So, this is in BCNF form and now, if you look at, but what did we see the key thing that we saw if we just go back, the key thing that we saw that there is ample redundancy of data, the same data is entered multiple times.

So, the consequence of that could be insertion anomaly. If we want to add a phone number to the same instructor, then we need to add two tuple because the instructor also has two children. If the instructor and three children will need to add three and unless this is maintained always, then we will have difficulty.

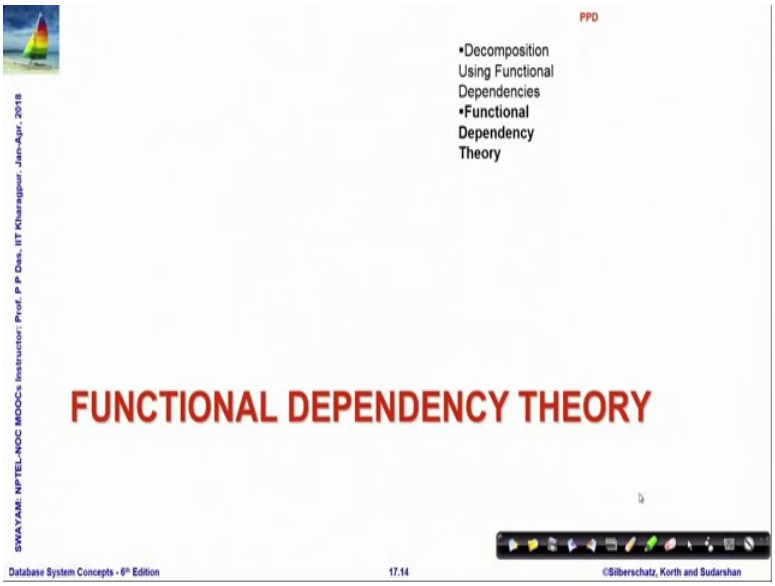
(Refer Slide Time: 14:15)



So, the redundancy consequences anomaly that we are getting into, so it could be better to decompose this to say that I make this orthogonal; I keep the child information with id and I keep the phone number information in the id separately. So, if I do that, then I can decompose it in this manner and if I decompose that, this have just shown that if you are dividing that table in two parts. So, naturally these are not required, neither are these required. So, these are the entries that I get and you can convince yourself that you can actually do a lossless join to get back the information.

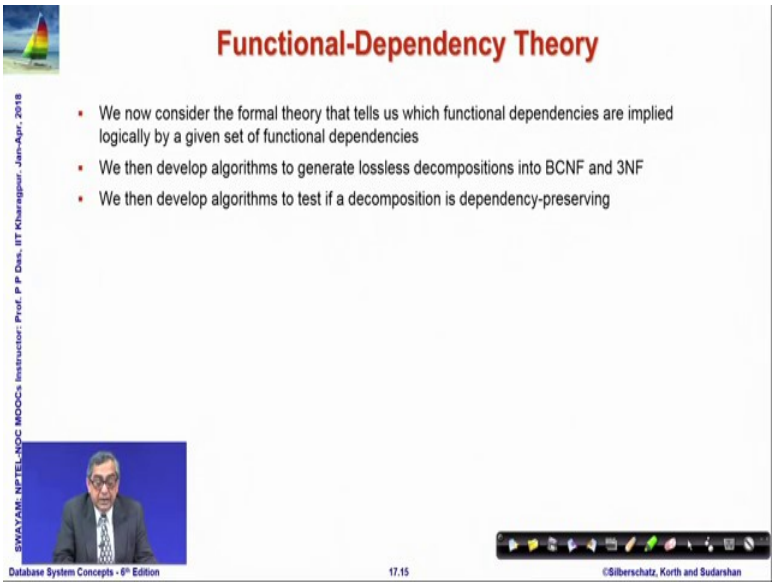
So, BCNF not necessarily give you good designs and we will see later on that there are other normal forms which can be used to improve on BCNF.

(Refer Slide Time: 15:05)



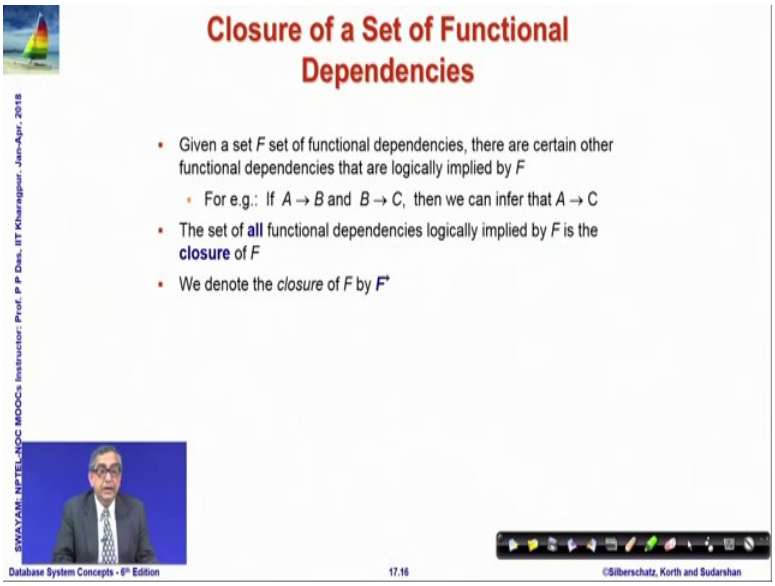
Now, let us for formally getting into how do we convert decomposed relation into a third normal form and how we assess that we need to understand more of the functional dependencies.

(Refer Slide Time: 15:20)



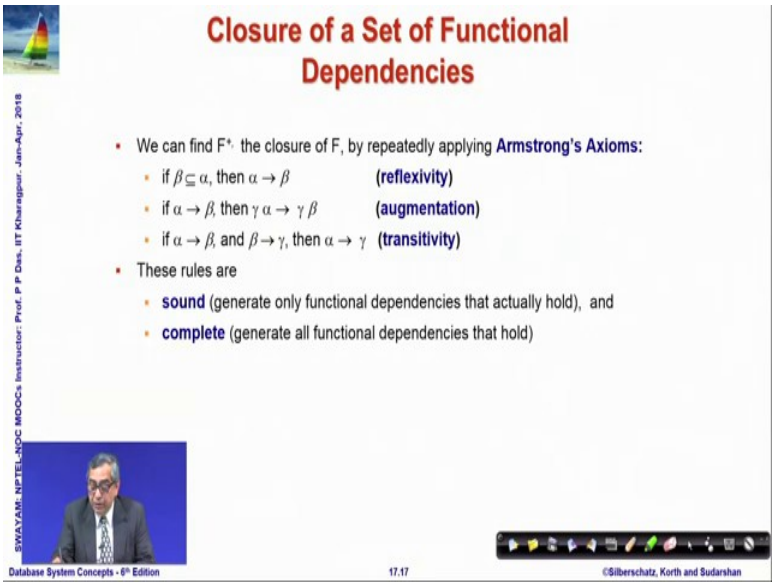
So, we will consider now a little bit of formal theory on them and then, develop algorithms that can generate lossless join decomposition into BCNF and 3 NF and we will also create algorithm to test if decomposition preserves the dependency.

(Refer Slide Time: 15:39)



So, just quickly to recap we have already introduced the closure set of a functional dependencies. It is all dependencies that are logically implied by it. Now, the question certainly is given a set how do I compute this closure of a set?

(Refer Slide Time: 15:57)



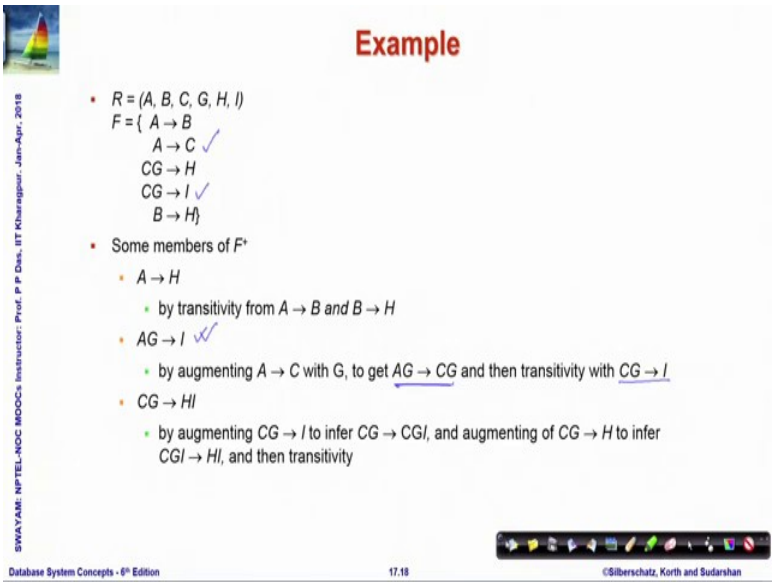
So, to do this we make use of three rules known by Armstrong's Axiom named after the person who first observed them. So, the first rule is reflexivity which says that if **β** $\subseteq$ **α**, then **α** → **β**. Always **α**→s **β**. So, this is basically reflexivity you can see is a different

way of saying specifying about trivial dependencies. Next comes important thing augmentation which says that if $\alpha \rightarrow \beta$, then $\gamma\ \alpha$ where γ is some set of attributes in R.

Then, $\gamma\ \alpha \rightarrow \gamma\ \beta$ which is very easy to see because $\alpha \rightarrow \beta$ means two tuples who match on α will necessarily match on β. Now, if that happens and whatever is γ if two tuples match on γ and α, then certainly they will match and γ and β because $\alpha \rightarrow \beta$ tells me that they will match on β and γ is the same set of attributes. So, augmentation also is easy. Then, we have transitivity which we earlier saw also if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then obviously $\alpha \rightarrow \gamma$.

So, these are the foundational rules observed which can be made used to compute the closure of the set of functional dependencies. Now, these rules as they say this is more for you know understanding the theory better. These rules are sound as well as complete. Soundness mean that if I use these rules repeatedly in a set of dependencies, then it generates functional dependencies all of which actually hold. So, it will never generate a functional dependency which is not correct, which will not hold and the second it is complete which means that if I keep on using these rules, then all functional dependencies that can at all hold will eventually get generated.

(Refer Slide Time: 18:06)

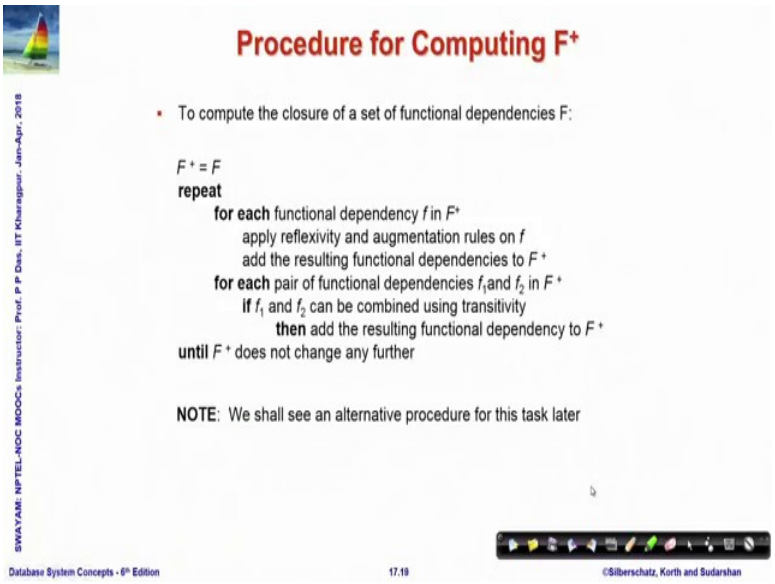


So, that is a very strong result and that is what leads to say the following example. So, when we are trying to compute the functional, the closure of the function set of functional dependencies here. So, there are six attributes in the set. There are six

different functional dependencies and we identify some members of the closure. For example, we can see that **A→ B** and **B→ H**. So, transitivity clearly shows that **A → H**, very clear. So, in the closure that must be there.

Similarly, we can see that **A→C**. Now, if we augment it with G, that is put G on both sides, then **AG→CG** and we know that **CG→I**. So, if we combine these two by transitivity, then we can get a new functional dependency which has **AG→I**. So, in this manner you can do the next one also and you can try to infer several other functional dependencies that can be inferred by different applications of the Armstrong's axiom, the three rules in any multiple different ways.

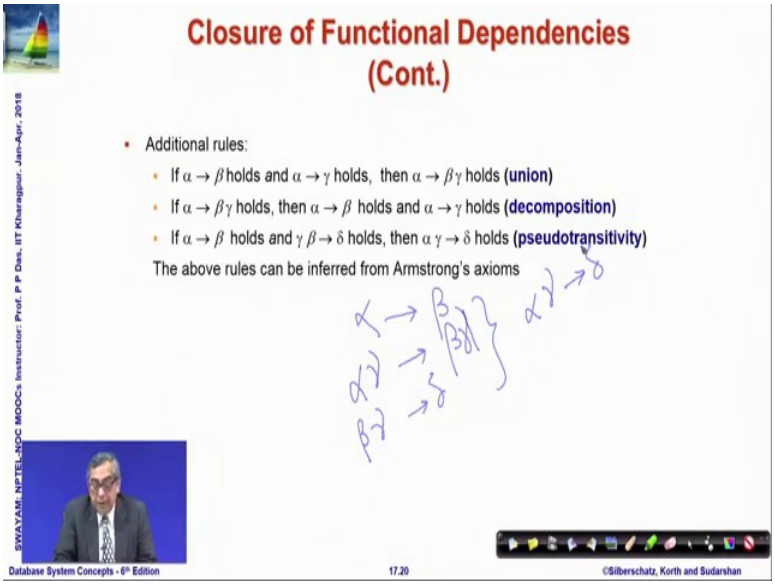
(Refer Slide Time: 19:36)



So, to get the closure what we need to do is, now very simple is certainly we will have a repetitive algorithm to get the closure. The first algorithm will start with the set of functional dependencies that we have. So, the closure must include the given set of functional dependencies. So, F^+ must have F. So, let us start with initial value of F^+ as F, then for every functional dependency is F^+. This is what we keep on repeating. Look at the outer loop, every functional dependency that we have F^+ now will apply reflexivity and augmentation and add the resulting functional dependency in F^+. It is possible that the same functional dependency gets generated and added multiple times does not matter. F^+ is a set. It will naturally eliminate duplicates.

Then, for each pair of functional dependencies because reflexivity and augmentation applies to one functional dependency only, but transitivity applies to two functional dependencies. So, for every pair of functional dependencies, we check whether they can be combined by transitivity. If they do, then the transitive closure of the transitive functional dependency that arise out of that is also added to F^+ and mind you more and more functional dependencies you add, there are more and more opportunities to apply the Armstrong's Axiom rules and newer functional dependencies will continue to get added, but eventually you reach a point where F^+ does not change any further and when that is achieved, we know that the functional, the closure of the functional dependencies have been obtained and that is our final set.

(Refer Slide Time: 21:32)



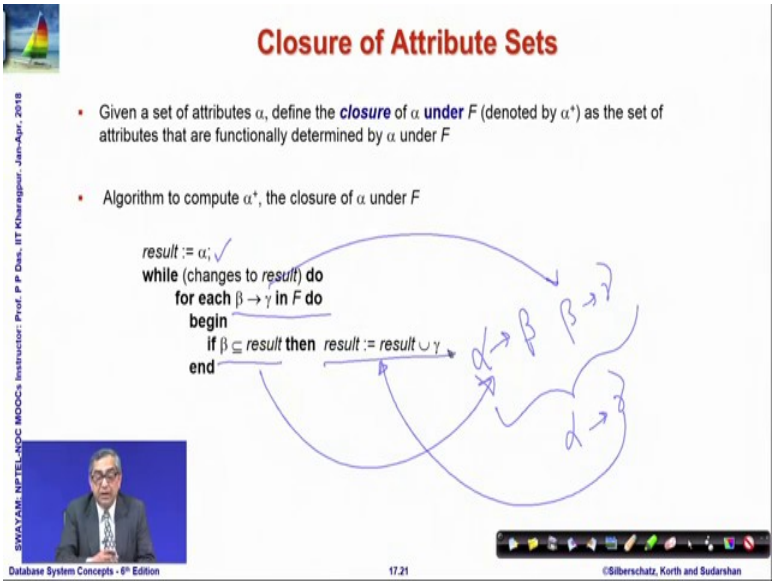
We can also observe that based on the rules of Armstrong, the Armstrong's Axioms we can also generate lot of derived rules. Some of those are shown here. For example, if A →, if **α** → **β** holds and if **α** → **γ**, that also holds, then **α** → **β** and **γ** together. This is called the **union** set. So, if there are two functional dependencies which are the same left hand side set of attribute, then we can take the **union** of their right hand side attributes and that functional dependency will hold obviously, it is trivial to prove this.

If **α** → **β γ**, then **α** → **β** holds and **α** → **γ** holds. This is called decomposition. So, kind of the other side of the **U** which also is trivial because **α** → **β γ** says if two tuples match on **α**, they match on **β** as well as **γ** attributes. So, obviously you take the first part, you get **α**

→ **β**. You take the second part of the observation, you get **α** → **γ**. So, that is a composition rule. The third is interesting. It is called the pseudo transitivity which says that **α** → **β** if that holds and **γ β** → **δ** if that holds, then **α γ** → **δ** which is not difficult to get because if this holds, then I can augment **γ** on both sides. I get **β γ** and then, I have given **β γ** → **δ**. So, if I combine these two in terms of transitivity, I get **α γ** → **δ**.

So, this is called pseudo transitivity because here you are adding another attribute in the transitivity. So, often times it becomes easier to make use of these additional rules to quickly get to the closer set.

(Refer Slide Time: 23:45)



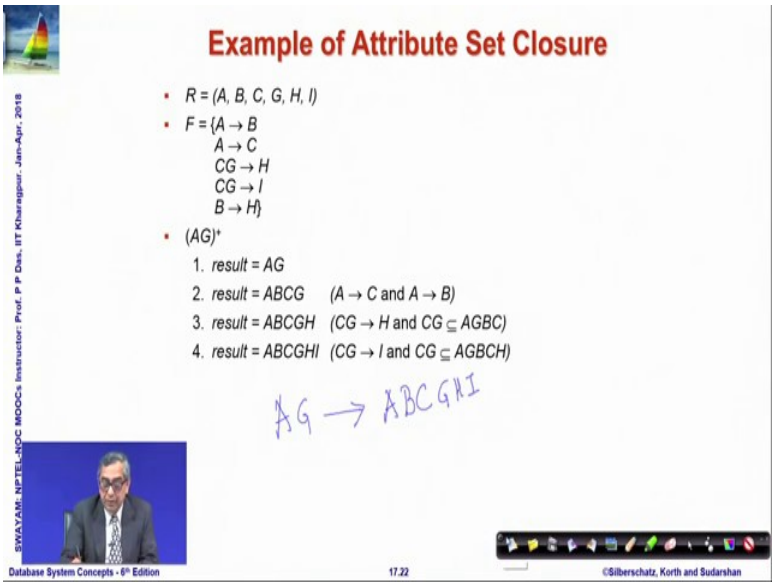
So, given a set of attributes we also compute the closure of a set of attributes. This is a second concept we have seen how to give the set of functional dependencies, how to compute the closure of the functional dependencies. Now, we are given a set of attributes and we want to define the closure of this set of functional, this set of attributes under the set of functional dependencies and as the closure of functional dependencies F is denoted by $\mathbf{F}^+$, the closure of a set of attributes **α** under F is denoted by $\boldsymbol{\alpha}^+$. So, this set of closure attributes of **α** is a set of attributes that are functionally determined by **α** under F. So, all set of attributes that functionally determined by **α** under the set of functional dependencies is member of **α**+.

So, the following simple algorithm can compute the closure naturally. Initially let us say the result is the final closure set. So, initially we can say that result can be initialized with

α because certainly the whole of **α** would necessarily belong to α^+ by the reflexivity condition, then for each functional dependency **β** → **γ**, we check if **β** is a subset the result. If **β** is a subset the current set of attributes that form result which mean that **α→β**, it will have to because result is the set of all attributes that **α** functionally determines. So, if **β** is a subset the result, then necessarily **α→β** is a consequence of this and we know that this is their **β→γ** combined by transitivity.

So, I know **α→γ**. If function **α→γ**, then it must get into the result and this is exactly what the statement is saying that take result and add **α**, add **γ**, the set of attributes **γ** to the result. How long should you do that? Naturally you will do that as long as over a full iteration of functional dependencies in F, if there is no change to the result, then you know that all future iterations will have no change. So, you reach a fixed point and you declare that the closure of the set of attributes have been obtained.

(Refer Slide Time: 26:44)



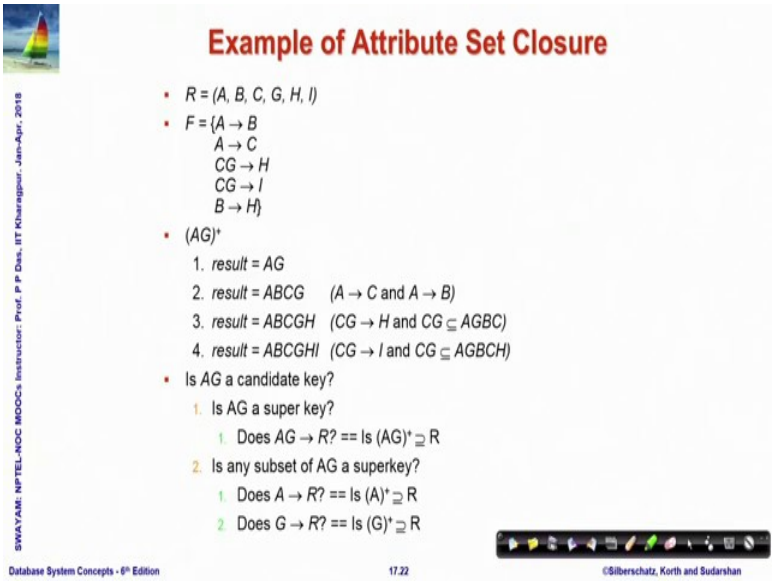
Now, this closure information is very interesting and we just show an example here based on the same set of attributes and same set of functional dependencies.

So, we are trying to find the closure of the set of attributes AG. So, AG^+ initially it will be AG. Now, since **A→C**, so given that I can say that C will get included in this set in the same iteration. If I look at **A→B**, so B will get included in this set. So, after this first iterative loop I will have the result as ABCG. If ABCG is there and I am looking at the next iteration, then **CG→H**. So, H comes into the set because CG is a subset that I

comes into the set because **CG → I** and at this point, it eventually ends in this case. In this particular example, you can see that all attributes have got included. So, you can see that it immediately gives you another information as a byproduct of the closure that closure of AG is all attributes which mean that AG is a key. It has to be a key because **AG→ABCGHI**.

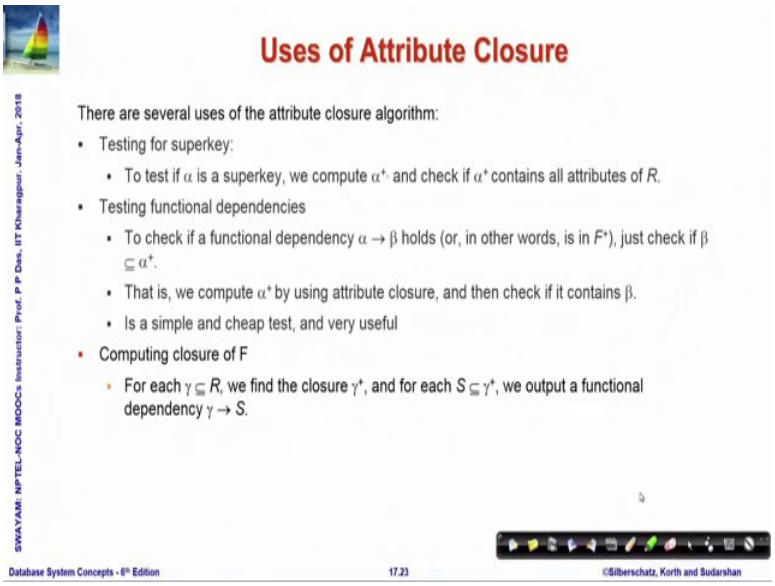
So, what is the meaning of $\mathbf{AG}^+$ being this? So, if the meaning of this is **AG →ABCGHI**, right, so we will see that this closure set has a lot of valuable information in this.

(Refer Slide Time: 28:31)



So, we can say that AG is a candidate key and because of this, we can also check whether AG is a super key or not. All that we need to do is drop some member from AG, we drop G and check whether **A→R** which means we check whether $\mathbf{A}^+$ = = R or not. We check we drop A from AG and check whether **G→R** which means $\mathbf{G}^+$ = =R and by that we can easily determine whether the set of attributes is a key or not.

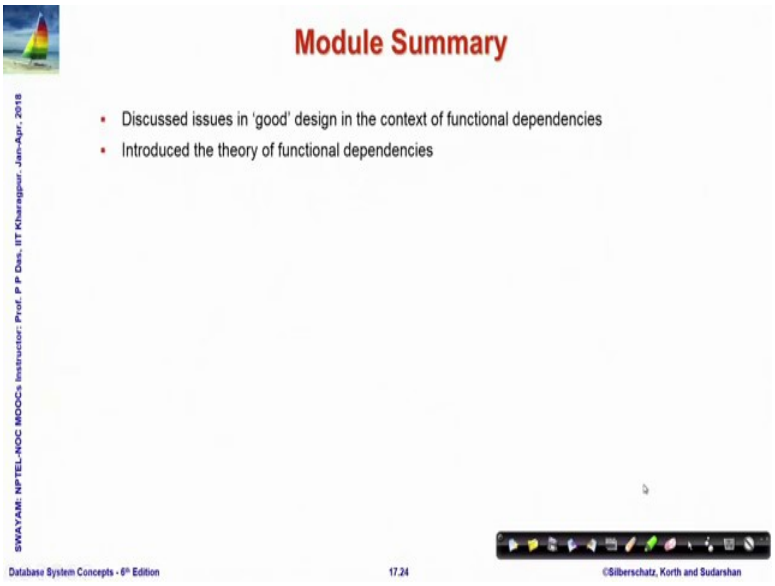
(Refer Slide Time: 29:14)



So, there are several ways. The attribute closure can be used as we have just seen. It helps you determine whether something is a super key. We can check for testing functional dependencies because if we have to check whether a functional dependency **α** → **β** hold, all that we will have to do is to compute the closure of the set of attributes **α** that is **α**+ and check whether **β** is a subset that. If it is, then certainly holds. If it is not, then it does not hold. So, it is simple and useful test that can be made use of.

So, it can also be used in computing the closure of F that for example, for every subset **γ** of R, if we find **γ**+ that is a closure of the set of attributes of **γ** and then, for each subset **γ**+, we know that there is a functional dependency **γ** →**S** which is just is the same statement being made in you know or in different forms and the closure of attributes is a very nice concept which help you play around in this multiple ways and we will see subsequently many of the algorithms for normalization.

(Refer Slide Time: 30:35)



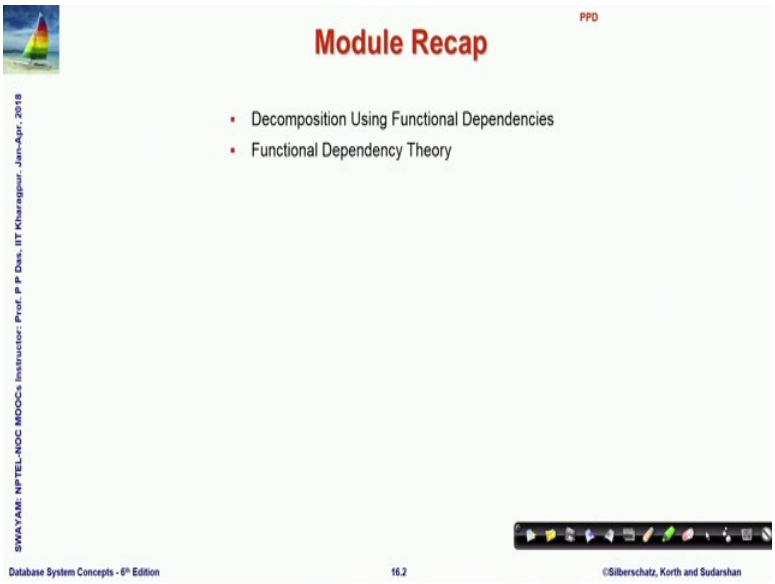
How they make effective use of this closure set, notion of both closure of functional dependencies and in very practical implementation algorithms, the closure of attributes. So, to summarize this module, we have discussed issues further issues in the good design in the context of functional dependencies and in the process, we have also extended the theory of functional dependencies and we will continue it this in the next module to get more insight into the algorithms that actually work with the functional dependencies.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture - 18
Relational Database Design (Contd.)

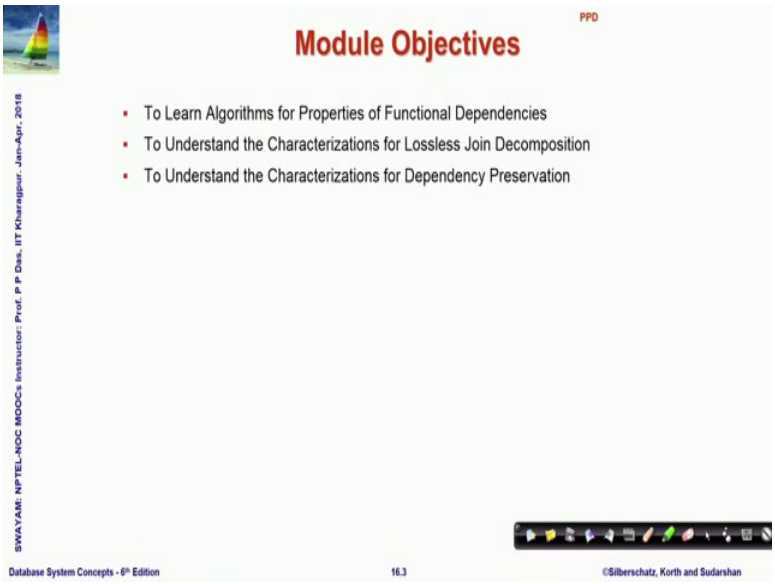
Welcome to module 18 of Database Management Systems. We have been discussing about relational database design. This is a part 3 of that.

(Refer Slide Time: 00:30)



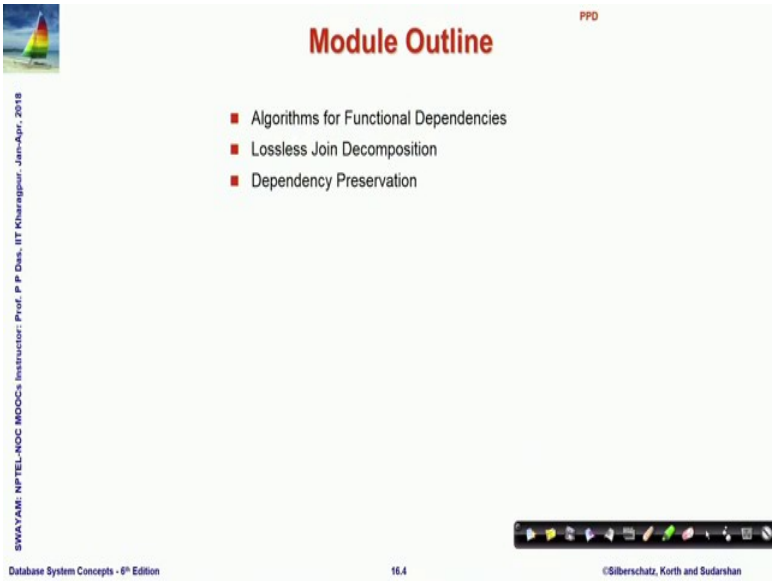
In the last module, we discussed about the Notion of functional dependency and decomposition based on that in an elementary level and certain bit of its theory.

(Refer Slide Time: 00:39)



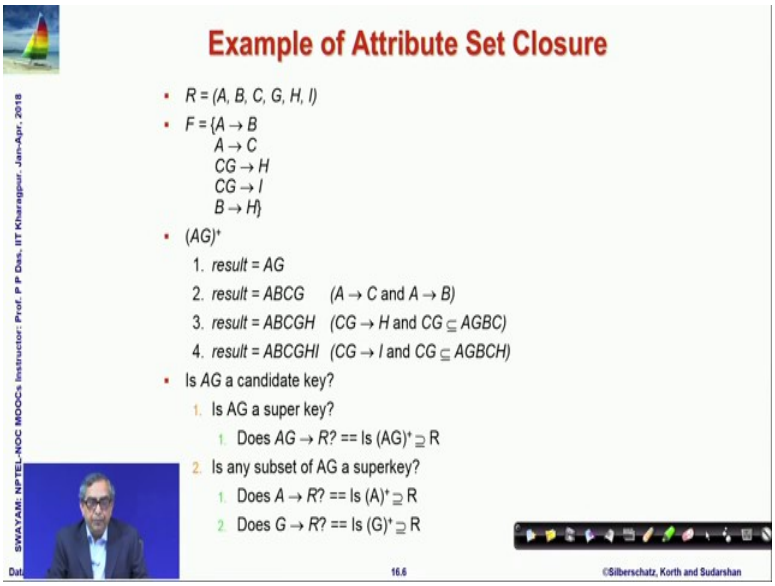
In this current module, we learnt different algorithms that use the functional dependencies and can make conclusions about the design or make changes to the design. We will also try to understand the characterization for lossless, join decomposition and the notion of deter dependency preservation.

(Refer Slide Time: 01:06)



Therefore, this module will have these three topics algorithms for functional dependencies, lossless join decomposition and dependency preservation.

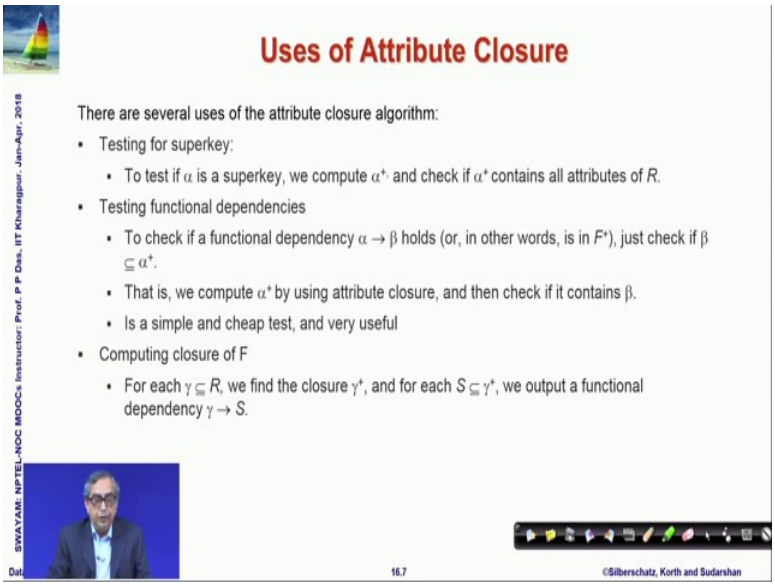
(Refer Slide Time: 01:15)



So, first we start with the algorithms and I quickly reproduce what we had ended in the last module in terms of computing the closure of a set of attributes. So, if we have a relation having these attributes and a set of functional dependencies, then for a given subset of attributes, in this case AG we can iteratively compute the closure set when no further changes can be done, and with using that we can make different conclusions.

For example, if our question is whether AG can be a candidate key, we would first like to check whether it is a super key that is whether its closure has all the attributes of R and we would like to check if we have taken a subset of AG. If we take just as a attribute A or attribute G whether the closure of that will actually work as a key or not.

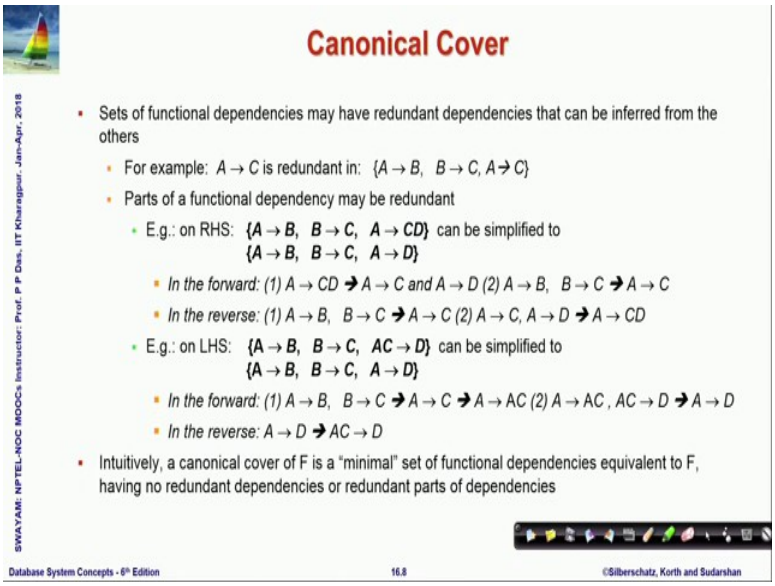
(Refer Slide Time: 02:12)



So, this algorithm of attribute closure turns out to be a very powerful one, where as we have just seen it can be used for checking super keys, the candidate keys, primary, non-primary attributes and so on. It can be used for checking functional dependencies. For example, let us suppose that if we have to check that whether if a particular functional dependency $\boldsymbol{\alpha \rightarrow \beta}$ holds, then rather in other words whether $\boldsymbol{\alpha \rightarrow \beta}$ is in the closure of the set of functional dependencies F, then all that we need to do is to compute α^+ that is a closure of the set of attributes on the left hand side of the dependency and check if β is a subset of that. If β is a subset of that, then I know that $\boldsymbol{\alpha \rightarrow \beta}$ actually holds.

So, in this manner it can also be used to compute the closure of the whole set of functional dependencies F. So, if I mean at least at A, rudimentary level we can think of that. If we take any subset of the set of attributes and find the closure and then, all attributes that belong to that closure set are actually functionally dependent and therefore, those functional dependencies will exist.

(Refer Slide Time: 03:42)



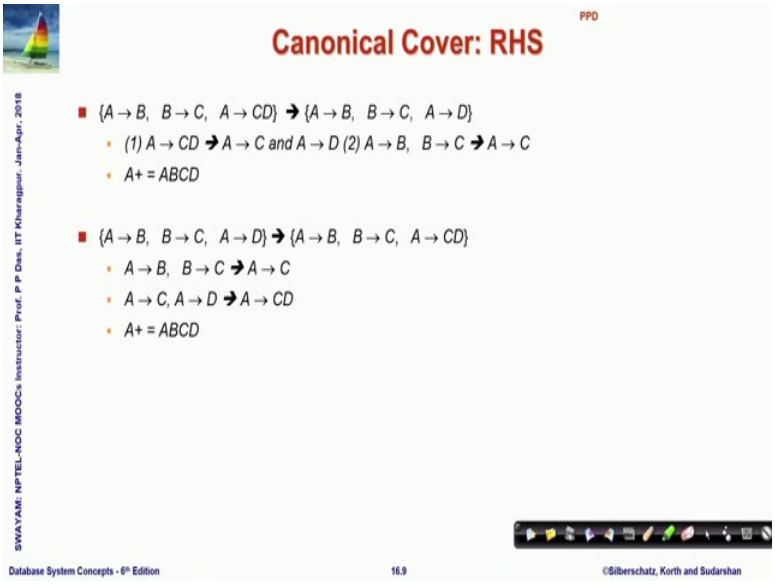
Now, we move forward from there and talk about what is known as a canonical cover. A set of functional dependencies may have a number of redundant dependencies also. So, we need to understand that because there are lot of dependencies which can be inferred from a certain set of dependencies, for example if you look into this set, you will easily understand that in this whole set if I actually have just this, we will be able to by transitivity, we will be able to conclude about **A → C**. So, in that way AC, **A → C** is a redundant dependency.

So, here I am just showing you some examples. For example, say I have a set of functional dependencies as this set and I want to know whether I can replace it by a simpler set here where this particular attribute on the right hand side of this dependency may be extraneous. So, if I have to do that, then what we need to perform is, we need to show that given the set of functional dependencies, the original set whether this can imply this set that is from this set of functional dependencies, whether I can logically conclude the simplified set.

So, using the rules we will need to do that I have worked that out here under the forward scheme and we would also need to establish that if I have the simplified set, then can I go to the original set that was given. So, if the simplified set also logically → the original set, then we can say that these are in a way equivalent and therefore, I would like to use a simpler set.

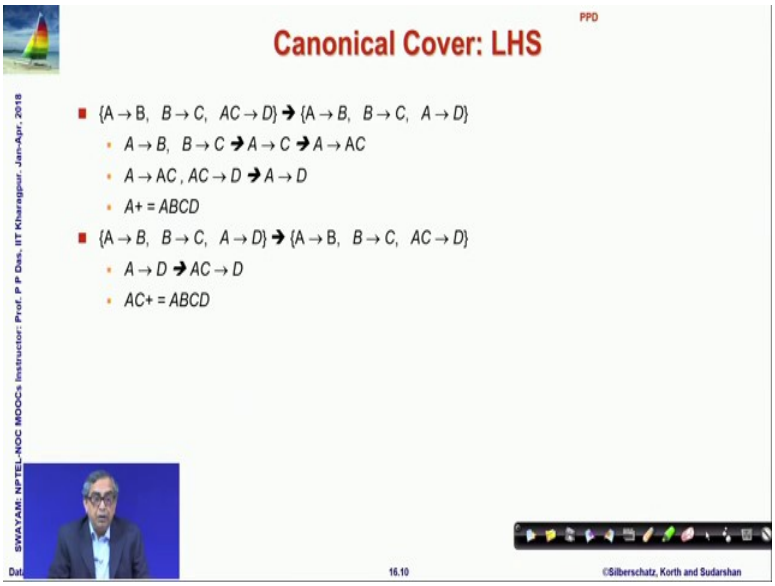
So, there is another example following here where I have another set given, where if we look into this, I would like to check whether I can get rid of this C on the left hand side and as it stands, we can actually do that and here in this whole process, I have shown it in terms of using the Armstrong's Axioms how you can prove this, but what we can do to systematize this whole process, we can again make use of the notion of closure of attributes and compute whether these two sets are equivalent, whether simplification can be done. So, we will say a cover is canonical. If it is in a sense minimal and still equivalent to the original set of dependencies and we will formally introduce what is minimal.

(Refer Slide Time: 06:44)



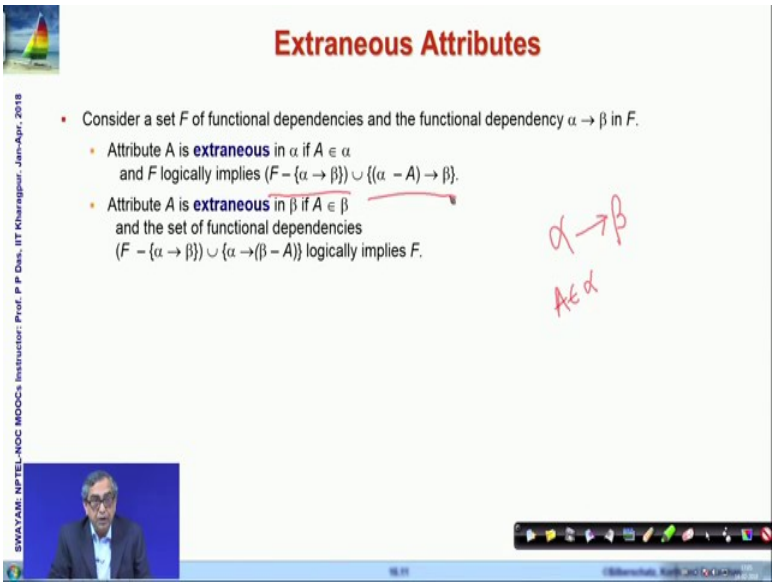
Before that let us just look at the same examples again. So, we are trying to show the forward direction in the first case and the reverse direction in the first case, but the only difference that I wanted to highlight is in terms of showing that you do not need to really explore on the Armstrong's Axioms, but what you can do is, you can simply take the left hand side attribute and compute it closure and see whether the right hand side is included. That is basically testing for whether the given functional dependency is actually implied.

(Refer Slide Time: 07:19)



Similar things can be done to simplify the left hand side also. So, this is the other example I showed and I am just showing you that how you conclude this based on the closure of attributes algorithm.

(Refer Slide Time: 07:32)



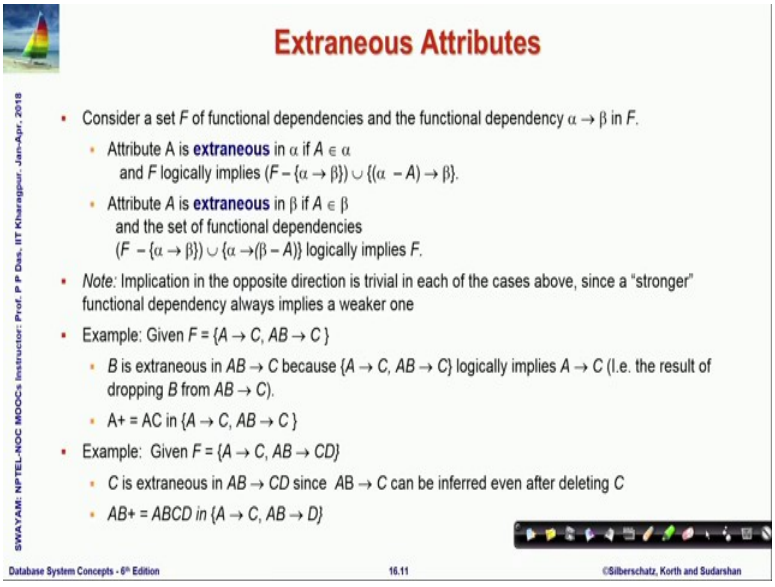
So, now I can formally define these possible removals. So, if I can remove an attribute as I have shown I can remove it from the right hand side or I can remove it from the left hand side. So, if an attribute can be removed, then it is called extraneous. So, if I have a functional dependency, let us say $\alpha \rightarrow \beta$ and I have an attribute $A \in \alpha$, then we can check

whether it is possible to remove A from α. So, to test that what we do is, we form a new set by removing the original functional dependency and adding the new functional dependency where the left hand side does not have that A and if F logically → this, then certainly we can conclude that A on the left hand side of the functional dependency was extraneous.

Similar thing can be done for checking if there is an extraneous attribute on the right hand side of a dependency and in this case, naturally what we will need to do is, we will need to work out the simpler set and then check whether F is implied by that because as you can understand that if you are making the left hand, if you are removing an attribute from the left hand side, then you are making your precondition softer.

So, you need to see whether that is implied by the original set and on the other hand, if you are removing something on the right hand side, then you are making your consequence simpler. So, you need to understand whether that set → the original set.

(Refer Slide Time: 09:27)

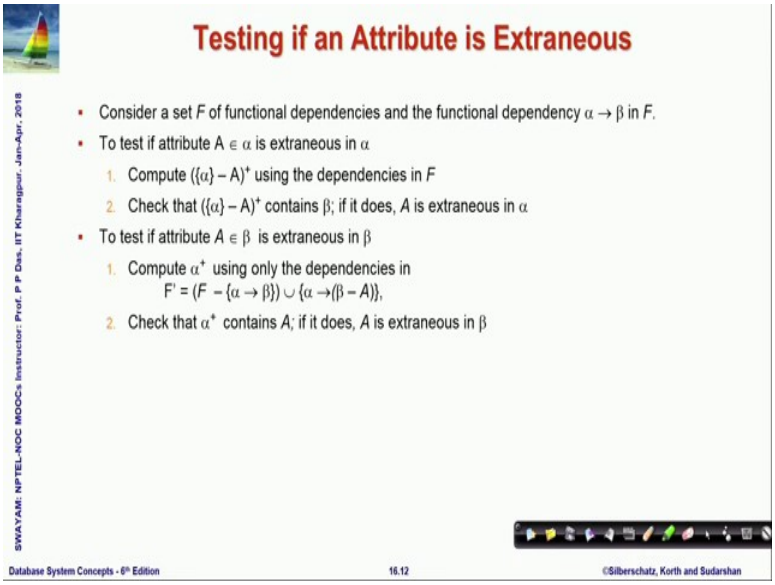


So, if you look into that and obviously, the other directions of this implication is not necessary to be proven because that will automatically follow because in the first case when I am removing an attribute, extraneous attribute from the left hand side of a functional dependency, naturally the set that I get that will always imply the original set because it is always possible to add additional attributes on the left hand side and so on. So, here are some examples worked out. So, here where I show that given a set AC and

AB → C, B is actually extraneous because as you can see if I remove B, then I get **A → C** which is originally already there in the set. You can establish that by computing the closure of the attribute set.

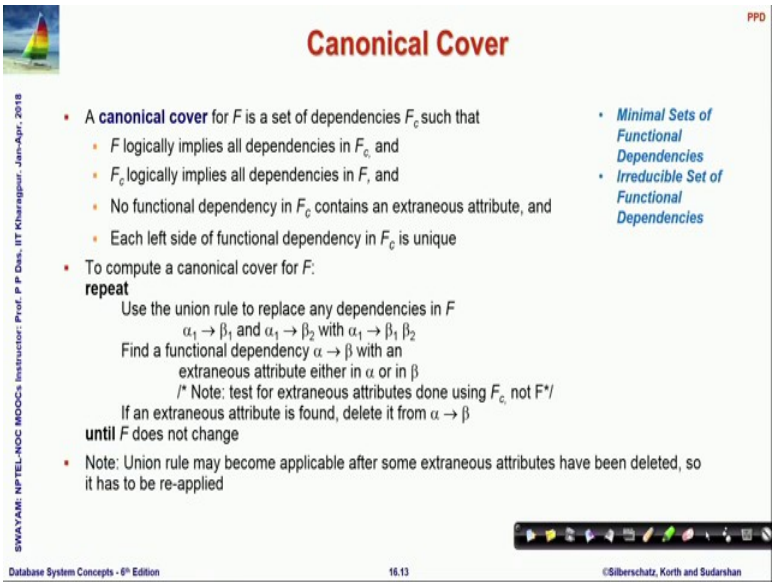
Another example where you are trying to see an extraneous attribute on the right hand side; so in this example, C on the right hand side of the set **AB → CD** is extraneous because it can be inferred even after because **AB → C** can be inferred even after deleting this C from the right hand side.

(Refer Slide Time: 10:42)



So, these are using this notion. We can formalize a test for whether an attribute is extraneous. So, this is the formal steps of the step are given here, but I am sure you have already understood through the example.

(Refer Slide Time: 10:58)

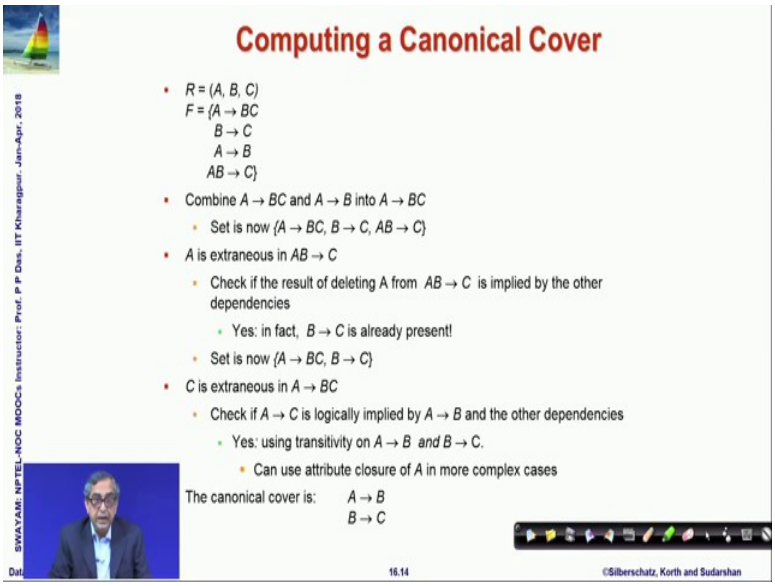


So, given this a canonical cover of a set of functional dependencies F, it is denoted by F_c will mean that it is a set which is equivalent to F which means F will logically imply all dependencies in Fc and Fc will logically imply all dependencies in F.

No functional dependency in Fc will contain any extraneous attribute. So, all of them will be required attributes and each left hand side of the functional dependency in F_c must be unique. So, it is a minimal set of functional dependencies. Please note on these two core points. A cover is canonical if it is a minimal set and it is an irreducible set.

So, neither you can remove any dependency nor you can remove any extraneous attribute from this dependency set. So, here is the algorithm. So, I am not going through the steps of the algorithm. You can go through that and convince yourself that it indeed computes the canonical cover and practice more on that.

(Refer Slide Time: 12:07)

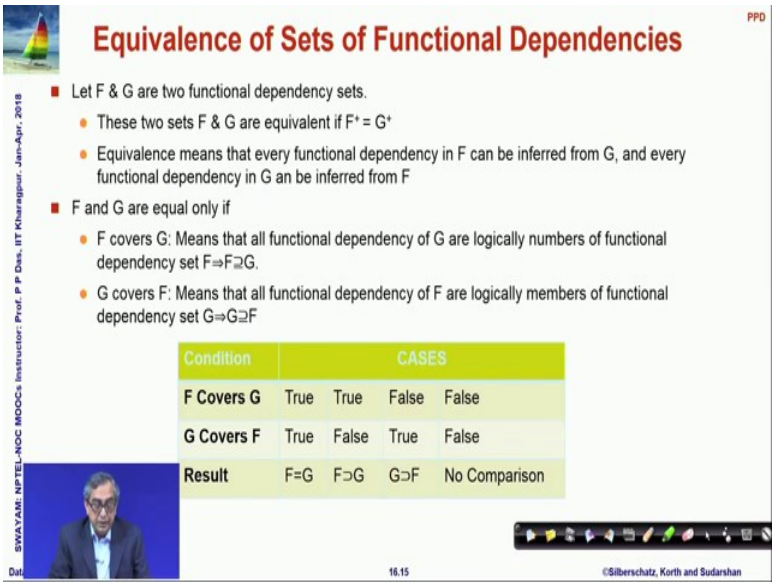


So, here I have shown an example where we want to compute the canonical cover here. So, first since all left hand sides have to be unique, so first we combine two, these two into in terms of **A → BC**. So, it becomes a simpler set. So, **A → B** is removed, then I would check for A being extraneous in **AB → C** and we find that it indeed is extraneous.

So, because B → C is already there, you can do the formal test in terms of the closure. So, the set gets even simpler. I will check if C is extraneous in **A → BC**. I find that it indeed is and again you can use transitivity to get here or can use attribute closure and finally, I get that the set of the original set F is covered by a canonical set where just you have **A → B** and **B → C**.

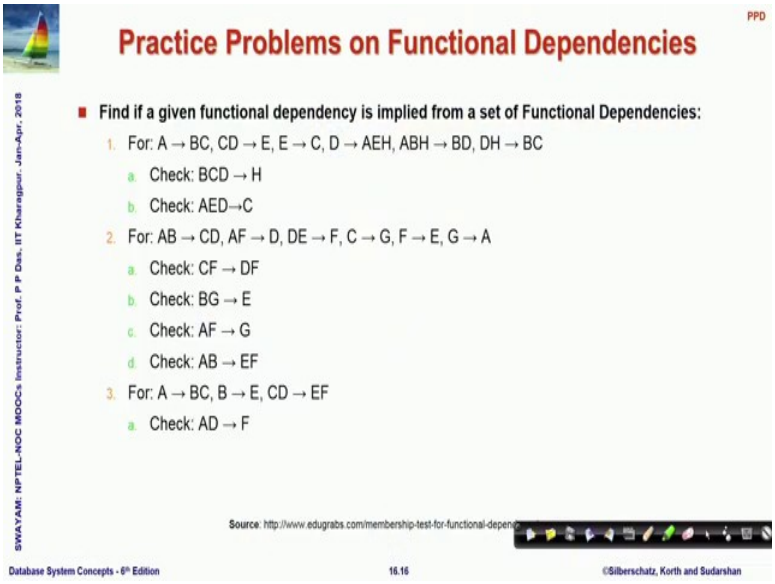
So, this set is logically implied by the original set and this set can logically imply the original set and we will often use the canonical cover for simplicity and for ease of application.

(Refer Slide Time: 13:32)



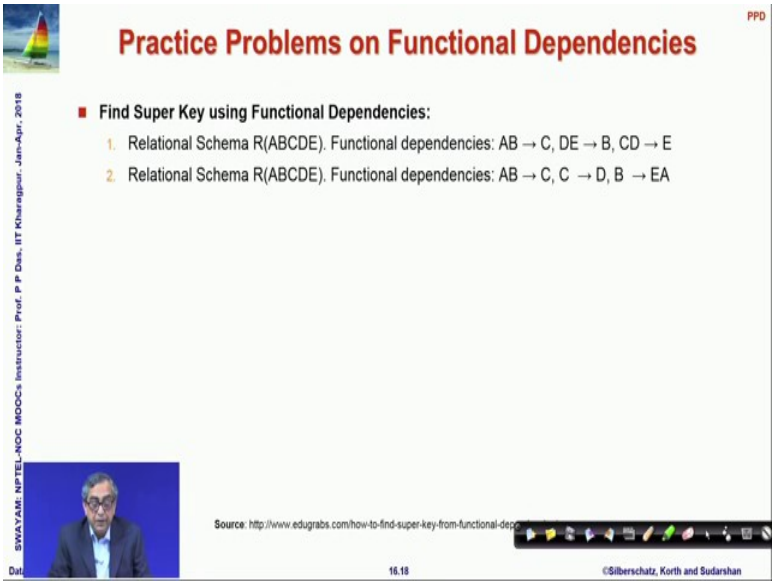
Naturally this is strongly using the underlying concept of equivalence of two sets of functional dependencies F and G. They are equivalent if their closures are equal or in other words, if F covers G and G covers F, that is F logically → G and G logically → F. So, this table shows you at different conditions where you can conclude whether F and G are equivalent sets of functional dependencies. So, they will have to, both covers have to be true for the sets to be equivalent.

(Refer Slide Time: 14:09)



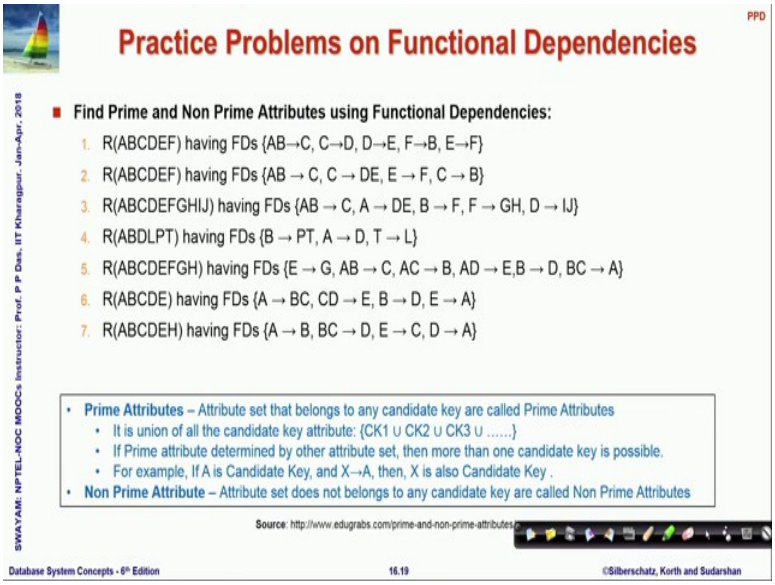
Next what I have done is, we have put a number of practice problems for various kind of things that you can do with functional dependencies. The first set of problems. Find in first set of problems you have to find if a given functional dependency is implied from a set of functional dependencies. So, there are three problems where three sets of functional dependencies are given and you are given to check one or more functional dependencies if it is implied from that set. So, use the attribute closure and the algorithm that we have discussed to practice these problems and become master of that.

(Refer Slide Time: 14:54)



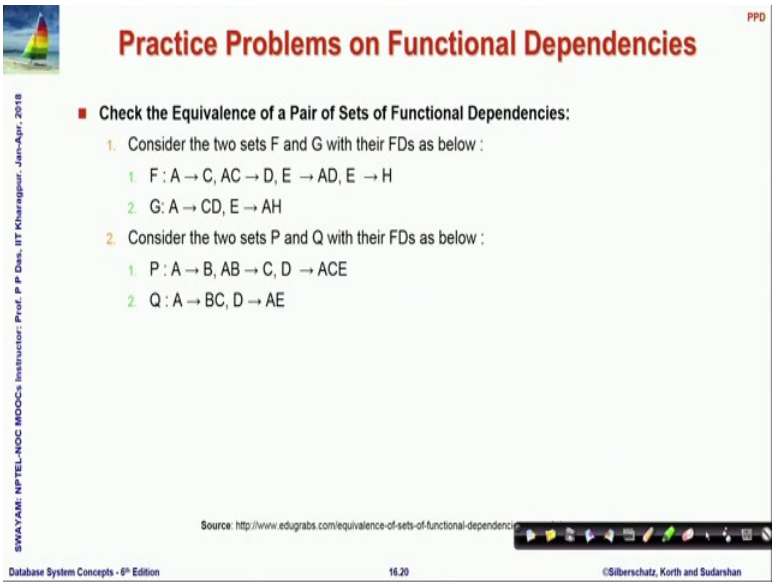
You can also check if you can find candidate key using the functional dependencies. The sets are given. Your task would be to find the candidate keys.

You can also use the algorithms to find super keys for a given set of functional dependencies. So, do practice these problems.



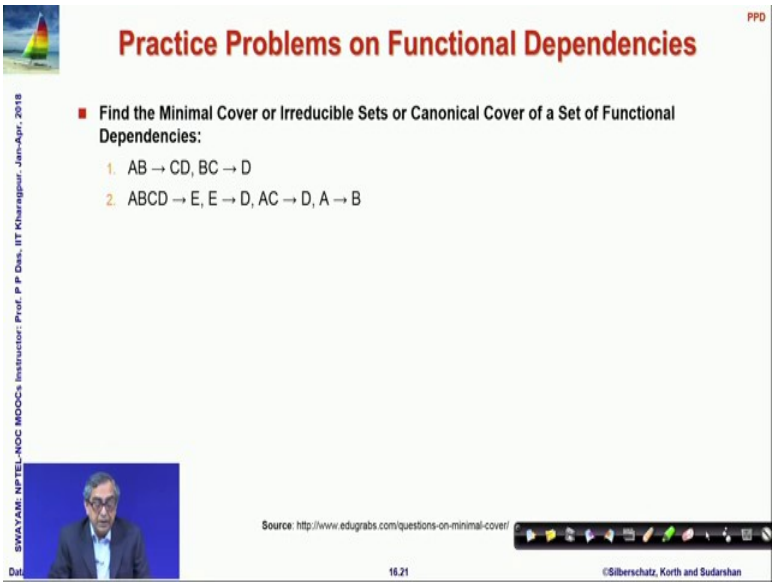
You can find prime and non-prime attributes using functional dependencies. Prime attributes are attributes that belong to any candidate key, not necessarily the same candidate key. All attributes that belong to some candidate key, you take a set together and you call them as a prime attribute and non prime attributes are those that do not belong to any candidate key at all. So, here your task is to find the prime and non-prime attributes using the sets of functional dependencies given.

(Refer Slide Time: 15:50)



You can check for equivalents for a pair of sets of functional dependencies. There are couple of problems given on that. So, please try them out.

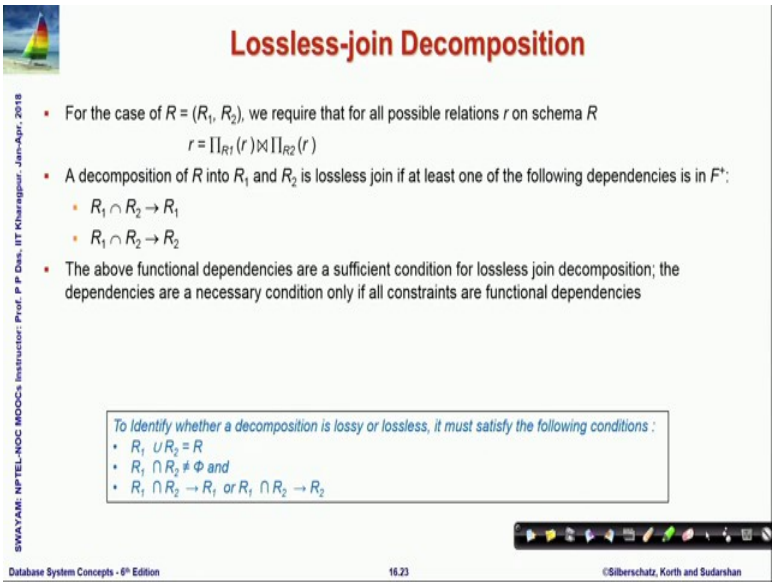
(Refer Slide Time: 16:01)



For here for the different sets, you have to compute the minimal cover or the irreducible set or canonical cover of the set of functional dependencies.

So, please practice on this problem, so that you become comfortable with using this algorithms for dealing easily with the functional dependency sets of functional dependencies, individual functional dependencies and so on. So, after this week is closed and your assignments are also done, then we will publish the solutions for these practice problems as well next let me take up a little characterization of the concept that we had introduced earlier in terms of the lossless join decomposition.

(Refer Slide Time: 16:48)

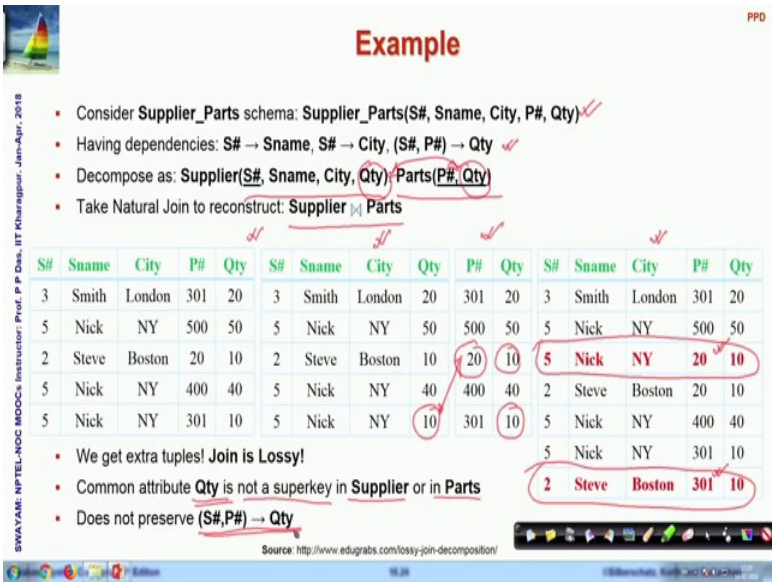


So, in the lossless join decomposition, the problem is say that you have a relational scheme R and you are trying to divide that into two relational schemes R1 and R2. So, both R1 and R2 are having a set of attributes and R naturally has a set of attributes which is a union of the attributes of R1 and R2, then is it possible that if I take a relation, project it on the attributes of R1 and on the attributes of R2, the two relations that we get. If I take a natural join of that, do I get back R?

If I do, then I say that I have a lossless join. If I do not, then I have lost some information due to this projection and re-computation of the original relation based using the natural join. This requirement of lossless join decomposition is determined if at least one of the following dependencies exist in the closure set of F which this is saying that if I do **R1** ∩ **R2**, that is A attributes which are common. You will recall that when we do natural join, it is this set of attributes which take part because these set of attributes will help you $\Pi_{R1}(r)$ $\Pi_{R2}(r)$.

So, if **R1** ∩ **R2** → **R1** or **R1** ∩ **R2** → **R2**, that is if the intersection set of attributes is a super key either in R1 or in R2 or both then, we say that the join will be a lossless join. Note that this is a sufficient condition which means that there could be some instances where this property is not satisfied yet the join is lossless, but we need guarantees for our design. So, we make use of the fact that if one of these conditions are satisfied, then it is a sufficient condition to say that the join will must, join will necessarily be lossless.

(Refer Slide Time: 19:10)



So, here I give you a quick example to show the idea. So, we have a supplier relationship here which has five attributes. Here is an instance of that and we know that these are the dependencies that hold the **supplier number → the supplier name** and the **(supplier city, supplier number, product number) → quantity** and we decompose them in this manner, we put a supplier relationship where we have the number, name, city and quantity of supplier and then, we have parts relation where we just have a product name and the quantity.

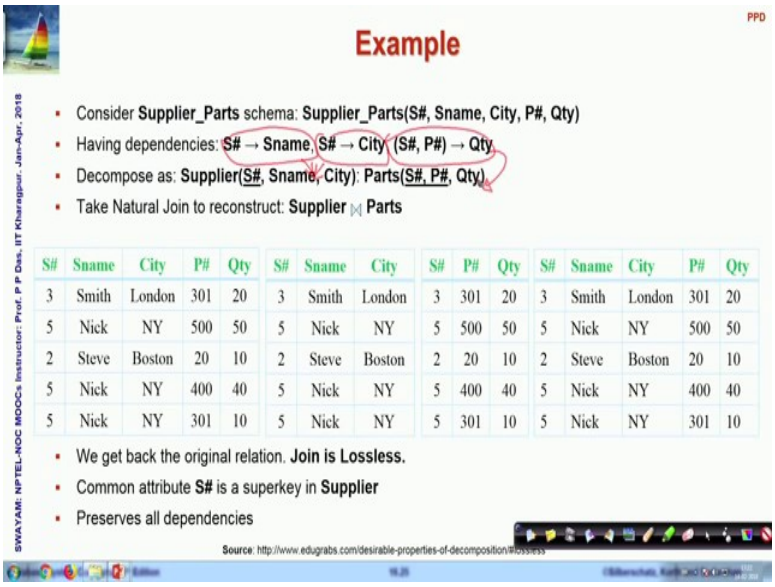
So, this is the projected supplier relation instance. This is a projected parts relation instance and then, we take a natural join to reconstruct. So, we are taking a natural join to reconstruct and we get this relationship. Now, our desire was that we must get back the original relation, but if you compare, you will find that this is not the case here. We have one tuple here and we have another tuple here. I have specifically highlighted them in red which were not there in the original relation.

They have come in because when I did the join naturally, the join had to be performed on this common attribute quantity and based on that value. So, Nick, 5 Nick NY, then we have 10, 5 Nick NY 10, this entry and we have two entries of 10 and 10 here. So, the combination of this with this where the product number is 20 is actually not present in the original instance of the relation and that is what shows up here a similar one exists here.

So, we get extra tuples and mind you though we are actually getting extra tuple, we will say that this join is lossy because if you get extra tuple, then you are losing information, you are losing correctness. So, being lossy is actually losing correctness. So, even though we have more tuples, we say that this is a lossy join and you can now go back and analyze it. The common attribute QTY is not a super key either in this or in this. So**, R1** ∩ **R2** → **R1** or **R1** ∩ **R2** → **R2** does not hold. So, it does not and in addition it also does not preserve this functional dependency because these are not, no more determined.

Now, let us see it. So, we saw a case where the join decomposition that we did and then, the subsequent join that we performed did not prove to be a lossless join. We lost information. So, let us take a look as to can we actually do a decomposition which will be lossless where we will not lose information.

(Refer Slide Time: 22:29)



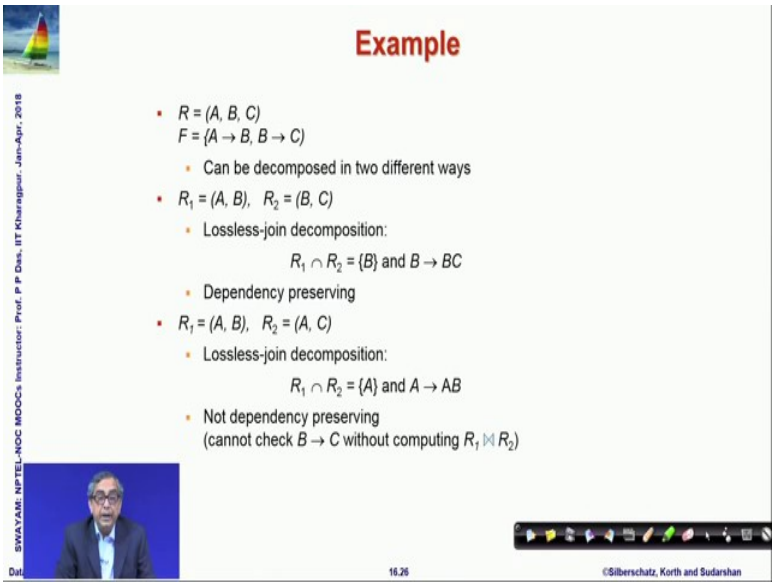
So, I take the same example the supplier, but the decomposition, the same set of dependencies also, but the decomposition is different. Now, we have name number, name and city in one supplier relation and supplier name, number, product number and quantity in the other parts relation and then, we again go back and perform the join.

Now, we find that from the original relation, this was the original relation, this is the projected supplier relation, these are projected parts relation and this is the natural join of these two relations. So, this is the natural join of these two relations and we find that they exactly match with the original relation.

So, we have not lost any information we get it back. So, we say that the join is lossless and the reason we could guarantee that is because if you look into the set of functional dependencies, you will find that S number, the supplier number is a key in the supplier relationship because **S#→ S name , S city**. So, **R1 ∩ R2 → R1** is true here and therefore, it actually gives you a lossless join.

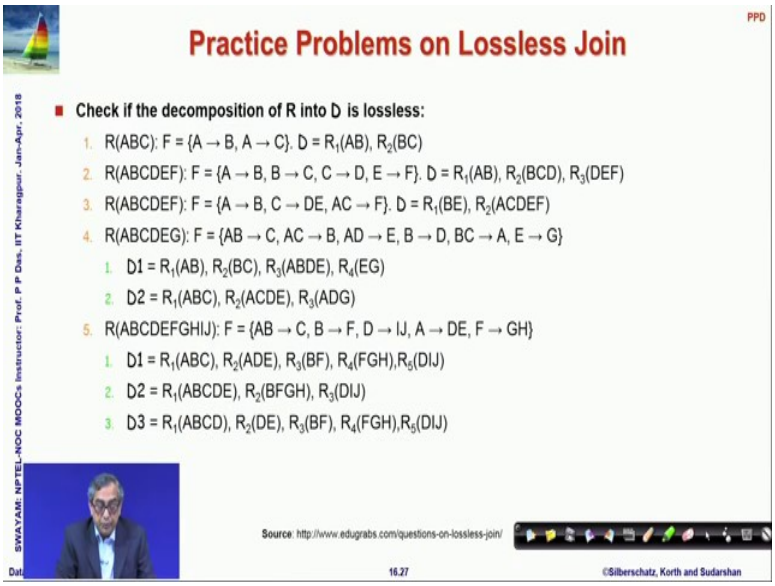
It also preserves all the dependencies because if you look into these dependencies, you can check for this dependency. In this relation, you can check for this dependency also in this relation and you can check for this dependency in also in the parts relation which is something which we were not able to do in the last decomposition that we have.

(Refer Slide Time: 24:33)



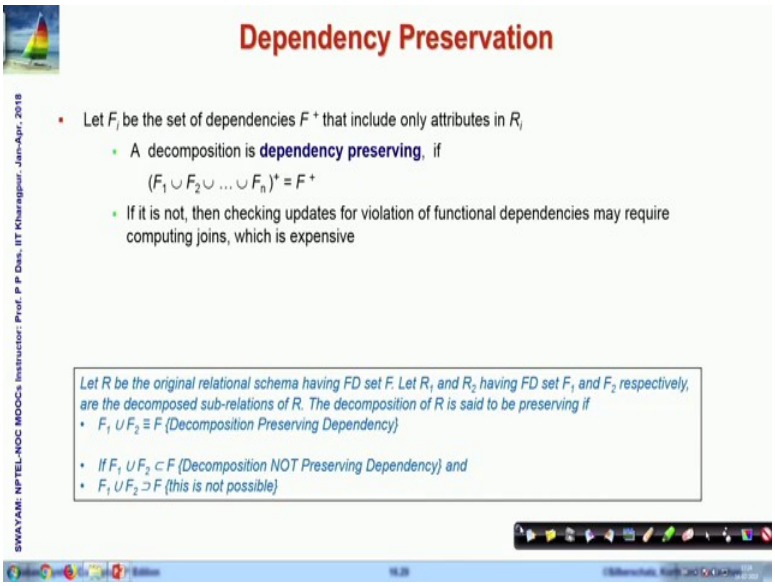
So, naturally this is a type of decomposition that we will prefer. So, here we have I have given some more examples which you can practice and I show that given a very simple schema having three attributes and two dependencies, one decomposition into AB and BC is lossless join decomposition whereas, the other one AB and AC is a lossy decomposition.

(Refer Slide Time: 24:57)



I have given a number of practice problems on lossless join, so that you can practice and become master of these kind of algorithm. Finally, let me quickly go over the dependency preservation concept.

(Refer Slide Time: 25:17)



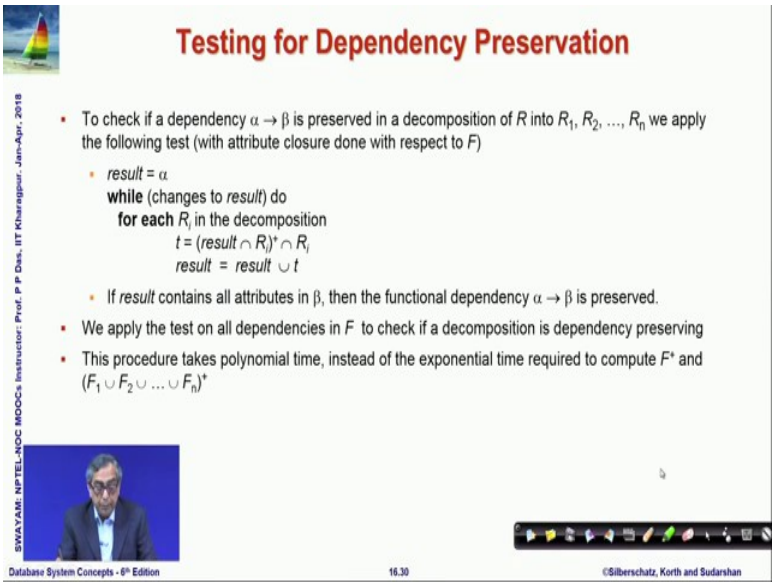
Dependency preservation is if you have a relation which you have decomposed into n different relations, so if you decompose a relation into a number of relations, then naturally all functional dependencies you cannot check on all the relations because a

dependency may involve attributes all of which may not be present in a particular decomposed relation that you have. It may be distributed amongst different.

So, when you do this decomposition, for every relation you get a new set of subset of functional dependencies. So, the decomposed relation Ri the i^{th} relation will have a set of dependencies $\mathbf{F_i}$ which is a subset of the original set F and involves only the attributes which exist in $\mathbf{R_i}$. So, the decomposition will be said to be dependency preserving if I can take the union of all these functional dependencies, what is projected on R1, on R2 and Rn, F1, F2, Fn. If we can take union of and if we take F, they must be equivalent sets which we know the requirement.

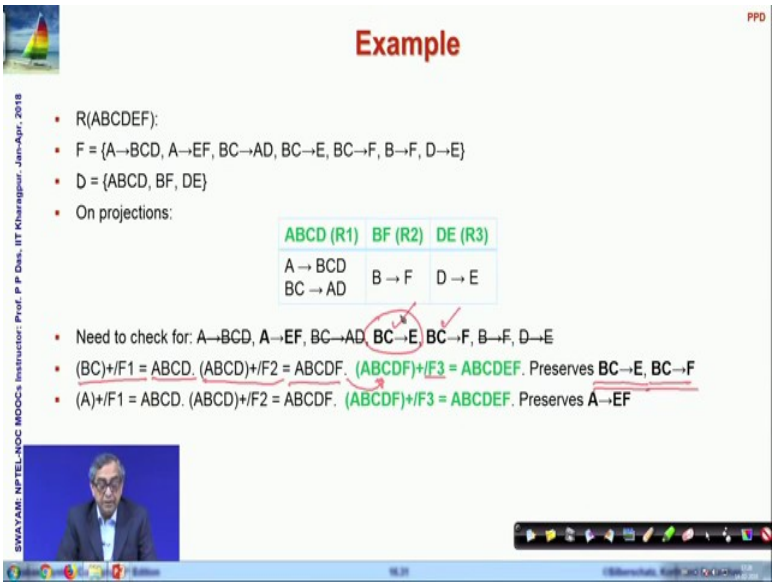
So, equivalence mean that their covers will have to be equal. If it is not, then some there will be at least one dependency which you will not be able to check in any one of the projected relations and to be able to check that, you will have to compute the natural join and that is as we know is a very expensive process and we would not be able to do that on a regular basis.

(Refer Slide Time: 27:12)



So, here I have written down the algorithm to test if a decomposition actually preserves the dependency or not. So, I will not go through the steps. I will leave that for you to understand, but what I will do, I will just show you a simple set of worked out example and reason on that.

(Refer Slide Time: 27:34)



So, I show you two different methods of doing this. So, here we have a set of attributes given the dependencies that work in that and a particular decomposition. So, given the set of attributes and the decomposition if we project, now if we project the set of functional dependencies and these are the sets that we get. So, on R1, we have two dependencies on R2, we have three dependence, one dependency and R3 we have one dependency again.

So, if we now think about the U of these and the closure for that, then we can see that these four dependencies which occur here and therefore, I have struck them off in this set. These four dependencies can be checked directly on the projected relations. So, that leaves us with three dependencies in the original set which cannot be checked on any one of R1,R2 or R3. For example, if you consider **BC → E**, then B exist on R1 and C also exist on R1, but E is not there. So, you cannot check that dependency on R1, you cannot check that on R2 because C and E do not exist and you cannot check them on R3, check it on R3 because none of them actually exist.

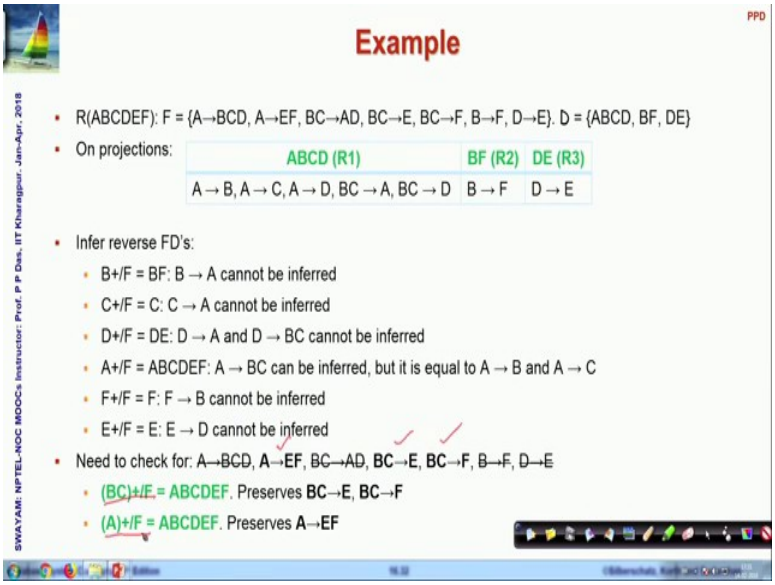
So, what we will need for the dependency preservation to hold is the dependencies which are already existing four dependencies that are struck off if they collectively can logically imply these dependencies, so that they can be checked. Then, we will be able to say that this is dependency preserving. So, what you do is something very simple. You want to say, you want to check whether this is preserved. So, we start with the left hand

side and compute the closure. The only difference you compute the closure first with the set of functional dependencies projected on R1, that is F1, the set closure set that you get, you take that and compute its closure with respect to the second set of functional dependencies F2.

The closure that you get, you take that and you compute the closure with respect to the third set of functional dependencies which is on R3 and that is your final closure set. So, this closure set includes the right hand side attribute E. So, we can conclude that **BC →E** and that relationship will be preserved because we have starting from BC. We have seen that in every projected relation what all implied functional dependencies that can be checked which is what the meaning of the closure set of attributes R and since that set eventually has E, we will know that this can be, this will be preserved.

This set also has F. So, the other one will also be preserved. So, this is preserved, this is preserved to check whether this dependency is preserved. We need to again repeat the process and find whether EF belongs to the final closure set which it does and therefore, we conclude that this decomposition is dependency preserving.

(Refer Slide Time: 31:04)



With the same example I will just show you a little different way of ah doing the same exercise. I have not written down the algorithm for this in longhand, but the example should be quite illustrative. So, we are what you do when you project, you check if some dependency has multiple attributes on the left hand, on the right hand side, then you

write them in a separately decomposed manner. So, **A → BCD** is written in terms of three dependencies. **A → B**, **B → C** and **C → D**. So, you make sure that all dependencies are written in a form where the right hand side has a single attribute, then you compute what is known as the reverse functional dependencies that is you take the right hand side and compute whether the right hand side can imply the left hand side.

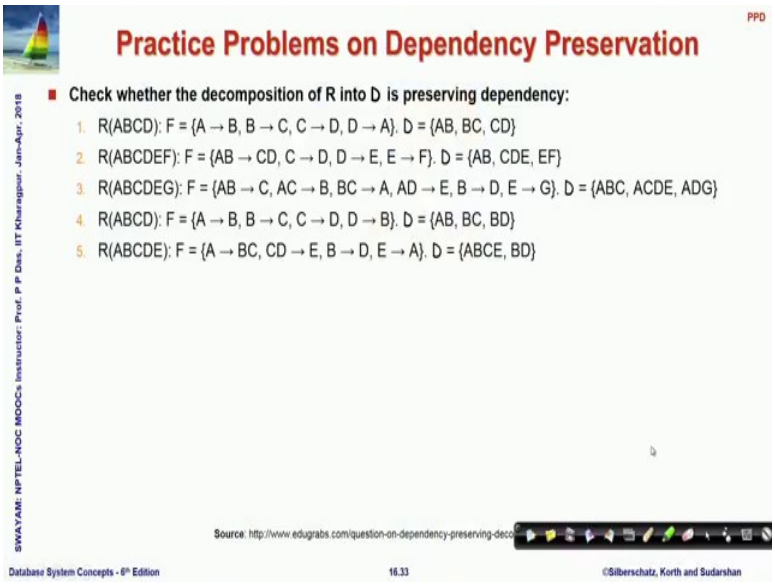
So, I will just show you one. So, in case the right hand side here is B, you have AB on the right hand side. So, you compute the closure with respect to F. The original set, not the projected set of D and you get BF. So, you know that this inverse, this reverse functional dependency which is **AB → A** which is the reverse dependency cannot be inferred and you do this for each of the right hand side single attribute and check if some, if the reverse dependencies can be inferred or not.

The interesting case occurs here where if you try to do the closure of A, you actually find that **A → BC** which is a reverse of this functional dependency can be inferred, but you do not consider that as a violation because it is you already have **A → B** and **A → C**. So, that logically implying that **A → BC**. So, it is not a new violation that is getting imposed.

So, with this your test for reverse functional dependencies is passed and then, you finally check for whether the three dependencies which are not part of the projected set of dependencies, you take the closure of the left hand side with respect to in this case. Again, the original set of functional dependencies, not the projected one and check if the right hand side belongs there. If they do, then combined with these two strategies you say that the set of functional dependencies are preserved under this decomposition.

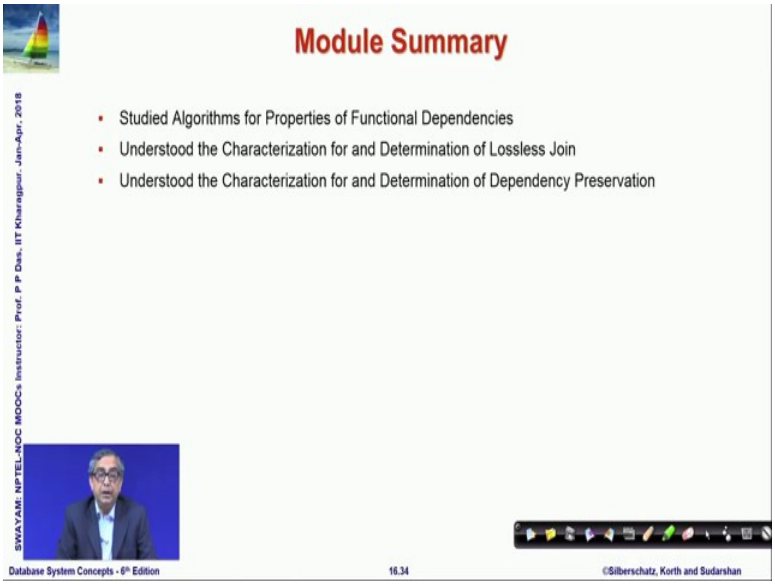
So, this is the process to follow. You can follow any one of the two approaches to solve.

(Refer Slide Time: 34:01)



So, I have given some practice problems on dependency preservation which you should practice on to.

(Refer Slide Time: 34:06)



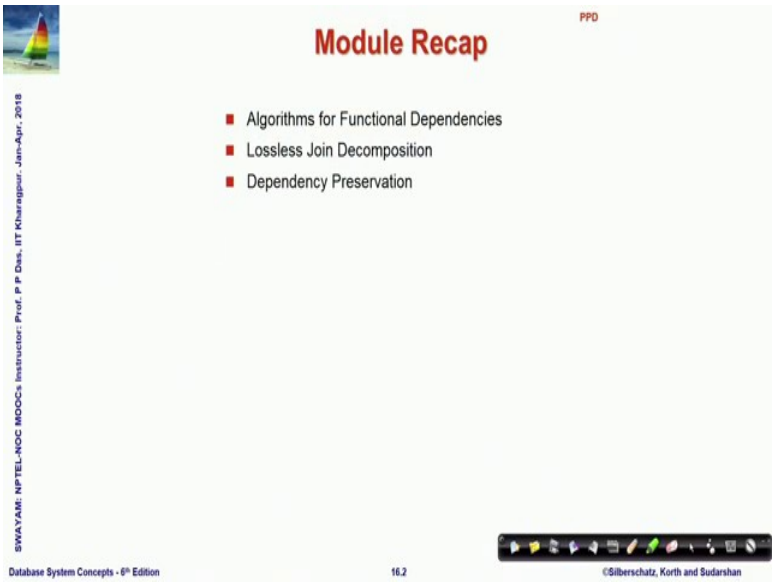
Summarize we have studied the algorithms for properties of functional dependencies and we have understood the characterization and determination algorithm for lossless join decomposition and for dependency preservation in a decomposition. In the coming module, we will make use of these and discuss about how to improve these designs of relational schemas through the use of different normal forms.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture - 19
Relational Database Design (Contd.)

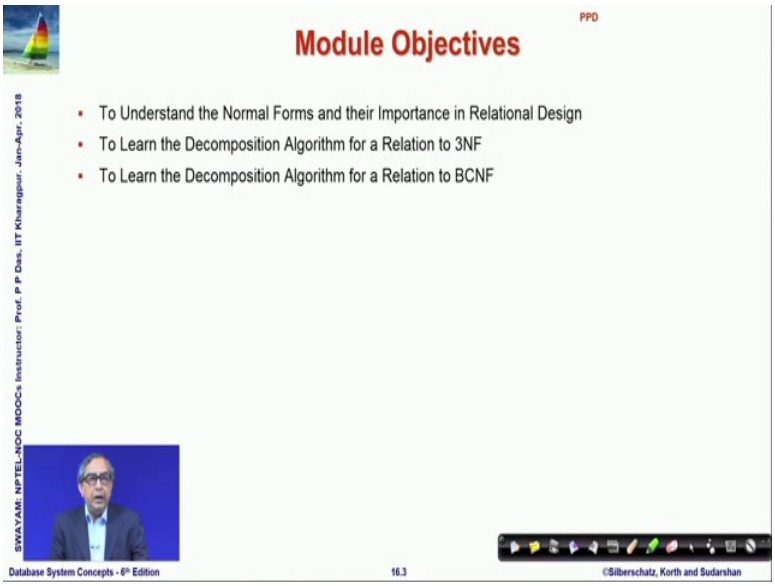
Welcome to module 19 of Database Management Systems; we have been discussing relational database design and this is the fourth part; fourth module in that series.

(Refer Slide Time: 00:38)



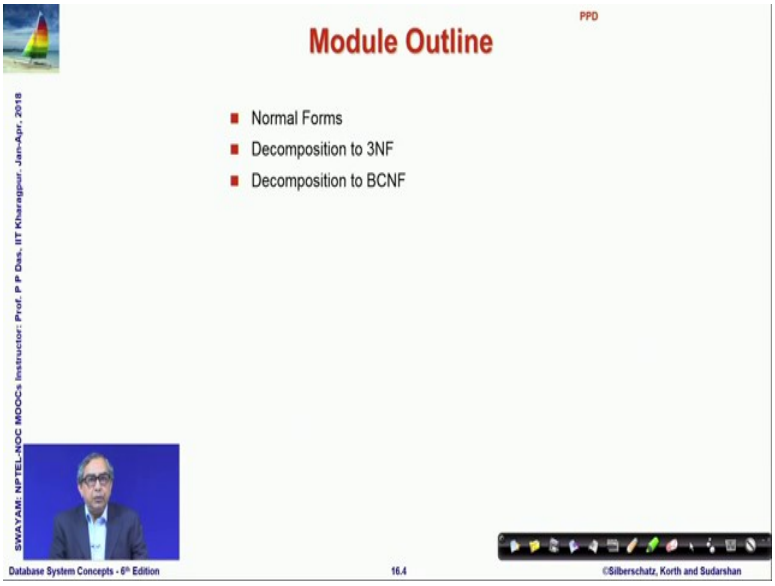
In the last module, we have discussed about algorithms for functional dependencies lossless joint decomposition and dependency preservation. So, based on this foundational algorithms and concepts.

(Refer Slide Time: 00:51)



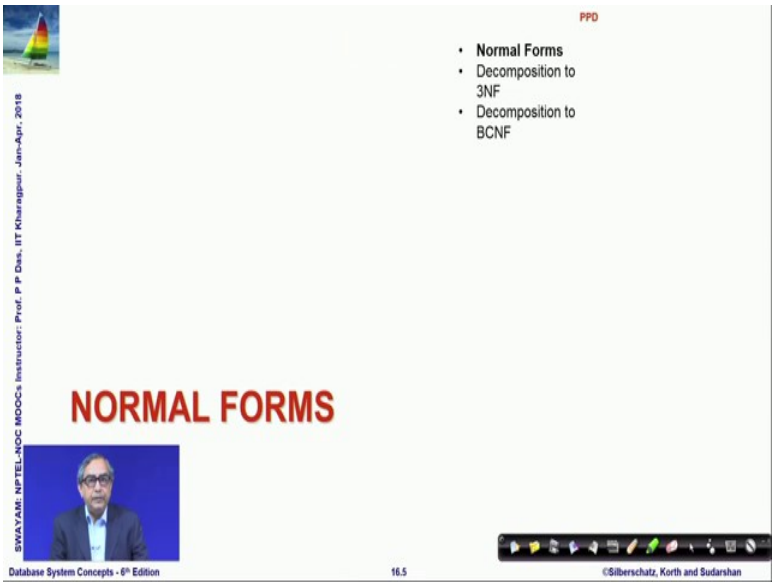
We will in today's module get into understanding the core design aspects of relational databases; that is a normal forms and how important they are in terms of the relational design. We would specifically learn about decomposition of a relational schema into the third normal form and into Boyce Codd BCNF form.

(Refer Slide Time: 01:20)



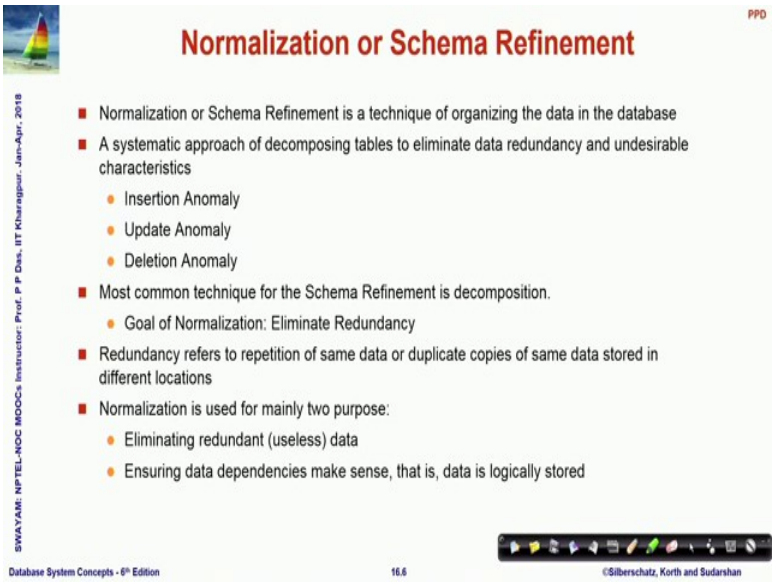
So, our topics will be the three normal forms decomposition of 3 NF and into BCNF.

(Refer Slide Time: 01:27)



So, starting with the normal forms.

(Refer Slide Time: 01:29)

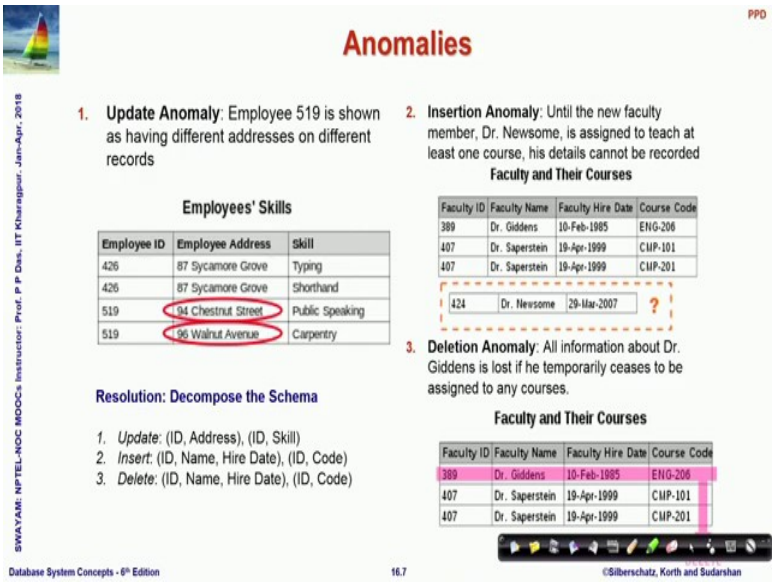


So, normal forms or normalization of a schema is a technique of refinement to organize the data in the database. So, the question naturally arises as to why do we need to do this refinement after we have done a design based on possibly the E-R diagram based approach that we had talked of we had identified the entities and we had identified the attributes for the entities their relationships; then why do we need to normalize?

The answer to this question lies in the fact that a design for a relational schema may give rise to a variety of anomalies in terms of the data. These are typically three anomalies which concerns us most the insertion, the update and the deletion anomaly. So, the anomalies happen when there is redundancy in the data in terms of the schema. And whether there will be redundant data and how much what kind of redundant data would be there depends on the design of the database schema depends on the design of the normal form that we are using for it.

But if we have redundancy then there is potential for anomalies and therefore, we want to reduce the redundancy and get rid of this anomaly.

(Refer Slide Time: 03:00)



So, we will quickly take a look into the anomalies that are that we are talking of first one is called an update anomaly. So, we are showing you a snapshot of an instance of a database which has three attributes and you can look at the row having two entries the last two rows for employee code 519 and there are two different addresses in these two different rows. So, if we know that the employee will have a unique address or in other words if **employee ID →employee address** address then this situation is not possible.

So, but when we try to update then it is for example, the employees address has changed. And while making that change this change will need to be incorporated in all the records having the same ID. And if because of some coding error or something we miss out to

update any of the address fields then we will have a difficulty and that difficulty is having inconsistent address data as in this case.

So, this is known as update anomaly similarly I could have an insertion anomaly which I am illustrating here in terms of another database schema which has four attributes. And we have faculty ID, name, the hiring date and the course name naturally given the faculty ID the faculty name and hire date should be unique. Now suppose a new faculty joins and as soon as the faculty joins he or she may not have an assigned course.

So, if we want to enter that record here we will not be able to do that because we do not have any value for the course code. So, either we use a null value or we cannot actually enter this value; this kind of situation is known as an insertion anomaly. Similarly I could have a deletion anomaly in the in the same table we are showing that in the table the first highlighted row; the for faculty ID 389 if that faculty stops taking any course for the time being.

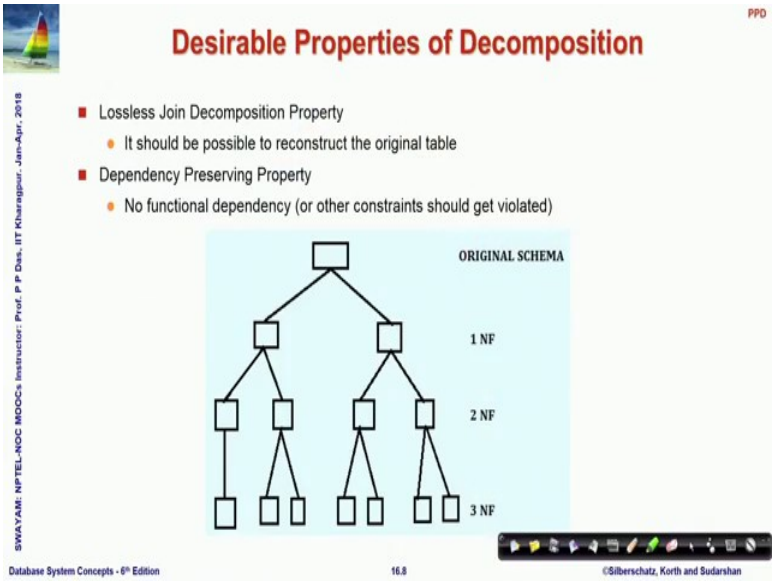
So, the association between 389 and the corresponding course code will be removed and once you remove that you remove this whole record in the process you actually lose the whole of the faculty information the ID, name and hire date. So, these are difficulties in these relational schemas and that lead to a whole lot of problems.

So, the resolution for this lie in terms of decomposing the schema that instead of having one relation, I will decompose this set of attributes into multiple different relations. So, for example, the update anomaly can be removed if we have two different tables; one that maintains ID with address and one that maintains ID with skill. So, in that case what will happen if the for every ID the address will not be repeated.

So, if the address is updated; it will be updated only at one place and it will not feature in the other table. Similarly, to avoid insert or delete anomaly the other table schema can be split into ID, name and hire date as one table and ID and code, rows code as another table. And you can you can easily understand that if this is split in this way then you cannot have an insert anomaly because you can insert a new faculty without assigning a course to him because that will feature in as a separate record in a different table similarly in the same way the deletion anomaly also disappears.

So, these anomalies are resultant of the redundant data that we are having and can be removed by taking care of the process of decomposition.

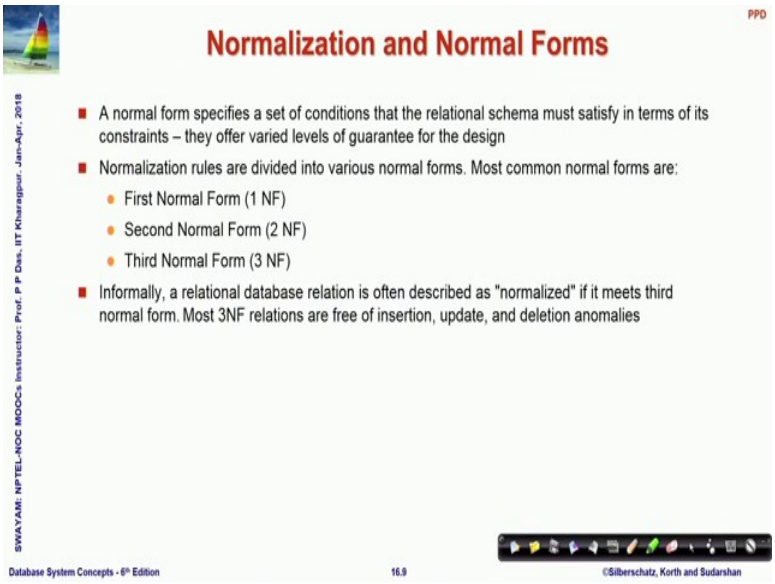
(Refer Slide Time: 07:03)



Now, when we decompose then we would desire certain properties to be hold held and we talked about this loosely earlier as well. We would require the lossless join decomposition property that it should be possible to take any instance of the two or more decomposed relations and join them by natural join using common set of attributes and get back the original instance of the relation if that does not happen then the relationship is lossy we have discussed it at length in the last module. At the same time we would want that all functional dependencies that hold must be; can must be testable in the decomposed set of relation.

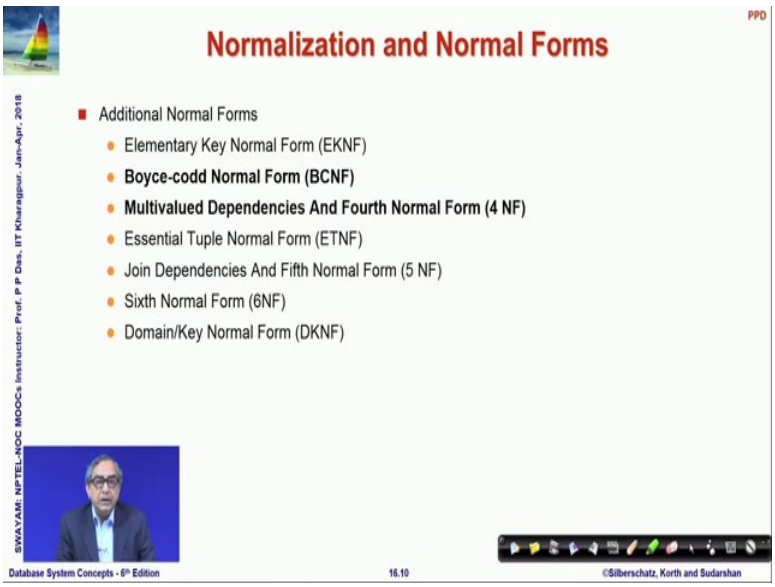
So, all functional dependencies when they are projected in terms of the decomposed set of relations; they must be testable within them. So, that to test for a dependency I do not need to carry out a join this is a point we discussed in the last module as well. So, based on that once you start with the original schema, you can check for what are the different possibilities or sources of redundancy define constraints based on that and step by step; you could convert a schema into a one normal form have more constraints put onto it convert it into two normal form have further constraints decompose it into third normal form and so, on.

(Refer Slide Time: 08:34)



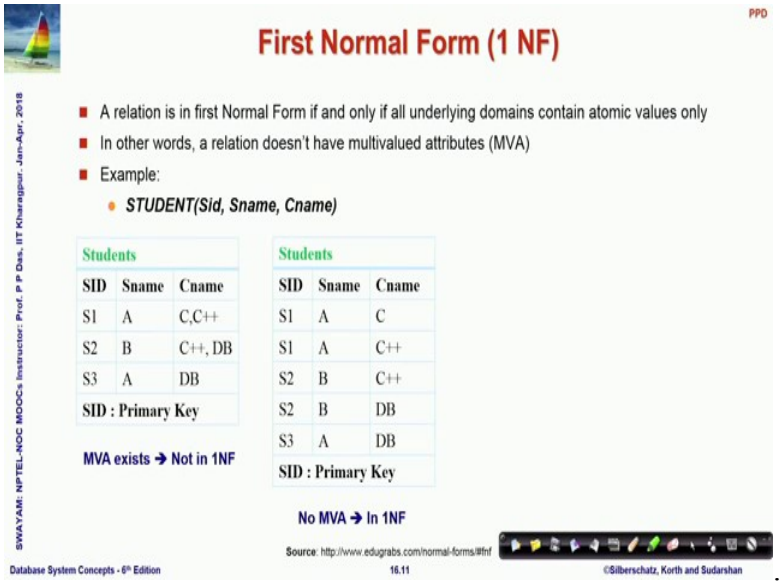
So, normalization is a process through which we do this kind of decomposition and make sure that once a relational schema is expressed in terms of a normal form; it satisfies a given set of properties that that normal form should adhere to. And the common normal forms are 1 NF, 2 NF and 3 NF and loosely speaking when we say if a database schema is normalized; we normal usually mean that it is in the 3 NF form a third normal form. And most third number form relations are free of insert, delete or update anomalies. So, that they are a good positive in the design.

(Refer Slide Time: 09:12)



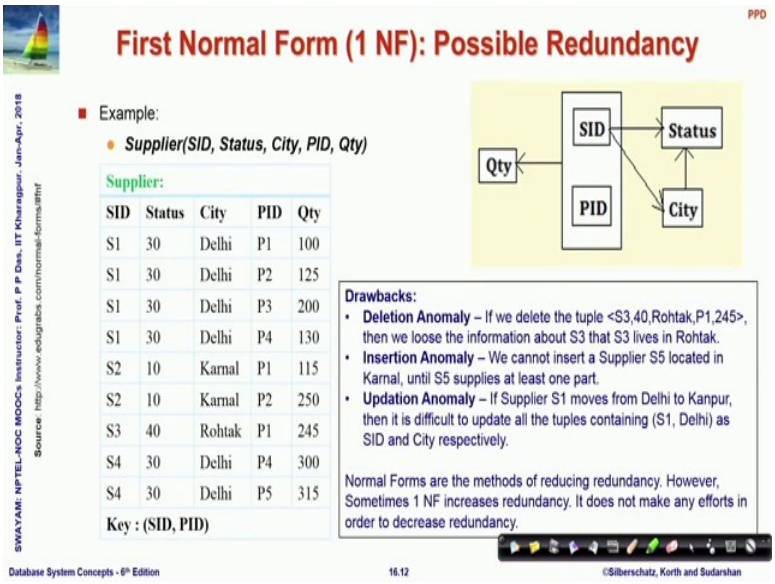
Of course, these are not the only normal forms as you can see there is a whole lot of lists of variety of normal forms; we will not study all of them we will study further in the next module the other two highlighted ones.

(Refer Slide Time: 09:26)



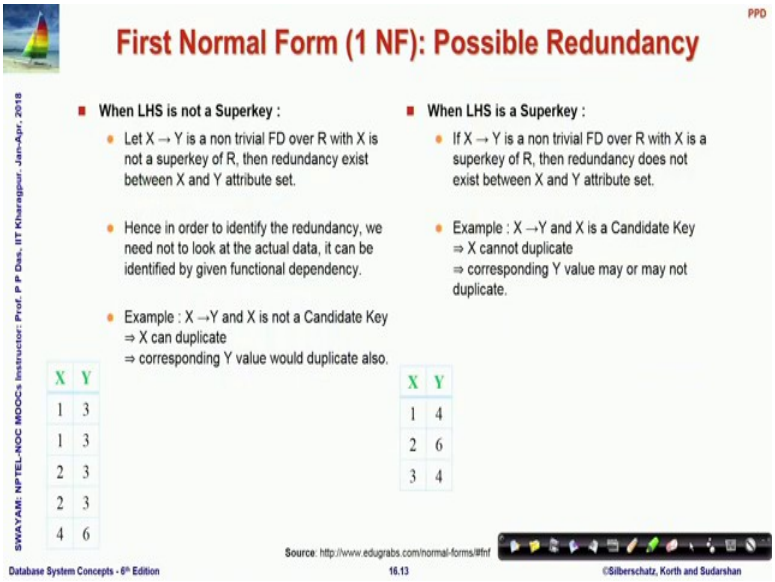
But first let us get started with the first normal form which we had talked about earlier as well; that first normal form is one where the multivalued attributes are not allowed. So, if you think about a think about a relationship where you have a student relationship between student her name and the courses taken by the student then since the students take multiple courses; the C name in this case can take multiple values. So, we do not allow that we expand them into different rows and that once we have done that we say that relation is in the one normal form.

(Refer Slide Time: 10:02)



But one normal form may give rise to a variety of different redundancies and therefore, anomalies. So, this is another instance; in fact, the earlier instances that you saw all of them were also in one normal form, but they had deletion insertion and update anomaly. So, here is another example where we are illustrating that.

(Refer Slide Time: 10:26)



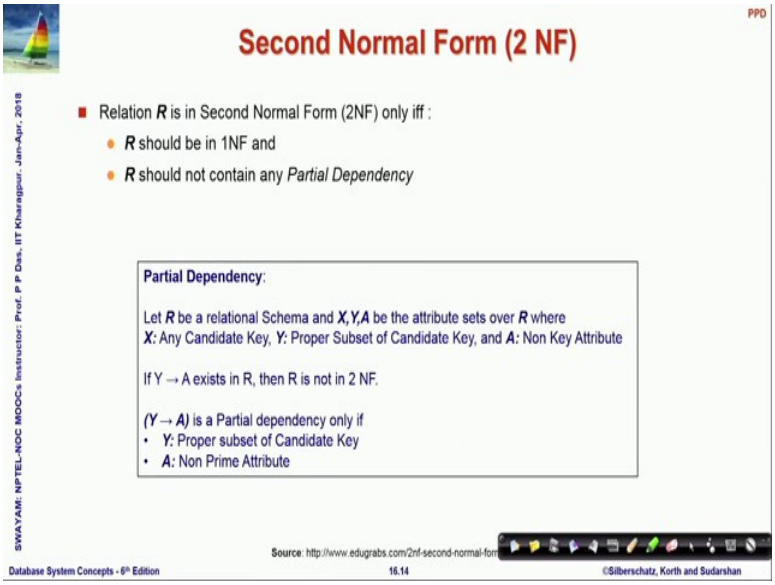
So, it is a possible that if I have a functional dependency $\mathbf{X \rightarrow Y}$ which is nontrivial functional dependency over the set of attributes and X is not a super key; then there exists a redundancy between X and Y attribute set. So, on the left the we have shown an

instance of this relationship on only on the X and Y attributes and you can see since X is not a key; I can have two rows having the value 1 in X.

And since the value is 1 in X; the value Y will be same for these two rows and we have redundancy of that please all. Please remember that X is not a super key; so, there are other attributes which actually form the super key and therefore, such instances are possible.

Whereas if you look at the right column where the left hand side X is a super key then such instances will not happen.

(Refer Slide Time: 11:22)

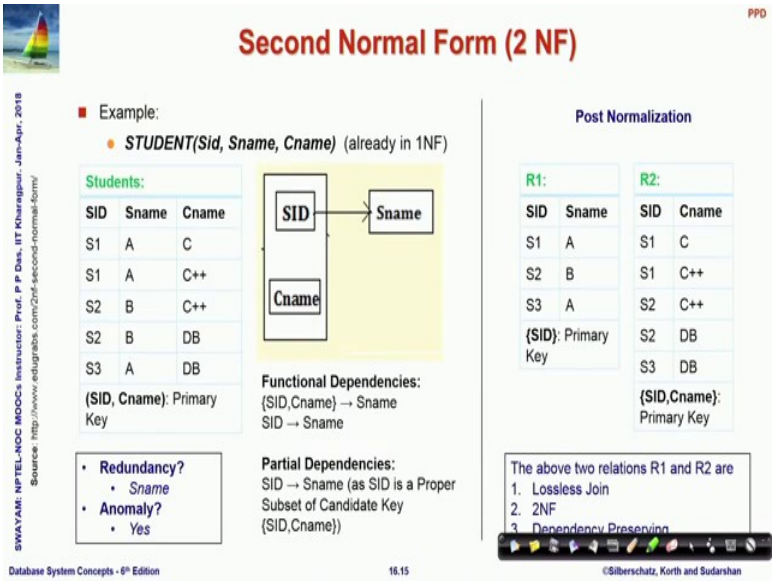


Moving on the second normal form which is obviously, a relation is in second normal form if it is in first normal form and it does not have any partial dependency. So, what is the partial dependency? I have given the definition here partial dependency why functionally determines A if that that can hold in the set of functional dependency then if I have that Y is a proper subset of a candidate key and A is a nonprime attribute in nonprime attribute is one which one nonprime attribute we defined in the last module is an attribute which does not feature in any of the candidate keys.

So, if Y is a proper subset of a candidate key which functionally determines a nonprime attribute; then this is known as a partial dependency and if there is partial dependency

then the relationship is not in second normal form. So, second normal form will require that the relation is in 1 NF and there is no partial dependency.

(Refer Slide Time: 12:25)

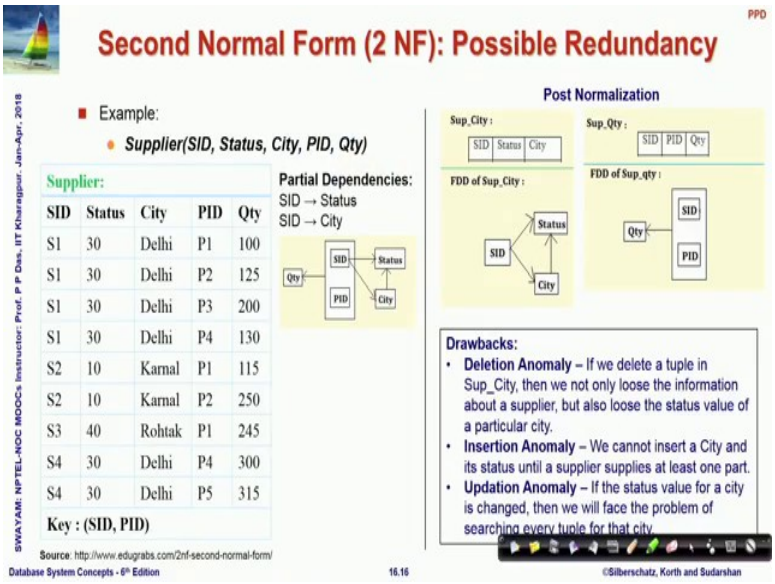


So, here I were showing an example where on the left you can see that SID and Cname together forms a key and **SID → Sname**. So, **(SID,Cname) → Sname** naturally **SID → Sname** is a partial dependency because the left hand side SID⊆ candidate key **{SID, Cname}**. And **Sname** is not featuring in any candidate key. So, Sname is actually a nonprime attribute and the result of that as you can see in the first two rows or in the third and fourth row you can see that Sname is repeated.

So, there is redundancy and therefore, consequently we will have anomalies that we have talked of, but we can normalize we can decompose this into two separate relations R1 and R2 as I am showing on the right; where you associate SID and Sname in one table and SID and Cname in other table. Naturally then the dependency that the partial dependency that you had disappears because **SID → Sname** in R1; now becomes is not a partial dependency because in that table SID becomes a primary key. So, it does not qualify as a partial dependency.

So, R1 and R2 both are in second normal form and you will get rid of the redundancy that you saw and this decomposition is ensures that it has a list lossless join incidentally; this is we have not guaranteed that it is in second normal form and it has also the dependency preservation.

(Refer Slide Time: 14:04)

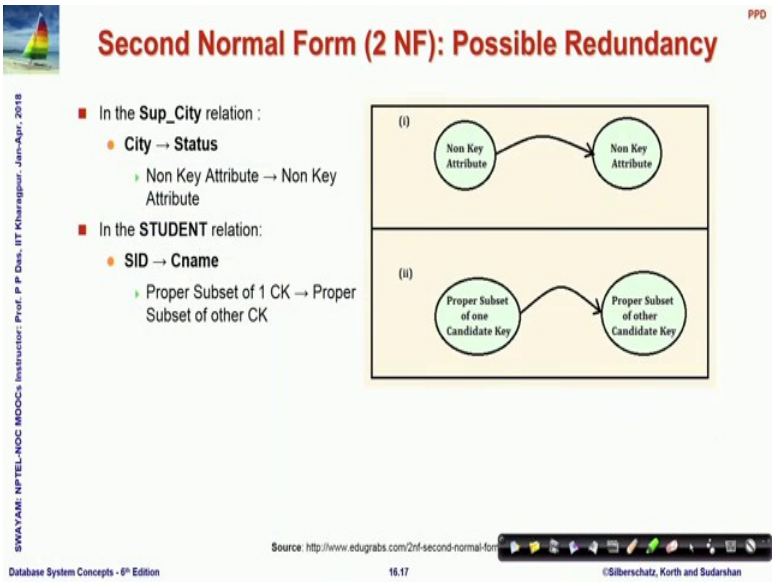


But it is possible again in second normal form a relation could be in second normal form yet it could have some possible redundancies. So, there is a design instance that I am showing with the supplier ID, SID the status key which are functionally determined by SID and the product and quantity values.

So, that in the table supplier SID and PID together form say key whereas, and as that happens you can clearly see that there is a lot of redundancy that you can see in terms of the status happening and which will cause you different anomalies to occur. So, if I normalize in the second normal form on the right then I will have a supplier city say with the three attributes SID, status and city and another supplier quantity which has SID, PID and quantity naturally in this there is no partial dependency anymore.

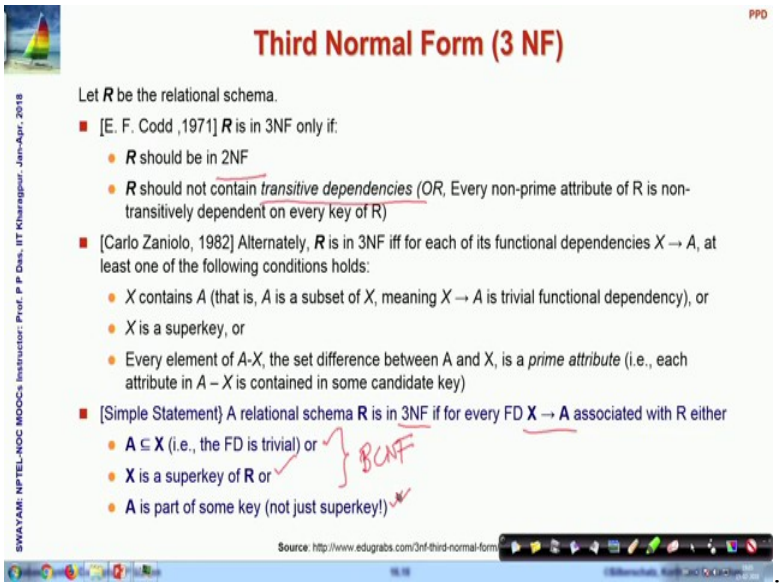
Earlier we had **SID → status** as a partial dependency because SID is a proper was a proper subset of the primary key which is SID CID, but after I normalize this dependency does not exist, but yet there will be redundancy in this relationship and there the status will continue to be redundant.

(Refer Slide Time: 15:30)



And for that reason we have to move on to the next type of normal form. So, this I am just explaining here as to what are the possible redundancy sources of possible redundancy that you can have in 2 NF.

(Refer Slide Time: 15:43)



In the 3 NF; third normal form what you define is your relation first of all has to be in 2 NF. So, we are looking at the first definition these are there are three forms of definitions given all of them are actually equivalent, you do not have to worry about why and how they are equivalent slowly you will start understanding.

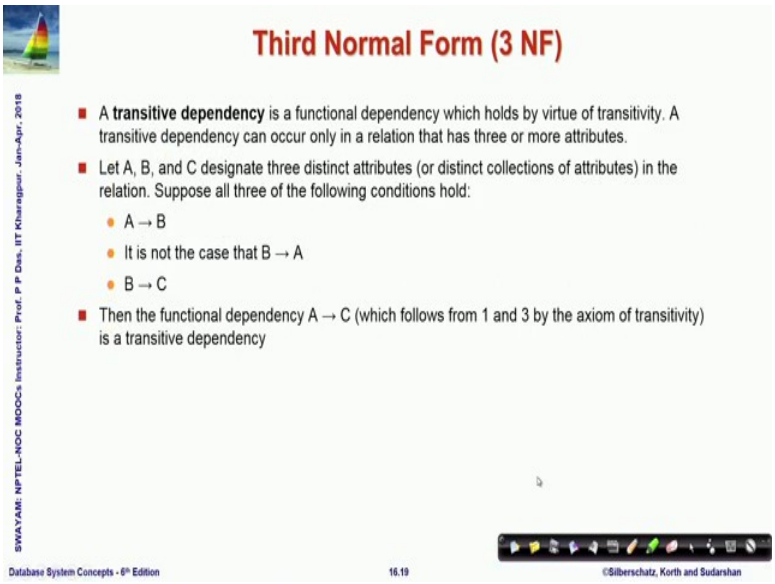
But we take it in three different forms because each form of the definition allow us to understand certain aspect of the three normal form. So, the first thing which is true for everything is it has to be in the second normal form and it should not contain any transitive dependency which means that I should not if I have **X → Y** and **Y → Z**; then I should not have **X → Z** which can be inferred transitively as you know through the angstrom axiom.

Alternately,there was an alternate definition given later on by Zanilo and I have stated a simpler simplified version of that at the bottom. So, we will say that a relational schema is in 3 NF if for every functional dependency **X → A** that holds on this schema either it is a trivial dependency which is A is a subset of X or X is a super key.

So, this is kind of the condition also as you had seen earlier this also is a condition to be in Boyce Codd normal form. So, you can easily understand the 3 NF is a any relation which is in 3 NF is also in the Boyce Codd normal form, but we add a fourth third condition where you say that we will say this is in 3 NF; even if the first two conditions are not satisfied, but a is a part of some key just note the wording is a part of some key not just the super key..

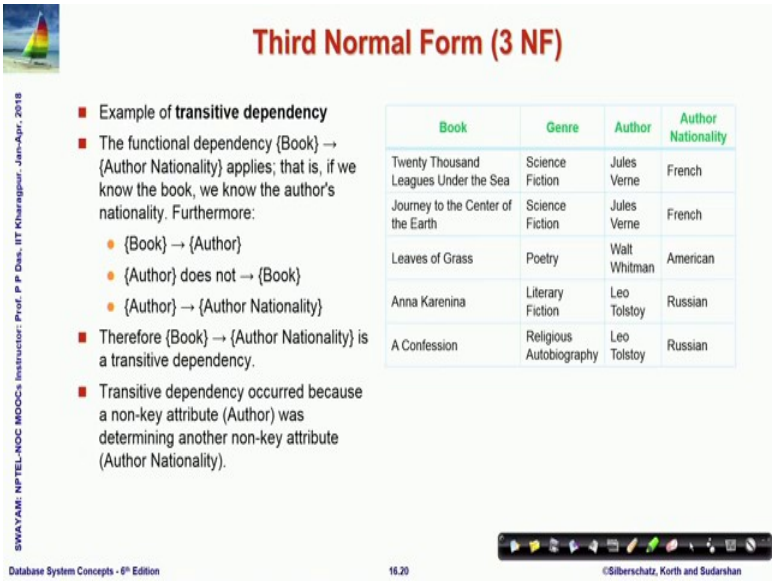
So, if A is a part of some key then and the first two conditions are also not are not satisfied even then we will say that the relation is in third normal form. So, to check for a relation to be in third normal form; we will actually check for whether any one of the three conditions hold.

(Refer Slide Time: 18:00)



So, this is a definition of transitive dependency which I have just loosely told you. So, I will skip over this.

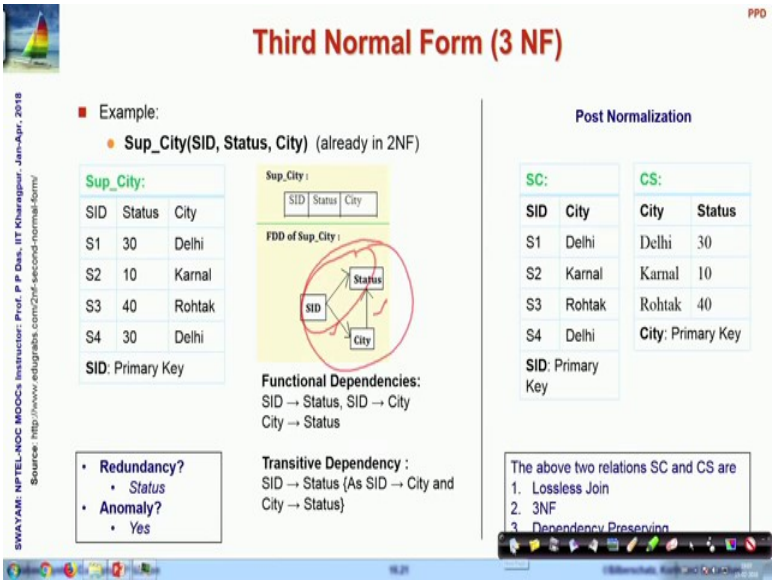
(Refer Slide Time: 18:09)



There is given another example of a very different kind of a relationship book, genre author and author nationality as you can understand. Given the book you know the author there is a functional dependency given the author do you know the author nationality and the, but author does not actually determine the book because the author may have written multiple books. But given that **book → author** and **author → author**

nationality we have that **book → author nationality** and therefore, we have redundancy possibility of redundancy in here which is a transitive redundancy due to this transitive dependency that we have.

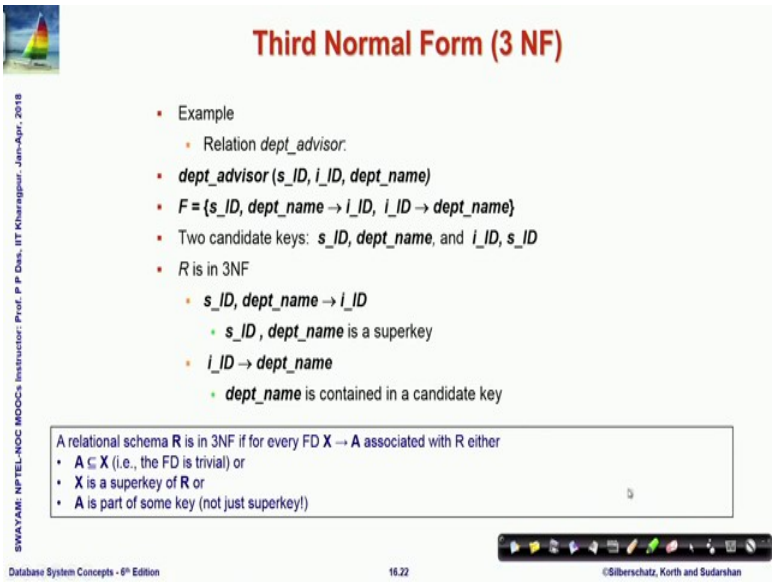
(Refer Slide Time: 18:48)



So, here is a the earlier example where you can as you can see clearly in this diagram you can if you note this diagram you can see that **SID → city** and **city → status**. So, this is it this is the transitive dependency that **SID → status**.

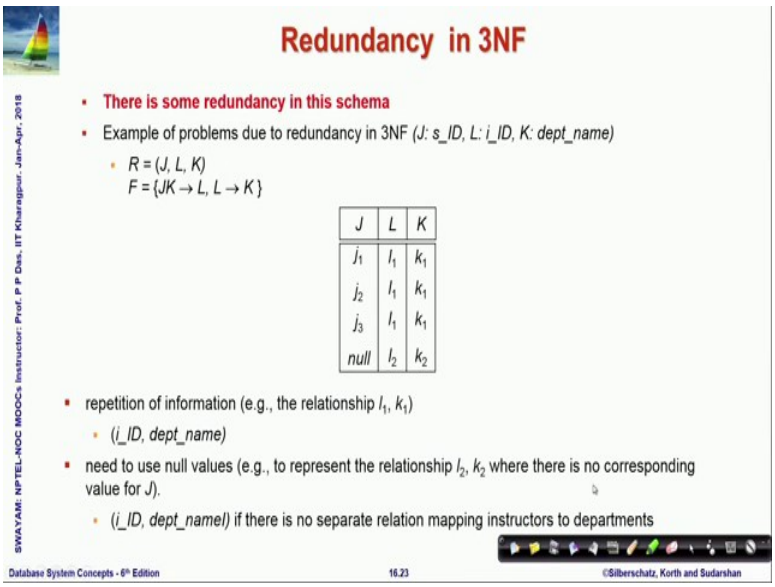
So, if that happens and status becomes redundant and therefore, there could be anomalies. And we can easily normalize by making them into SID and city and city and status. And in that naturally that that redundancy goes away because you have no more the transitive dependency in the relationship; you only have **SID → the city** which is a primary key in SC and **city → status** which is the primary key in the CS.

(Refer Slide Time: 19:50)



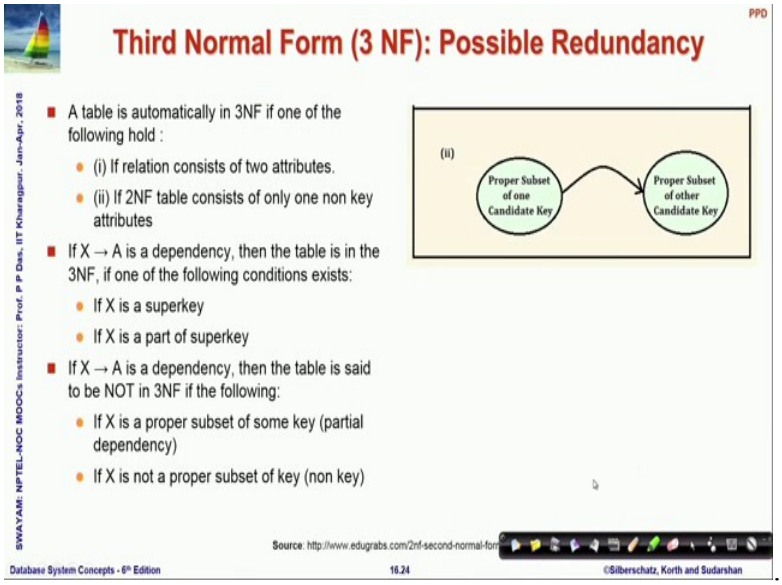
So, there are these are other examples that that you can go through where we have I have taken the example of a student ID , i_ID and the department name and shown that what kind of problems, you might get into in this. In this case you can see that the relationship actually is in the there because there are two candidate keys and. So, this s_ID department name is a super key and this relationship is in the third normal form. Because **i_ID → department name** is contained in a candidate key. So, that is the it is a it is in 3 NF due to the third condition that we have had shown.

(Refer Slide Time: 20:41)



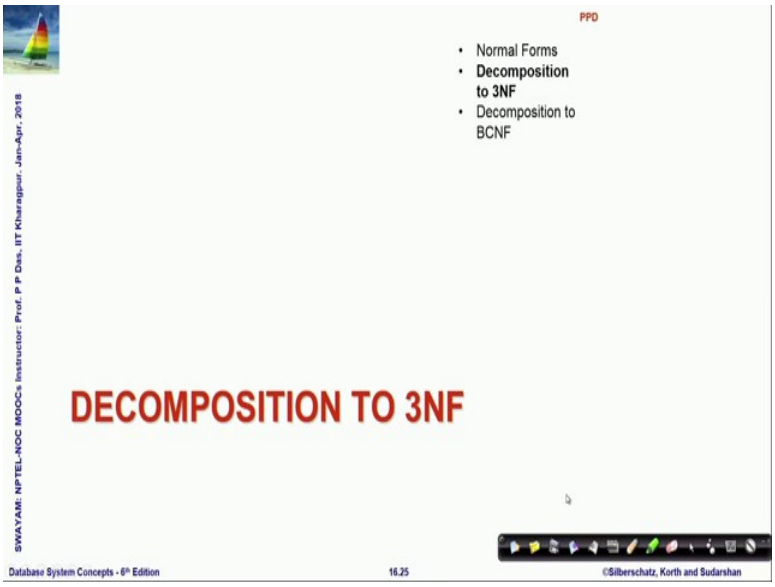
So, when you, but this is a where you can there is some redundancy in this schema that you can observe. So, this is just constructed and you have been because of this redundancy you have been able to we have had to use null values in this case.

(Refer Slide Time: 21:02)



So, in a third normal form there is possible redundancy coming in and these are the different cases that we have to check through.

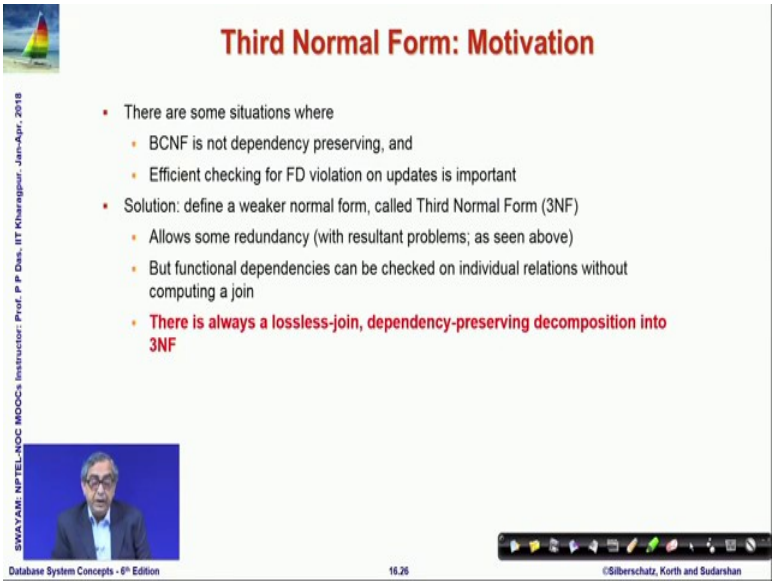
(Refer Slide Time: 21:13)



So, next what ; so, we have seen the different normal forms first normal form no multivalued attribute then the second normal form no partial dependency then the third normal form where you do not have any transitive dependency. So, all these are cascading definitions. So, in third normal form you have no multivalued attribute, no partial dependency and no transitive dependency.

So, now what will take a look into is how if I am given a relational schema and if it is violating any one or more of this condition. So, that the schema is not in the three normal form, third normal form then how can we decompose it into the third normal form?

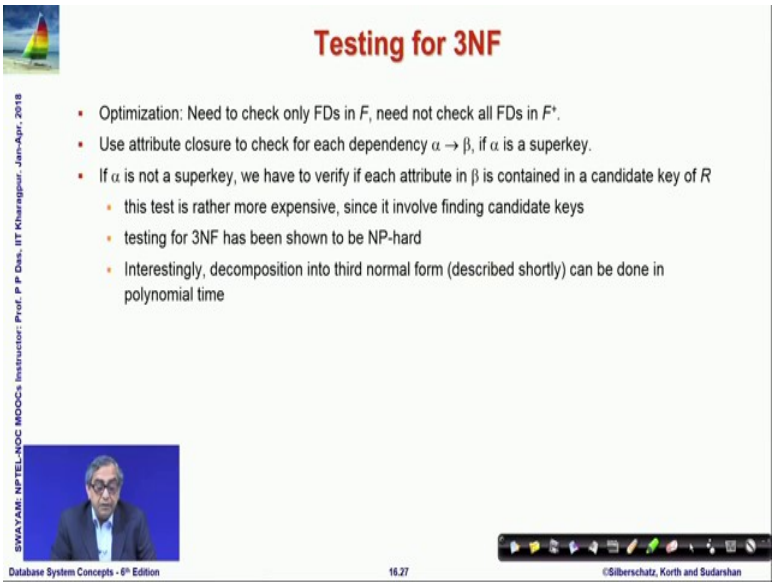
(Refer Slide Time: 21:55)



So, the question naturally is certainly is can it always be done is the basic question that can I always decompose a schema into third normal form the answer is yes, you can and that is always a lossless join and dependency preserving decomposition into third normal form which is of great value.

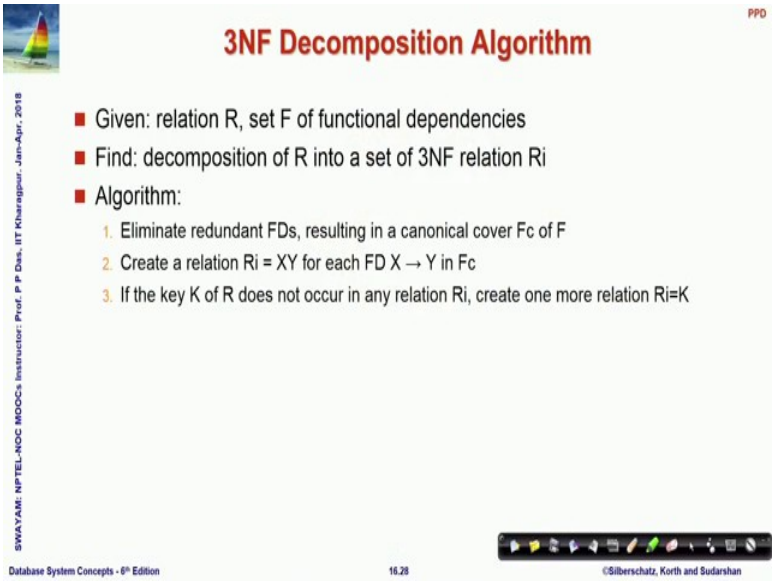
Because that is we said is that desirable properties of our decomposition and if you recall our discussions in the earlier part of the relational design modules, then you would recall that Boyce Codd normal form also we had discussed at the early stages. And that gives you a decomposition which is lossless join, but it does not guarantee preservation of the dependencies where third normal form does that.

(Refer Slide Time: 22:49)



So, naturally there are different algorithms first the question is can you test if a relationship is in third normal form; I will not go into the details of that and the computer science result here is testing for third normal form is an NP hard problem. So, there is no known polynomial time algorithm for that, but the interesting thing is the actually that decomposition can be done in very simply in polynomial time.

(Refer Slide Time: 23:17)

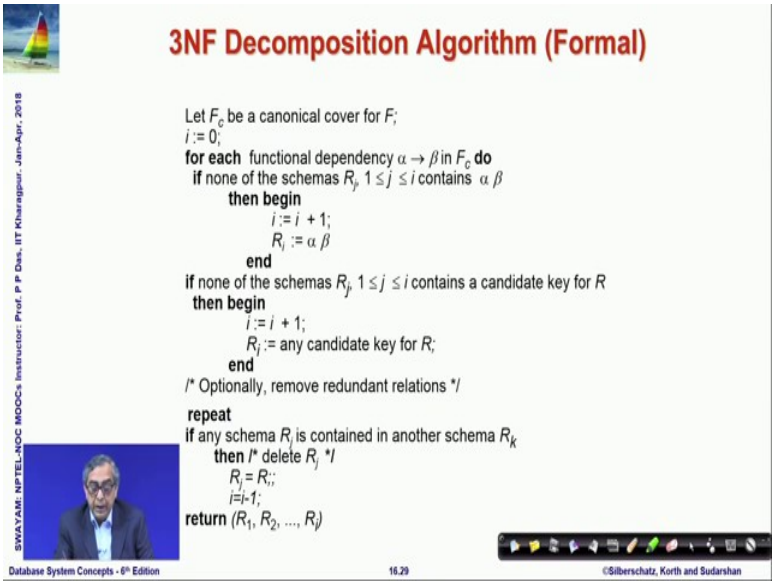


So, what do you have what is the decomposition algorithm very written in very simple terms you want to you have given a relation R and a set of functional dependencies that

hold on you. So, you first compute a canonical cover you know what is a canonical cover. So, you compute a canonical cover you eliminate extraneous attributes eliminate redundant FDs and you have the canonical cover Fc from F then you create for every functional dependency **X** → **Y** that exists in the canonical cover.

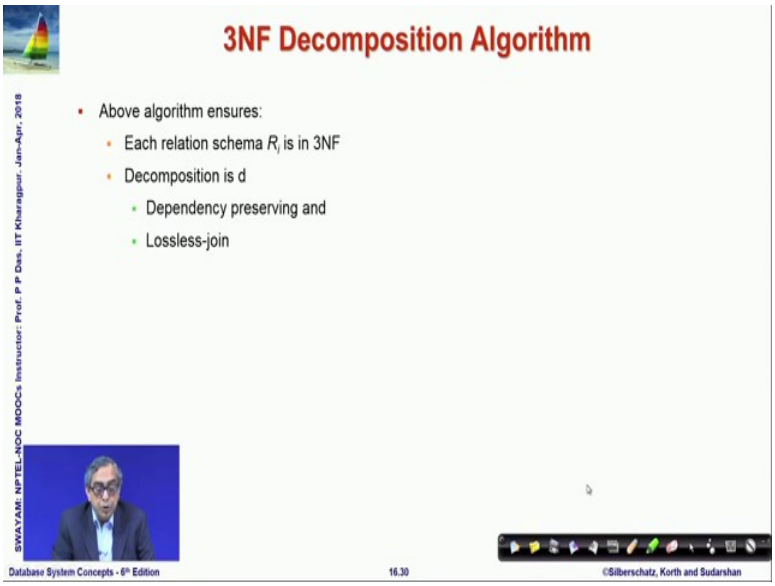
You compute you make a relation say the i^{th} relation taking **X U Y**. So, you call it the relation XY and you do that for all the functional dependencies in the cover. And after that if you find that the key does not occur in any one of these decomposed relations as generated, then you generate one separate relation to represent the key.

(Refer Slide Time: 24:19)



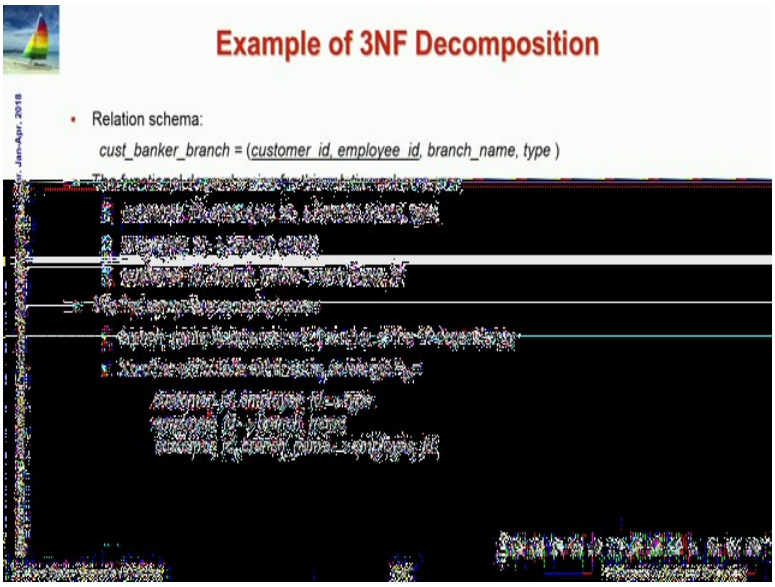
That is a very simple algorithm and I just wrote it in simple hand. So, that you can understand it easily, but here is the formal algorithm. So, if you are interested to rigor I mean in the in the rigor of how 3 NF decomposition will happen here is the algorithm, but I will not go through these in steps.

(Refer Slide Time: 24:37)



So, that ensures that each relation R_i that I have decomposed and generated is actually in third normal form and this decomposition is dependency preserving and is lossless join we are not proving that but we are just using that result.

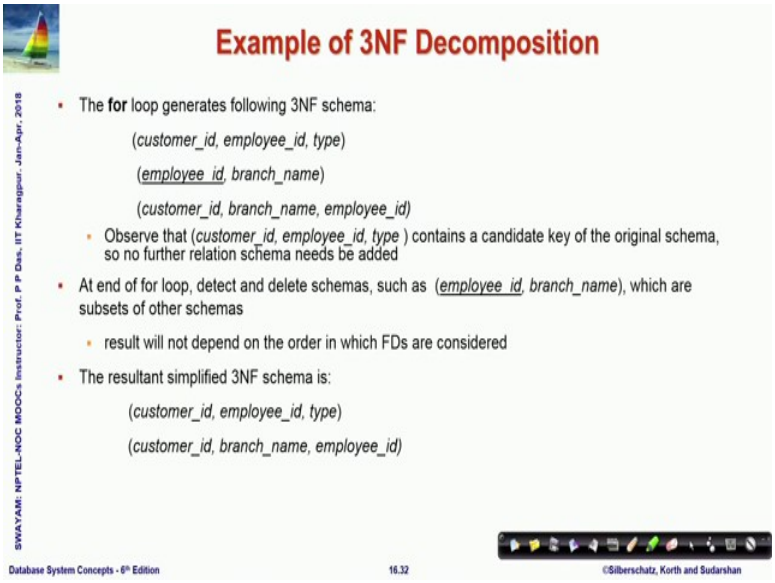
(Refer Slide Time: 24:54)



So, here is an example of a schema; so, we have a customer_banker_branch. So, these are the four attributes and these are the different functional dependencies that exist. Now naturally given this first thing you will have to do is first thing you have to do is to look at the different to look at taking the canonical cover the minimal cover.

So, if you compute try to compute the minimal cover; you will find that branch name actually is extraneous in the first dependency. So, you can remove that and there is nothing else.

(Refer Slide Time: 25:36)



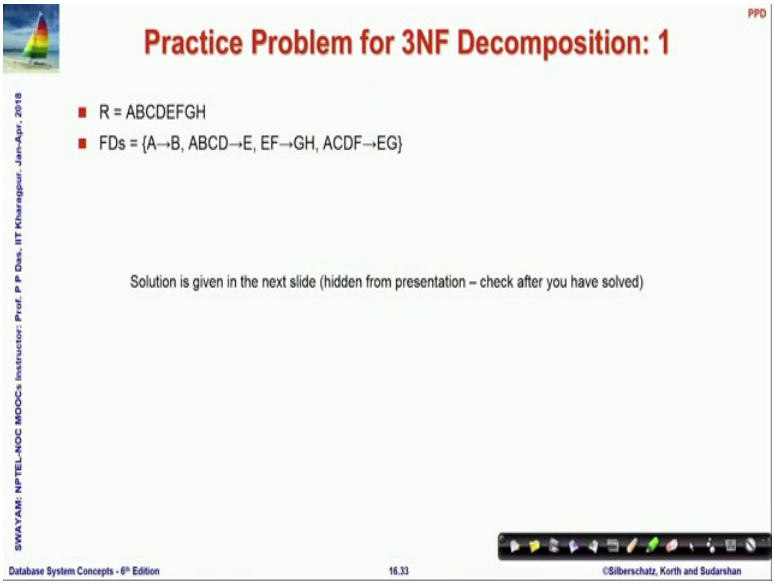
So, your canonical cover turns out to be this set of dependencies and then you go over and for each one of them. So, you take each one the first one is **(customer ID, employee ID) → type**. So, for that you generate a schema customer ID, employee ID and type again you take the second functional dependency **employee ID → branch name**. So, create employee ID and branch name as a different schema and in this way you will generate three decomposed schema in the third normal form.

Now, once you have done that then you find that your if you look into the original key it was customer ID and employee ID and you find that here in the third; second and the third you already have that. So, you do not need to add a separate relation for accommodating the key and also the third relation. So, we can now declare that no further key needs to be added and we have the final 3 NF decomposition..

So, at the end of the fault detect and delete. So, this is this is a stated in terms of the detailed algorithm, but this is you can say that the employee ID and branch name the second relation in the decomposition is actually a subset of the third relation. So, you can remove that as well. So, you will be left with only two relations in this decompose

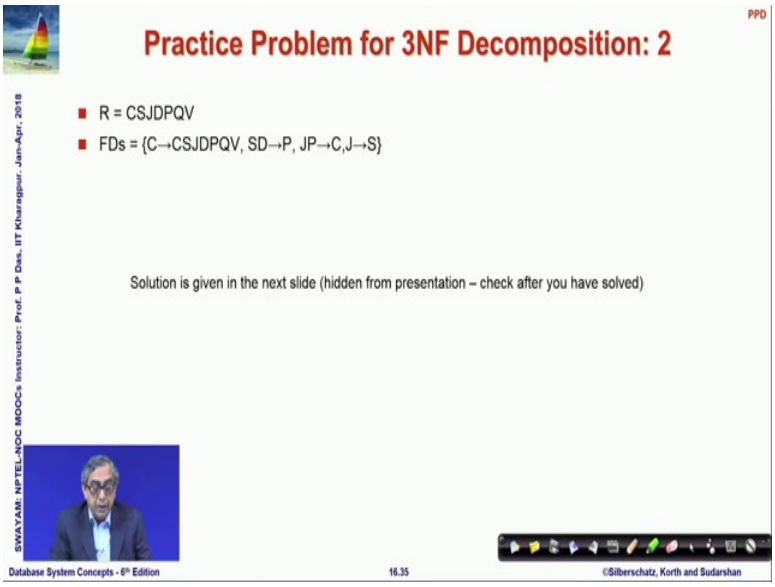
schema which both of which are in third normal form and this decomposition is guaranteed you lossless join and dependency preservation.

(Refer Slide Time: 27:17)



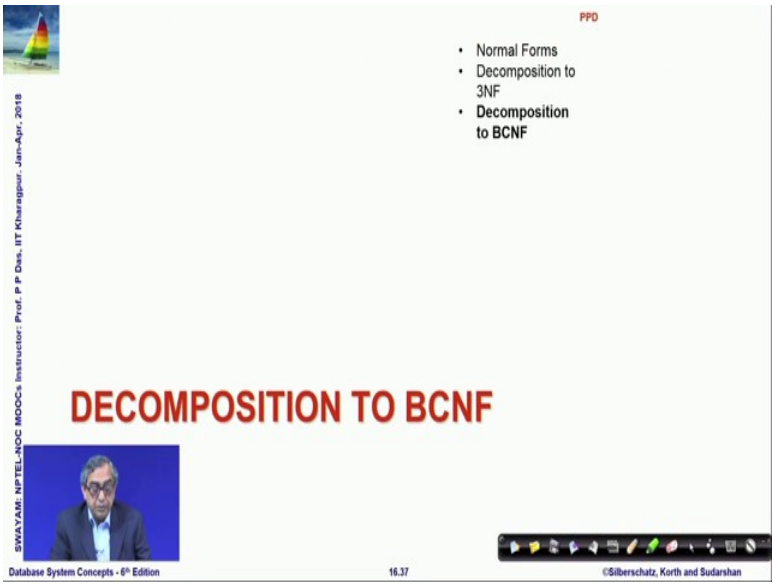
So, I have given some practice problems for you I have also given the solution, but the solution is not in the current run of the presentation; you will get see them in the presentation as hidden slides. So, you first try solving them and once you have solved them then you look at the solution in the slide.

(Refer Slide Time: 27:37)



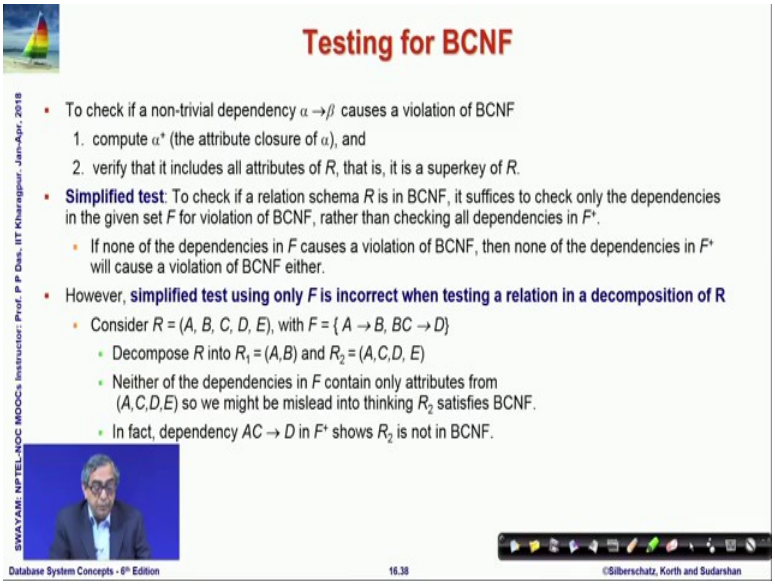
So, there are two problems; so, this is a second one and you can solve them in that way.

(Refer Slide Time: 27:40)



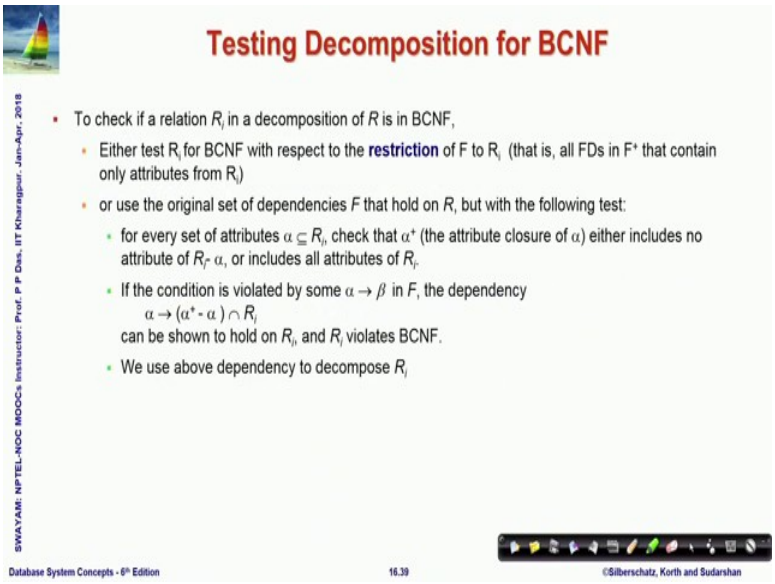
Next is the we will quickly recap on the decomposition of BCNF Boyce Codd normal form which we had seen earlier.

(Refer Slide Time: 27:49)



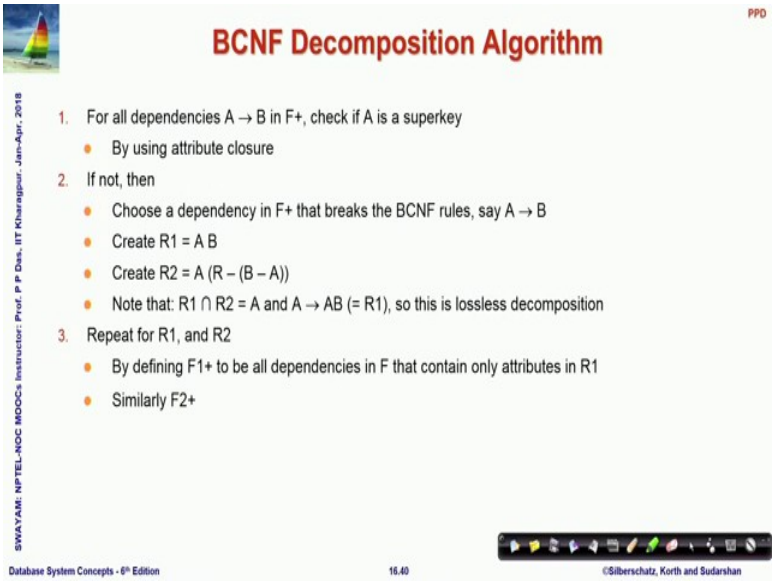
And we know that the Boyce Codd normal form guarantees that there will have be every dependency that exists must be either trivial or the left hand side must be a super key. So, using the algorithms, you can test for the Boyce Codd normal form which is described here I am not going through in steps.

(Refer Slide Time: 28:08)



And here is the more detailed formal algorithm to find determine whether a Boyce Codd normal form is in a decomposed form is in Boyce Codd.

(Refer Slide Time: 28:18)

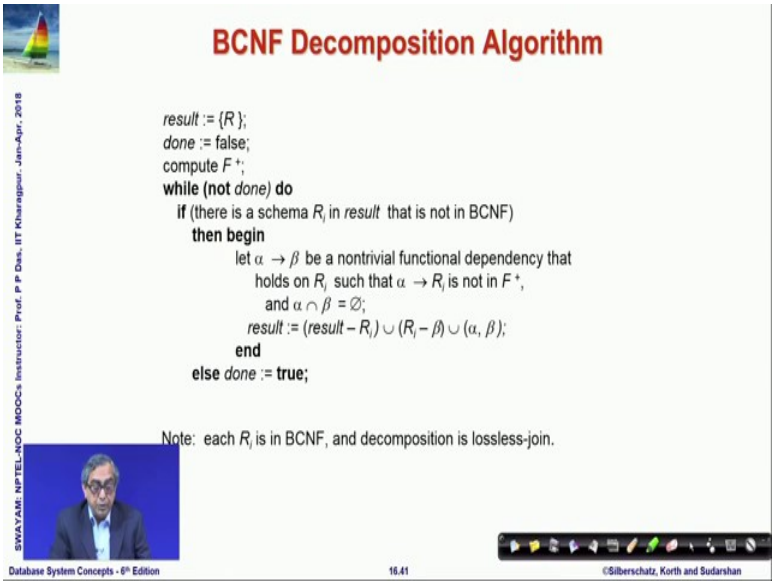


So, I will just quickly recap on the algorithm to do that naturally for all dependencies you first determine the super key and check if **A → B** is a super key or not if it and that you can easily do using attribute cover. If it is not a super key then you choose a dependency **A → B** which violates and you form by Boyce Codd goes in every step it decomposes one relation into two separate relation.

So, one that you take by taking U of the attributes of A and B and the other where you take out B minus A; these attributes this difference attributes you take out from R and then you add A and make the other relationship. Naturally in between these two A is a common attribute and since and that will determine A B because **A → B**.

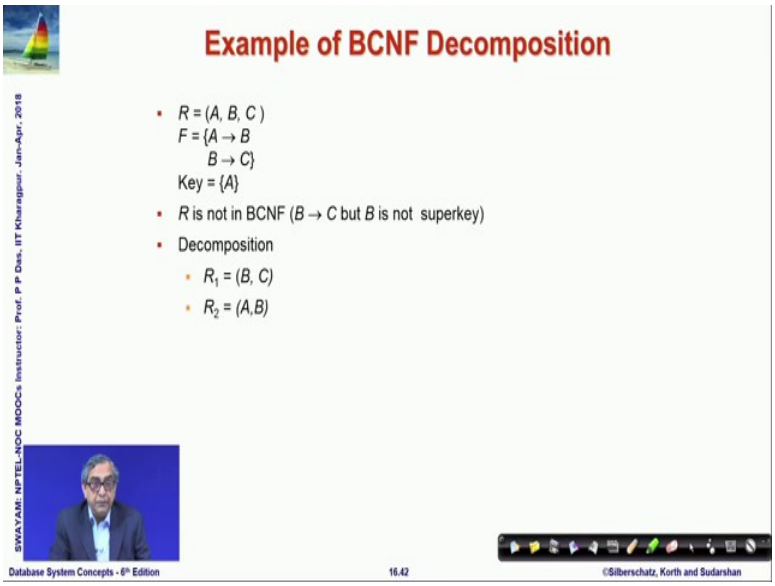
So, **A →A B** that is whole of R1. So, naturally the lossless join is guaranteed and you repeat that keep on doing that for the resultant relations that you have got. keep on decomposing them till you finally, close and you have no more violating dependency and you will have a decomposition into Boyce Codd normal form.

(Refer Slide Time: 29:39)



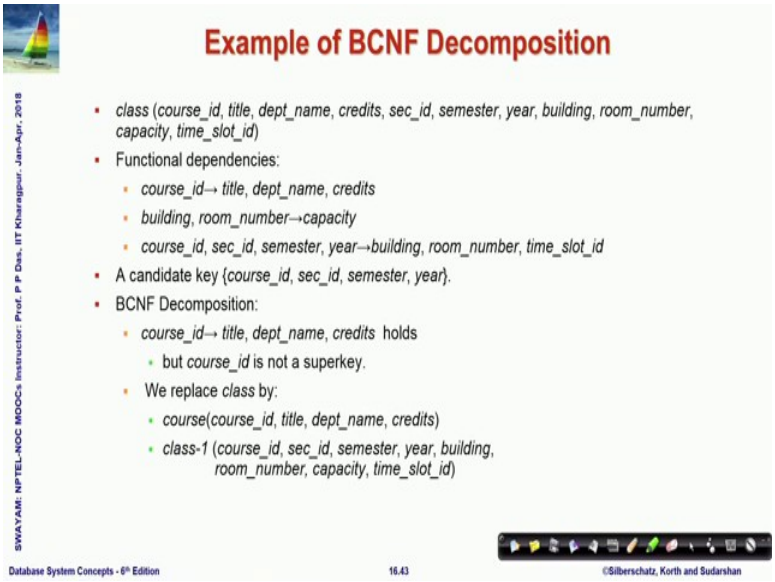
Here is the formal algorithm again for you to go by steps if you are interested.

(Refer Slide Time: 29:46)



Otherwise you know how to do this; again I have shown another example here which is showing that how to decompose in BCNF. So, you should practice this that is why I have work them out in steps here. So, here **A → B**; **B → C** naturally A is the key; R is not in BCNF because **B → C** is a functional dependency where B is not a super key.

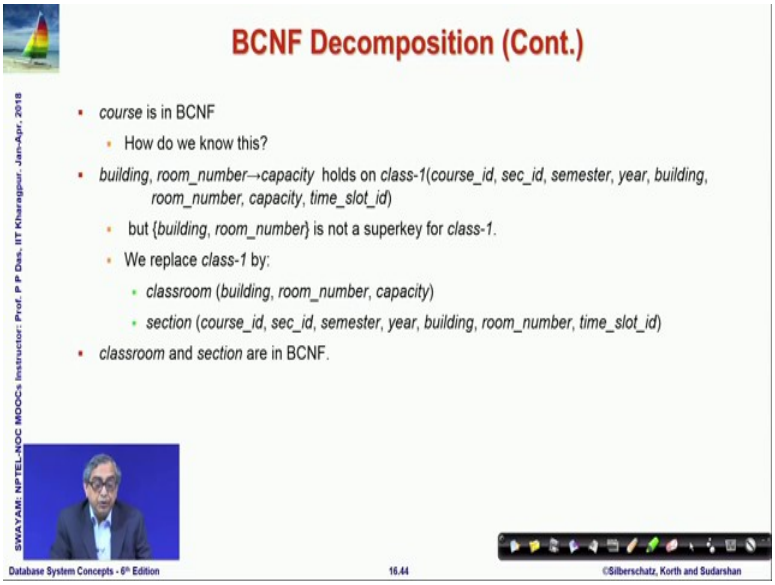
(Refer Slide Time: 30:15)



So, you can decompose them in terms of. So, you can decompose in terms of BC as one relation and AB as another relation. Here is another example a more detailed one of a class relationship which has a whole set of attributes and these functional dependencies

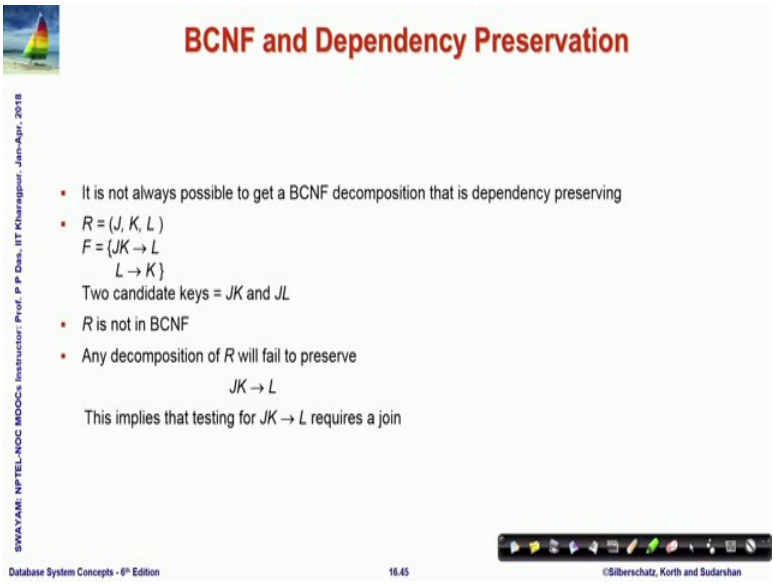
and based on that the candidate key is course ID, section ID, semester and year and you can proceed with the BCNF decomposition; taking the first functional dependency that holds, but the left hand side the course ID is not a super key. So, you will replace it by a one relation; which is say new course relation and a new class relation which is the remaining attributes.

(Refer Slide Time: 31:06)



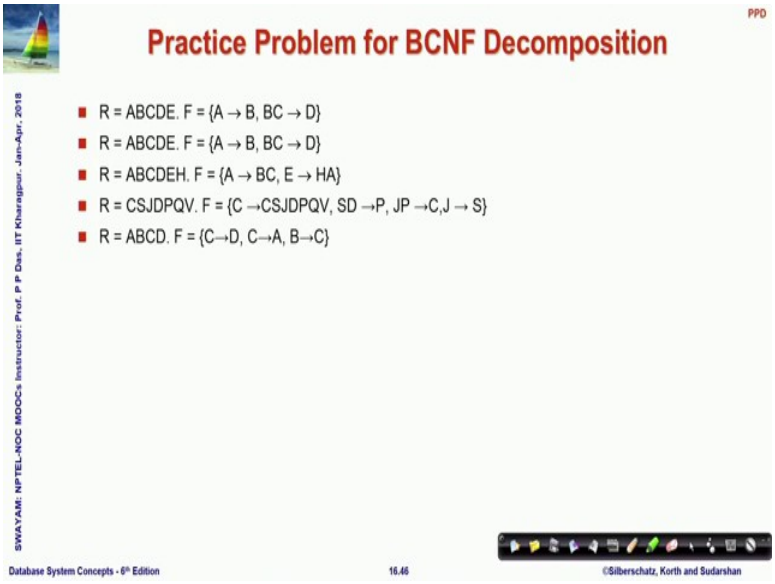
And then you get convinced that course is in BCNF, but the other one the class is not because **(building,room number)→ capacity** where building room number together is not a super key. So, you split it again and you replace class 1 in terms of two new relations class room and section and both of them are in BCNF and you are done with this.

(Refer Slide Time: 31:31)



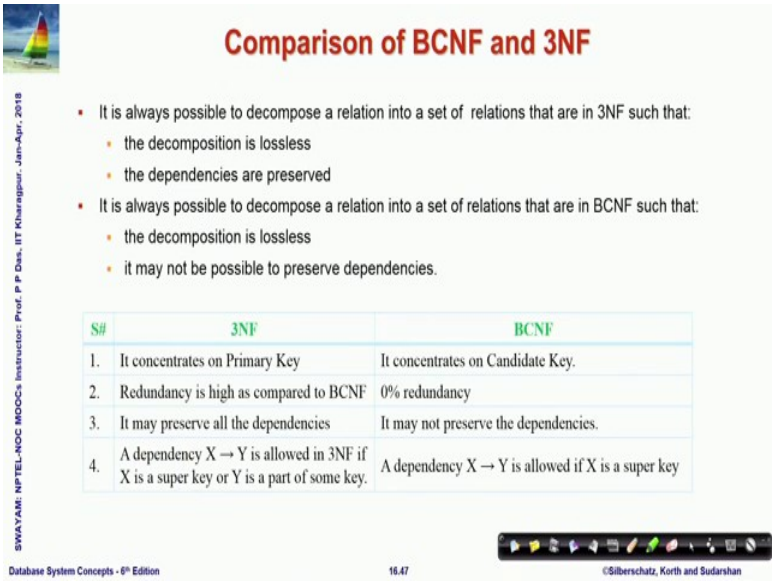
But BCNF as I would again warning you BCNF does not preserve dependence it gives you lossless join, but it does not preserve the dependencies. So, it is not always possible to decompose into BCNF with dependency preservation. So, here is an example which we saw little earlier and there are two candidate keys R is not in BCNF, you can clearly see and any decomposition will fail **JK → L** and that will require a join. So, this will not preserve the dependencies in terms of the decomposition.

(Refer Slide Time: 32:06)



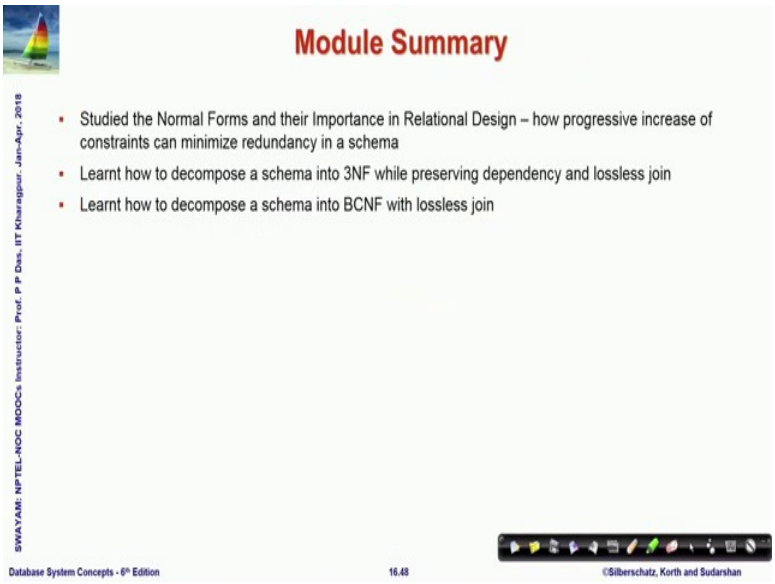
Again, I have given a set of practice problems here which we you should try and get confident in terms of the Boyce Codd from normal form normalization.

(Refer Slide Time: 32:19)



Now, it is always possible to decompose a relation into a set of relation in 3 NF; if the decomposition is lossless and the dependencies are preserved. Whereas, in case of BCNF it is not possible; so, here is a table which summarizes the relative comparison between Boyce Codd and third normal form because they are the common once Boyce Codd naturally is more strict it gives you lesser dependent lesser redundancies, but it cannot guarantee that your dependencies will be preserved. So, more often we will accept 3 NF as an acceptable normalized decomposition with some redundancy still existing it is possible and we cannot get rid of them.

(Refer Slide Time: 33:09)



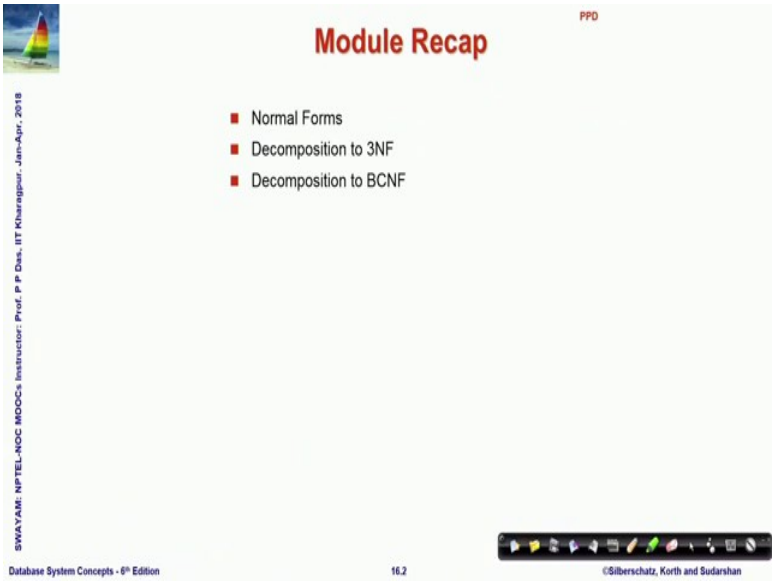
So, we have studied about the normal forms and their importance and how progressively we can increase the constraints to minimize redundancy in the schema and learned how to decompose a schema into third normal form and also in the Boyce Codd normal form.

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture – 20
Relational Database Design (Contd.)

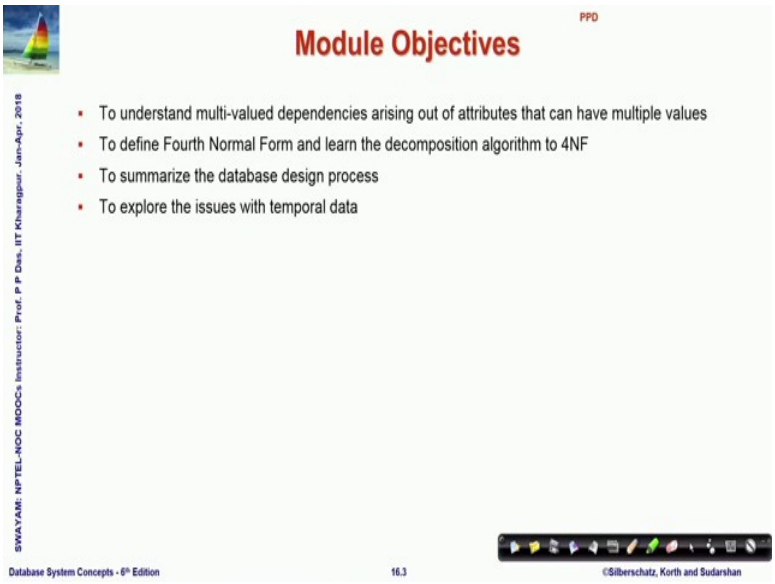
Welcome to module 20 of Database Management Systems. We have been discussing about relational database design, since the last 4 modules and this will be the concluding part of relational database design.

(Refer Slide Time: 00:33)



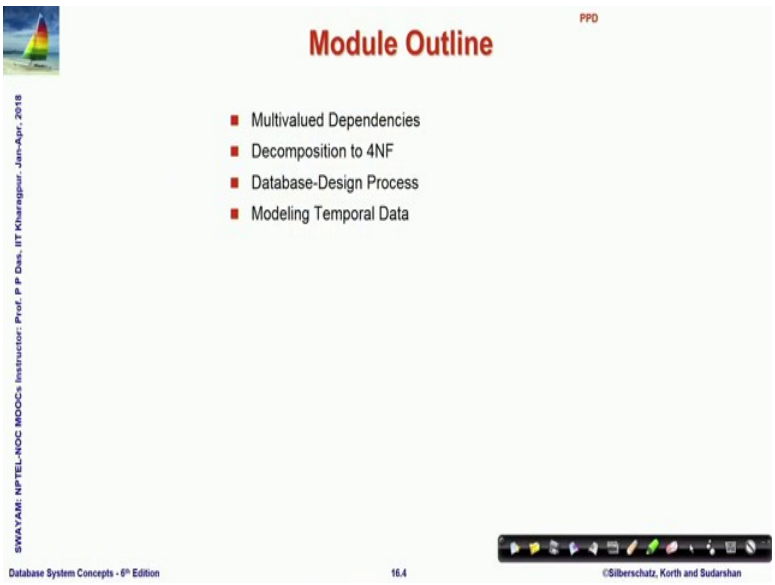
In the last module, we have seen some very key concepts of relational design, that of normal forms Third and Boyce Codd normal form specifically and how to decompose into them? And how do we get benefit in terms of doing this kind of decomposition? In removing the anomalies by reducing the redundancy in the design.

(Refer Slide Time: 00:59)

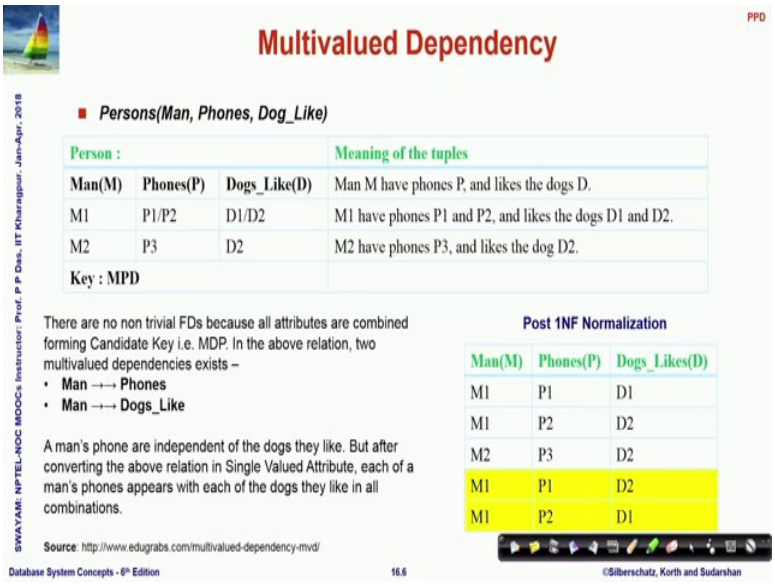


In view of that in the background of that, in this module will we would try to understand a new kind of data dependency, an additional kind of data dependency, which is called multivalued dependency. Which can occur when an attribute can have can take multiple possible values, which we had eliminated in the first normal form together and based on that, we will define 4th normal form and decomposition into 4 NF and then we will summarize this whole set of discussions of relational database design, and talk little bit about what happens, when you have temporal data in your system.

(Refer Slide Time: 01:39)



(Refer Slide Time: 01:47)



So, these are the outline points, based on our objectives and we start with the discussion of multivalued dependency. Consider a situation like this. So, here we are trying to represent an individual instead of persons with three attributes, man which is an may be id or name of that person, phones and dog like. So, the idea is that the here persons and they can have one, two, three any number of phones, which is true for all of us and then a person may have any number of dogs that he or she likes.

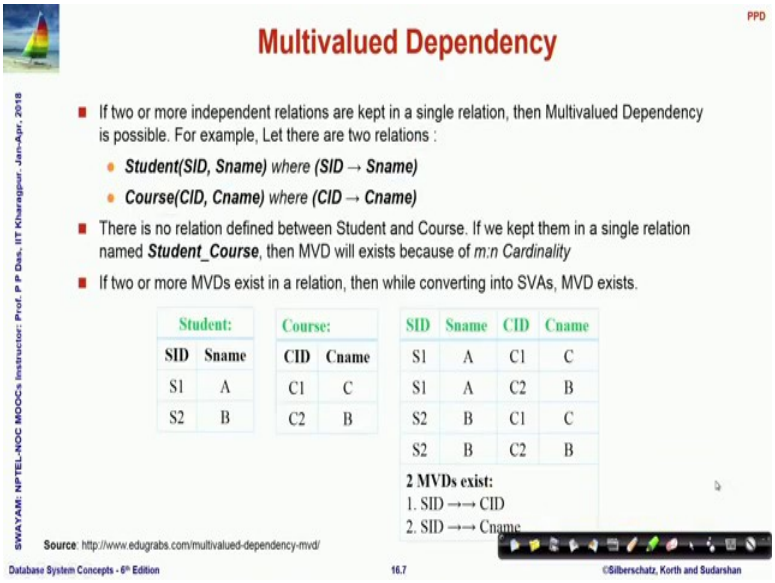
So, both of these phones and dog like D, P and D attributes can take multiple values and ah. So, if we if we if you look at the 1 NF normalized form here. So, in 1 NF what we do? We create separate rows for them. So, we have created separate rows for M 1 against P 1 and P 1 or P 2 in phones and similarly for D 1 and D 2. So, once we have done that then we have here, we can see I have highlighted with yellow, you can see the different redundancies that are arising.

Because, since I have phones and dog liking attributes. So, it is possible that if phone takes 2 values and dog like takes 2 values, then actually 4 different combinations of them are possible. But in reality, it may be in reality it may be actually these 2 are true, that M 1 has phone P 1 and likes dog D 1 M 1 has phone P 2 and likes dog D 2, but you could also have such redundant tuples coming in, because there they are now valid.

So, this is the situation which we try to try to capture, in terms of what you see here multivalued dependencies, where which is row shown in terms of double arrows as you

can see here. So, **man →→ phones**, **man →→ dog likes**. So, there are two different multi valued dependencies in this case. So, this multi valued dependency adds a new source of redundancy in our data, and that is very real in various models of our system.

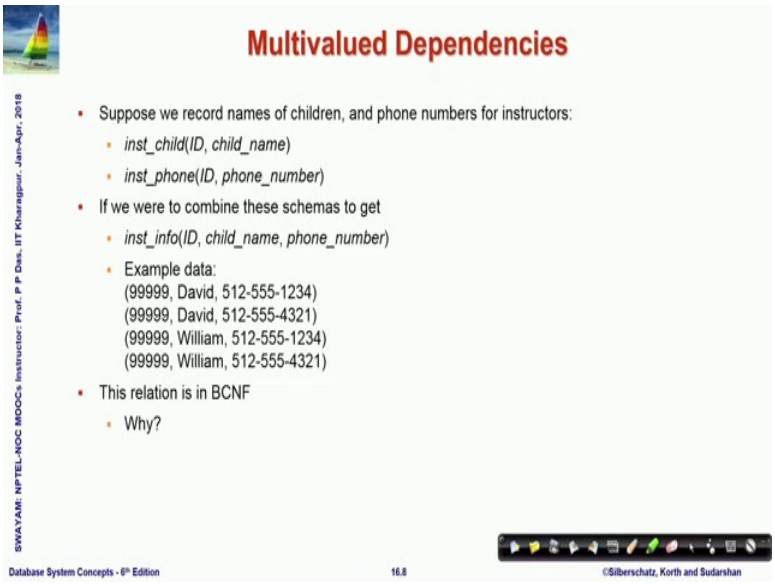
(Refer Slide Time: 04:19)



So, let us move on. So, this is just another example, we have two different relations Student give the student id and name, Courses giving course id and name and the corresponding functional dependencies in them, you can see two instances of that. But if we as such, there is no relationship between student and course, but if we choose to keep them in a single relation, say Student_Course which I have shown on bottom right here. Then there will be you can see lot of redundancies coming in, because since I they can be in terms of all different combinations. So, S 1 has in name A may be taking course C 1 having name C, but again the S 1 having name A could be taking course C 2 having name B, you just do not know which one is correct.

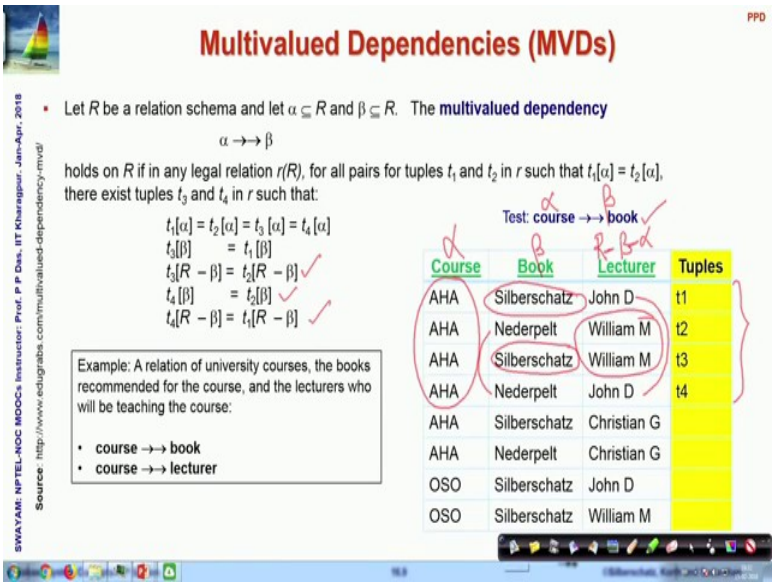
So, you can see that here again you have 2 multiple value dependencies, one where **SID →→ CID** and **SID →→ Cname**. So, these are the two different multiple values that you can, find against the SID and this is ah. So, if two or more multi valued dependencies exist in a relation, then while we convert the we convert multivalued attributes into single valued attributes, then the multi value dependency will show up. So, that is the basic problem that we would like to address.

(Refer Slide Time: 05:59)



This is another example of two relations, where the ID and child are together, when ID and phone_number are together. So, naturally if I combine them into a single relation, you have a set of possibilities of multiple different tuples. Because given an ID there could be multiple children, there given an id there could be multiple phone numbers. Mind you, this relation of isn't info is still in Boyce Codd normal form, because there is no dependence there is no functional dependency that holds on this relation. So, the key of this relation is the union of all the three attributes and therefore, that being the key and no functional dependency holding on it, naturally vacuously makes it Boyce Codd normal form, but you can still see that there are redundancy in that is data.

(Refer Slide Time: 06:48)



So now, let us define multivalued dependency in a formal way and. So, we say that **α →→ β**, naturally α and β both have to be subsets of the given set of attributes. When we say that? When there are for all pairs of tuples t1 and t2 such that they match on the fields of α, this till this point it looks like functional dependencies. There exists two more tuples t3 and t4 such that this condition sold, what are the conditions? Look, carefully here we say that all of them match on the α attributes which is fine, then you say that t3 matches with t1 in the β attributes and t3 matches on the remaining attributes with t2. Similarly, t4 matches with t2 in the β attributes and t4 matches with t1 on the remaining attributes.

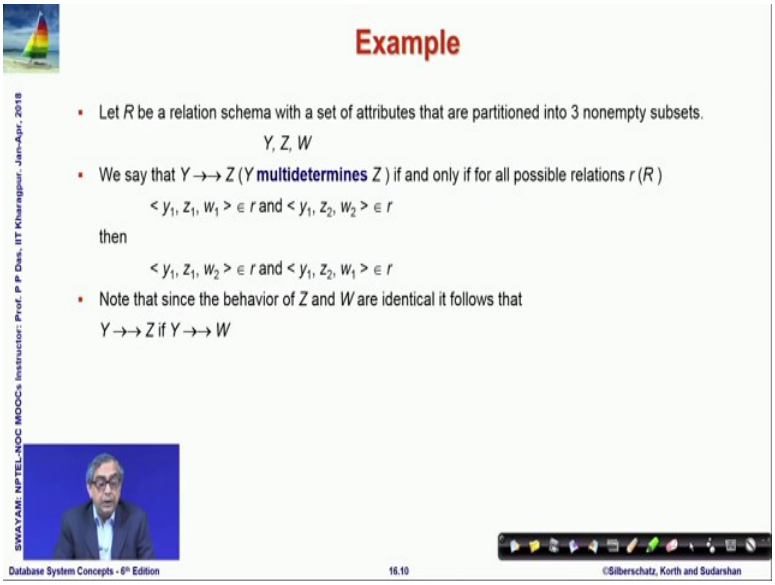
So, let us look at an example, gets confusing. So, here is course book and lecturer ah. So, it is a relationship of university courses known naturally, every course has multiple recommended books and every course has been taken by multiple different lecturers from time to time. So, course can have multiple books. So, there is a multivalued dependency here, it can be taught by multiple lectures.

So, there is a multivalued dependency here and therefore, I can have an instance of this particular relation and I am just showing you, how to test for the multivalued dependency **course →→ book**. So, these are the two, four tuples I have marked t1, t2, t3, t4 if you look into the first condition. So, this is your α I am checking for. So, this is α this is β. So, this is β and this is ah. So, to say **R –(β – α)** ok.

So, the first condition that all these tuples will have to match on α yes, they do, all four of them have AHA here. So, that is fine take at the second condition t3 on β is Silberschatz and t1 on β is also Silberschatz. So, they match and t3 on the remaining attributes remaining attributes are, if I take out β if I take out book it is AHA it is course and the lecturer that is remaining. Now it already matches on the course. So, I do not have to check for that, but. So, I can just check for whether it matches on lecturer, between some checking for this rule, whether t3 and t2 match yes t3 and t2 match, they have the same name for the lecturer.

Look at the next one which is t4 and t2 match on β, t4 and t2 match on β yes, they have the same name of the book, and whether t4 and t1 match on the lecturer, this rule t4 and t 1 match on the lecturer this rule. So, it also satisfies. So, I can say that this relation has holds the multivalued dependency course multi determining book. In a similar way you can you can mark your t1, t2, t3, t4 on this and check for course multi determining the lecturer, actually we will we will soon state that; if **course** →→ **book**, then it is trivial that course will also multi determine lecturer.

(Refer Slide Time: 10:36)

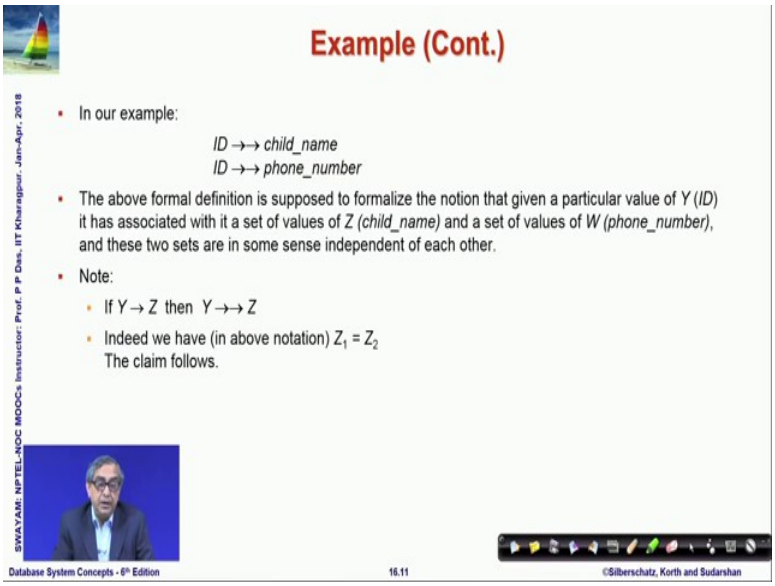


So, this is just to tell you if you have 3 non-empty sets of attributes Y, Z and W and then we say, Y →→ Z, if and only if there are these are the possible relations. That I can have Y 1 and Z 1 W 1 in a relation and Y 1 and Z 2, W 2 in the relation, then I can have

Y 1, Z 1 with W 2 and Y 1, Z 2 with W 1, that is you can basically take the cross of these to other 2 attributes and those are r tuples, possible tuples in your relation and ah.

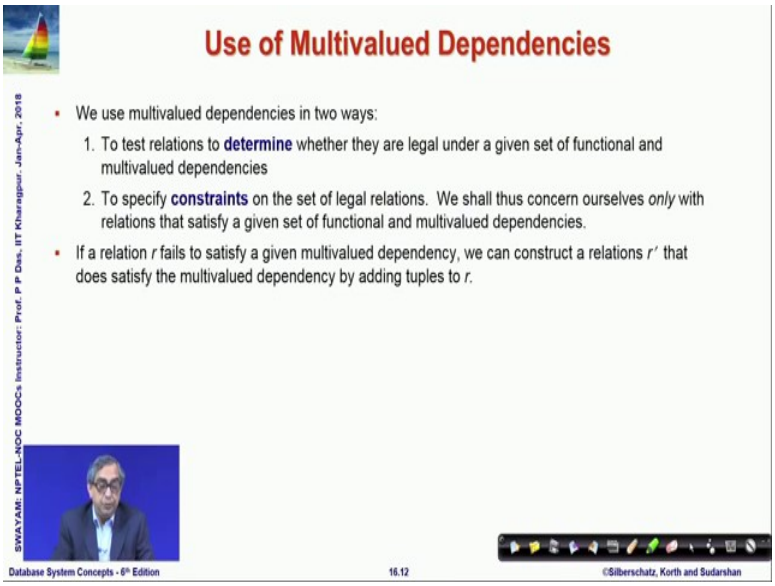
So, you can you can naturally if you read it in little in a different way, then you can observe that since the behavior of Z and W are identical they are switchable. So, if **Y →→ Z**, then you can you have ZY multi determining W and vice versa. So, this is; what is a core observation in terms of the multi value dependencies

(Refer Slide Time: 11:40)



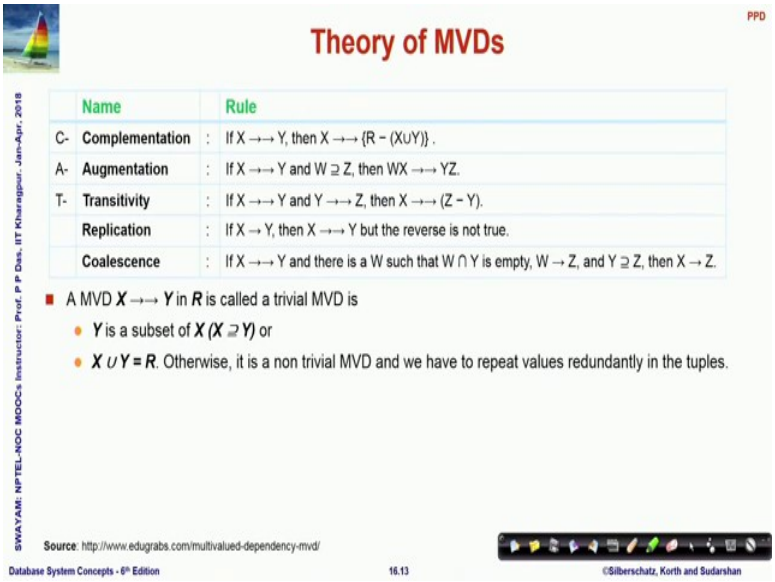
So, this is in terms of our example, you can now clearly understand that **ID →→ child name,ID →→ phone number**, in the earlier example that we took and ah. So, we can also note that if there is a functional dependency, **Y→Z** then; obviously, **Y →→Z**, that is that is just quite obvious.

(Refer Slide Time: 12:06)



So, we have to we can make use of multi value dependency to specify, further constraints to remove redundancies and defining what is legal in a relation. And if a relation fails to satisfy a given multivalued dependency, then we can construct a relation R primed, that does satisfy the multi valued dependency by adding tuples to that r right.

(Refer Slide Time: 12:32)



Now, once having defined the notion of multi valued dependency, we next proceed to check, how do we reason about that. So, I would remind you about functional

dependencies, and the different rules of ah functional dependencies Armstrong's rules, that we had introduced the all of these of augmentation transitivity and all that.

So, in terms of functional dependencies we have three rules, commonly called the cat rules. Which purely involve the functional dependencies, first is a complementation which is a kind which we have just discussed shown, that if **X →→ Y**, then **X →→ R – (X UY)** →→ the remaining set of attributes.

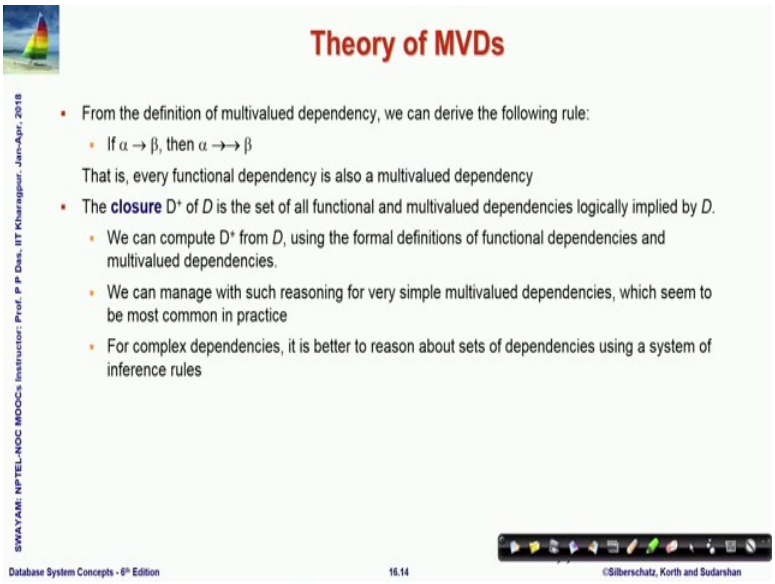
Augmentation that is I can augment any multivalued dependency with left and putting attributes on the left and right-hand side, as long as I put all attributes that I put on the right-hand side, I put them on the left-hand side. I may put more attributes on the left-hand side, but all attributes that I put on the right-hand side here Z must be a subset of the attributes that I put on the left-hand side, that augmentation is possible. Transitivity is manifesting in a little different way, if **X →→ Y** and **Y →→ Z** then, **X →→ Z -Y**.

So, these are the these are the 3 rules which are basically these three are rules that, involve only multi valued dependencies and the other two rules, actually involve the relationship between multi value dependency the replication rule and the coalescence rule, which are between the multi value dependency and the functional dependency. We are not going deeper into that further, or trying to take specific examples and show how they work.

I just want you to know that such rules exist through which, you can define similar algorithms for multivalued dependency also, as we did for functional dependency like as you can understand the most critical algorithm to define would be the algorithm of closure, which can again be used in the situation where I have functional as well as multivalued dependency.

So, just we will keep that, in little bit advance space of this course. So, just know that such things exist, but we are not going into the details of that. Finally, for a multivalued dependency where, **X→Y** we call that MVD to be trivial. If either **Y ⊂ X** which is the notion we used for functional dependencies or there is a second condition here, that the **X U Y** that left hand right hand side gives you the whole set of attributes, otherwise it is a non-trivial multivalued dependency and we have to repeat the values. So, these are the two conditions, if they satisfy then we know that we have a trivial multi value dependency and we do not want to deal with that.

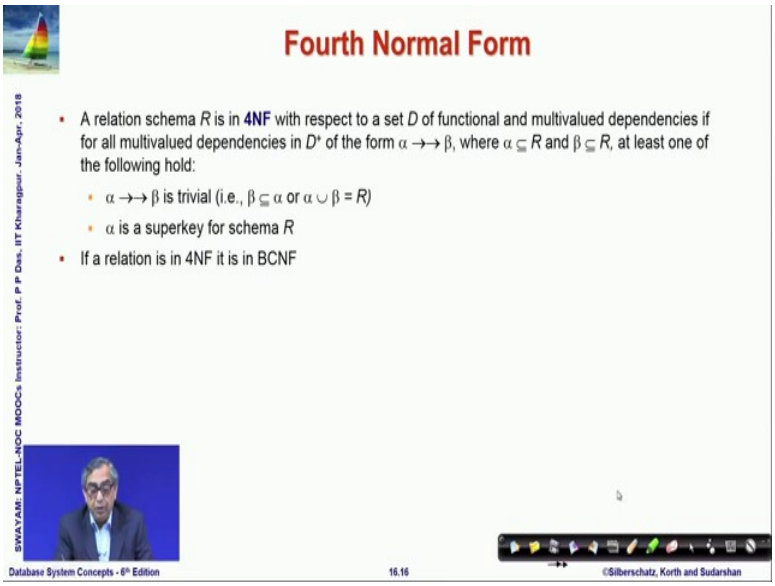
(Refer Slide Time: 15:40)



So, there is little bit of references to the theory given here, I have mentioned that there are closure algorithms ah. So, that given a set of dependencies I will now generalize and say dependencies, which means that there could be functional dependencies, as well as multivalued dependencies. Given a set of dependencies you can define a closure of all of these functional and multivalued dependencies together, that are implied by the given set and we can have all those parallel definitions of closure of the dependencies, the minimal cover, canonical cover and so on.

So, I just want you to note that these things have been defined and the existent theory, but will be beyond the current course that we are pursuing. So, it is now that we have is we have seen an another additional source of redundancy in our data, in terms of multiple values and in terms of the multi value dependency that hold.

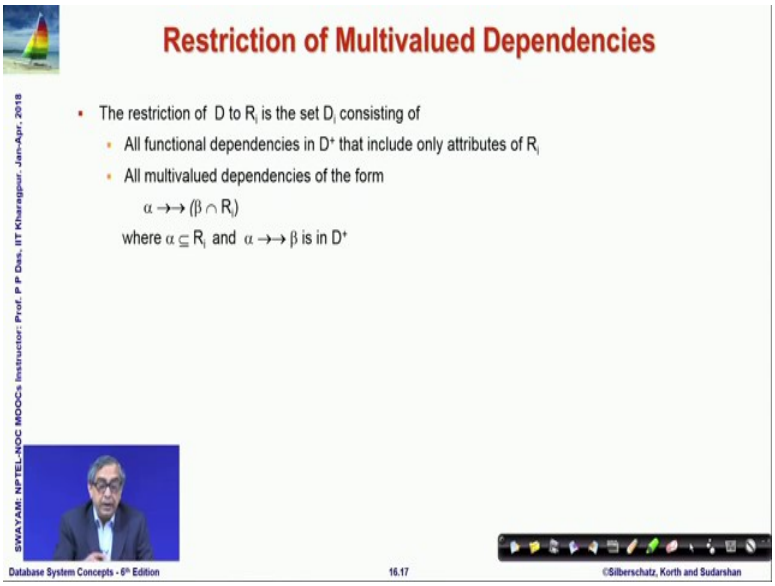
(Refer Slide Time: 16:50)



So, we would now like to look into if such dependencies exist, then how do you decompose a relation to satisfy that the redundancy caused by such dependencies are not affecting us. So, such a normal form it is beyond the third normal form is called to be said to be a 4th normal form or 4 NF. Where you say that a relation is in 4 NF if, every multi valued dependency $\boldsymbol{\alpha \rightarrow\rightarrow \beta}$, in the closure of the set of dependencies is either trivial, trivial means that left hand side is a subset of the right-hand side or the union of the left and right-hand side gives you the whole set of attributes.

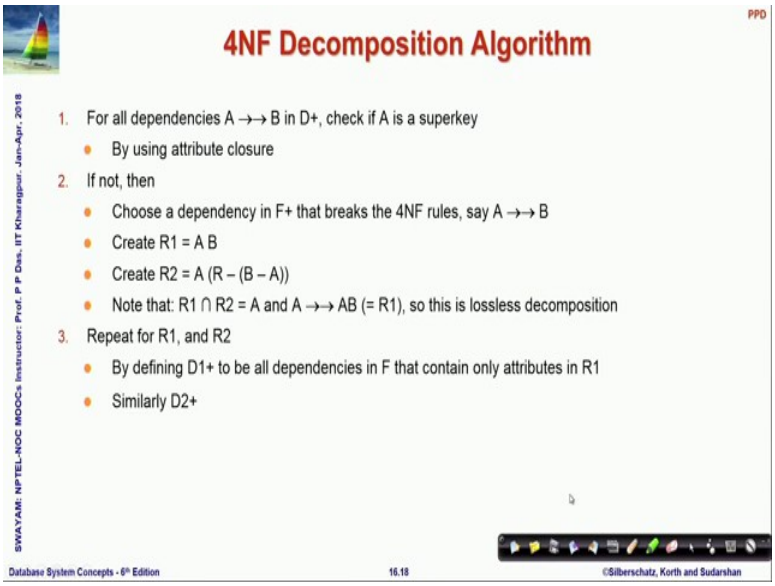
So, it is either trivial every dependency is either trivial, or α left-hand side is a superset of the schema R, you can very well relate that this is just a little twist on the definition of the Boyce Codd normal form, where the second condition was identical and only thing in the first condition instead of MVD, you had a functional dependency. So, when we have this, we say we are a relation would be in the in the 4th normal form. Naturally, if a relation is in 4th normal form, it is trivial that it will be in the Boyce Codd normal form, but the reverse will not be necessarily true.

(Refer Slide Time: 18:19)



So, again the same set of concepts that, if I have a set of dependencies and you have a decomposed relation then smaller relation, then I can project that set of dependencies, in terms of a particular subset of the attributes and here is the condition that is given.

(Refer Slide Time: 18:43)

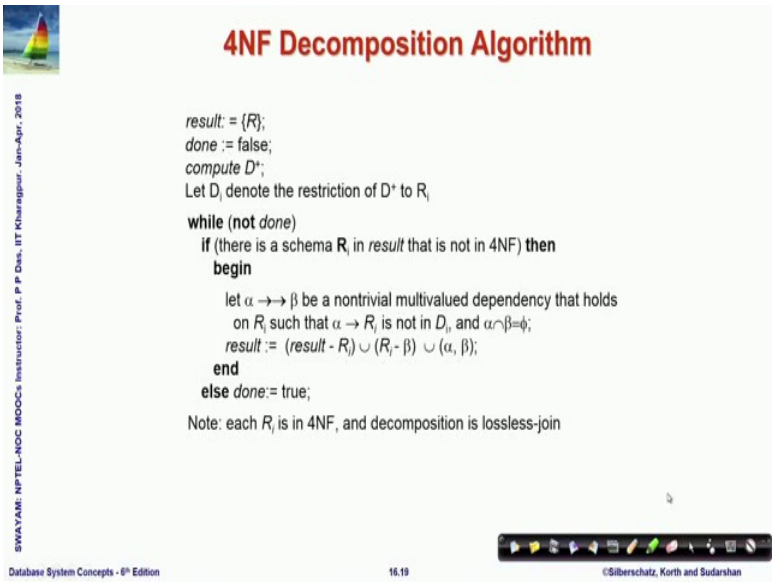


So, the decomposition algorithm into 4 NF is exactly like the decomposition algorithm of the Boyce Codd Normal form BCNF. Only difference being that, now you may be doing this crucial step of 2A decomposition, for every multivalued dependency also earlier we were doing this only for the functional dependency.

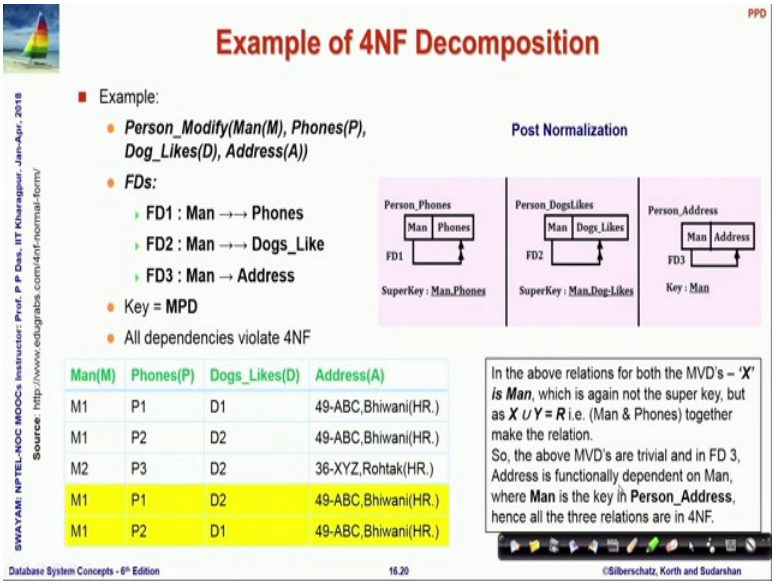
So, now if there is any offending multivalued dependency, which is not satisfying the 4 NF form? We can decompose the relation in terms of R 1 and R 2, as in here which is exactly like the Boyce Codd normal form and then the rest of it is simple. If ah if it is you know by this another important point, that you that you must note is in this process you actually guarantee lossless join.

So, this also continues to be in lossless join, with every decomposition and then you keep on repeating till all dependencies in F, in your set has been dealt with the attributes in R 1 and have converted them into the 4 NF form. So, you have a total 4 NF decomposition happening.

(Refer Slide Time: 20:04)



(Refer Slide Time: 20:13)



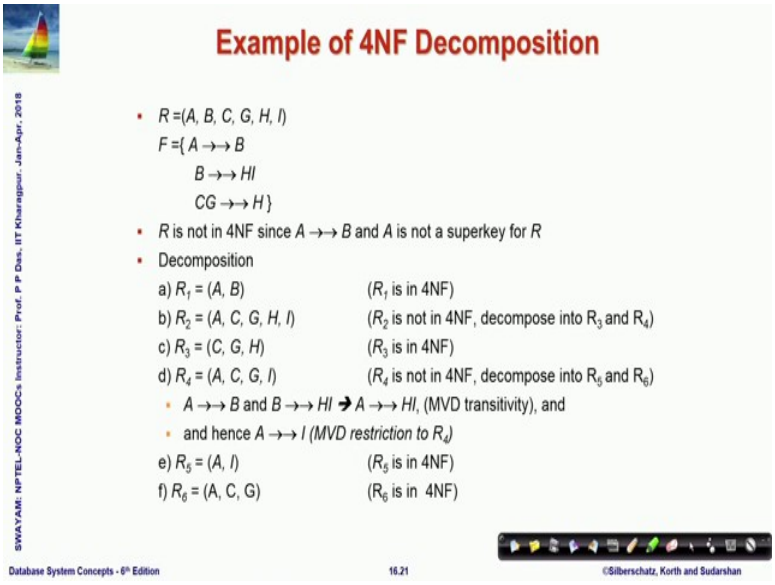
Ah let us take a this here, is the like before here is a formal algorithm for those who would be interested, to formally study the steps. Ah here I am just showing examples of 4 NF decomposition. So, we started this discussion with a person relational scheme, having man, phone and dog likes MPD, I have added I have just modified and I have added another attribute address. So, that in addition to the multi value dependencies, I can also have a functional dependency. So, we have two multivalued dependencies like before, man multi determining phones and man multi determining dog like, but now we have a functional dependency man determining address the key continues to be MPD.

So, all of these dependencies will violate the 4 NF, because none of them satisfy the either of the condition, that none of them are trivial and on for none of them left hand side is a super key because key is MPD. So, you can see that in on instances of this, relational schema you will have multiple redundant records, in the actual instance. So, on the right we normalize we normalize by taking FD 1; **man U phones** that gives you the first relation and then the rest of it. Then again you split based on FD 2, you have the second relation in the decomposition man and dog like and the third one gets generated as a byproduct of that, which is man and address.

And you have three relations now, which together represent the original relation each one of them is in 4th normal form. Actually, what happens is, when you when you have decomposed then, FD1 in this has become a relation where, the multivalued dependency **man →→phones** can be checked in terms of a functional dependency itself, and that that is what gives you the multi value dependency. And since it is multivalued so, man and

phones together continues to form the key, similarly in the second one the man and dog like is the key. Because you just have the multivalued dependency and given the same man, you will have multiple dogs whom he or she likes, but in the third one in the person address where you have man and address you have only man as the key, because man is a functional dependency that holds.

(Refer Slide Time: 23:01)

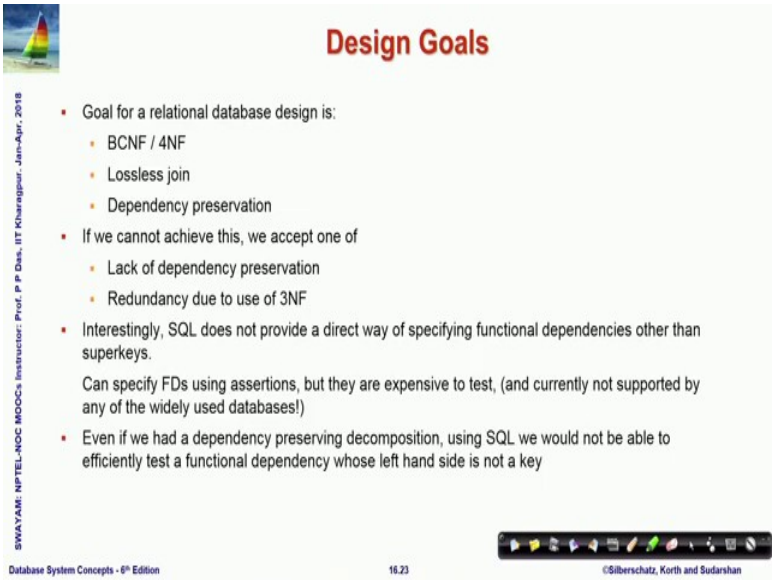


So, this is a simple illustration of decomposition into 4 NF, here is a little more elaborate one, again this is a we have I have worked through the steps. So, there are three multivalued dependencies and you can see that, **A →→B** is not does not is not who does not hold the condition of 4 NF. So, you have to decompose, you decompose get R 1 which is in 4 NF and the remaining R 2 which is also not in 4 NF. So, decompose in R 3 which is in 4 NF and R 4, we can R 4 is not in 4 NF you decompose R 4 into R 5 and R 6 and work through that, and you will be able to see that R 5 is in 4 NF and R 6 also is in 4 NF, which gives a complete multivalued decomposition of this whole set.

Naturally with that, we will conclude our discussion on the decomposition process, there are there would be some more aspects to look at and there is lot of more normal forms that exist. But this is for all practical purposes; a database is normalized, when it is represented in terms of the third normal form. And I have discussed still I have discussed the 4th normal form, because in some places people prefer to represent also in 4th normal form.

So, that they guarantee that they have even less redundancy in the data, but leaving that, let us quickly take a round in terms of the what we have done so far and what is a basic overall design process that we should be following.

(Refer Slide Time: 24:43)



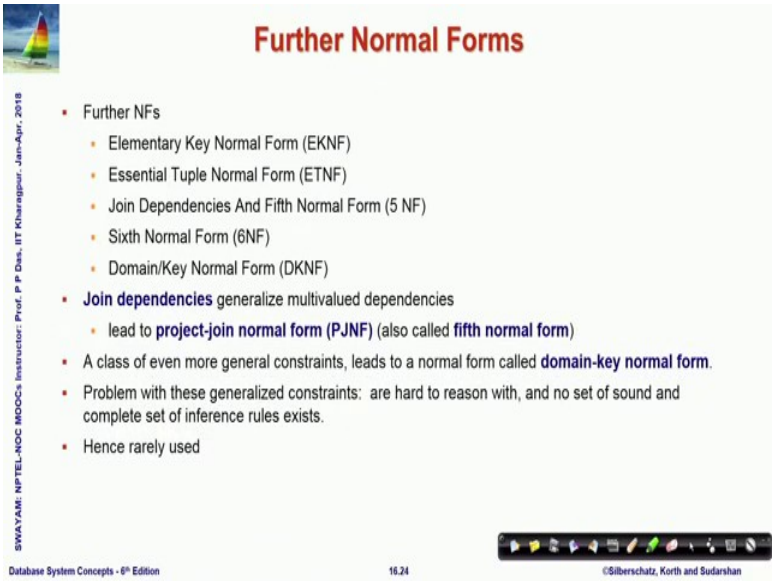
So, again to remind you the goal for our design is to have a relational database which is in BCNF or 3 NF has a lossless join due to the decomposition, and dependency preservation. If we cannot achieve that, I am I am sorry earlier what I meant is BCNF or 4 NF not BCNF and 3 NF. So, the idea would be I have a decomposition in BCNF and 4 NF lossless join and dependency preservation which may not be achievable. If I cannot achieve that then I go I have to sacrifice either, the lack of dependency preservation. So, dependencies will have to be checked using natural join or, I will allow a little bit of redundancy and use the third normal form where I have the guarantee.

Now, at this point you must wonder and note that SQL, the language in which we are doing the creation and update and the query processing. That SQL does not provide any directory of specifying or checking any dependency, other than the functional other than the functional dependency that checks the super key. Super key is the only functional dependency that SQL would check, no other functional dependency or multivalued dependency and other type dependencies are can be specified or checked in SQL. You can do that using assertions, in the in the while discussing SQL I were talked about assertions we can do that using assertions, but that too is very expensive to test. So, it is

not usually supported by any of the databases, which are widely used because that slows down your every process very, very much.

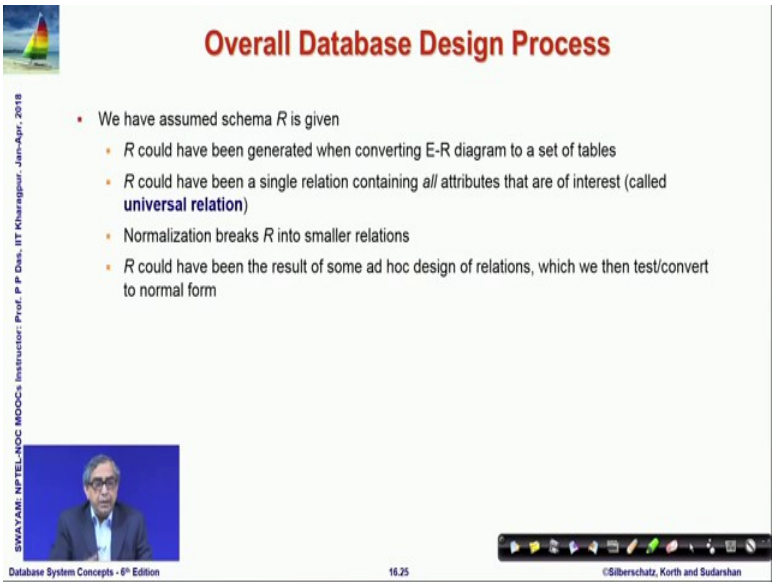
So, you can understand that in terms of your design goals, you have to do a very good job to make sure that, your functional and multivalued dependencies are accurately expressed in the design and accordingly the schemas are normalized in the proper ways satisfying BCNF or 4 NF or 3 NF normal forms. But because, while you will actually have instances there will not be a practical way, to see if you are violating any one or more of these ah rules of dependencies that you have set.

(Refer Slide Time: 27:09)



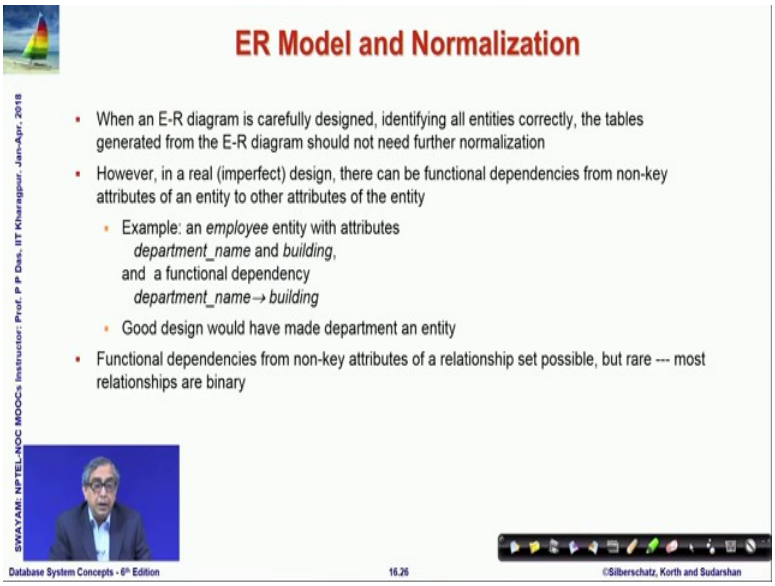
So, as I mentioned there are actually these are not the only forms, there are various other normal forms as well and fifth normal form, 6 normal form and so on, but it is very rarely these are very rarely used. It is not easy to decompose into these normal forms and by this decomposition does not give you enough returns in terms of the reduction of redundancy and removal of anomalies, that people often would have motivation to do them, but you should know that such normal forms exist.

(Refer Slide Time: 27:44)



So, in the overall process if we look, at I mean what we have been doing is there are several tracks that we could be taking one possible thing is, the whole set of attributes have been generated while we have converted or relation has been generated. When you have converted the entity relationship diagram, the UML or the ER diagram into a set of tables, that is how we got our set of attributes or the relational schema R, it is also possible that we just started with a single relation containing all attributes, which is called the universal relation? And then normalization will break them into smaller relations. It could have been or could have been the result of some adds of design of relations also, and then you convert them.

(Refer Slide Time: 28:33)



So, there are possible all different possible tracks that can happen. So, if we have taken the ER model track, then frankly speaking if the ER model is carefully designed, then every entity defined in that ER model will have only the dependency; which are the determining super key.

So, just recall the employee department building kind of situation we discussed earlier. So, an employee entity has attributes department name and building, and there is a functional dependency from department_name to building. So, what it means that in the entity relationship diagram itself we didn't do a good job. If we had done a good job then we would have identified that the department itself is an entity and therefore, would not feature as an attribute on the employee. So, it would have been I mean right there, we would have if we had called it as a separate entity, then that is equivalent of what we are doing now taking the relation and then breaking it down through decomposition.

So, functional dependencies from non-key attributes of a relationship are possible ah, but are rare. So, mostly the relationships are binary, and if you do a careful design of the ER model then many of these deep exercise of normalization you will not have to go through.