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)

PPD

Module Recap

- Decomposition Using Functional Dependencies
- Functional Dependency Theory

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition | 16.2 | ©Silberschatz, Korth and Sudarshan

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)

Module Objectives

PPD

- To Learn Algorithms for Properties of Functional Dependencies
- To Understand the Characterizations for Lossless Join Decomposition
- To Understand the Characterizations for Dependency Preservation

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 16.3 ©Silberschatz, Korth and Sudarshan

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)

Module Outline

PPD

- Algorithms for Functional Dependencies
- Lossless Join Decomposition
- Dependency Preservation

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 16.4 ©Silberschatz, Korth and Sudarshan

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

(Refer Slide Time: 01:15)

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \to B$
  $A \to C$
  $CG \to H$
  $CG \to I$
  $B \to H\}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \to C$ and $A \to B)$
  3. $result = ABCGH$ $(CG \to H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \to I$ and $CG \subseteq AGBCH)$
- Is AG a candidate key?
  1. Is AG a super key?
     1. Does $AG \to R$? == Is $(AG)^+ \supseteq R$
  2. Is any subset of AG a superkey?
     1. Does $A \to R$? == Is $(A)^+ \supseteq R$
     2. Does $G \to R$? == Is $(G)^+ \supseteq R$

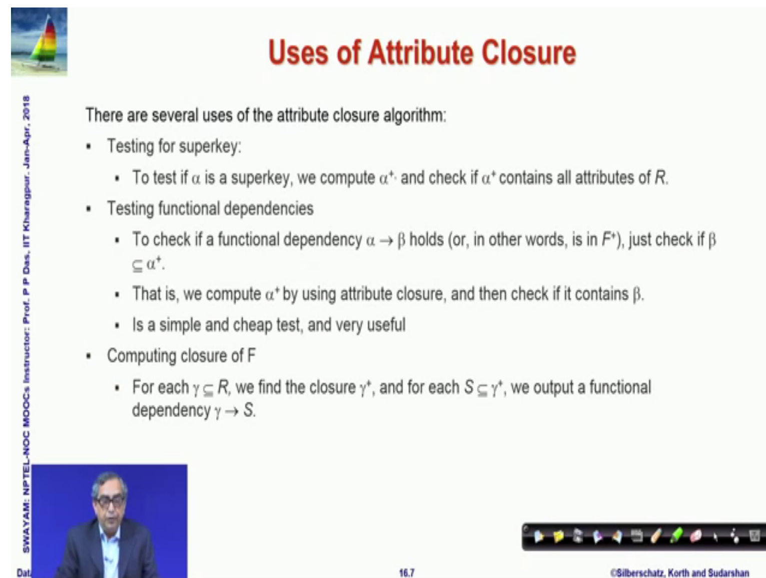
SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

16.6 ©Silberschatz, Korth and Sudarshan

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 art 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 alpha determines beta holds, then rather in other words whether alpha determines beta is in the closure of the set of functional dependencies F, then all that we need to do is to compute alpha plus that is a closure of the set of attributes on the left hand side of the dependency and check if beta is a subset of that. If beta is a subset of that, then I know that alpha determines 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)

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \to C$ is redundant in: $\{A \to B,\ B \to C, A \to C\}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS: $\{A \to B,\ B \to C,\ A \to CD\}$ can be simplified to $\{A \to B,\ B \to C,\ A \to D\}$
      - *In the forward:* (1) $A \to CD \Rightarrow A \to C$ and $A \to D$ (2) $A \to B,\ B \to C \Rightarrow A \to C$
      - *In the reverse:* (1) $A \to B,\ B \to C \Rightarrow A \to C$ (2) $A \to C, A \to D \Rightarrow A \to CD$
    - E.g.: on LHS: $\{A \to B,\ B \to C,\ AC \to D\}$ can be simplified to $\{A \to B,\ B \to C,\ A \to D\}$
      - *In the forward:* (1) $A \to B,\ B \to C \Rightarrow A \to C \Rightarrow A \to AC$ (2) $A \to AC$, $AC \to D \Rightarrow A \to D$
      - *In the reverse:* $A \to D \Rightarrow AC \to D$
- Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 16.8 ©Silberschatz, Korth and Sudarshan

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 determining C. So, in that way ACA determining 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 implies 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 Armstrongs 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 Armstrongs 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)

Canonical Cover: LHS

- $\{A \rightarrow B,\ B \rightarrow C,\ AC \rightarrow D\} \Rightarrow \{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow D\}$
  - $A \rightarrow B,\ B \rightarrow C \Rightarrow A \rightarrow C \Rightarrow A \rightarrow AC$
  - $A \rightarrow AC,\ AC \rightarrow D \Rightarrow A \rightarrow D$
  - $A+ = ABCD$
- $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow D\} \Rightarrow \{A \rightarrow B,\ B \rightarrow C,\ AC \rightarrow D\}$
  - $A \rightarrow D \Rightarrow AC \rightarrow D$
  - $AC+ = ABCD$

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

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

(Refer Slide Time: 07:32)

Extraneous Attributes

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
  - Attribute A is **extraneous** in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute A is **extraneous** in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

$\alpha \rightarrow \beta$

$A \in \alpha$

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

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 functionally determines beta and I have an

attribute A which belongs to alpha, then we can check whether it is possible to remove A from alpha. 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 implies 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 implies the original set.

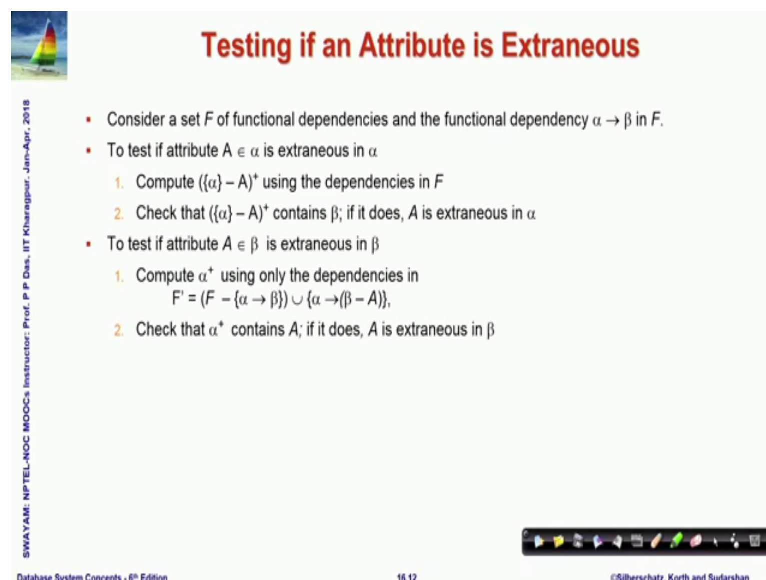
(Refer Slide Time: 09:27)

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

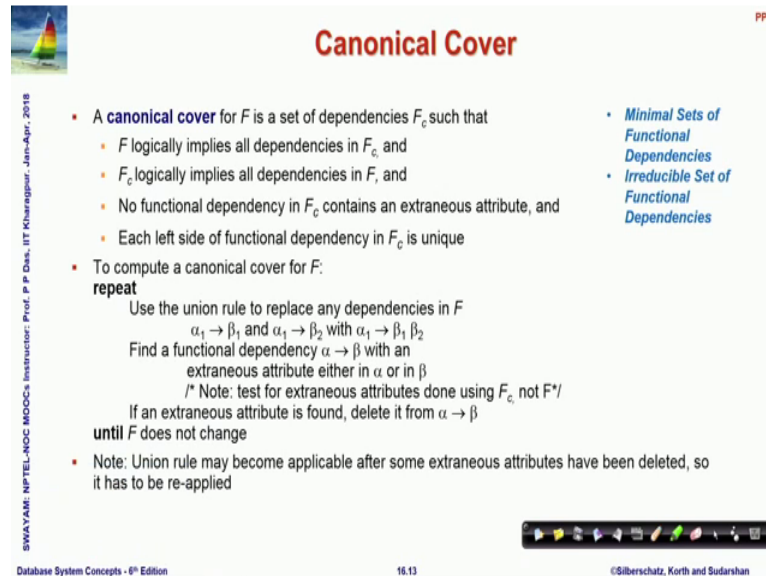
Another example where you are trying to see an extraneous attribute on the right hand side; so in this example, C on the right hand side of the set AB determining CD is extraneous because it can be inferred even after because AB determining C can be inferred even after deleting this C from the right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

Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.
      - Can use attribute closure of A in more complex cases

The canonical cover is: $A \rightarrow B$
$B \rightarrow C$

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

16.14

©Silberschatz, Korth and Sudarshan

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 determining BC. So, it becomes a simpler set. So, A determining B is removed, then I would check for A being extraneous in AB determining C and we find that it indeed is extraneous.

So, because B determining 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 determining 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 determining B and B determining 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)

PPD

Equivalence of Sets of Functional Dependencies

- Let F & G are two functional dependency sets.
  - These two sets F & G are equivalent if $F^+ = G^+$
  - Equivalence means that every functional dependency in F can be inferred from G, and every functional dependency in G an be inferred from F
- F and G are equal only if
  - F covers G: Means that all functional dependency of G are logically numbers of functional dependency set $F \Rightarrow F \supseteq G$.
  - G covers F: Means that all functional dependency of F are logically members of functional dependency set $G \Rightarrow G \supseteq F$

Condition	CASES			
F Covers G	True	True	False	False
G Covers F	True	False	True	False
Result	F=G	$F \supset G$	$G \supset F$	No Comparison

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Naturally this is strongly using the underlying concept of equivalence of two sets of functional dependencies F and G. They are equivalent if there closures are equal or in other words, if F covers G and G covers F, that is F logically implies G and G logically implies 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)

PPD

Practice Problems on Functional Dependencies

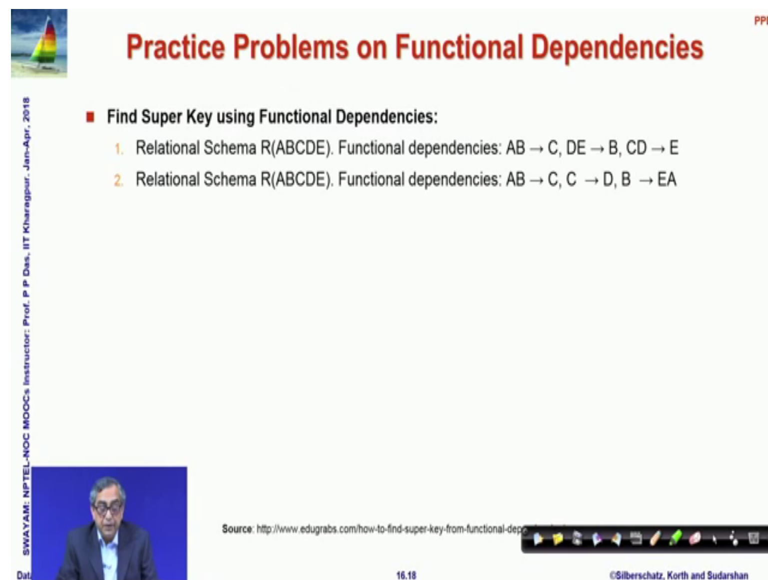
- **Find if a given functional dependency is implied from a set of Functional Dependencies:**
  1. For: A → BC, CD → E, E → C, D → AEH, ABH → BD, DH → BC
     a. Check: BCD → H
     b. Check: AED→C
  2. For: AB → CD, AF → D, DE → F, C → G, F → E, G → A
     a. Check: CF → DF
     b. Check: BG → E
     c. Check: AF → G
     d. Check: AB → EF
  3. For: A → BC, B → E, CD → EF
     a. Check: AD → F

Source: http://www.edugrabs.com/membership-test-for-functional-depen...

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

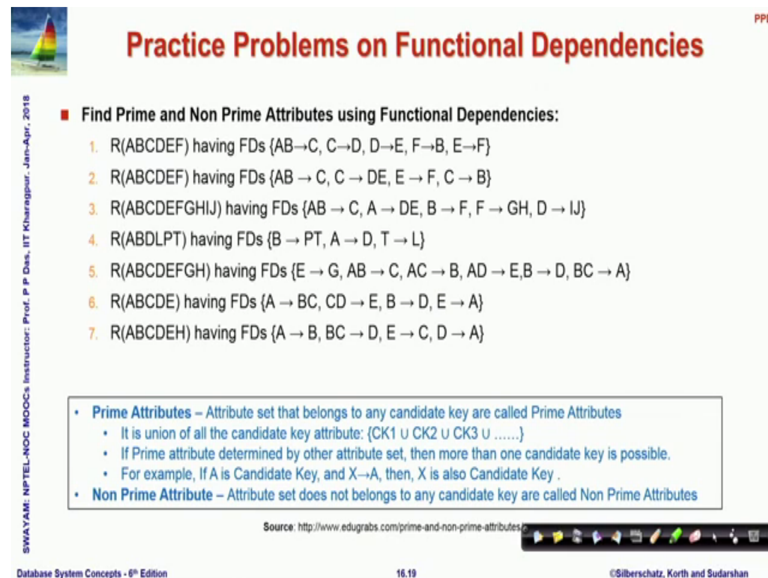
Next what I have done is, we have put a number of practice problems for various kind of things that you can do with functional dependencies. The first set of problems. Find in first set of problems you have to find if a given functional dependency is implied from a set of functional dependencies. So, 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)



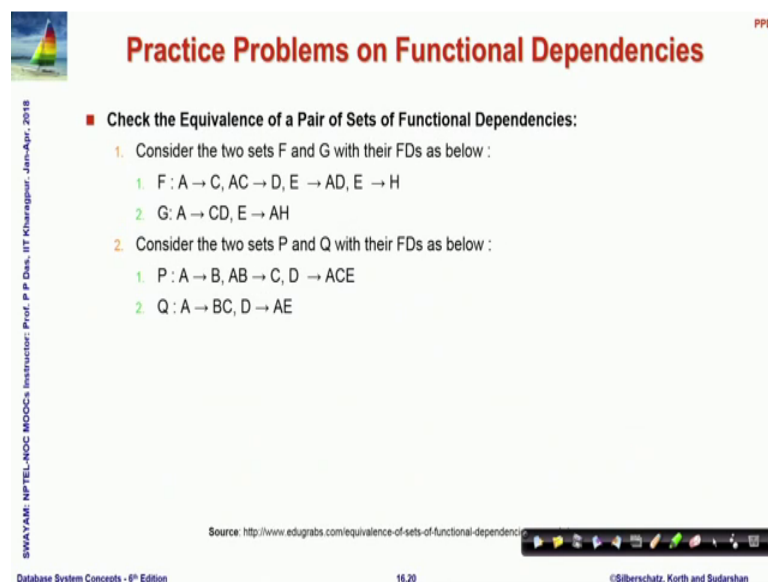
You can also check if you can find candidate key using the functional dependencies. The sets are given. Your task would be to find the candidate keys.

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)

PPD

Practice Problems on Functional Dependencies

- Find the Minimal Cover or Irreducible Sets or Canonical Cover of a Set of Functional Dependencies:
    1. $AB \rightarrow CD$, $BC \rightarrow D$
    2. $ABCD \rightarrow E$, $E \rightarrow D$, $AC \rightarrow D$, $A \rightarrow B$

Source: http://www.edugrabs.com/questions-on-minimal-cover/

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

16.21

©Silberschatz, Korth and Sudarshan

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)

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

To Identify whether a decomposition is lossy or lossless, it must satisfy the following conditions :
- $R_1 \cup R_2 = R$
- $R_1 \cap R_2 \neq \Phi$ and
- $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 16.23 ©Silberschatz, Korth and Sudarshan

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 recomputation 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 intersection 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 compute the join between projection on R1 and the projection of R2.

So, if this intersection set of attributes uniquely determines R1 or it uniquely determines 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 lost layer, 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)

Example

- Consider **Supplier_Parts** schema: **Supplier_Parts(S#, Sname, City, P#, Qty)**
- Having dependencies: **S# → Sname, S# → City, (S#, P#) → Qty**
- Decompose as: **Supplier(S#, Sname, City, Qty)**: **Parts(P#, Qty)**
- Take Natural Join to reconstruct: **Supplier ⋈ Parts**

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

S#	Sname	City	Qty
3	Smith	London	20
5	Nick	NY	50
2	Steve	Boston	10
5	Nick	NY	40
5	Nick	NY	10

P#	Qty
301	20
500	50
20	10
400	40
301	10

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
5	Nick	NY	20	10
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10
2	Steve	Boston	301	10

- We get extra tuples! **Join is Lossy!**
- Common attribute **Qty** is not a superkey in **Supplier** or in **Parts**
- Does not preserve **(S#,P#) → Qty**

Source: http://www.edugrabs.com/lossy-join-decomposition/

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 determines the supplier name and the supplier city and supplier number and product number together determines the 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 intersection R2 implying R1 or implying 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)

PPD

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Example

- Consider **Supplier_Parts** schema: **Supplier_Parts(S#, Sname, City, P#, Qty)**
- Having dependencies: **S# → Sname, S# → City, (S#, P#) → Qty**
- Decompose as: **Supplier(S#, Sname, City)**: **Parts(S#, P#, Qty)**
- Take Natural Join to reconstruct: **Supplier ⋈ Parts**

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

S#	Sname	City
3	Smith	London
5	Nick	NY
2	Steve	Boston
5	Nick	NY
5	Nick	NY

S#	P#	Qty
3	301	20
5	500	50
2	20	10
5	400	40
5	301	10

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

- We get back the original relation. **Join is Lossless.**
- Common attribute **S#** is a superkey in **Supplier**
- Preserves all dependencies

Source: http://www.edugrabs.com/desirable-properties-of-decomposition/#lossless

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 it functionally determines S name as well as S city. So, R1 intersection R2 functionally determining 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)

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  - Can be decomposed in two different ways
- $R_1 = (A, B)$, $R_2 = (B, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving
- $R_1 = (A, B)$, $R_2 = (A, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

16.26 ©Silberschatz, Korth and Sudarshan

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 decomposotion.

(Refer Slide Time: 24:57)

Practice Problems on Lossless Join

- Check if the decomposition of R into D is lossless:
  1. R(ABC): F = {A → B, A → C}. D = R_1(AB), R_2(BC)
  2. R(ABCDEF): F = {A → B, B → C, C → D, E → F}. D = R_1(AB), R_2(BCD), R_3(DEF)
  3. R(ABCDEF): F = {A → B, C → DE, AC → F}. D = R_1(BE), R_2(ACDEF)
  4. R(ABCDEG): F = {AB → C, AC → B, AD → E, B → D, BC → A, E → G}
     1. D1 = R_1(AB), R_2(BC), R_3(ABDE), R_4(EG)
     2. D2 = R_1(ABC), R_2(ACDE), R_3(ADG)
  5. R(ABCDEFGHIJ): F = {AB → C, B → F, D → IJ, A → DE, F → GH}
     1. D1 = R_1(ABC), R_2(ADE), R_3(BF), R_4(FGH),R_5(DIJ)
     2. D2 = R_1(ABCDE), R_2(BFGH), R_3(DIJ)
     3. D3 = R_1(ABCD), R_2(DE), R_3(BF), R_4(FGH),R_5(DIJ)

Source: http://www.edugrabs.com/questions-on-lossless-join/

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

- Let F_i be the set of dependencies F^+ that include only attributes in R_i
  - A decomposition is **dependency preserving**, if
    $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive

Let R be the original relational schema having FD set F. Let R_1 and R_2 having FD set F_1 and F_2 respectively, are the decomposed sub-relations of R. The decomposition of R is said to be preserving if
- *$F_1 \cup F_2 \equiv F$ {Decomposition Preserving Dependency}*
- *If $F_1 \cup F_2 \subset F$ {Decomposition NOT Preserving Dependency} and*
- *$F_1 \cup F_2 \supset F$ {this is not possible}*

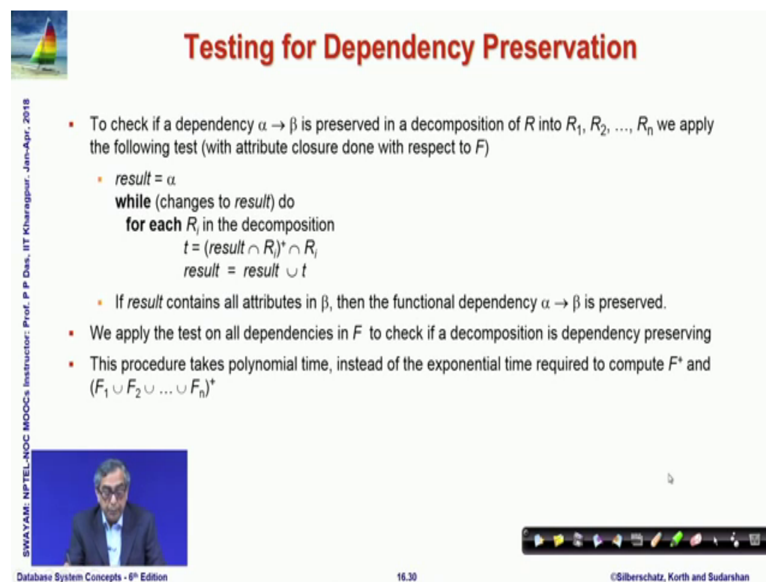
Dependency preservation is if you have a relation which you have decomposed into n different relations, so if you decompose a relation into a number of relations, then naturally all functional dependencies you cannot check on all the relations because a

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 R i the ith relation will have a set of dependencies F i which is a subset of the original set F and involves only the attributes which exist in 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 R 1, on R 2 and R n, F 1, F 2, F n. 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 r 3 we have one dependency again.

So, if we now think about the union 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 determining 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 indeed functionally will determine 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)

PPD

Example

- R(ABCDEF): F = {A→BCD, A→EF, BC→AD, BC→E, BC→F, B→F, D→E}. D = {ABCD, BF, DE}
- On projections:

ABCD (R1)	BF (R2)	DE (R3)
A → B, A → C, A → D, BC → A, BC → D	B → F	D → E

- Infer reverse FD's:
  - B+/F = BF: B → A cannot be inferred
  - C+/F = C: C → A cannot be inferred
  - D+/F = DE: D → A and D → BC cannot be inferred
  - A+/F = ABCDEF: A → BC can be inferred, but it is equal to A → B and A → C
  - F+/F = F: F → B cannot be inferred
  - E+/F = E: E → D cannot be inferred
- Need to check for: ~~A→BCD~~, **A→EF**, ~~BC→AD~~, **BC→E**, **BC→F**, ~~B→F~~, ~~D→E~~
  - (BC)+/F = ABCDEF. Preserves **BC→E**, **BC→F**
  - (A)+/F = ABCDEF. Preserves **A→EF**

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

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 implies determines BCD is written in terms of three dependencies. A implies B, B implies C and C implies 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 functionally determines 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 determines 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 determining B and A determining C. So, that logically implies that A determines 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)

Practice Problems on Dependency Preservation

- Check whether the decomposition of R into D is preserving dependency:
  1. R(ABCD): F = {A → B, B → C, C → D, D → A}. D = {AB, BC, CD}
  2. R(ABCDEF): F = {AB → CD, C → D, D → E, E → F}. D = {AB, CDE, EF}
  3. R(ABCDEG): F = {AB → C, AC → B, BC → A, AD → E, B → D, E → G}. D = {ABC, ACDE, ADG}
  4. R(ABCD): F = {A → B, B → C, C → D, D → B}. D = {AB, BC, BD}
  5. R(ABCDE): F = {A → BC, CD → E, B → D, E → A}. D = {ABCE, BD}

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

(Refer Slide Time: 34:06)

Module Summary

- Studied Algorithms for Properties of Functional Dependencies
- Understood the Characterization for and Determination of Lossless Join
- Understood the Characterization for and Determination of Dependency Preservation

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.