

## Lecture 21: Functional Dependency and Normal Forms

### A proper look at the ideas from last lecture

- Functional Dependencies
- Keys

# Functional Dependency

When the values of some attributes  $X$  determine the values of some other attributes  $Y$ , we call this a *functional dependency*, and write

$$X \longrightarrow Y$$

# Functional Dependency

When the values of some attributes  $X$  determine the values of some other attributes  $Y$ , we call this a *functional dependency*, and write

$$X \longrightarrow Y$$

The attribute sets  $X$  and  $Y$  must both be subsets of the same relation schema  $R$ .

# Functional Dependency

When the values of some attributes  $X$  determine the values of some other attributes  $Y$ , we call this a *functional dependency*, and write

$$X \longrightarrow Y$$

The attribute sets  $X$  and  $Y$  must both be subsets of the same relation schema  $R$ .

In the shipments example,  $SUPPLIER \longrightarrow CITY$ .

# Functional Dependency

When the values of some attributes  $X$  determine the values of some other attributes  $Y$ , we call this a *functional dependency*, and write

$$X \longrightarrow Y$$

The attribute sets  $X$  and  $Y$  must both be subsets of the same relation schema  $R$ .

In the shipments example,  $SUPPLIER \longrightarrow CITY$ .

Both sides can have multiple attributes  
 $SUPPLIER, PART \longrightarrow QUANTITY, DATE\_RECD$ .

## Functional dependency more precisely

- A functional dependency is a constraint

If relation  $R$  is an instance of relation schema  $\mathbf{R}$ ,

## Functional dependency more precisely

- A functional dependency is a constraint
- It says whether or not a given relation is admissible

If relation  $R$  is an instance of relation schema  $\mathbf{R}$ ,

## Functional dependency more precisely

- A functional dependency is a constraint
- It says whether or not a given relation is admissible

If relation  $R$  is an instance of relation schema  $\mathbf{R}$ ,

## Functional dependency more precisely

- A functional dependency is a constraint
- It says whether or not a given relation is admissible

If relation  $R$  is an instance of relation schema  $\mathbf{R}$ ,  
and  $X$  and  $Y$  are subsets of the attributes of  $\mathbf{R}$ ,

## Functional dependency more precisely

- A functional dependency is a constraint
- It says whether or not a given relation is admissible

If relation  $R$  is an instance of relation schema  $\mathbf{R}$ ,  
and  $X$  and  $Y$  are subsets of the attributes of  $\mathbf{R}$ ,  
then  $X \longrightarrow Y$  says that, for any two tuples  $s, t \in R$ ,

## Functional dependency more precisely

- A functional dependency is a constraint
- It says whether or not a given relation is admissible

If relation  $R$  is an instance of relation schema  $\mathbf{R}$ ,  
and  $X$  and  $Y$  are subsets of the attributes of  $\mathbf{R}$ ,  
then  $X \longrightarrow Y$  says that, for any two tuples  $s, t \in R$ ,

$$\text{if } \pi_X(s) = \pi_X(t) \text{ then } \pi_Y(s) = \pi_Y(t)$$

## Functionally Dependent Attributes are not (necessarily) Derived Attributes

- sometimes its possible to calculate the value of an attribute from the value of other attributes

## Functionally Dependent Attributes are not (necessarily) Derived Attributes

- sometimes its possible to calculate the value of an attribute from the value of other attributes
- that is there is a single function  $f$  we can apply to the attributes  $X$  to calculate the value of  $A$ , *for any state of that schema*

## Functionally Dependent Attributes are not (necessarily) Derived Attributes

- sometimes its possible to calculate the value of an attribute from the value of other attributes
- that is there is a single function  $f$  we can apply to the attributes  $X$  to calculate the value of  $A$ , *for any state of that schema*
- functional dependency is weaker than this, there can be a different function for each state.

## Functionally Dependent Attributes are not (necessarily) Derived Attributes

- sometimes its possible to calculate the value of an attribute from the value of other attributes
- that is there is a single function  $f$  we can apply to the attributes  $X$  to calculate the value of  $A$ , *for any state of that schema*
- functional dependency is weaker than this, there can be a different function for each state.

## Functionally Dependent Attributes are not (necessarily) Derived Attributes

- sometimes its possible to calculate the value of an attribute from the value of other attributes
- that is there is a single function  $f$  we can apply to the attributes  $X$  to calculate the value of  $A$ , *for any state of that schema*
- functional dependency is weaker than this, there can be a different function for each state.

eg, a SUPPLIER might move from one CITY to another.

## Functionally Dependent Attributes are not (necessarily) Derived Attributes

- sometimes its possible to calculate the value of an attribute from the value of other attributes
- that is there is a single function  $f$  we can apply to the attributes  $X$  to calculate the value of  $A$ , *for any state of that schema*
- functional dependency is weaker than this, there can be a different function for each state.

eg, a SUPPLIER might move from one CITY to another.

Different states, different functions  $SUPPLIER \longrightarrow CITY$

## Functionally Dependent Attributes are not (necessarily) Derived Attributes

- sometimes its possible to calculate the value of an attribute from the value of other attributes
- that is there is a single function  $f$  we can apply to the attributes  $X$  to calculate the value of  $A$ , *for any state of that schema*
- functional dependency is weaker than this, there can be a different function for each state.

eg, a SUPPLIER might move from one CITY to another.

Different states, different functions  $SUPPLIER \longrightarrow CITY$

$CITY$  is not *derivable* from  $SUPPLIER$ , just dependent on it

But

Are derived attributes necessarily functionally dependent on the attributes they are derived from?

But

Are derived attributes necessarily functionally dependent on the attributes they are derived from?

yes

# Functional Dependencies and Logical Design

- functional dependencies are knowledge about the subject matter of the database

# Functional Dependencies and Logical Design

- functional dependencies are knowledge about the subject matter of the database
- a database is a lot of “facts”

# Functional Dependencies and Logical Design

- functional dependencies are knowledge about the subject matter of the database
- a database is a lot of “facts”
- we want a database schema that can store every possible combination of relevant facts

# Functional Dependencies and Logical Design

- functional dependencies are knowledge about the subject matter of the database
- a database is a lot of “facts”
- we want a database schema that can store every possible combination of relevant facts
- we do not want the database to be able to store impossible combinations of facts

# Functional Dependencies and Logical Design

- functional dependencies are knowledge about the subject matter of the database
- a database is a lot of “facts”
- we want a database schema that can store every possible combination of relevant facts
- we do not want the database to be able to store impossible combinations of facts
- functional dependencies tell us about what is possible

# Functional Dependencies and Logical Design

- functional dependencies are knowledge about the subject matter of the database
- a database is a lot of “facts”
- we want a database schema that can store every possible combination of relevant facts
- we do not want the database to be able to store impossible combinations of facts
- functional dependencies tell us about what is possible
- to find them, we don't think about databases, but about the world we want to represent

# Functional Dependencies and Logical Design

- functional dependencies are knowledge about the subject matter of the database
- a database is a lot of “facts”
- we want a database schema that can store every possible combination of relevant facts
- we do not want the database to be able to store impossible combinations of facts
- functional dependencies tell us about what is possible
- to find them, we don't think about databases, but about the world we want to represent
- its an alternative to UML/ER for capturing design information

# Functional Dependencies and Logical Design

- functional dependencies are knowledge about the subject matter of the database
- a database is a lot of “facts”
- we want a database schema that can store every possible combination of relevant facts
- we do not want the database to be able to store impossible combinations of facts
- functional dependencies tell us about what is possible
- to find them, we don't think about databases, but about the world we want to represent
- its an alternative to UML/ER for capturing design information
- we will sometimes abbreviate *functional dependency* to FD

## Deduction of Functional Dependencies

There are *rules* like deductive rules in logic, which say that if some functional dependencies are true, then some other functional dependency must be true as well.

## Deduction of Functional Dependencies

There are *rules* like deductive rules in logic, which say that if some functional dependencies are true, then some other functional dependency must be true as well.

Using these rules, we can find a *minimal* set of functional dependencies, equivalent to the set we started with.

## Deduction of Functional Dependencies

There are *rules* like deductive rules in logic, which say that if some functional dependencies are true, then some other functional dependency must be true as well.

Using these rules, we can find a *minimal* set of functional dependencies, equivalent to the set we started with.

This is useful for some design algorithms.

## Deduction of Functional Dependencies

There are *rules* like deductive rules in logic, which say that if some functional dependencies are true, then some other functional dependency must be true as well.

Using these rules, we can find a *minimal* set of functional dependencies, equivalent to the set we started with.

This is useful for some design algorithms.

We will do this in the next labs.

## Deduction of Functional Dependencies

There are *rules* like deductive rules in logic, which say that if some functional dependencies are true, then some other functional dependency must be true as well.

Using these rules, we can find a *minimal* set of functional dependencies, equivalent to the set we started with.

This is useful for some design algorithms.

We will do this in the next labs.

We write  $XY$  for  $X \cup Y$ , the union of attributes in  $X$  and  $Y$ .

## Transitive Rule

if  $X \longrightarrow Y$  and  $Y \longrightarrow Z$  then  $X \longrightarrow Z$

## Transitive Rule

if  $X \longrightarrow Y$  and  $Y \longrightarrow Z$  then  $X \longrightarrow Z$

Because when you compose two functions, you get another function.

## Transitive Rule

if  $X \longrightarrow Y$  and  $Y \longrightarrow Z$  then  $X \longrightarrow Z$

Because when you compose two functions, you get another function.

eg. if we have functions  $f : X \longrightarrow Y$  and  $g : Y \longrightarrow Z$  then we have a function  $g \circ f : X \longrightarrow Z$ , where  $(g \circ f)(x) = g(f(x))$ .

## Projective Rule

if  $X \longrightarrow YZ$  then  $X \longrightarrow Y$

## Projective Rule

if  $X \longrightarrow YZ$  then  $X \longrightarrow Y$

ie, if  $X$  determines  $Y$  and  $Z$ , then  $X$  determines  $Y$

## Projective Rule

if  $X \longrightarrow YZ$  then  $X \longrightarrow Y$

ie, if  $X$  determines  $Y$  and  $Z$ , then  $X$  determines  $Y$

therefore we can break each FD  $X \longrightarrow Y$  down to  $X \longrightarrow A_i$   
for  $i = 1..n$  where  $Y = \{A_1, \dots, A_n\}$

## Reflexive and Augmentation Rules

$$XY \longrightarrow X$$

## Reflexive and Augmentation Rules

$$XY \longrightarrow X$$

Because if two tuples which agree on  $X$  and  $Y$ , then they agree on  $X$ .

## Reflexive and Augmentation Rules

$$XY \longrightarrow X$$

Because if two tuples which agree on  $X$  and  $Y$ , then they agree on  $X$ .

And for the same reason,

## Reflexive and Augmentation Rules

$$XY \longrightarrow X$$

Because if two tuples which agree on  $X$  and  $Y$ , then they agree on  $X$ .

And for the same reason,

$$\text{If } X \longrightarrow Y \text{ then } XZ \longrightarrow Y$$

## Reflexive and Augmentation Rules

$$XY \longrightarrow X$$

Because if two tuples which agree on  $X$  and  $Y$ , then they agree on  $X$ .

And for the same reason,

$$\text{If } X \longrightarrow Y \text{ then } XZ \longrightarrow Y$$

[See E&N 10.2.2 about these rules]

# Keys and Functions

Recall that a function is a relation with exactly one pair for each member of the domain.

## Keys and Functions

Recall that a function is a relation with exactly one pair for each member of the domain.

Recall that declaring attributes  $X$  to be a *key* means that there can only one tuple with any given values for the attributes in  $X$ .

## Keys and Functions

Recall that a function is a relation with exactly one pair for each member of the domain.

Recall that declaring attributes  $X$  to be a *key* means that there can only one tuple with any given values for the attributes in  $X$ .

This means that *everything is functionally dependent on the key*

## Admissible States

[E&N §5.2.2] has a lot of definitions about keys, which we have not really needed till now. They are repeated in [E&N §10.3.3] because now we need them to define the higher normal forms.

## Admissible States

[E&N §5.2.2] has a lot of definitions about keys, which we have not really needed till now. They are repeated in [E&N §10.3.3] because now we need them to define the higher normal forms.

We have looked at primary keys and foreign keys. Declaring one of these is to declare a *constraint*. It restricts the range of relations we consider to be valid states of the schema.

## Admissible States

[E&N §5.2.2] has a lot of definitions about keys, which we have not really needed till now. They are repeated in [E&N §10.3.3] because now we need them to define the higher normal forms.

We have looked at primary keys and foreign keys. Declaring one of these is to declare a *constraint*. It restricts the range of relations we consider to be valid states of the schema.

That is a “pure” database view, considering just the mathematical structure, and not its connection with the real world.

## Admissible States

[E&N §5.2.2] has a lot of definitions about keys, which we have not really needed till now. They are repeated in [E&N §10.3.3] because now we need them to define the higher normal forms.

We have looked at primary keys and foreign keys. Declaring one of these is to declare a *constraint*. It restricts the range of relations we consider to be valid states of the schema.

That is a “pure” database view, considering just the mathematical structure, and not its connection with the real world.

The following definitions assume that we already have restrictions on what states are *admissible*, based on what is possible in the world the database represents.

## A Bunch of Keys - Super Key

## A Bunch of Keys - Super Key

A **subset of the attributes**  $SK$  of a relation schema is a *super key* iff for every admissible state, we have

$$\pi_{SK}(t_1) = \pi_{SK}(t_2) \longrightarrow t_1 = t_2$$

## A Bunch of Keys - Super Key

A **subset of the attributes**  $SK$  of a relation schema is a *super key* iff for every admissible state, we have

$$\pi_{SK}(t_1) = \pi_{SK}(t_2) \longrightarrow t_1 = t_2$$

- That is, if two tuples have the same primary key values, then they are the same tuple.

## A Bunch of Keys - Super Key

A **subset of the attributes**  $SK$  of a relation schema is a *super key* iff for every admissible state, we have

$$\pi_{SK}(t_1) = \pi_{SK}(t_2) \longrightarrow t_1 = t_2$$

- That is, if two tuples have the same primary key values, then they are the same tuple.
- This is usually because the things represented by the tuple are *uniquely identified* by the values of those attributes.

## A Bunch of Keys - Super Key

A **subset of the attributes**  $SK$  of a relation schema is a *super key* iff for every admissible state, we have

$$\pi_{SK}(t_1) = \pi_{SK}(t_2) \longrightarrow t_1 = t_2$$

- That is, if two tuples have the same primary key values, then they are the same tuple.
- This is usually because the things represented by the tuple are *uniquely identified* by the values of those attributes.
- Another way of putting it: **distinct** tuples always have different primary key values.

## A Bunch of Keys - Super Key

A **subset of the attributes**  $SK$  of a relation schema is a *super key* iff for every admissible state, we have

$$\pi_{SK}(t_1) = \pi_{SK}(t_2) \longrightarrow t_1 = t_2$$

- That is, if two tuples have the same primary key values, then they are the same tuple.
- This is usually because the things represented by the tuple are *uniquely identified* by the values of those attributes.
- Another way of putting it: distinct tuples always have different primary key values.
- in symbols  $t_1 \neq t_2 \longrightarrow \pi_{SK}(t_1) \neq \pi_{SK}(t_2)$

A Bunch of Keys - Key

## A Bunch of Keys - Key

A superkey  $K$  is called a *key* iff no proper subset of  $K$  is a superkey.

## A Bunch of Keys - Key

A superkey  $K$  is called a *key* iff no proper subset of  $K$  is a superkey.

That is, if you take any of the attributes out of  $K$ , then there is not enough to uniquely identify tuples.

## A Bunch of Keys - Key

A superkey  $K$  is called a *key* iff no proper subset of  $K$  is a superkey.

That is, if you take any of the attributes out of  $K$ , then there is not enough to uniquely identify tuples.

*A key is a minimal superkey.*

## A Bunch of Keys - Key

A superkey  $K$  is called a *key* iff no proper subset of  $K$  is a superkey.

That is, if you take any of the attributes out of  $K$ , then there is not enough to uniquely identify tuples.

*A key is a minimal superkey.*

Keys are also called *candidate keys*, because the *primary key* will be chosen from among them.

## A Bunch of Keys - Prime Attributes

A *prime* attribute is one that is a member of some candidate key.

## Second Normal Form

$X \longrightarrow Y$  is a *full* functional dependency iff there is no proper subset  $X'$  of  $X$  where  $X' \longrightarrow Y$ .

## Second Normal Form

$X \longrightarrow Y$  is a *full* functional dependency iff there is no proper subset  $X'$  of  $X$  where  $X' \longrightarrow Y$ .

In the example, *CITY* is functionally dependent on the key *SUPPLIER, PART*, but not *fully* functionally dependent on it.

## Second Normal Form

$X \longrightarrow Y$  is a *full* functional dependency iff there is no proper subset  $X'$  of  $X$  where  $X' \longrightarrow Y$ .

In the example, *CITY* is functionally dependent on the key *SUPPLIER, PART*, but not *fully* functionally dependent on it.

Because its dependent on *SUPPLIER*.

## Second Normal Form

$X \longrightarrow Y$  is a *full* functional dependency iff there is no proper subset  $X'$  of  $X$  where  $X' \longrightarrow Y$ .

In the example, *CITY* is functionally dependent on the key *SUPPLIER*, *PART*, but not not *fully* functionally dependent on it.

Because its dependent on *SUPPLIER*.

“A relation is in *second normal form* (2NF) if and only if it is in 1NF and every nonkey (non-prime) attribute is fully dependent on the primary key.” (Date)

## Normalisation to 2NF

- the problem is the nonkey attributes that are not fully dependent on the primary key

## Normalisation to 2NF

- the problem is the nonkey attributes that are not fully dependent on the primary key
- so we project them out with the part of the primary key they *are* dependent on

## Normalisation to 2NF

- the problem is the nonkey attributes that are not fully dependent on the primary key
- so we project them out with the part of the primary key they *are* dependent on
  - in our case  $SECOND := \pi_{SUPPLIER,CITY,STATUS}(FIRST)$

## Normalisation to 2NF

- the problem is the nonkey attributes that are not fully dependent on the primary key
- so we project them out with the part of the primary key they *are* dependent on
  - in our case  $SECOND := \pi_{SUPPLIER,CITY,STATUS}(FIRST)$
- then we don't need them in the original table, so drop them by projecting out the rest

## Normalisation to 2NF

- the problem is the nonkey attributes that are not fully dependent on the primary key
- so we project them out with the part of the primary key they *are* dependent on
  - in our case  $SECOND := \pi_{SUPPLIER,CITY,STATUS}(FIRST)$
- then we don't need them in the original table, so drop them by projecting out the rest
  - in our case  $SHIPMENT := \pi_{SUPPLIER,PART,QUANTITY}(FIRST)$

## Normalisation to 2NF

- the problem is the nonkey attributes that are not fully dependent on the primary key
- so we project them out with the part of the primary key they *are* dependent on
  - in our case  $SECOND := \pi_{SUPPLIER,CITY,STATUS}(FIRST)$
- then we don't need them in the original table, so drop them by projecting out the rest
  - in our case  $SHIPMENT := \pi_{SUPPLIER,PART,QUANTITY}(FIRST)$
- the part of the primary key that they were dependent on becomes a foreign key there

## Normalisation to 2NF

- the problem is the nonkey attributes that are not fully dependent on the primary key
- so we project them out with the part of the primary key they *are* dependent on
  - in our case  $SECOND := \pi_{SUPPLIER,CITY,STATUS}(FIRST)$
- then we don't need them in the original table, so drop them by projecting out the rest
  - in our case  $SHIPMENT := \pi_{SUPPLIER,PART,QUANTITY}(FIRST)$
- the part of the primary key that they were dependent on becomes a foreign key there
  - in our case, *SUPPLIER* becomes a foreign key in *SHIPMENT*, referencing *SECOND.SUPPLIER*

## Normalisation to 2NF

- the problem is the nonkey attributes that are not fully dependent on the primary key
- so we project them out with the part of the primary key they *are* dependent on
  - in our case  $SECOND := \pi_{SUPPLIER,CITY,STATUS}(FIRST)$
- then we don't need them in the original table, so drop them by projecting out the rest
  - in our case  $SHIPMENT := \pi_{SUPPLIER,PART,QUANTITY}(FIRST)$
- the part of the primary key that they were dependent on becomes a foreign key there
  - in our case, *SUPPLIER* becomes a foreign key in *SHIPMENT*, referencing *SECOND.SUPPLIER*
- *note: 1NF relations not in 2NF must have a composite primary key*

## 2NF Data

<u>SUPPLIER</u>	<u>STATUS</u>	<u>CITY</u>
s1	20	London
s2	10	Paris
s3	10	Paris
s4	20	London
s5	30	Athens

<u>SUPPLIER</u>	<u>PART</u>	<u>QUANTITY</u>
s1	p1	300
s1	p2	200
:	:	:
s3	p2	200
s4	p2	200
s4	p4	300
s4	p5	400

- Now we can record the location of supplier s5, even though we don't have any orders from them at the moment.

## 2NF Data

<u>SUPPLIER</u>	<u>STATUS</u>	<u>CITY</u>
s1	20	London
s2	10	Paris
s3	10	Paris
s4	20	London
s5	30	Athens

<u>SUPPLIER</u>	<u>PART</u>	<u>QUANTITY</u>
s1	p1	300
s1	p2	200
:	:	:
s3	p2	200
s4	p2	200
s4	p4	300
s4	p5	400

- Now we can record the location of supplier s5, even though we don't have any orders from them at the moment.
- A natural join on *SUPPLIER* recovers our original table *FIRST* - we have not lost anything.

## Still not good enough!

- the example is now in 2NF, but problems remain

## Still not good enough!

- the example is now in 2NF, but problems remain
- like suppliers city before, now we can not record a cities status unless we have a supplier there

## Still not good enough!

- the example is now in 2NF, but problems remain
- like suppliers city before, now we can not record a cities status unless we have a supplier there
- and we must ensure that redundant copies of this fact agree

## Still not good enough!

- the example is now in 2NF, but problems remain
- like suppliers city before, now we can not record a cities status unless we have a supplier there
- and we must ensure that redundant copies of this fact agree
- these problems arise because *STATUS* is dependent on a non-key attribute

## Still not good enough!

- the example is now in 2NF, but problems remain
- like suppliers city before, now we can not record a cities status unless we have a supplier there
- and we must ensure that redundant copies of this fact agree
- these problems arise because *STATUS* is dependent on a non-key attribute
- it is therefore *transitively* dependent on the primary key

## Still not good enough!

- the example is now in 2NF, but problems remain
- like suppliers city before, now we can not record a cities status unless we have a supplier there
- and we must ensure that redundant copies of this fact agree
- these problems arise because *STATUS* is dependent on a non-key attribute
- it is therefore *transitively* dependent on the primary key
- (notice that “fact” seems to be defined by “functional dependency” - what would the logical atomists say?)

## 3NF

“A relation is in *third normal form* (3NF) if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key”

- Our new table violates this, because *CITY*  $\longrightarrow$  *STATUS*

## 3NF

“A relation is in *third normal form* (3NF) if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key”

- Our new table violates this, because  $CITY \longrightarrow STATUS$
- and therefore  $SUPPLIER \longrightarrow CITY \longrightarrow STATUS$ , a transitive dependency on the primary key

## 3NF - Data

<u>SUPPLIER</u>	CITY
s1	London
s2	Paris
s3	Paris
s4	London
s5	Athens

*CITY* is a foreign key referencing *CITY* in the following table.

<u>CITY</u>	STATUS
Athens	30
London	20
Paris	10
Rome	50

- Now we can record the status of Rome, even though we don't have any suppliers there at the moment.

## 3NF - Data

<u>SUPPLIER</u>	CITY
s1	London
s2	Paris
s3	Paris
s4	London
s5	Athens

*CITY* is a foreign key referencing *CITY* in the following table.

<u>CITY</u>	STATUS
Athens	30
London	20
Paris	10
Rome	50

- Now we can record the status of Rome, even though we don't have any suppliers there at the moment.
- A natural join on *CITY* recovers our original table  
*SECOND* - we have not lost anything.