ST. XAVIER’S COLLEGE

**Maitighar, Kathmandu**

**(Affiliated to Tribhuvan University)**



**Database Management System**

**Theory Assignment #10**

**Submitted By**

Alok Shrestha

013BSCIT005

B.Sc. CSIT Year II/IV Semester

**Submitted To**

Er. Sanjay Kumar Yadav

Lecturer

Department of Computer Science

St. Xavier’s College

Maitighar, Kathmandu

**Submitted On**

October 2015

# Functional Dependencies

**Basic Concepts**

1. Functional dependencies.

* Functional dependencies are a constraint on the set of legal relations in a database.
* They allow us to express facts about the real world we are modeling.
* The notion generalizes the idea of a superkey.
* Let tex2html_wrap_inline1054 and tex2html_wrap_inline1056 .
* Then the functional dependency tex2html_wrap_inline1058 holds on *R* if in any legal relation *r*(*R*), for all pairs of tuples tex2html_wrap_inline940 and tex2html_wrap_inline946 in *r* such that tex2html_wrap_inline1070 , it is also the case that tex2html_wrap_inline1072 .
* Using this notation, we say *K* is a superkey of *R* if tex2html_wrap_inline1078 .
* In other words, *K* is a superkey of *R* if, whenever tex2html_wrap_inline1084 , then tex2html_wrap_inline1086 (and thus tex2html_wrap_inline1088 ).

1. Functional dependencies allow us to express constraints that cannot be expressed with superkeys.
2. Consider the scheme

*Loan-info-schema = (bname, loan#, cname, amount)*

if a loan may be made jointly to several people (e.g. husband and wife) then we would not expect *loan#* to be a superkey. That is, there is no such dependency

*loan#* tex2html_wrap_inline1090 *cname*

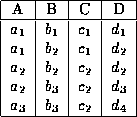
We do however expect the functional dependency

*loan#* tex2html_wrap_inline1090 *amount*

*loan#* tex2html_wrap_inline1090 *bname*

to hold, as a loan number can only be associated with one amount and one branch.

1. A set *F* of functional dependencies can be used in two ways:
   * To specify constraints on the set of legal relations. (Does *F* hold on *R*?)
   * To test relations to see if they are legal under a given set of functional dependencies. (Does *r* satisfy *F*?)
2. Figure 6.2 shows a relation *r* that we can examine.

     
**Figure 6.2:**   Sample relation *r*.

1. We can see that tex2html_wrap_inline1150 is satisfied (in this particular relation), but tex2html_wrap_inline1152 is not. tex2html_wrap_inline1154 is also satisfied.
2. Functional dependencies are called **trivial** if they are satisfied by all relations.
3. In general, a functional dependency tex2html_wrap_inline1058 is trivial if tex2html_wrap_inline1158 .
4. In the *customer* relation of figure 5.4, we see that tex2html_wrap_inline1160 is satisfied by this relation. However, as in the real world two cities can have streets with the same names (e.g. Main, Broadway, etc.), we would not include this functional dependency in our list meant to hold on *Customer-scheme*.
5. The list of functional dependencies for the example database is:
   * On *Branch-scheme:*

*bname tex2html_wrap_inline1090 bcity*

*bname tex2html_wrap_inline1090 assets*

* + On *Customer-scheme:*

*cname tex2html_wrap_inline1090 ccity*

*cname tex2html_wrap_inline1090 street*

* + On *Loan-scheme:*

*loan# tex2html_wrap_inline1090 amount*

*loan# tex2html_wrap_inline1090 bname*

* + On *Account-scheme:*

*account# tex2html_wrap_inline1090 balance*

*account# tex2html_wrap_inline1090 bname*

There are no functional dependencies for *Borrower-schema*, nor for *Depositor-schema*.

**Closure of a Set of Functional Dependencies**

1. We need to consider *all* functional dependencies that hold. Given a set *F* of functional dependencies, we can prove that certain other ones also hold. We say these ones are **logically implied** by *F*.
2. Suppose we are given a relation scheme *R*=(*A*,*B*,*C*,*G*,*H*,*I*), and the set of functional dependencies:

*A tex2html_wrap_inline1090 B*

*A tex2html_wrap_inline1090 C*

*CG tex2html_wrap_inline1090 H*

*CG tex2html_wrap_inline1090 I*

*B tex2html_wrap_inline1090 H*

Then the functional dependency tex2html_wrap_inline1194 is logically implied.

1. To see why, let tex2html_wrap_inline940 and tex2html_wrap_inline946 be tuples such that tex2html_wrap_inline1200

As we are given *A tex2html_wrap_inline1090 B*, it follows that we must also have tex2html_wrap_inline1204

Further, since we also have *B tex2html_wrap_inline1090 H*, we must also have tex2html_wrap_inline1208

Thus, whenever two tuples have the same value on *A*, they must also have the same value on *H*, and we can say that *A tex2html_wrap_inline1090 H*.

1. The **closure** of a set *F* of functional dependencies is the set of all functional dependencies logically implied by *F*.
2. We denote the closure of *F* by tex2html_wrap_inline1222 .
3. To compute tex2html_wrap_inline1222 , we can use some rules of inference called **Armstrong's Axioms**:
   * **Reflexivity rule:** if tex2html_wrap_inline958 is a set of attributes and tex2html_wrap_inline1158 , then tex2html_wrap_inline1058 holds.
   * **Augmentation rule:** if tex2html_wrap_inline1058 holds, and tex2html_wrap_inline1234 is a set of attributes, then tex2html_wrap_inline1236 holds.
   * **Transitivity rule:** if tex2html_wrap_inline1058 holds, and tex2html_wrap_inline1240 holds, then tex2html_wrap_inline1242 holds.
4. These rules are **sound** because they do not generate any incorrect functional dependencies. They are also **complete** as they generate all of tex2html_wrap_inline1222 .
5. To make life easier we can use some additional rules, derivable from Armstrong's Axioms:
   * **Union rule:** if tex2html_wrap_inline1058 and tex2html_wrap_inline1242 , then tex2html_wrap_inline1250 holds.
   * **Decomposition rule:** if tex2html_wrap_inline1250 holds, then tex2html_wrap_inline1058 and tex2html_wrap_inline1242 both hold.
   * **Pseudotransitivity rule:** if tex2html_wrap_inline1058 holds, and tex2html_wrap_inline1260 holds, then tex2html_wrap_inline1262 holds.
6. Applying these rules to the scheme and set *F* mentioned above, we can derive the following:
   * *A tex2html_wrap_inline1090 H*, as we saw by the transitivity rule.
   * *CG tex2html_wrap_inline1090 HI*by the union rule.
   * *AG tex2html_wrap_inline1090 I*by several steps:
     + Note that *A tex2html_wrap_inline1090 C*holds.
     + Then *AG tex2html_wrap_inline1090 CG*, by the augmentation rule.
     + Now by transitivity, *AG tex2html_wrap_inline1090 I*.

(You might notice that this is actually pseudotransivity if done in one step.)

**Closure of Attribute Sets**

1. To test whether a set of attributes tex2html_wrap_inline958 is a superkey, we need to find the set of attributes functionally determined by tex2html_wrap_inline958 .
2. Let tex2html_wrap_inline958 be a set of attributes. We call the set of attributes determined by tex2html_wrap_inline958 under a set *F* of functional dependencies the **closure** of tex2html_wrap_inline958 under *F*, denoted tex2html_wrap_inline1292 .
3. The following algorithm computes tex2html_wrap_inline1292 :

*result* := tex2html_wrap_inline958

**while** (changes to *result*) **do**

**for each** functional dependency tex2html_wrap_inline1240

**in**  *F* **do**

**begin**

**if**  tex2html_wrap_inline1302 *result*

**then**  *result* := *result* tex2html_wrap_inline1304

**end**

1. If we use this algorithm on our example to calculate tex2html_wrap_inline1306 then we find:
   * We start with *result* = AG.
   * *A tex2html_wrap_inline1090 B*causes us to include B in *result*.
   * *A tex2html_wrap_inline1090 C*causes *result* to become ABCG.
   * *CG tex2html_wrap_inline1090 H*causes *result* to become ABCGH.
   * *CG tex2html_wrap_inline1090 I*causes *result* to become ABCGHI.
   * The next time we execute the while loop, no new attributes are added, and the algorithm terminates.
2. This algorithm has worst case behavior quadratic in the size of *F*. There is a linear algorithm that is more complicated.

# 

# Decomposition

1. The previous example might seem to suggest that we should decompose schema as much as possible.

Careless decomposition, however, may lead to another form of bad design.

1. Consider a design where Lending-schema is decomposed into two schemas

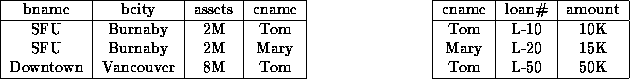
Branch-customer-schema = (bname, bcity, assets, cname)

Customer-loan-schema = (cname, loan#, amount)

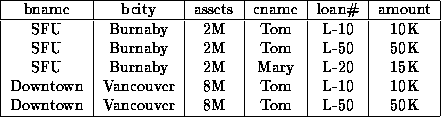
1. We construct our new relations from lending by:

branch-customer = tex2html_wrap_inline1540

customer-loan = tex2html_wrap_inline1542

  **Figure 7.2:**   The decomposed lending relation.

1. It appears that we can reconstruct the lending relation by performing a natural join on the two new schemas.
2. Figure 7.3 shows what we get by computing branch-customer tex2html_wrap_inline1544 customer-loan.

     
**Figure 7.3:**   Join of the decomposed relations.

1. We notice that there are tuples in branch-customer tex2html_wrap_inline1544 customer-loan that are not in lending.
2. How did this happen?
   * The intersection of the two schemas is cname, so the natural join is made on the basis of equality in the cname.
   * If two lendings are for the same customer, there will be four tuples in the natural join.
   * Two of these tuples will be spurious - they will not appear in the original lending relation, and should not appear in the database.
   * Although we have **more** tuples in the join, we have **less** information.
   * Because of this, we call this a **lossy** or **lossy-join decomposition**.
   * A decomposition that is not lossy-join is called a **lossless-join decomposition**.
   * The only way we could make a connection between branch-customer and customer-loan was through cname.
3. When we decomposed Lending-schema into Branch-schema and Loan-info-schema, we will not have a similar problem. Why not?

Branch-schema = (bname, bcity, assets)

Branch-loan-schema = (bname, cname, loan#, amount)

* + The only way we could represent a relationship between tuples in the two relations is through bname.
  + This will not cause problems.
  + For a given branch name, there is exactly one assets value and branch city.

1. For a given branch name, there is exactly one assets value and exactly one bcity; whereas a similar statement associated with a loan depends on the customer, not on the amount of the loan (which is not unique).
2. We'll make a more formal definition of lossless-join:
   * Let *R* be a relation schema.
   * A set of relation schemas tex2html_wrap_inline1550 is a **decomposition** of *R* if

displaymath1530

* + That is, every attribute in *R* appears in at least one tex2html_wrap_inline1556 for tex2html_wrap_inline1558 .
  + Let *r* be a relation on *R*, and let displaymath1531
  + That is, tex2html_wrap_inline1564 is the database that results from decomposing *R* into tex2html_wrap_inline1568 .
  + It is always the case that:

displaymath1532

* + To see why this is, consider a tuple tex2html_wrap_inline1570 .
    - When we compute the relations tex2html_wrap_inline1564 , the tuple *t* gives rise to one tuple tex2html_wrap_inline1576 in each tex2html_wrap_inline1578 .
    - These *n* tuples combine together to regenerate *t* when we compute the natural join of the tex2html_wrap_inline1578 .
    - Thus every tuple in *r* appears in tex2html_wrap_inline1588 .
  + However, in general, displaymath1533
  + We saw an example of this inequality in our decomposition of lending into branch-customer and customer-loan.
  + In order to have a lossless-join decomposition, we need to impose some constraints on the set of possible relations.
  + Let *C* represent a set of constraints on the database.
  + A decomposition tex2html_wrap_inline1550 of a relation schema *R* is a **lossless-join decomposition** for *R* if, for all relations *r* on schema *R* that are legal under *C*: displaymath1534

1. In other words, a lossless-join decomposition is one in which, for any legal relation *r*, if we decompose *r* and then ``recompose'' *r*, we get what we started with - no more and no less.

# 

# Normalization Using Functional Dependencies

We can use functional dependencies to design a relational database in which most of the problems we have seen do not occur.

Using functional dependencies, we can define several **normal forms** which represent ``good'' database designs.

**Desirable Properties of Decomposition**

1. We'll take another look at the schema

*Lending-schema = (bname, assets, bcity, loan#, cname, amount)*

which we saw was a bad design.

1. The set of functional dependencies we required to hold on this schema was:

*bname* tex2html_wrap_inline1526 *assets bcity*

*loan#* tex2html_wrap_inline1526 *amount bname*

1. If we decompose it into

*Branch-schema = (bname, assets, bcity)*

*Loan-info-schema = (bname, loan#, amount)*

*Borrow-schema = (cname, loan#)*

we claim this decomposition has several desirable properties.

### Lossless-Join Decomposition

1. We claim the above decomposition is lossless. How can we decide whether a decomposition is lossless?
   * Let *R* be a relation schema.
   * Let *F* be a set of functional dependencies on *R*.
   * Let tex2html_wrap_inline1620 and tex2html_wrap_inline1622 form a decomposition of *R*.
   * The decomposition is a lossless-join decomposition of *R* if at least one of the following functional dependencies are in tex2html_wrap_inline1628 :
     1. tex2html_wrap_inline1630
     2. tex2html_wrap_inline1632

Why is this true? Simply put, it ensures that the attributes involved in the natural join ( tex2html_wrap_inline1634 ) are a candidate key for at least one of the two relations.

This ensures that we can never get the situation where spurious tuples are generated, as for any value on the join attributes there will be a unique tuple in **one** of the relations.

1. We'll now show our decomposition is lossless-join by showing a set of steps that generate the decomposition:
   * First we decompose Lending-schema into

Branch-schema = (bname, bcity, assets)

Loan-info-schema = (bname, cname, loan#, amount)

* + Since bname tex2html_wrap_inline1526 assets bcity, the augmentation rule for functional dependencies implies that

bname tex2html_wrap_inline1526 bname assets bcity

* + Since Branch-schema tex2html_wrap_inline1640 Borrow-schema = bname, our decomposition is lossless join.
  + Next we decompose Borrow-schema into

Loan-schema = (bname, loan#, amount)

Borrow-schema = (cname, loan#)

* + As loan# is the common attribute, and

loan# tex2html_wrap_inline1526 amount bname

This is also a lossless-join decomposition.

### 

### Dependency Preservation

1. Another desirable property in database design is **dependency preservation**.
   * We would like to check easily that updates to the database do not result in illegal relations being created.
   * It would be nice if our design allowed us to check updates without having to compute natural joins.
   * To know whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually.
   * Let *F* be a set of functional dependencies on schema *R*.
   * Let tex2html_wrap_inline1550 be a decomposition of *R*.
   * The **restriction** of *F* to tex2html_wrap_inline1556 is the set of all functional dependencies in tex2html_wrap_inline1628 that include only attributes of tex2html_wrap_inline1556 .
   * Functional dependencies in a restriction can be tested in one relation, as they involve attributes in one relation schema.
   * The set of restrictions tex2html_wrap_inline1660 is the set of dependencies that can be checked efficiently.
   * We need to know whether testing only the restrictions is sufficient.
   * Let tex2html_wrap_inline1662 .
   * *F*' is a set of functional dependencies on schema *R*, but in general, tex2html_wrap_inline1668.
   * However, it may be that tex2html_wrap_inline1670 .
   * If this is so, then every functional dependency in *F* is implied by *F*', and if *F*' is satisfied, then *F* must also be satisfied.
   * A decomposition having the property that tex2html_wrap_inline1670 is a **dependency-preserving** decomposition.
2. The algorithm for testing dependency preservation follows this method:

compute tex2html_wrap_inline1628

**for each** schema tex2html_wrap_inline1556 in *D* **do**

**begin**

tex2html_wrap_inline1688 := the restriction of tex2html_wrap_inline1628 to tex2html_wrap_inline1556 ;

**end**

tex2html_wrap_inline1694

**for each** restriction tex2html_wrap_inline1688 **do**

**begin**

tex2html_wrap_inline1698

**end**

compute tex2html_wrap_inline1700 ;

**if** ( tex2html_wrap_inline1670 ) **then** return (true)

**else** return (false);

1. We can now show that our decomposition of Lending-schema is dependency preserving.
   * The functional dependency

bname tex2html_wrap_inline1526 assets bcity

can be tested in one relation on Branch-schema.

* + The functional dependency

loan# tex2html_wrap_inline1526 amount bname

can be tested in Loan-schema.

1. As the above example shows, it is often easier not to apply the algorithm shown to test dependency preservation, as computing tex2html_wrap_inline1628 takes exponential time.
2. **An Easier Way To Test For Dependency Preservation**

Really we only need to know whether the functional dependencies in *F* and not in *F*' are implied by those in *F*'.

In other words, are the functional dependencies not easily checkable logically implied by those that are?

Rather than compute tex2html_wrap_inline1628 and tex2html_wrap_inline1700 , and see whether they are equal, we can do this:

* + Find *F* - *F*', the functional dependencies not checkable in one relation.
  + See whether this set is obtainable from *F*' by using Armstrong's Axioms.
  + This should take a great deal less work, as we have (usually) just a few functional dependencies to work on.

Use this simpler method on exams and assignments (unless you have exponential time available to you).