**ST. XAVIER’S COLLEGE**

(Affiliated to Tribhuwan University)

Maitighar, Kathmandu

****

**database management system**

**LAb ASSIGNMENT #10**

**SUBMITTED BY:**

Dikita Tuladhar

013BSCIT018

4th Semester/2nd Year

**SUBMITTED TO:**

|  |  |
| --- | --- |
| Er. Sanjay Kumar Yadhav  Lecturer  Department of Computer Science  St. Xavier’s College, Maitighar |  |

**Date of Submission**: 8th October, 2015

1. **Functional Dependencies**
   1. **Basic Concepts**

Functional dependency is a relationship that exists when one attribute uniquely determines another attribute. If R is a relation with attributes X and Y, a functional dependency between the attributes is represented as X->Y, which specifies Y is functionally dependent on X.

In relational database theory, a functional dependency is a constraint between two sets of attributes in a relation from a database.

Given a relation R, a set of attributes X in R is said to functionally determine another set of attributes Y, also in R, (written X → Y) if, and only if, each X value is associated with precisely one Y value; R is then said to satisfy the functional dependency X → Y. Equivalently, the projection \pi\_{X,Y}R is a function, i.e. Y is a function of X.[1][2] In simple words, if the values for the X attributes are known (say they are x), then the values for the Y attributes corresponding to x can be determined by looking them up in any tuple of R containing x. Customarily X is called the determinant set and Y the dependent set. A functional dependency FD: X → Y is called trivial if Y is a subset of X.

In other words, a dependency FD: X → Y means that the values of Y are determined by the values of X. Two tuples sharing the same values of X will necessarily have the same values of Y.

Functional dependency in a database serves as a constraint between two sets of attributes. Defining functional dependency is an important part of relational database design and contributes to aspect normalization.

* 1. **Closure of a Set of Functional Dependencies**

"The closure of F, denoted as F+, is the set of all regular Functional Dependencies that can be derived from F".

This is used to discover some of the hidden functional dependencies so as to design a better database.

Consider the Armstrong's axioms developed by William W.Armstrong. Those axioms provide simpler technique to identify set of other functional dependencies (hidden). Let us list all the axioms along with the additional rules once again.

1. Reflexivity rule

If X is a set of attributes, and Y is subset of X, then we would say, X → Y.

1. Augmentation rule

If X → Y, and Z is another set of attributes, then we write XZ → XY.

1. Transitivity rule

If X → Y, and Y → Z, then X → Z is true.

1. Union rule

If X → Y and X → Z, then X → YZ is true.

1. Decomposition rule

Its reverse of Rule 4. If X → YZ, then X → Y, and X → Z are true

1. Pseudo transitivity rule

If X → Y and ZY → A are true, then XZ → A is also true.

Let us consider set F of functional dependencies hold on a relation R. We can derive additional functional dependencies from the set of given functional dependencies. But, still we may have some more hidden functional dependencies. We could derive some of the additional hidden functional dependencies from F on applying the above listed rules. In other words, we could deduce a new functional dependency from two or more functional dependencies of set F. Suppose, R is a relation with attributes (A, B, C, D, E, F) and with the identified set F of functional dependencies as follows;

F = { A → B, A → C, CD → E, B → E, CD → F }

Let us apply the above listed rules on every functional dependency of F to identify the member of F+ (the closure of functional dependency is termed as F+, i.e, set F added with new members resulting in F+).

1. A → E is logically implied. From our F we have two FDs A → B and B → E. By applying Transitivity rule, we could infer A → E. That is, if A can uniquely determine B and B can uniquely determine E then A can determine E (through B).

2. A → BC is logically implied. It can be inferred from the FDs A → B and A → C using Union rule.

3. CD → EF is logically implied by FDs CD → E and CD → F using Union rule.

4. AD → F is logically implied by FDs A → C and CD → F using Pseudo transitivity rule.

Like this, we can continue identifying the new members for F+. The procedure to find set F+ can be written as follows in pseudo code.

|  |
| --- |
| F+ = F  Repeat  For each functional dependency FD in F+  Apply reflexivity and augmentation rules on f  Add the result to F+  For each pair of functional dependencies f1 and f2 in F+  If f1 and f2 can be combined using Transitivity  Add the result to F+  Until F+ does not change any further. |

This procedure does not show the additional rules Union, Decomposition, and Pseudo transitivity. The reason is, these additional rules are actually inferred from basic axioms. Also additional rules can be proved using Armstrong’s axioms. The procedure can be stopped when we started to get the functional dependencies already existing in F+.

For a relation with set of n attributes, there are 2n+1 possible functional dependencies. For example, if R has 3 attributes, then 23+1 = 16 functional dependencies are possible.

* 1. **Closure of Attribute Sets**

Given a set α of attributes of R and a set of functional dependencies F, we need a way to find all of the attributes of R that are functionally determined by α. This set of attributes is called the closure of α under F and is denoted α +. Finding α + is useful because:

• if α + = R, then α is a superkey for R

• if we find α + for all α ⊆ R, we've computed F+ (except that we'd need to use decomposition to get all of it).

An algorithm for computing α +:

result := α

repeat

temp := result

for each functional dependency **β → γ** in F do

if **β**  ⊆ result then

result := result **∪ γ**

until temp = result

1. **Decomposition**
   1. **Lossless-Join Dependencies**

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 R1 and R2 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 F+:

R1 ᶸ R2 -> R1

R1 ∩ R2 -> R2

Why is this true? Simply put, it ensures that the attributes involved in the natural join (R1 ∩ R2) 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.

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.

* 1. Dependency Preservation

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.

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);

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*.

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.

**An Easier Way To Test For Dependency Preservation**

Really we only need to know whether the functional dependencies in *F* and not ed 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.