ST. XAVIER’S COLLEGE

**(Affiliated to Tribhuvan University)**

**Maitighar, Kathmandu**

****

**Database Management System**

**Theory Assignment #10**

**SUBMITTED BY:**

Sneha Prasai

013BSCCSIT040

**SUBMITTED TO**

|  |  |
| --- | --- |
| **Er. Sanjay Kr. Yadav**  **( Lecturer )** |  |
| **Department of Computer Science** | |

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

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 super key.

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 super key of *R* if tex2html_wrap_inline1078 .

In other words, *K* is a super key of *R* if, whenever tex2html_wrap_inline1084 , then tex2html_wrap_inline1086 (and thus tex2html_wrap_inline1088 ).

Functional dependencies allow us to express constraints that cannot be expressed with super keys.

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.

* 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*?)

Functional dependencies are called **trivial** if they are satisfied by all relations.

In general, a functional dependency tex2html_wrap_inline1058 is trivial if tex2html_wrap_inline1158 .

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

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*.

The **closure** of a set *F* of functional dependencies is the set of all functional dependencies logically implied by *F*.

We denote the closure of *F* by tex2html_wrap_inline1222 .

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.

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 .

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.
  1. **Closure of Attribute Sets**

To test whether a set of attributes tex2html_wrap_inline958 is a superkey, we need to find the set of attributes functionally determined by A .

Let A be a set of attributes. We call the set of attributes determined by A under a set *F* of functional dependencies the **closure** of A under *F*, denoted A+ .

The following algorithm will help us in finding the closure of an attribute;

*result :=* A*;*

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

***for each*** *functional dependency* B →C

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

***if*** B ⊆ *result*

***then*** *result := result* ∪ C*;*

***end***

Let us discuss this algorithm with an example;

Assume a relation schema R = (A, B, C) with the set of functional dependencies F = {A → B, B →C}. Now, we can find the attribute closure of attribute A as follows;

Step 1: We start with the attribute in question as the initial result. Hence, result = A.

Step 2: Take the first FD A → B. Its left hand side (i.e, A) is in the result, hence the right hand side can be included with the result. This lead to result = AB.

Step 3: Take the second FD B → C. Its left hand side (i.e, B) is in the result (or subset of result), hence the right hand side can be included with the result. Now, result = ABC.

We have no more attributes. Hence the algorithm exits. As the result, A+ includes all the attributes of relation R. now we would say A+ is ABC. And, A is one of the keys of the relation R

1. **Decomposition**

It is the process of breaking down in parts or elements. Decomposing is the act of breaking tables down in order to achieve higher normal form. Decompositions should always be lossless. This confirms that information in the original relation can be accurately reconstructed based on the decomposed relations. Decomposition should be considered “GOOD” it must also preserve functional dependencies.

* 1. **Lossless- Join Decomposition**

Lossless means functioning without a loss. In other words, retain everything. It is important for databases to have this feature.

**Definition:**

Let { R1, R2 } be a decomposition of R (R1 U R2 = R); the decomposition is lossless if for every legal instance r of R:

r = ΠR1(r) |X| ΠR2(r)

Lossless join property is necessary if the decomposed relation is to be recovered from its decomposition.

Let R be a schema and F be a set of FD’s on R, and α = (R1, R2) be a decomposition of R. Then α has a lossless join with respect to F if

R1 ∩ R2 -> R1 (or R1 - R2 ) or

R2 ∩ R1 -> R2 (or R2 - R1 )

where such FD exist in Closure of F

Note: This is a sufficient condition, but not a necessary condition.

Example:

R = (City, Street, Zip) F = {CS -> Z, Z -> C}

R1 = (CZ) R2 = (SZ)

R1 ∩ R2 = Z , R1 – R2 = (SZ)

check Z -> C in F ? Yes

Therefore, the decomposition to be (CZ) (SZ) is

*lossless join decomposition.*

* 1. **Dependency Preservation**

Getting lossless decomposition is necessary. But of course, we also want to keep dependencies, since losing a dependency means, that the corresponding constraint can be check only through natural join of the appropriate resultant relation in the decomposition. This would be very expensive, so, our aim is to get a lossless dependency preserving decomposition.

**Definition:**

A decomposition D = {R1, …, Rm} of R is dependency-preserving with respect to a set F of FDs

if (F1 ∪ … ∪ Fm)+ = F+,

Where Fi means the projection of the dependency set F onto Ri.

Fi =Π Ri(F+)

denotes a set of FDs X → Y in F+ such that all attributes in X ∪ Y are contained in Ri:

Fi=Π Ri(F+) ={ X→Y| {X,Y}⊆ Ri and X→Y ∈ F+ }

**Testing:**

* If decomposition is not dependency-preserving, some dependency is lost in the decomposition.
* One way to verify that a dependency is not lost is to take joins of two or more relations in the decomposition to get a relation that contains all of the attributes in the dependency under consideration and then check that the dependency holds on the result of the joins.
* 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.

**Example 1:**

Consider relation ABCD, with FD’s :

A ->B, B ->C, C ->D

* + Decompose into two relations: ABC and CD.
  + ABC supports the FD’s A->B, B->C.
  + CD supports the FD C->D.
  + All the original dependencies are preserved

**Example 2:**

Consider relation ABCD, with FD’s:

A ->B, B ->C, C->D

* + Decompose into two relations: ACD and BC.
  + ACD supports the FD B ->C and implied FD A ->C.
  + BC supports the FD B->C.
  + However, no relation supports A ->B.
  + So the dependency is not preserved.