COP 4710: Database Systems
Spring 2004

-Day 11 – February 11, 2004 –
Introduction to Normalization – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop4710/spr2004

School of Electrical Engineering and Computer Science
University of Central Florida

Third Normal Form (3NF)

- Third Normal Form (3NF) is based on the concept of a transitive dependency.
- Given a relation scheme R with a set of functional dependencies F and subset $X \subseteq R$ and an attribute $A \in R$. A is said to be transitively dependent on X if there exists $Y \subseteq R$ with $X \rightarrow Y$, $Y\ X \not\rightarrow X$ and $Y \rightarrow A$ and $A \notin X \cup Y$.
- An alternative definition for a transitive dependency is: a functional dependency $X \rightarrow Y$ in a relation scheme R is a transitive dependency if there is a set of attributes $Z \subseteq R$ where Z is not a subset of any key of R and yet both $X \rightarrow Z$ and $Z \rightarrow Y$ hold in F.

Third Normal Form (3NF) (cont.)

- A relation scheme R is in 3NF with respect to a set of functional dependencies F, if whenever $X \rightarrow A$ holds either: (1) X is a superkey of R or (2) A is a prime attribute.
- Alternative definition: A relation scheme R is in 3NF with respect to a set of functional dependencies F if no non-prime attribute is transitively dependent on any key of R.

Example: Let R = (A, B, C, D)

$$K = \{AB\}, F = \{AB \rightarrow CD,\ C \rightarrow D,\ D \rightarrow C\}$$

then R is not in 3NF since $C \rightarrow D$ holds and C is not a superkey of R.

Alternatively, R is not in 3NF since $AB \rightarrow C$ and $C \rightarrow D$ and thus D is a non-prime attribute which is transitively dependent on the key AB.

Why Third Normal Form?

- What does 3NF do for us? Consider the following database:

  assign(flight, day, pilot-id, pilot-name)
  K = {flight day}
  F = {pilot-id → pilot-name, pilot-name → pilot-id}

flight	day	pilot-id	pilot-name
112	Feb.11	317	Mark
112	Feb. 12	246	Kristi
114	Feb.13	317	Mark

Why Third Normal Form? (cont.)

flight	day	pilot-id	pilot-name
112	Feb.11	317	Mark
112	Feb. 12	246	Kristi
114	Feb.13	317	Mark
112	Feb. 11	319	Mark

Since {flight day} is key, clearly {flight day} $\rightarrow$ pilot-name. But in F we also know that pilot-name $\rightarrow$ pilot-id, and we have that {flight day} $\rightarrow$ pilot-id.

Now suppose the highlighted tuple is added to this instance. is added. The fd pilot-name $\rightarrow$ pilot-id is violated by this insertion. A transitive dependency exists since: pilot-id $\rightarrow$ pilot-name holds and pilot-id is not a superkey.

Boyce-Codd Normal Form (BCNF)

- Boyce-Codd Normal Form (BCNF) is a more stringent form of 3NF.
- A relation scheme R is in Boyce-Codd Normal Form with respect to a set of functional dependencies F if whenever $X \rightarrow A$ hold and A ? X, then X is a superkey of R.

Example: Let R = (A, B, C)

$F = \{AB \rightarrow C,\ C \rightarrow A\}$

$K = \{AB\}$

R is not in BCNF since $C \rightarrow A$ holds and C is not a superkey of R.

Boyce-Codd Normal Form (BCNF) (cont.)

- Notice that the only difference in the definitions of 3NF and BCNF is that BCNF drops the allowance for A in $X \rightarrow A$ to be prime.
- An interesting side note to BCNF is that Boyce and Codd originally intended this normal form to be a simpler form of 3NF. In other words, it was supposed to be between 2NF and 3NF. However, it was quickly proven to be a more strict definition of 3NF and thus it wound up being between 3NF and 4NF.
- In practice, most relational schemes that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in the schema where X is not a superkey and A is prime, will the schema be in 3NF but not in BCNF.

Moving Towards Relational Decomposition

- The basic goal of relational database design should be to ensure that every relation in the database is either in 3NF or BCNF.
- 1NF and 2NF do not remove a sufficient number of the update anomalies to make a significant difference, whereas 3NF and BCNF eliminate most of the update anomalies.
- As we've mentioned before, in addition to ensuring the relation schemas are in either 3NF or BCNF, the designer must also ensure that the decomposition of the original database schema into the 3NF or BCNF schemas guarantees that the decomposition have (1) the lossless join property (also called a non-additive join property) and (2) the functional dependencies are preserved across the decomposition.

Moving Towards Relational Decomposition (cont.)

- There are decomposition algorithms that will guarantee a 3NF decomposition which ensures both the lossless join property and preservation of the functional dependencies.
- However, there is no algorithm which will guarantee a BCNF decomposition which ensures both the lossless join property and preserves the functional dependencies. There is an algorithm that will guarantee BCNF and the lossless join property, but this algorithm cannot guarantee that the dependencies will be preserved.
- It is for this reason that many times, 3NF is as strong a normal form as will be possible for a certain set of schemas, since an attempt to force BCNF may result in the non-preservation of the dependencies.
- In the next few pages we'll look at these two properties more closely.

Preservation of the Functional Dependencies

- Whenever an update is made to the database, the DBMS must be able to verify that the update will not result in an illegal instance with respect to the functional dependencies in F^+.
- To check updates in an efficient manner the database must be designed with a set of schemas which allows for this verification to occur without necessitating join operations.
- If an fd is "lost", the only way to enforce the constraint would be to effect a join of two or more relations in the decomposition to get a "relation" that includes all of the determinant and consequent attributes of the lost fd into a single table, then verify that the dependency still holds after the update occurs. Obviously, this requires too much effort to be practical or efficient.

Preservation of the Functional Dependencies (cont.)

- Informally, the preservation of the dependencies means that if $X \rightarrow Y$ from F appears either explicitly in one of the relational schemas in the decomposition scheme or can be inferred from the dependencies that appear in some relational schema within the decomposition scheme, then the original set of dependencies would be preserved on the decomposition scheme.

- It is important to note that what is required to preserve the dependencies is not that every fd in F be explicitly present in some relation schema in the decomposition, but rather the union of all the dependencies that hold on all of the individual relation schemas in the decomposition be equivalent to F (recall what equivalency means in this context).

Preservation of the Functional Dependencies (cont.)

- The projection of a set of functional dependencies onto a set of attributes Z, denoted F[Z] (also sometime as $\pi_Z(F)$), is the set of functional dependencies $X \rightarrow Y$ in F^+ such that $X \cup Y \subseteq Z$.

- A decomposition scheme $\gamma = \{R_1, R_2, \ldots, R_m\}$ is dependency preserving with respect to a set of fds F if the union of the projection of F onto each R_i $(1 \leq i \leq m)$ in γ is equivalent to F.

$$(F[R_1] \cup F[R_2] \cup \ldots \cup F[R_m])^+ = F^+$$

Preservation of the Functional Dependencies (cont.)

- It is **always** possible to find a dependency preserving decomposition scheme D with respect to a set of fds F such that each relation schema in D is in 3NF.
- In a few pages, we will see an algorithm that guarantees a 3NF decomposition in which the dependencies are preserved.

Algorithm for Testing the Preservation of Dependencies

Algorithm Preserve
// input: a decomposition D= (R_1, R_2, …, R_k), a set of fds F, an fd $X \rightarrow Y$
// output: true if D preserves F, false otherwise

Preserve (D , F, $X \rightarrow Y$)
  Z = X;
  while (changes to Z occur) do
    for i = 1 to k do // there are k schemas in D
      $Z = Z \cup ((Z \cap R_i)^+ \cap R_i)$
    endfor;
  endwhile;
  if $Y \subseteq Z$
    then return true; // $Z \stackrel{?}{\rightarrow} X \rightarrow Y$
    else return false;
end.

How Algorithm Preserves Works

- The set Z which is computed is basically the following: $G = \bigcup_{i=1}^{k} F[R_i]$
- Note that G is not actually computed but merely tested to see if G covers F. To test if G covers F we need to consider each fd $X \rightarrow Y$ in F and determine if X_G^+ contains Y.
- Thus, the technique is to compute X_G^+ without having G available by repeatedly considering the effect of closing F with respect to the projections of F onto the various R_i.

A Hugmongously Big Example

Let R = (A, B, C, D)

F = {A→B, B→C, C→D, D→A}

D = {(AB), (BC), (CD)}

$G = F[AB] \cup F[BC] \cup F[CD]$ $Z = Z \cup ((Z \cap R_i)^+ \cap R_i)$

Test for each fd in F.

Test for A→B

Z = A,

$= \{A\} \cup ((A \cap AB)^+ \cap AB)$

$= \{A\} \cup ((A)^+ \cap AB)$

$= \{A\} \cup (ABCD \cap AB)$

$= \{A\} \cup \{AB\}$

$= \mathbf{\{AB\}}$

A Hugmongously Big Example (cont.)

$Z = \{AB\}$
$= \{AB\} \cup ((AB \cap BC)^+ \cap BC)$
$= \{AB\} \cup ((B)^+ \cap BC)$
$= \{AB\} \cup (BCDA \cap BC)$
$= \{AB\} \cup \{BC\}$
$= \mathbf{\{ABC\}}$

$Z = \{ABC\}$
$= \{ABC\} \cup ((ABC \cap CD)^+ \cap CD)$
$= \{ABC\} \cup ((C)^+ \cap CD)$
$= \{ABC\} \cup (CDAB \cap CD)$
$= \{ABC\} \cup \{CD\}$
$= \mathbf{\{ABCD\}}$

G covers A ® B

A Hugmongously Big Example (cont.)

Test for $B \rightarrow C$

$Z = B,$
$= \{B\} \cup ((B \cap AB)^+ \cap AB)$
$= \{B\} \cup ((B)^+ \cap AB)$
$= \{B\} \cup (BCDA \cap AB)$
$= \{B\} \cup \{AB\}$
$= \mathbf{\{AB\}}$

$Z = \{AB\}$
$= \{AB\} \cup ((AB \cap BC)^+ \cap BC)$
$= \{AB\} \cup ((B)^+ \cap BC)$
$= \{AB\} \cup (BCDA \cap BC)$
$= \{AB\} \cup \{BC\}$
$= \mathbf{\{ABC\}}$

$Z = \{ABC\}$
$= \{ABC\} \cup ((ABC \cap CD)^+ \cap CD)$
$= \{ABC\} \cup ((C)^+ \cap CD)$
$= \{ABC\} \cup (CDAB \cap CD)$
$= \{ABC\} \cup \{CD\}$
$= \mathbf{\{ABC\}}$

So G covers B ® C

A Hugmongously Big Example (cont.)

Test for C→D

Z = C,
= {C} ∪ ((C ∩ AB)$^+$ ∩ AB)
= {C} ∪ ((∅)$^+$ ∩ AB)
= {C} ∪ (∅)
= **{C}**

Z = {C}
= {C} ∪ ((C ∩ BC)$^+$ ∩ BC)
= {C} ∪ ((C)$^+$ ∩ BC)
= {C} ∪ (CDAB ∩ BC)
= {C} ∪ {BC}
= **{BC}**

Z = {BC}
= {BC} ∪ ((BC ∩ CD)$^+$ ∩ CD)
= {BC} ∪ ((C)$^+$ ∩ CD)
= {BC} ∪ (CDAB ∩ CD)
= {BC} ∪ {CD}
= **{BCD}** **So G covers C ® D**

A Hugmongously Big Example (cont.)

Test for $D \rightarrow A$

$Z = D,$
$= \{D\} \cup ((D \cap AB)^+ \cap AB)$
$= \{D\} \cup ((\varnothing)^+ \cap AB)$
$= \{D\} \cup (\varnothing)$
$= \mathbf{\{D\}}$

$Z = \{D\}$
$= \{D\} \cup ((D \cap BC)^+ \cap BC)$
$= \{D\} \cup ((\varnothing)^+ \cap BC)$
$= \{D\} \cup (\varnothing)$
$= \mathbf{\{D\}}$

$Z = \{D\}$
$= \{D\} \cup ((D \cap CD)^+ \cap CD)$
$= \{D\} \cup ((D)^+ \cap CD)$
$= \{D\} \cup (DABC \cap CD)$
$= \{D\} \cup \{CD\}$
$= \mathbf{\{DC\}}$

Changes made to G so continue.

A Hugmongously Big Example (cont.)

Test for D→A continues on a second pass through D.

Z = DC,
= {DC} ∪ ((DC ∩ AB)$^+$ ∩ AB)
= {DC} ∪ ((∅)$^+$ ∩ AB)
= {DC} ∪ (∅)
= **{DC}**

Z = {DC}
= {DC} ∪ ((DC ∩ BC)$^+$ ∩ BC)
= {DC} ∪ ((C)$^+$ ∩ BC)
= {D} ∪ (CDAB ∩ BC)
= {D} ∪ (BC)
= **{DBC}**

Z = {DBC}
= {DBC} ∪ ((DBC ∩ CD)$^+$ ∩ CD)
= {DBC} ∪ ((CD)$^+$ ∩ CD)
= {DBC} ∪ (CDAB ∩ CD)
= {DBC} ∪ {CD}
= **{DBC}**

Again changes made to G so continue.

A Hugmongously Big Example (cont.)

Test for D→A continues on a third pass through D.

Z = DBC,

= {DBC} ∪ ((DBC ∩ AB)$^+$ ∩ AB)

= {DBC} ∪ ((B)$^+$ ∩ AB)

= {DBC} ∪ (BCDA ∩ AB)

= {DBC} ∪ (AB)

= **{DBCA}**

Finally, we've included every attribute in R.
Thus, G covers D →A.

Thus, D preserves the functional dependencies in F.

Practice Problem: Determine if D preserves the dependencies in F given:

R = (C, S, Z)

F = {CS →Z, Z→C}

D = {(SZ0, (CZ)}

Solution in next set of notes!

Algorithm for Testing for the Lossless Join Property

Algorithm Lossless
// input: a relation schema R= (A_1, A_2, …, A_n), a set of fds F, a decomposition
// scheme D = {R_1, R_2, ..., R_k)
// output: true if D has the lossless join property, false otherwise

Lossless (R, F, D)
Create a matrix of *n* columns and *k* rows where column *y* corresponds to attribute A_y ($1 \leq y \leq n$) and row *x* corresponds to relation schema R_x ($1 \leq x \leq k$). Call this matrix T.

Fill the matrix according to: in T_{xy} put the symbol a_y if A_y is in R_x and the symbol b_{xy} if not.

Repeatedly "consider" each fd $X \rightarrow Y$ in F until no more changes can be made to T.
Each time an fd is considered, look for rows in T which agree on all of the columns corresponding to the attributes in X. Equate all of the rows which agree in the X value on the Y values according to: If any of the Y symbols is a_y make them all a_y, if none of them are a_y equate them arbitrarily to one of the b_{xy} values.

If after making all possible changes to T one of the rows has become $a_1a_2...a_n$ then return yes, otherwise return no.
end.

Testing for a Lossless Join - Example

Let R = (A, B, C, D, E)

F = {A→C, B→C, C→D, DE→C, CE→A}

D = {(AD), (AB), (BE), (CDE), (AE)}

initial matrix T:

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{23}	b_{24}	b_{25}
(BE)	b_{31}	a_2	b_{33}	b_{34}	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	b_{53}	b_{54}	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

$A \rightarrow C$: equates b_{13}, b_{23}, b_{53}. Arbitrarily we'll set them all to b_{13} as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	b_{24}	b_{25}
(BE)	b_{31}	a_2	b_{33}	b_{34}	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	b_{13}	b_{54}	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

B→C: equates b_{13}, b_{33}.. We'll set them all to b_{13} as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	b_{24}	b_{25}
(BE)	b_{31}	a_2	b_{13}	b_{34}	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	b_{13}	b_{54}	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

C→D: equates a_4, b_{24}, b_{34}, b_{54}. We set them all to a_4 as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	a_4	b_{25}
(BE)	b_{31}	a_2	b_{13}	a_4	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	b_{13}	a_4	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

DE→C: equates a_3, b_{13}. We set them both to a_3 as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	a_4	b_{25}
(BE)	b_{31}	a_2	a_3	a_4	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	a_3	a_4	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

CE→A: equates b_{31}, b_{41}, $a_{1.}$. We set them all to a_1 as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	a_4	b_{25}
(BE)	a_1	a_2	a_3	a_4	a_5
(CDE)	a_1	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	a_3	a_4	a_5

Testing for a Lossless Join – Example (cont.)

First pass through F is now complete. However row (BE) has become all a_is, so stop and return true, this decomposition has the lossless join property.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	a_4	b_{25}
(BE)	a_1	a_2	a_3	a_4	a_5
(CDE)	a_1	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	a_3	a_4	a_5