Relational Database Design

Notice

(Welcome back from Quiet Week)

Project 1: Due on the April 1st (not an April fools' joke)

Clarification on Project Late Penalties

From The Previous Lecture

Anomalies can be removed from relation designs by decomposing them until they are in a normal form.

Decomposition

Definition (**Decomposition**): A decomposition of a relation scheme, R, is a set of relation schemes $\{R_1, \ldots, R_n\}$ such that $R_i \subseteq R$ for each i, and $\bigcup_{i=1}^{n} R_i = R$

On Decompositions

Important: it is improper to assess the quality of your decompositions by independently checking to see if the resulting relations are in a higher form.

A good decomposition should also have the following two properties.

1. the dependency preservation property
2. the nonadditive (or lossless) join property

Together, they gives us desirable decompositions

Goal of Relational Database Design

Goal for a relational database design is:

- BCNF.
- Lossless join.
- Dependency preservation.

If we cannot achieve this, we accept one of

- Lack of dependency preservation
- Redundancy due to use of 3NF

Property	3NF	BCNF
Elimination of redundancy due to functional dependency	Most	Yes
Lossless Join	Yes	Yes
Dependency preservation due to functional dependency	Yes	Maybe

Projection of F

Given a set of initial dependencies *F* on *R*:

Let R be decomposed in to R_i … R_m

Definition (Projection): The **projection** of *F* on R_i, denoted by $\pi_{R_i}(F)$ where R_i is a subset of *R*, is the set of dependencies *X* → *Y* in *F*+ such that the attributes in *X* ∪ *Y* are all contained in R_i.

To simplify notations, we also denote the projection of F on R_i as F_i.

In simple English: F_i is the set of dependencies F^+ that include only attributes in R_i.
(Hence a projection of F)

Projection of F Example

Definition (Projection): The **projection** of *F* on R_i, denoted by $\pi_{R_i}(F)$ where R_i is a subset of *R*, is the set of dependencies *X* → *Y* in *F*+ such that the attributes in $X \cup Y$ are all contained in R_i.

Example

R = (A, B, C, D, E, G, M)

F = { A → BC, D → EG, M → A }

What are the projections of R1 and R2?

R_1 = (A, B, C, M) and R_2 = (C, D, E, G)

π_{R_1} = { A → BC, M → A}, π_{R_2} = {D → EG} (Projections of R1 and R2)

(Can be similarly denoted as F_1 = { A → BC, M → A}, F_2 = {D → EG})

Dependency Preservation

A property that a decomposition *D* should possess is the dependency preservation property.

Given a set of dependencies *F* on *R*:

Recall (Projection): The **projection** of *F* on R_i, denoted by $\pi_{R_i}(F)$ where R_i is a subset of *R*, is the set of dependencies $X \rightarrow Y$ in *F*+ such that the attributes in $X \cup Y$ are all contained in R_i.

(Dependency Preservation) We say that a decomposition
$D = \{R_1, R_2, ..., R_m\}$ of *R*
is dependency-preserving with respect to *F on R* if the union of the projections of *F* on each R_i in *D* is equivalent to *F*

that is, $((\pi_{R_1}(F)) \cup ... \cup (\pi_{R_m}(F)))+ = F+$.

Comment

Drawback of Not Dependency Preserving Decomposition

We established previously that functional dependencies must be enforced for a relational schema.

Question: What if a relation isn't dependency preserving? Do we stop enforcing that functional dependency?

Answer: To check that a lost dependency hold:

- Take the join of the relations in the decomposition
- Until we have a relation that includes left-hand side and right-hand side attributes of the lost dependency
- Check if the dependency is enforced that way

Not Practical

Dependency Preservation Example (1)

(Dependency Preservation) A decomposition is dependency preserving if $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$

R = (A, B, C, D, E, G, M)

Consider F = { A → BC, D → EG, M → A }

Decomposed into

R_1= (A, B, C, M) and R_2 = (C, D, E, G)

$\pi_{R_1}(F)$ = { A → BC, M → A}, $\pi_{R_2}(F)$ = {D → EG}

(Question: Is this decomposition dependency preserving?)

Let F' = $\pi_{R_1}(F)$ U $\pi_{R_2}(F)$.

F'+ = F+, Thus it is dependency preserving.

(Question: Must F' be the same as F ?)

Dependency Preservation Example (2)

(Dependency Preservation) We say that a decomposition *D* = {R_1, R_2, ..., R_m } of *R* is dependency-preserving with respect to *F on R* if the union of the projections of *F* on each R_i in *D* is equivalent to *F*; that is, (($\pi_{R_1}(F)$) ∪ ... ∪ ($\pi_{R_m}(F)$))+ = *F+*.

R = (A, B, C, D, E, G, M)

Consider F = { A → BC, D → EG, M → A, M → D }

Decomposition into R_1 and R_2

R_1= (A, B, C, M) and R_2 = (C, D, E, G);

F_1 = { A → BC, M → A}, F_2 = {D → EG}

(Question: is R1 and R2 dependency preserving w.r.t to F? (It seems like we lost M → D))

Dependency Preservation Example (2)

(Dependency Preservation) We say that a decomposition *D* = {R_1, R_2, ..., R_m } of *R* is dependency-preserving with respect to *F on R* if the union of the projections of *F* on each R_i in *D* is equivalent to *F*; that is, (($\pi_{R_1}(F)$) ∪ ... ∪ ($\pi_{R_m}(F)$))+ = *F+*.

R = (A, B, C, D, E, G, M)
Consider F = { A → BC, D → EG, M → A, M → D }
Decomposition into R_1 and R_2
R_1= (A, B, C, M) and R_2 = (C, D, E, G);
F_1 = { A → BC, M → A}, F_2 = {D → EG}

We only checked if F_1 U F_2 is the same as F , this is not always sufficient.

Approach: We need to verify if M→D is inferred by F_1 U F_2

Answer: Since $M^+ |_{F1 \cup F2}$ = {M, A, B, C}, Therefore, M→D is not inferred by F_1 U F_2. Hence, R_1 and R_2 are not dependency preserving regarding F. a

Dependency Preservation Example (3)

Third Example:

R = (A, B, C, D, E, G, M)
Consider F = {A → BC, D → EG, M → A , M→C, C→ D, M→ D}
Decomposition into R_1 and R_2
R_1= (A, B, C, M) and R_2 = (C, D, E, G)
F_1 = {A → BC, M→ A, M→ C}, F_2 = {D→ EG, C→ D}

(Question: Is this dependency preserving?)

Once again F_1 U F_2 is not the same as F

We can verified that M→ D is inferred by F_1 and F_2.

Thus, F+ = (F_1 U F_2)+ (they are equivalent)

Hence, R_1 and R_2 are dependency preserving regarding F.

Lossless Join Property

Another property that a decomposition *D* should possess is the lossless join property.

Definition (**Lossless Join Property**): Formally, a decomposition *D* = {*R*1, *R*2, ..., *Rm*} of *R* has the lossless join property with respect to the set of dependencies *F* on *R* if, for *every* relation state *r* of *R* that satisfies *F*, the following holds, where * is the NATURAL JOIN of all the relations in *D*: *(π*R*1(*r*), ..., π*Rm*(*r*)) = *r.*

Lossless Join Property

Simplified explanation : A decomposition {$R_1, \ldots, R_n$} of R is a *lossless join* decomposition with respect to a set F of FD's if for every relation instance r that satisfies F: $r = \pi_{R_1}(r) \bowtie \cdots \bowtie \pi_{R_n}(r)$.

Recall

Property	3NF	BCNF
Elimination of redundancy due to functional dependency	Most	Yes
Lossless Join	**Yes**	**Yes**
Dependency preservation due to functional dependency	Yes	Maybe

Both the 3NF and BCNF can ensure lossless join property holds.

Lossy Join Decomposition(cont)

Suppose that we decompose the following relation:

STUDENT_ADVISOR

Name	Department	Advisor
Jones	Comp Sci	Smith
Ng	Chemistry	Turner
Martin	Physics	Bosky
Dulles	Decision Sci	Hall
Duke	Mathematics	James
James	Comp Sci	Clark
Evan	Comp Sci	Smith
Baxter	English	Bronte

With dependencies $\{Name \rightarrow Department, Name \rightarrow Advisor, Advisor \rightarrow Department\}$, into two relations:

A Lossy Join Decomposition(cont)

STUDENT_ADVISOR

Name	Department	Advisor
Jones	Comp Sci	Smith
Ng	Chemistry	Turner
Martin	Physics	Bosky
Dulles	Decision Sci	Hall
Duke	Mathematics	James
James	Comp Sci	Clark
Evan	Comp Sci	Smith
Baxter	English	Bronte

STUDENT_DEPARTMENT

Name	Department
Jones	Comp Sci
Ng	Chemistry
Martin	Physics
Duke	Mathematics
Dulles	Decision Sci
James	Comp Sci
Evan	Comp Sci
Baxter	English

DEPARTMENT_ADVISOR

Department	Advisor
Comp Sci	Smith
Chemistry	Turner
Physics	Bosky
Decision Sci	Hall
Mathematics	James
Comp Sci	Clark
English	Bronte

A Lossy Join Decomposition(cont)

There is a simple test to see if a decomposition is lossy by check if this dependency exists.

Test: A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

This only works for binary decompositions.

A Lossy Join Decomposition(cont)

There is a simple test to see if a decomposition is lossy by check if this dependency exists.

Test: A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

Exercise

$\{Name \rightarrow Department, Name \rightarrow Advisor, Advisor \rightarrow Department\}$

Is this a lossless join? No

Lossless Join Property

Note: the above test only applies for simple binary decompositions

We restate the theorem: The decomposition $\{R_1, R_2\}$ of R is lossless iff the common attributes $R_1 \cap R_2$ form a superkey for either R_1 or R_2.

Exercise: Given R(A,B,C) and F = {A→ B}.

Is the decomposition into R_1(A,B) and R_2(A,C) lossless? Yes

A Lossy Join Decomposition(cont)

Name	Department	Advisor
Jones	Comp Sci	Smith
Jones	Comp Sci	Clark*
Ng	Chemistry	Turner
Martin	Physics	Bosky
Dulles	Decision Sci	Hall
Duke	Mathematics	James
James	Comp Sci	Smith*
James	Comp Sci	Clark
Evan	Comp Sci	Smith
Evan	Comp Sci	Clark*
Baxter	English	Bronte

Informally: This is lossy because is not the same as the original relation, thus, the decomposition is lossy.

Formally: A decomposition $\{R_1, \ldots, R_n\}$ of R is a ***lossy*** *join* decomposition with respect to a set F of FD's if for every relation instance r that satisfies F:

$r \subset \pi_{R_1}(r) \bowtie \cdots \bowtie \pi_{R_n}(r)$.

Lossless Join Property

Note:

- The word loss in lossless refers to loss of information
- The word loss in lossless dose not refer to a loss of tuples

In fact…

- A decomposition without the lossless join property leads to **additional spurious** tuples after NATURAL JOIN operations
  - These additional tuples contribute to erroneous or invalid information
- A decomposition with a lossless join property will not lead to additional tuples; Therefore, it is also known as **non-additive join**.

Test Lossless Join property

This previous test works on **binary** decompositions, below is the general solution to testing lossless join property

Algorithm TEST_LJ

1. Create a matrix *S,* each element $s_{i,j} \in S$ corresponds the relation R_i and the attribute A_j, such that: $s_{j,i}$ = a if $A_i \in R_j$, otherwise $s_{j,i}$ = b.
2. Repeat the following process untill (1) S has no change OR (2) one row is made up entirely of "a" symbols.
   i. For each X→ Y , choose the rows where the elements corresponding to X take the value a.
   ii. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y .

Verdict: Decomposition is *lossless* if one row is entirely made up by "a" values.

Testing lossless join property(cont)

Example 1: R = (A,B,C,D),
F = {$A \rightarrow B$, $A \rightarrow C$, $C \rightarrow D$}.
Let R_1 = (*A,B,C*), R_2 = (*C,D*).

	A	B	C	D
R_1	a	a	a	b
R_2	b	b	a	a

Note: rows 1 and 2 of S agree on {*C*}, which is the left-hand side of $C \rightarrow D$. Therefore, change the *D* value on rows 1 to a, matching the value from row 2.

Now row 1 is entirely *a's*, so the decomposition is lossless.

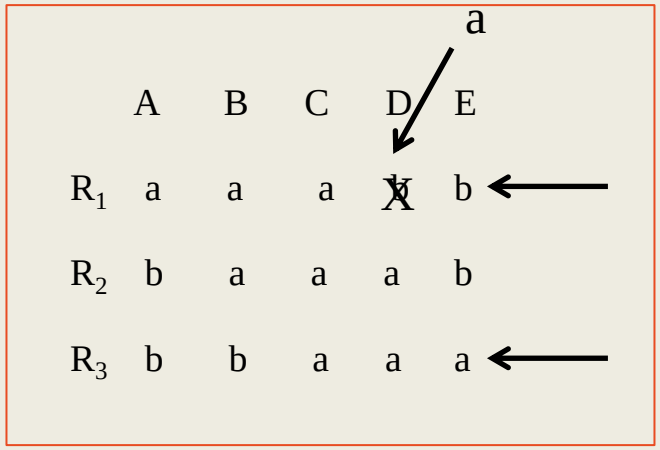
CHEAT SHEET: Algorithm TEST_LJ

1. Create a matrix *S*, each element $s_{i,j} \in S$ corresponds the relation R_i and the attribute A_j, such that: $s_{j,i}$ = a if $A_i \in R_j$, otherwise $s_{j,i}$ = b.
2. Repeat the following process till S has no change or one row is made up entirely of "a" symbols.
   1. For each X→ Y , choose the rows where the elements corresponding to X take the value a.
   2. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y .

Testing lossless join property(cont)

Example 2: R = (A,B,C,D,E),
F = {$AB \rightarrow CD$, $A \rightarrow E$, $C \rightarrow D$}.

Let R_1 = (*A,B,C*),
R_2 = (*B,C,D*) and
R_3 = (*C,D,E*).



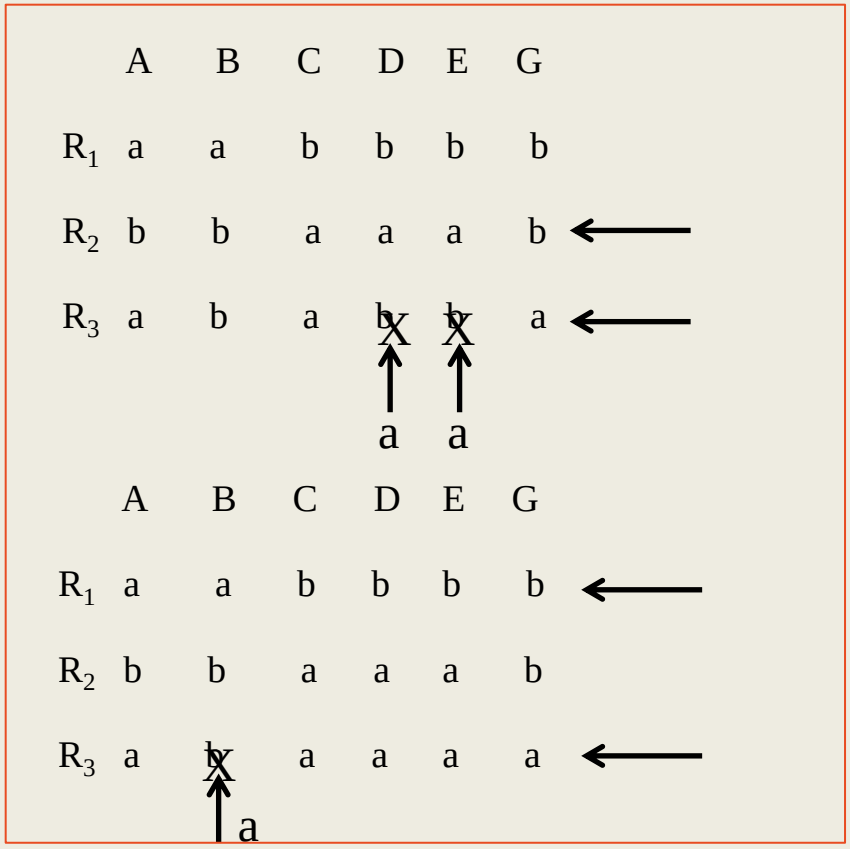
Not lossless join

CHEAT SHEET: Algorithm TEST_LJ

1. Create a matrix *S*, each element $s_{i,j} \in S$ corresponds the relation R_i and the attribute A_j, such that: $s_{j,i}$ = a if $A_i \in R_j$, otherwise $s_{j,i}$ = b.
2. Repeat the following process till S has no change or one row is made up entirely of "a" symbols.
   1. For each X→ Y , choose the rows where the elements corresponding to X take the value a.
   2. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y .

Testing lossless join property(cont)

Example 3: R = (A,B,C,D,E,G),
F = {$C \rightarrow DE$, $A \rightarrow B$, $AB \rightarrow G$}.
Let R_1 = (*A,B*), R_2 = (*C,D,E*) and R_3 = (*A,C,G*).



CHEAT SHEET: Algorithm TEST_LJ

1. Create a matrix *S*, each element $s_{i,j} \in S$ corresponds the relation R_i and the attribute A_j, such that: $s_{j,i}$ = a if $A_i \in R_j$, otherwise $s_{j,i}$ = b.
2. Repeat the following process till S has no change or one row is made up entirely of "a" symbols.
   1. For each X→ Y , choose the rows where the elements corresponding to X take the value a.
   2. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y .

Checkpoint

Previous:

1. The test for lossless join property
2. The dependency preservation property

Next:

1. The method to decompose to 3NF and BCNF
2. Minimal Cover and Equivalance

Testing for BCNF

Testing of a relation schema *R* to see if it satisfies BCNF can be simplified in some cases:

- To check if a nontrivial dependency α → β causes a violation of BCNF, compute α+ (the attribute closure of α), and verify that it includes all attributes of *R*; that is, it is a superkey for *R*.
- To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than check all dependencies in *F* +.

Testing for BCNF

NOTE: We cannot use F to test relations Ri (decomposed from R) for violation of BCNF. It may not suffice.

Consider R(A, B, C, D, E) with F = {A → B, BC →D}.
Suppose R is decomposed into R1 = (A, B) and R2 = (A, C, D, E).

Neither of the dependencies in F contains only attributes from R2.
So R2 is in BCNF? No, AC → D is in F+.

Example above : A X →Y violating BCNF is not always in F.

It passing with respect to the projection of F on Ri

Testing Decomposition for BCNF

An alternative BCNF test is sometimes easier than computing every dependency in *F*+. To check if a relation schema *Ri* in a decomposition of *R* is truly in BCNF, we apply this test:

For each subset *X* of R_i, computer X^+.

- $X \rightarrow (X^+|_{Ri} - X)$ violates BCNF, if $X^+|_{Ri} - X \neq \emptyset$ and $R_i - X^+ \neq \emptyset$.
- This will show if R_i violates BCNF.

Explanation:

- $X^+|_{Ri} - X = \emptyset$ means each F.D with X as the left-hand side is trivial;
- $R_i - X^+ = \emptyset$ means X is a superkey of R_i

Lossless Decomposition into BCNF

Algorithm TO_BCNF

- D := $\{R_1, R_2, \ldots R_n\}$
- **While** (there exists a $R_i \in$ D and R_i is not in BCNF) **Do**

  1 . find a X →Y in R_i that violates BCNF;

  2. replace R_i in D by (R_i − Y) and (X ∪ Y);

Lossless Decomposition into BCNF

Example: (From Desai 6.31)

Find a BCNF decomposition of the relation scheme below:

SHIPPING (*Ship , Capacity , Date , Cargo , Value*)

F consists of:

Ship → *Capacity*

{*Ship , Date*} → *Cargo*

{*Cargo , Capacity*} → *Value*

We know this relation is not in BCNF

Algorithm TO_BCNF

D := {R_1,R_2, ...R_n}

While (there exists a $R_i \in$ D and R_i is not in BCNF) **Do**

1 . find a X →Y in R_i that violates BCNF;

2. replace R_i in *D* by (R_i – Y) and (X ∪ Y);

Lossless Decomposition into BCNF (V1)

From *Ship*→ *Capacity*, we decompose *SHIPPING* into R_{1A} and R_{2A}

R_{1A}(*Ship , Date , Cargo , Value*) with Key: {*Ship,Date*}

A nontrivial FD in F^+ violates BCNF: {*Ship , Cargo*} → Value

and

R_{2A}(*Ship , Capacity*) with Key: {*Ship*}

Only one nontrivial FD in F^+: *Ship* → *Capacity*

SHIPPING (*Ship , Capacity , Date , Cargo , Value*)
F consists of: Ship → Capacity, {*Ship , Date*}→ *Cargo,* {*Cargo , Capacity*}→ *Value*

Lossless Decomposition into BCNF (V1)

R_1 is not in BCNF so we must decompose it further into R_{11A} and R_{12A}

R_{11A} (*Ship , Date , Cargo*) with Key: {*Ship,Date*}

Only one nontrivial FD in F^+ with single attribute on the right side: {*Ship , Date*} →Cargo

and

R_{12A} (*Ship , Cargo , Value*) with Key: {*Ship,Cargo*}

Only one nontrivial FD in F^+ with single attribute on the right side: {*Ship,Cargo*} → *Value*

This is in BCNF, and the decomposition is lossless but not dependency preserving (the FD {*Capacity, Cargo*} → *Value*) has been lost.

SHIPPING (*Ship , Capacity , Date , Cargo , Value*)
F consists of: Ship → Capacity, {*Ship , Date*}→ *Cargo*, {*Cargo , Capacity*}→ *Value*

Lossless Decomposition into BCNF (V2)

Or we could have chosen {*Cargo , Capacity*} →*Value*, which would give us:

R_{1B} (*Ship , Capacity , Date , Cargo*) with Key: {*Ship,Date*}

A nontrivial FD in F^+ violates BCNF: *Ship* → *Capacity*

and

R_{2B} (*Cargo , Capacity , Value*) with Key: {*Cargo,Capacity*}

Only one nontrivial FD in F^+ with single attribute on the right side: {*Cargo , Capacity*} → *Value*

Once again, R_{1B} is not in BCNF so we must decompose it further…

SHIPPING (*Ship , Capacity , Date , Cargo , Value*)
F consists of: Ship → Capacity, {*Ship , Date*}→ *Cargo,* {*Cargo , Capacity*}→ *Value*

Lossless Decomposition into BCNF (V2)

R_1 is not in BCNF so we must decompose it further into R_{11B} and R_{12B}

R_{11B} (*Ship , Date , Cargo*) with Key: {*Ship,Date*}

Only one nontrivial FD in F^+ with single attribute on the right side: {*Ship , Date*} → *Cargo*

and

R_{12B} (*Ship , Capacity*) with Key: {*Ship*}

Only one nontrivial FD in F^+: *Ship* → *Capacity*

This is in BCNF, and the decomposition is both lossless and dependency preserving.

SHIPPING (*Ship , Capacity , Date , Cargo , Value*)
F consists of: Ship → Capacity {*Ship , Date*}→ *Cargo,* {*Cargo , Capacity*}→ *Value*

Lossless Decomposition into BCNF

With this algorithm from the previous slide…

We get a decomposition *D* of *R* that does the following:

- May not preserves dependencies
- Has the lossless join property
- Is such that each resulting relation schema in the decomposition is in BCNF

Property	3NF	BCNF
Elimination of redundancy due to functional dependency	Most	Yes
Lossless Join	Yes	Yes
Dependency preservation due to functional dependency	Yes	**Maybe**

Practice

F = { A→B, A→C, A→D, C→E, E→D, C→G },

R1 = (C, D, E, G), R2 = (A, B, C, D)

R11 = (C, E, G), R12 = (E, D) because of E → D

R21 = (A, B, C), R22 = (C, D) because of C → D

Equivalence

Definition (**equivalence**): Two sets of functional dependencies *E* and *F* are equivalent if *E*+ = *F*+.

Equivalence can also be understood via cover defined as follows

Definition (**cover**): A set of functional dependencies *F* is said to cover another set of functional dependencies *E* if every FD in *E* is also in *F*+; that is, if every dependency in *E* can be inferred from *F*; alternatively, we can say that *E* is covered by *F*.

Exaplaination (**equivalence**): Therefore, equivalence means that every FD in *E* can be inferred from F, and every FD in *F* can be inferred from *E*; that is, *E* is equivalent to *F* if both the conditions—*E* covers *F AND F* covers *E*—hold

Minimal Cover

Definition (**equivalence**): Two sets of functional dependencies *E* and *F* are equivalent if *E*+ = *F*+.

Definition. A minimal cover F_{min} of a set of functional dependencies *E* is a minimal set of dependencies (in the standard canonical form and without redundancy) that is **equivalent** to *E*.

Property: If any dependency from the set *F* is removed; this property is lost *F*

A *minimal cover* for F is a minimal set of FD's F_{min} such that $F^+ = F^+_{min}$.

Minimal Cover

A set *F* of FD's is minimal if

1. Every FD $X \rightarrow Y$ in F is simple: *Y* consists of a single attribute,
2. Every FD $X \rightarrow A$ in F is *left-reduced*: there is no proper subset $Y \subset X$ such that $X \rightarrow A$ can be replaced with $Y \rightarrow A$.
3. No FD in F can be removed; that is, there is no FD $X \rightarrow A$ in F such that $(F - \{X \rightarrow A\})^+ = F^+$.

Prereq. for Algorithm (1)

(Condition one)

Algorithm Reduce_right

- INPUT: *F*.
- OUTPUT: right side reduced *F'*.
- For each FD $X \rightarrow Y \in F$ where Y = $\{A_1, A_2, \ldots, A_k\}$, we use all X $\rightarrow \{A_i\}$ (for $1 \leq i \leq k$) to replace $X \rightarrow Y$.

Prereq. for Algorithm (2)

(Condition two)

Algorithm Reduce_left

- INPUT: right side reduced *F*.
- OUTPUT: right and left side reduced *F'*.
- For each $X \rightarrow \{A\} \in F$ where X = $\{A_i : 1 \leq i \leq k\}$, do the following. For i = 1 to k, replace X with $X - \{A_i\}$ if $A \in (X - \{A_i\})^+$.

Prereq for Algorithm (3)

(Condition three)

Algorithm Reduce_redundancy

- INPUT: right and left side reduced *F*.
- OUTPUT: a minimum cover *F'* of *F*.
- For each FD $X \rightarrow \{A\} \in F$, remove it from *F* if: $A \in X^+$ with respect to $F - \{X \rightarrow \{A\}\}$.

Algorithm for Minimum Cover

Algorithm Min_Cover

Input: a set F of functional dependencies.

Step 1: *Reduce right side*.

Apply Algorithm Reduce Right to F.

Step 2: *Reduce left side*.

Apply Algorithm Reduce Left to the output of Step 1.

Step 3: *Remove redundant* FDs.
Apply Algorithm Remove_redundency to the output of Step 2.

Computing a Minimum Cover (Step 1)

Step 1: Reduce Right: For each FD $X \rightarrow Y \in F$ where Y = $\{A_1, A_2, ..., A_k\}$, we use all X $\rightarrow \{A_i\}$ (for $1 \leq i \leq k$) to replace $X \rightarrow Y$.

Practice:

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

F = {A → BCD, B → CDE, AC → E}

At the end of step 1 we have : F' = {A → B, A→C, A → D, B→ C, B→ D, B→ E, AC → E}

Computing a Minimum Cover (Step 2)

Step 2: Reduce Left: For each $X \rightarrow \{A\} \in F$ where X = $\{A_i : 1 \leq i \leq k\}$, do the following. For i = 1 to k, replace X with $X - \{A_i\}$ if $A \in (X - \{A_i\})^+$.

From Step 1, we had: F' = {A → B, A→C, A → D, B→ C, B→ D, B→ E, AC → E}

AC → E

C^+ = {C}; thus C → E is not inferred by F'.

Hence, AC → E cannot be replaced by C → E.

A^+ = {A, B, C, D, E}; thus, A→ E is inferred by F'.

Hence, AC→ E can be replaced by A → E.

We now have F'' = {A→ B, A→C, A→ D, A→ E, B→ C, B→D, B→ E}

Computing a Minimum Cover (Step 3)

Step 3: Reduce_redundancy: For each FD $X \rightarrow \{A\} \in F$, remove it from F if: $A \in X^+$ with respect to $F - \{X \rightarrow \{A\}\}$.

From Step 2, we had: F'' = {A→ B, A→C, A→ D, A→ E, B→ C, B→D, B→ E}

$A+|_{F'' - \{A \rightarrow B\}}$ = {A, C, D, E}; thus A→B is not inferred by F'' – {A→B}.

That is, A→B is not redundant.

$A+|_{F'' - \{A \rightarrow C\}}$ = {A, B, C, D, E}; thus, A→ C is redundant.

Thus, we can remove A→C from F'' to obtain F'''.

We find that we can remove A→D and A→E but not the others.

Thus, F_{min}={A→B, B→C, B→D, B→E}.

A Note on Finding Minimum Cover

There can be more than one possible minimum cover.

We can always find *at least one* minimal cover *F* for any set of dependencies *E* using this algorithm.

3NF Decomposition Algorithm

Algorithm 3NF decomposition

1. Find a minimal cover *G* for *F*.
2. For each left-hand-side *X* of a functional dependency that appears in *G*, create a relation schema in *D* with attributes $\{X \cup \{A1\} \cup \{A2\} \ldots \cup \{Ak\}\}$, where $X \rightarrow A1$, $X \rightarrow A2$, ..., $X \rightarrow Ak$ are the only dependencies in *G* with *X* as left-hand-side (*X* is the key to this relation).
3. If none of the relation schemas in *D* contains a key of *R*, then create one more relation schema in *D* that contains attributes that form a key of *R*.
4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation *R* is considered redundant if *R* is a projection of another relation *S* in the schema; alternately, *R* is subsumed by *S*.

3NF Decomposition Algorithm

With this algorithm from the previous slide…

We get a decomposition *D* of *R* that does the following:

- Preserves dependencies
- Has the nonadditive join property
- Is such that each resulting relation schema in the decomposition is in 3NF

3NF Decomposition Algorithm

Example ONE:

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

F_{min}={A→B, B→C, B→D, B→E}.

Candidate key: (A, G)

R_1 = (A, B), R_2 = (B, C, D, E)

R_3 = (A, G)

Algorithm 3NF decomposition

1. Find a minimal cover *G* for *F* (use Algorithm 16.2).
2. For each left-hand-side *X* of a functional dependency that appears in *G*, create a relation schema in *D* with attributes {*X* ∪ {*A*1} ∪ {*A*2} ... ∪ {*Ak*} }, where *X* → *A*1, *X* → *A*2, ..., *X* → *Ak* are the only dependencies in *G* with *X* as left-hand-side (*X* is the key of this relation).
3. If none of the relation schemas in *D* contains a key of *R*, then create one more relation schema in *D* that contains attributes that form a key of *R*.
4. (Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation *R* is considered redundant if *R* is a projection of another relation *S* in the schema; alternately, *R* is subsumed by *S*.)

3NF Decomposition Algorithm

Example TWO: (From Desai 6.31)

Following from the *SHIPPING* relation. The functional dependencies already form a canonical cover.

- From *Ship*→*Capacity*, derive R_1(*Ship*,*Capacity*),
- From {*Ship,Date*} → *Cargo*, derive

  R_2(*Ship , Date , Cargo*),
- From {*Capacity,Cargo*} → *Value*, derive

  R_3(*Capacity , Cargo , Value*).
- There are no attributes not yet included and the original key {*Ship,Date*} is included in R_2.

SHIPPING (*Ship , Capacity , Date , Cargo , Value*)
F consists of: Ship → Capacity, {*Ship , Date*}→ *Cargo*, {*Cargo , Capacity*}→ *Value*

3NF Decomposition Algorithm

Example THREE: Apply the algorithm to the LOTS example given earlier.

One possible minimal cover is

{ Property_Id→Lot_No,

Property_Id → Area, {City,Lot_No} → Property_Id,

Area → Price, Area → City, City → Tax_Rate }.

This gives the decomposition:

R_1 (<u>Property_Id</u> , Lot_No , Area)

R_2 (<u>City , Lot_No</u> , Property_Id)

R_3 (<u>Area</u> , Price , City)

R_4 (<u>City</u> , Tax_Rate)

Summary

1. Data redundancies are undesirable as they create the potential for update anomalies.
2. One way to remove such redundancies is to normalize a design, guided by FD's.
3. BCNF removes all redundancies due to FDs, but a dependency preserving decomposition cannot always be found.
4. A dependency preserving, lossless decomposition into 3NF can always be found, but some redundancies may remain.
5. Even where a dependency preserving, lossless decomposition that removes all redundancies can be found, it may not be possible, for efficiency reasons, to remove all redundancies.

Learning Outcome

1. Checking for important decomposition properties
   I. Checking for the dependency preserving property
   II. Checking for the lossless join property

I. Lossless decomposition into BCNF algorithm

1. Lossless and Dependency Preserving 3NF decomposition algorithm

Some slides inspired by Silberchtz, Korth and Sudarshan; from Database System Concepts 6th Editions