Comp 5311 Database Management Systems

7. Functional Dependencies

Functional Dependencies (FD) - Definition

- Let R be a relation scheme and X, Y be sets of attributes in R.
- A functional dependency from X to Y exists if and only if:
  - For every instance of |R| of R, if two tuples in |R| agree on the values of the attributes in X, then they agree on the values of the attributes in Y
- We write $X \rightarrow Y$ and say that X determines Y
- Example on PGStudent (sid, name, supervisor_id, specialization):
  - {supervisor_id} $\rightarrow$ {specialization} means
    - If two student records have the same supervisor (e.g., Dimitris), then their specialization (e.g., Databases) must be the same
    - On the other hand, if the supervisors of 2 students are different, we do not care about their specializations (they may be the same or different).
- Sometimes, we omit the brackets for simplicity:
  - supervisor_id $\rightarrow$ specialization

Trivial FDs

- A functional dependency $X \rightarrow Y$ is trivial if Y is a subset of X
  - {name, supervisor_id} $\rightarrow$ {name}
    - If two records have the same values on both the name and supervisor_id attributes, then they obviously have the same supervisor_id.
    - Trivial dependencies hold for all relation instances

- A functional dependency $X \rightarrow Y$ is non-trivial if $Y \cap X = \varnothing$
  - {supervisor_id} $\rightarrow$ {specialization}
    - Non-trivial FDs are given in the form of constraints when designing a database.
      - For instance, the specialization of a students must be the same as that of the supervisor.
    - They constrain the set of legal relation instances. For instance, if I try to insert two students under the same supervisor with different specializations, the insertion will be rejected by the DBMS
- Some FDs are neither trivial nor non-trivial.

Functional Dependencies and Keys

- A FD is a generalization of the notion of a *key*.

- For PGStudent (<u>sid</u>, name, supervisor_id, specialization), we write:
- {sid} $\rightarrow$ {name, supervisor_id, specialization}
  - The sid determines all attributes (i.e., the entire record)
  - If two tuples in the relation student have the same sid, then they must have the same values on all attributes.
  - In other words they must be the same tuple (since the relational model does not allow duplicate records)

Superkeys and Candidate Keys using FD

- A set of attributes that determines the entire tuple is a **superkey**
  - {sid, name} is a superkey for the PGstudent table.
  - Also {sid, name, supervisor_id} etc.
- A minimal set of attributes that determines the entire tuple is a **candidate key**
  - {sid, name} is not a candidate key because I can remove the name.
  - sid is a candidate key – so is HKID (provided that it is stored in the table).
- If there are multiple candidate keys, the DB designer chooses designates one as the **primary key**.

Closure of a Set of Functional Dependencies

- Given a set of functional dependencies F, there are certain other functional dependencies that are logically implied by F.
- The set of all functional dependencies *logically implied* by F is the closure of F.
- We denote the closure of F by F^+.
- We can find all of F^+ by applying Armstrong's Axioms:
  - if $Y \subseteq X$, then $X \rightarrow Y$ (*reflexivity*)
  - if $X \rightarrow Y$, then $ZX \rightarrow ZY$ (*augmentation*)
  - if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ (*transitivity*)

  these rules are sound and complete.

Examples of Armstrong's Axioms

- if $Y \subseteq X$, then $X \rightarrow Y$ (*reflexivity* generates trivial FDs)
  name → name
  name, supervisor_id → name
  name, supervisor_id → supervisor_id

- if $X \rightarrow Y$, then $ZX \rightarrow ZY$ (*augmentation*)
  sid → name (given)
  supervisor_id, sid → supervisor_id, name

- if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ (*transitivity*)
  sid → supervisor_id (given)
  supervisor_id → specialization (given)
  sid → specialization

Additional Rules

- We can further simplify computation of F^+ by using the following additional rules.
  - If $X \rightarrow Y$ holds and $X \rightarrow Z$ holds, then $X \rightarrow YZ$ holds (*union*)
  - If $X \rightarrow YZ$ holds, then $X \rightarrow Y$ holds and $X \rightarrow Z$ holds (*decomposition*)
  - If $X \rightarrow Y$ holds and $ZY \rightarrow W$ holds, then $ZX \rightarrow W$ holds (*pseudotransitivity*)

- The above rules can be inferred from Armstrong's axioms.

  E.g., pseudotransitivity

$X \rightarrow Y$, $ZY \rightarrow W$	(given)
$ZX \rightarrow ZY$	(by augmentation)
$ZX \rightarrow W$	(by transitivity)

Example of FDs in the closure F^+

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- some members of F^+
  - $A \rightarrow H$ — $A \rightarrow B$; $B \rightarrow H$
  - $AG \rightarrow I$ — $A \rightarrow C$; $AG \rightarrow CG$; $CG \rightarrow I$
  - $CG \rightarrow HI$

Closure of Attribute Sets

- The closure of X under F (denoted by X^+) is the set of attributes that are functionally determined by X under F:

  $$X \rightarrow Y \text{ is in } F^+ \Leftrightarrow Y \subseteq X^+$$

  X is a set of attributes

  Given sid
  If sid $\rightarrow$ name
  then name is part of sid^+
  i.e., sid^+= {sid, name, …}

  If sid $\rightarrow$ supervisor_id
  then supervisor_id is part of sid^+
  i.e., sid^+= {sid, name, supervisor_id, …}

  If sid $\rightarrow$ specialization then continue ….
  Else stop

Algorithm for Computing Attribute Closure

- Input:
  R a relation scheme
  F a set of functional dependencies
  $X \subset R$ (the set of attributes for which we want to compute the closure)
- Output:
  X^+ the closure of X w.r.t. F

$X^{(0)} := X$
Repeat
  $X^{(i+1)} := X^{(i)} \cup Z$, where Z is the set of attributes such that there exists $Y \rightarrow Z$ in F, and $Y \subset X^{(i)}$
Until $X^{(i+1)} := X^{(i)}$
Return $X^{(i+1)}$

Closure of a Set of Attributes: Example

- $R = \{A,B,C,D,E,G\}$
- $F = \{ \{A,B\}\rightarrow\{C\}, \{C\}\rightarrow\{A\}, \{B,C\}\rightarrow\{D\}, \{A,C,D\}\rightarrow\{B\}, \{D\}\rightarrow\{E,G\}, \{B,E\}\rightarrow\{C\}, \{C,G\}\rightarrow\{B,D\}, \{C,E\}\rightarrow\{A,G\}\}$
- $X = \{B,D\}$

- $X^{(0)} = \{B,D\}$

  $\{D\}\rightarrow\{E,G\}$,
- $X^{(1)} = \{B,D,E,G\}$,

  $\{B,E\}\rightarrow\{C\}$
- $X^{(2)} = \{B,C,D,E,G\}$,

  $\{C\}\rightarrow\{A\}$
- $X^{(3)} = \{A,B,C,D,E,G\}$
- $X^{(4)} = X^{(3)}$

Uses of Attribute Closure

- Testing for superkey
  - To test if X is a superkey, we compute X^{+}, and check if X^{+} contains all attributes of *R*.
- Testing functional dependencies
  - To check if a functional dependency $X \rightarrow Y$ holds (or, in other words, $X \rightarrow Y$ is in F^{+}), just check if $Y \subseteq X^{+}$.
- Computing the closure of F
  - For each subset $X \subseteq R$, we find the closure X^{+}, and for each $Y \subseteq X^{+}$, we output a functional dependency $X \rightarrow Y$.
- Computing if two sets of functional dependencies F and G are **equivalent**, i.e., F+ = G+
  - For each functional dependency $Y \rightarrow Z$ in F
    - Compute Y+ with respect to G
    - If $Z \subseteq Y+$ then $Y \rightarrow Z$ is in G+
  - And vice versa

Redundancy of FDs

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - {A}→{C} is redundant in: {{A}→{B}, {B}→{C},{A}→ {C}}

- Parts of a functional dependency may be redundant
  - Example of extraneous/redundant attribute on RHS:

    {{A}→{B}, {B}→{C}, {A}→{C,D}} can be simplified to
    {{A}→{B}, {B}→{C}, {A}→{D}}
    (because {A}→{C} is inferred from {A} → {B}, {B}→{C})

  - Example of extraneous/redundant attribute on LHS:

    {{A}→{B}, {B}→{C}, {A,C}→{D}} can be simplified to
    {{A}→{B}, {B}→{C}, {A}→{D}}
    (because of {A}→{C})

Canonical Cover

- A *canonical cover* for F is a set of dependencies F_c such that
  - F and F_c are equivalent
  - F_c contains no redundancy
  - Each left side of functional dependency in F_c is unique.
    - For instance, if we have two FD $X \rightarrow Y$, $X \rightarrow Z$, we convert them to $X \rightarrow YZ$.
- Algorithm for canonical cover of F:

  repeat

  Use the union rule to replace any dependencies in F

  $X_1 \rightarrow Y_1$ and $X_1 \rightarrow Y_2$ with $X_1 \rightarrow Y_1 Y_2$

  Find a functional dependency $X \rightarrow Y$ with an extraneous attribute either in X or in Y

  If an extraneous attribute is found, delete it from $X \rightarrow Y$

  until F does not change
- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

Example of Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$ because of $B \rightarrow C$.
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$ because of $A \rightarrow B$ and $B \rightarrow$ C.
- The canonical cover is:

$$A \rightarrow B$$
$$B \rightarrow C$$

Pitfalls in Relational Database Design

- Relational database design requires that we find a “good” collection of relation schemas.
- Functional dependencies can be used to refine ER diagrams or independently (i.e., by performing repetitive decompositions on a "universal" relation that contains all attributes).
- A bad design may lead to several problems.

Problems of Bad Design

Assume the position determines the salary:
position → salary

T1

first_name	last_name	address	department	position	salary
Dewi	Srijaya	12a Jln Lempeng	Toys	**clerk**	**2000**
Izabel	Leong	10 Outram Park	Sports	trainee	1200
John	Smith	107 Clementi Rd	Toys	**clerk**	**2000**
Axel	Bayer	55 Cuscaden Rd	Sports	trainee	1200
Winny	Lee	10 West Coast Rd	Sports	**manager**	**2500**
Sylvia	Tok	22 East Coast Lane	Toys	**manager**	**2600**
Eric	Wei	100 Jurong drive	Toys	assistant manager	**2200**
?	?	?	?	security guard	1500

key

Redundant storage

Update anomaly

Potential deletion anomaly

Insertion anomaly

Decomposition Example

T2

first_name	last_name	address	department	position
Dewi	Srijaya	12a Jln lempeng	Toys	clerk
Izabel	Leong	10 Outram Park	Sports	trainee
John	Smith	107 Clementi Rd	Toys	clerk
Axel	Bayer	55 Cuscaden Rd	Sports	trainee
Winny	Lee	10 West Coast Rd	Sports	manager
Sylvia	Tok	22 East Coast Lane	Toys	manager
Eric	Wei	100 Jurong drive	Toys	assistant manager

T3

position	salary
clerk	2000
trainee	1200
manager	2500
assistant manager	2200
security guard	1500

- No Redundant storage
- No Update anomaly
- No Deletion anomaly
- No Insertion anomaly

Normalization

- Normalization is the process of decomposing a relation schema R into **fragments** (i.e., smaller tables) R_1, R_2,.., R_n. Our goals are:
  - Lossless decomposition: The fragments should contain the same information as the original table. Otherwise decomposition results in information loss.
  - Dependency preservation: Dependencies should be preserved within each R_i , i.e., otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.
  - Good form: The fragments R_i should not involve redundancy. Roughly speaking, a table has redundancy if there is a FD where the LHS is not a key (more on this later).

Lossless Join Decomposition

- A decomposition is **lossless** (aka lossless join) if we can recover the initial table
- In general a decomposition of R into R_1 and R_2 is lossless if and only if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
  - In other words, the common attribute of R_1 and R_2 must be a candidate key for R_1 or R_2.

- Is the previous decomposition example (T2, T3) lossless?
  - Yes because the common attribute of T2, T3 is position and it determines the salary; therefore it is a key for T3.

Example of a Lossy Decomposition

- Decompose R = (A,B,C) into R_1 = (A,B) and R_2 = (B,C)

r

A	B	C
a	1	*m*
a	2	*n*
b	1	*p*

➡

$\Pi_{A,B}(r)$

A	B
a	1
a	2
b	1

$\Pi_{B,C}(r)$

B	C
1	*m*
2	*n*
1	*p*

$\Pi_{A,B}(r) \bowtie \Pi_{B,C}(r)$

A	B	C
a	1	*m*
a	1	*p*
a	2	*n*
b	1	*m*
b	1	*p*

It is a lossy decomposition:
two extraneous tuples.
You get more, not less!!
B is not a key of either small table

Dependency Preserving Decomposition

- The decomposition of a relation scheme R with FDs F is a set of tables (fragments) R_i with FDs F_i
- F_i is the subset of dependencies in F^+ (the closure of F) that include only attributes in R_i.
- The decomposition is dependency preserving if and only if

$$(\cup_i F_i)^+ = F^+$$

Non-Dependency Preserving Decomposition Example

R = (A, B, C), F = {{A}→{B}, {B}→{C}, {A}→{C}}. Key: A

There is a dependency {B}→ {C}, where the LHS is not the key, meaning that there can be considerable redundancy in R.

Solution: Break it in two tables R1(A,B), R2(A,C) (normalization)

A	B	C
1	2	3
2	2	3
3	2	3
4	2	4

A	B
1	2
2	2
3	2
4	2

A	C
1	3
2	3
3	3
4	4

The decomposition is lossless because the common attribute A is a key for R1 (and R2)

The decomposition is not dependency preserving because F1={{A}→{B}}, F2={{A}→{C}} and $(F1 \cup F2)^+ \neq F^+$. We lost the FD {B}→{C}.

In practical terms, each FD is implemented as an assertion, which it is checked when there are updates. In the above example, in order to find violations, we have to join R1 and R2. Can be very expensive.

Dependency Preserving Decomposition Example

R = (A, B, C), F = {{A}→{B}, {B}→{C}, {A}→{C}}. Key: A

Break R in two tables R1(A,B), R2(B,C)

A	B	C
1	2	3
2	2	3
3	2	3
4	2	4

A	B
1	2
2	2
3	2
4	2

B	C
2	3
2	**4**

The decomposition is lossless because the common attribute B is a key for R2

The decomposition is dependency preserving because F1={{A}→{B}}, F2={{B}→{C}} and (F1∪F2)+=F+

Violations can be found by inspecting the individual tables, without performing a join.