CS 186 Section 8: Functional Dependencies and Normalization

Vikram Sreekanti

Functional Dependencies: Basics

- *X* —> *Y* (read as “*X* determines *Y*”)
  - Given any two tuples, *r1* and *r2*, if the *X* values are the same, then the *Y* values must be the same.
- *K* —> *{all attributes of R}*
  - This means that *K* is a superkey for *R*
- Closures
  - F^+ (read as “the closure of *F*”)
    - This is the set of all FDs that are implied by *F*, including trivial dependencies.

Armstrong's Axioms

- **Reflexivity**: if $X \supseteq Y$, then X —> Y
- **Augmentation**: if X —> Y, then XZ —> YZ $\forall$ Z
- **Transitivity**: if X —> Y and Y —> Z, X —> Z
- **Union**: if X —> Y and X —> Z, X -> YZ
- **Decomposition**: if X —> YZ, X —> Y and X —> Z

Common Mistake

$XA \longrightarrow YA$ ***does not mean*** $X \longrightarrow A$

Attribute Closures

- Attribute Closure (aka "X^+")

```
X+ := X
while (X+ changed on prev. iteration):
    for U –> V in F:
        if U in X+:
            add V to X
```

Functional Dependency Exercises

Consider the `Works_In (SSN, parking_lot, department_id, since)` relation. If SSN is a key for this relation, what is the functional dependency that we can infer from that?

`S -> SPDI`

Functional Dependency Exercises

Consider the `Works_In (SSN, parking_lot, department_id, since)` relation. If employees in the same department are put in the same parking lot, what else can we infer?

D –> P

Functional Flights

Determine the set of FDs inherent to this schema.

```
Flights(Flight_no, Date, fRom, To, Plane_id),
    ForeignKey(Plane_id)
Planes(Plane_id, tYpe)
Seat(Seat_no, Plane_id, Legroom), ForeignKey(Plane_id)
```

FD -> RTP

P -> Y

SP -> L

Attribute Closure Exercises

Now consider the attribute set R = ABCDE and the FD set `F = {AB -> C, A -> D, D -> E, AC -> B}`. Compute the closure for the following attributes:

`A:` ADE

`AB:` ABCDE

`B:` B

`DE:` DE

Boyce-Codd Normal Form

- A relation is in BCNF if…
  - $\forall$ *X* —> *A* in *F*, either:
    - $A \supseteq X$, or
    - *X* is a superkey for *R*
- In English:
  - A relation is in BCNF iff the only non-trivial FDs over *R* are the key constraints.

Schema Decomposition

- What do you do if a schema isn't in BCNF? You "decompose" the schema.

- For a relation R, if *X* —> *A* is an FD that violates BCNF, decompose it into *R* - *A* and *XA*.

- Repeat if necessary.

Schema Decomposition Example

`R = ABCEG ; F = { AB –> C, AC –> B, BC –> A, E –> G }`

Step 1: Determine keys.

There are none!

Step 2: Decompositions:

`AC –> B => ABEG, ABC`

`E –> G => ABE, EG`

Final Result: `ABE, EG, ABC`

Question: Why didn’t we apply `AC –> B` and `BC –> A`?

Pitfalls of Decompositions

- May be impossible to reconstruct the original relation — “lossiness”.
- Dependency checking may require joins.
- Queries become more expensive.

More on Decompositions

- Lossless decompositions: if you decompose *R* into *X* and *Y*, then π_X `(R) JOIN` π_Y `(R) == R`
  - All decompositions that deal with redundancy should be lossless!
- Dependency Preservation: if you decompose *R* into *X* and *Y*, $(F_X \text{ U } F_Y) = F^+$
  - BCNF may not always have dependency preserving decompositions

Third Normal Form

- A relation is in BCNF if…
  - $\forall$ *X* —> *A* in *F*, either:
    - $A \supseteq X$, or
    - *X* is a superkey for *R*
    - *A* is a (**<u>minimal</u>**) candidate key for *R*
- Why do we care about 3NF if we have BCNF?
  - You can ***<u>always</u>*** do a lossless-join, dependency-preserving decomposition into 3NF.

3NF Decomposition

- There's a bunch of theory behind this, but the gist of it is that you want the **minimal cover** of a set.
- How do you find the minimal cover of F?
  - For the clarity's sake, write out each dependency with only one element on each RHS.
    - e.g. A —> BC => A —> B, A —> C
  - Try to remove attributes from the LHS of each dependency.
    - If you can remove an attribute and use the rest of the dependencies to derive the right side, then you can remove it.
    - Otherwise, leave it and move on.
  - Get rid of redundant dependencies.
    - You hide a rule. You then find the closure of the LHS of the rule using the remaining rules. If you can derive it, the rule is redundant.
- Note that the minimal cover you end up with will depend on the order you do these checks in.

Minimal Cover Example

```
F = { A -> B, ABCD -> E, EF -> GH, ACDF -> EG }

A -> B
ABCD -> E
EF -> G
EF -> H
ACDF -> E
ACDF -> G
```

Minimal Cover Example

```
F = { A -> B, ABCD -> E, EF -> GH, ACDF -> EG }

A -> B
ABCD -> E
EF -> G
EF -> H
ACDF -> E
ACDF -> G
```

Minimal Cover Example

```
F = { A —> B, ABCD —> E, EF —> GH, ACDF —> EG }

A —> B
ACD —> E
EF —> G
EF —> H
ACD —> E
ACDF —> G
```

Minimal Cover Example

```
F = { A -> B, ABCD -> E, EF -> GH, ACDF -> EG }

A -> B
ACD -> E
EF -> G
EF -> H
ACD -> E
ACDF -> G
```

Minimal Cover Example

```
F = { A —> B, ABCD —> E, EF —> GH, ACDF —> EG }

A —> B
ACD —> E
EF —> G
EF —> H
ACDF —> G
```

Minimal Cover Example

F = { A -> B, ABCD -> E, EF -> GH, ACDF -> EG }

A -> B
ACD -> E
EF -> G
EF -> H
ACDF -> G

Minimal Cover Example

```
F = { A -> B, ABCD -> E, EF -> GH, ACDF -> EG }

A -> B
ACD -> E
EF -> G
EF -> H
```

Decomposition Exercise

```
R = ABCDEFG
F = { AB -> CD, C -> EF, G -> A, G -> F, CE -> F }
```

Decompose this into BCNF.

Primary Keys:
None!

Decomposition:

```
AB -> C => ABEFG, ABCD
G -> A => BEFG, AG, ABCD
G -> F => BEG, FG, AG, ABCD
```

Decomposition Exercise

Decomposition:

`AB -> C => ABEFG, ABCD`

`G -> A => BEFG, AG, ABCD`

`G -> F => BEG, FG, AG, ABCD`

Is this decomposition dependency preserving?

No. There is no way to derive `CE -> F` or `C -> EF`.

Minimal Cover Exercise

```
R = ABCDEFG
F = { AB —> CD, C —> EF, G —> A, G —> F, CE —> F }
```

Derive a minimal cover for this set of FDs.

Minimal Cover Exercise

```
R = ABCDEFG
F = { AB —> CD, C —> EF, G —> A, G —> F, CE —> F }
```

Derive a minimal cover for this set of FDs.

```
AB —> C
AB —> D
C —> E
C —> F
G —> A
G —> F
CE —> F
```

Minimal Cover Exercise

```
R = ABCDEFG
F = { AB —> CD, C —> EF, G —> A, G —> F, CE —> F }
```

Derive a minimal cover for this set of FDs.

```
AB —> C
AB —> D
C —> E
C —> F
G —> A
G —> F
CE —> F
```

Minimal Cover Exercise

```
R = ABCDEFG
F = { AB –> CD, C –> EF, G –> A, G –> F, CE –> F }
```

Derive a minimal cover for this set of FDs.

```
AB –> C
AB –> D
C –> E
C –> F
G –> A
G –> F
C –> F
```

Minimal Cover Exercise

```
R = ABCDEFG
F = { AB -> CD, C -> EF, G -> A, G -> F, CE -> F }
```

Derive a minimal cover for this set of FDs.

```
AB -> C
AB -> D
C -> E
C -> F
G -> A
G -> F
C -> F
```

Minimal Cover Exercise

```
R = ABCDEFG
F = { AB -> CD, C -> EF, G -> A, G -> F, CE -> F }
```

Derive a minimal cover for this set of FDs.

```
AB -> C
AB -> D
C -> E
C -> F
G -> A
G -> F
```

This *is* your 3NF Decomposition.