CompSci 516
Database Systems

Lecture 7
Relational Calculus (revisit)
And
Normal Forms

Instructor: Sudeepa Roy

Announcements

- HW1 Deadlines!
  – Today: parser and Q1-Q3
  – Q4: next Tuesday
  – Q5 (3 RA questions will be posted today): next Thursday
- 2 late days with penalty apply for individual deadlines
  – If you are still parsing XML
    - Remember to start early next time from first day
    - HW2 and HW3 typically take more time and effort!

Today's topic

- Revisit RC
- Finish Normalization
- From Thursday: Database Internals

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke, and with the help of slides by Dr. Magda Balazinska and Dr. Dan Suciu

Relational Calculus (RC)
(Revisit from Lecture 4)

Logic Notations

- ∃ There exists
- ∀ For all
- ∧ Logical AND
- ∨ Logical OR
- ¬ NOT
- ⇒ Implies

TRC: example

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- Find the name and age of all sailors with a rating above 7

∃ There exists

{P | ∃ S ϵ Sailors (S.rating > 7 ∧ P.sname = S.sname ∧ P.age = S.age)}

- P is a tuple variable
  – with exactly two fields sname and age (schema of the output relation)
  – P.sname = S.sname ∧ P.age = S.age gives values to the fields of an answer tuple
- Use parentheses, ∀ ∃ ∨ ∧ > < = ≠ ¬ etc as necessary
- A ⇒ B is very useful too
  – next slide

$A \Rightarrow B$

- A "implies" B
- Equivalently, if A is true, B must be true
- Equivalently, $\neg A \lor B$, i.e.
  - either A is false (then B can be anything)
  - otherwise (i.e. A is true) B must be true

Useful Logical Equivalences

$\exists$	There exists
$\forall$	For all
$\land$	Logical AND
$\lor$	Logical OR
$\neg$	NOT

- $\forall x\ P(x) \;=\; \neg\exists x\ [\neg P(x)]$
- $\neg(P \lor Q) \;=\; \neg P \land \neg Q$
- $\neg(P \land Q) \;=\; \neg P \lor \neg Q$

de Morgan's laws

  - Similarly, $\neg(\neg P \lor Q) = P \land \neg Q$ etc.
- $A \Rightarrow B \;=\; \neg A \lor B$

TRC: example

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- Find the names of sailors who have reserved at least two boats

TRC: example

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- Find the names of sailors who have reserved at least two boats

$\{P \mid \exists S \in Sailors\ (\exists R1 \in Reserves\ \exists R2 \in Reserves\ (S.sid = R1.sid \land S.sid = R2.sid \land R1.bid \neq R2.bid) \land P.sname = S.sname)\}$

TRC: example

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- Find the names of sailors who have reserved all boats

TRC: example

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- Find the names of sailors who have reserved all boats

$\{P \mid \exists S \in Sailors\ [\forall B \in Boats\ (\exists R \in Reserves\ (S.sid = R.sid \land R.bid = B.bid))] \land (P.sname = S.sname)\}$

TRC: example

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- Find the names of sailors who have reserved all red boats

How will you change the previous TRC expression?

TRC: example

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- Find the names of sailors who have reserved all red boats

{P | ∃ S ϵ Sailors (∀B ϵ Boats (B.color = 'red' ⇒ (∃ R ϵ Reserves (S.sid = R.sid ⋀ R.bid = B.bid))) ⋀ P.sname = S.sname)}

Recall that A ⇒ B is logically equivalent to ¬ A ⋁ B

so ⇒ can be avoided, but it is cleaner and more intuitive

More Examples: RC

- The famous "Drinker-Beer-Bar" example!

UNDERSTAND THE DIFFERENCE IN ANSWERS FOR ALL FOUR DRINKERS

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker Category 1

Find drinkers that frequent some bar that serves some beer they like.

…

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker Category 1

Find drinkers that frequent some bar that serves some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker ∧ ∃ S ϵ Serves ∃ L ϵ Likes (F.drinker = L.drinker) ∧ (F.bar = S.bar) ∧ (S.beer =L.beer))}

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker Category 2

Find drinkers that frequent some bar that serves some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker ∧ ∃ S ϵ Serves ∃ L ϵ Likes (F.drinker = L.drinker) ∧ (F.bar = S.bar) ∧ (S.beer =L.beer))}

Find drinkers that frequent only bars that serves some beer they like.

…

Free HW question hint!

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker Category 2

Find drinkers that frequent some bar that serves some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker ∧ ∃ S ϵ Serves ∃ L ϵ Likes (F.drinker = L.drinker) ∧ (F.bar = S.bar) ∧ (S.beer =L.beer))}

Find drinkers that frequent only bars that serve some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ F1 ϵ Frequents (F.drinker = F1.drinker) ⇒ ∃ S ϵ Serves ∃ L ϵ Likes [(F1.bar = S.bar) ∧ (F1.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

19

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker Category 3

Find drinkers that frequent some bar that serves some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker ∧ ∃ S ϵ Serves ∃ L ϵ Likes (F.drinker = L.drinker) ∧ (F.bar = S.bar) ∧ (S.beer =L.beer))}

Find drinkers that frequent only bars that serve some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ F1 ϵ Frequents (F.drinker = F1.drinker) ⇒ ∃ S ϵ Serves ∃ L ϵ Likes [(F1.bar = S.bar) ∧ (F1.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

Find drinkers that frequent some bar that serves only beers they like.

…

20

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker Category 3

Find drinkers that frequent some bar that serves some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker ∧ ∃ S ϵ Serves ∃ L ϵ Likes (F.drinker = L.drinker) ∧ (F.bar = S.bar) ∧ (S.beer =L.beer))}

Find drinkers that frequent only bars that serve some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ F1 ϵ Frequents (F.drinker = F1.drinker) ⇒ ∃ S ϵ Serves ∃ L ϵ Likes [(F1.bar = S.bar) ∧ (F1.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

Find drinkers that frequent some bar that serves only beers they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ S ϵ Serves (F.bar = S.bar) ⇒ ∃ L ϵ Likes [(F.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

21

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker Category 4

Find drinkers that frequent some bar that serves some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker ∧ ∃ S ϵ Serves ∃ L ϵ Likes (F.drinker = L.drinker) ∧ (F.bar = S.bar) ∧ (S.beer =L.beer))}

Find drinkers that frequent only bars that serve some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ F1 ϵ Frequents (F.drinker = F1.drinker) ⇒ ∃ S ϵ Serves ∃ L ϵ Likes [(F1.bar = S.bar) ∧ (F1.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

Find drinkers that frequent some bar that serves only beers they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ S ϵ Serves (F.bar = S.bar) ⇒ ∃ L ϵ Likes [(F.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

Find drinkers that frequent only bars that serve only beer they like.

…

22

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker Category 4

Find drinkers that frequent some bar that serves some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker ∧ ∃ S ϵ Serves ∃ L ϵ Likes (F.drinker = L.drinker) ∧ (F.bar = S.bar) ∧ (S.beer =L.beer))}

Find drinkers that frequent only bars that serve some beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ F1 ϵ Frequents (F.drinker = F1.drinker) ⇒ ∃ S ϵ Serves ∃ L ϵ Likes [(F1.bar = S.bar) ∧ (F1.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

Find drinkers that frequent some bar that serves only beers they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ S ϵ Serves (F.bar = S.bar) ⇒ ∃ L ϵ Likes [(F.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

Find drinkers that frequent only bars that serve only beer they like.

{x | ∃F ϵ Frequents (F.drinker = x.drinker) ∧ [∀ F1 ϵ Frequents (F.drinker = F1.drinker) ⇒ [∀ S ϵ Serves (F1.bar = S.bar) ⇒ ∃ L ϵ Likes [(F.drinker = L.drinker) ∧ (S.beer =L.beer)]]}

23

Why should we care about RC

- RC is declarative, like SQL, and unlike RA (which is operational)
- Gives foundation of database queries in first-order logic
  - you cannot express all aggregates in RC, e.g. cardinality of a relation or sum (possible in extended RA and SQL)
  - still can express conditions like "at least two tuples" (or any constant)
- RC expression may be much simpler than SQL queries
  - and easier to check for correctness than SQL
  - power to use ∀ and ⇒
  - then you can systematically go to a "correct" SQL or RA query

Duke CS, Fall 2019 CompSci 516: Database Systems 24

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Drinker category 5!

From RC to SQL

Query: Find drinkers that like some beer (so much) that they frequent all bars that serve it

$$\{x \mid \exists L \in \text{Likes}\ (L.\text{drinker} = x.\text{drinker}) \wedge [\ \forall S \in \text{Serves}\ (L.\text{beer} = S.\text{beer}) \Rightarrow \exists F \in \text{Frequents}\ [(F.\text{drinker} = L.\text{drinker}) \wedge (S.\text{beer} = L.\text{beer})]\ ]\}$$

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to SQL (or RA)

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$\{x \mid \exists L \in \text{Likes}\ (L.\text{drinker} = x.\text{drinker}) \wedge [\ \forall S \in \text{Serves}\ [\ (L.\text{beer} = S.\text{beer}) \Rightarrow \exists F \in \text{Frequents}\ [(F.\text{drinker} = L.\text{drinker}) \wedge (S.\text{beer} = L.\text{beer})]\ ]\ ]\}$$

$$\equiv \{x \mid \exists L \in \text{Likes}\ (L.\text{drinker} = x.\text{drinker}) \wedge [\ \forall S \in \text{Serves}\ [\neg\ (L.\text{beer} = S.\text{beer}) \vee [\exists F \in \text{Frequents}\ [(F.\text{drinker} = L.\text{drinker}) \wedge (S.\text{beer} = L.\text{beer})]\ ]\ ]\}$$

Step 1: Replace ∀ with ∃ using de Morgan's Laws

$\forall x\ P(x)$ same as $\neg\exists x\ \neg P(x)$

$$Q(x) = \exists y.\ \text{Likes}(x, y) \wedge [\ \neg\exists S \in \text{Serves}\ [(L.\text{beer} = S.\text{beer}) \wedge \neg\ [\exists F \in \text{Frequents}\ [(F.\text{drinker} = L.\text{drinker}) \wedge (S.\text{beer} = L.\text{beer})]\ ])$$

SQL or RA does not have ∀!
Now you got all ∃ and ¬ expressible in RA/SQL

$\neg(\neg P \vee Q)$ same as $P \wedge \neg Q$

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to SQL

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

$$Q(x) = \exists y.\ \text{Likes}(x, y) \wedge \neg\exists S \in \text{Serves}\ [(L.\text{beer} = S.\text{beer}) \wedge \neg\ [\exists F \in \text{Frequents}\ [(F.\text{drinker} = L.\text{drinker}) \wedge (S.\text{beer} = L.\text{beer})]\ ])$$

Step 2: Translate into SQL

```
SELECT DISTINCT L.drinker
FROM Likes L
WHERE not exists
  (SELECT S.bar
   FROM Serves S
   WHERE L.beer=S.beer
       AND not exists (SELECT *
                       FROM Frequents F
                       WHERE F.drinker=L.drinker
                          AND F.bar=S.bar))
```

We will see a "methodical and correct" translation trough "safe queries" in Datalog

Database Normalization

Recap from Lecture-5

ssn (S)	name (N)	lot (L)	rating (R)	hourly-wage (W)	hours-worked (H)
111-11-1111	Attishoo	48	8	10	40
222-22-2222	Smiley	22	8	10	30
333-33-3333	Smethurst	35	5	7	30
444-44-4444	Guldu	35	5	7	32
555-55-5555	Madayan	35	8	10	40

Redundancy is bad!
(well…not always?)

1. Redundant storage
2. Update anomalies
3. Insertion anomalies
4. Deletion anomalies

Solution: Decomposition!

Be careful about "Lossy decomposition"!
(on blackboard)

Schema is forcing to store (complex) associations among tuples
Nulls may or may not help

Decompositions should be used judiciously

1. Do we need to decompose a relation?
   – Several "normal forms" exist to identify possible redundancy at different granularity
   – If a relation is not in one of them, may need to decompose further
2. What are the problems with decomposition?
   – Bad decompositions: e.g., Lossy decompositions
   – Performance issues -- decomposition may both
     • help performance (for updates, some queries accessing part of data), or
     • hurt performance (new joins may be needed for some queries)

Functional Dependencies (FDs)

- A functional dependency (FD) X → Y holds over relation R if, for every allowable instance *r* of R:
  - i.e., given two tuples in *r*, if the X values agree, then the Y values must also agree
  - X and Y are *sets* of attributes
  - $t1 \in r,\ t2 \in r,\ \Pi_X(t1) = \Pi_X(t2)$ implies $\Pi_Y(t1) = \Pi_Y(t2)$

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

What is a (possible) FD here?

Functional Dependencies (FDs)

- A functional dependency (FD) X → Y holds over relation R if, for every allowable instance *r* of R:
  - i.e., given two tuples in *r*, if the X values agree, then the Y values must also agree
  - X and Y are *sets* of attributes
  - $t1 \in r,\ t2 \in r,\ \Pi_X(t1) = \Pi_X(t2)$ implies $\Pi_Y(t1) = \Pi_Y(t2)$

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

What is a (possible) FD here?

AB → C

Note that, AB is not a key

Can we detect FDs from an instance?

- An FD is a statement about all allowable relation instances
  - Must be identified based on semantics of application
  - Given some allowable instance *r1* of R, we can check if it violates some FD *f*, but we cannot tell if *f* holds over R
- K is a candidate key for R means that K →R
  - denoting R = all attributes of R too
  - However, S →R does not require S to be minimal
  - e.g. S can be a superkey

FD from a key

- Consider a relation R(A,B, C, D) where AB is a key
- Which FD must hold on R?
- AB → ABCD
- However, S → ABCD does not mean S is a key. Why?
  - S can be a superkey!
  - E.g., ABC → ABCD in R, but ABC is not a key

Armstrong's Axioms

- X, Y, Z are sets of attributes

1. Reflexivity: If X ⊇ Y, then X → Y, e.g., ABC → AB
2. Augmentation: If X → Y, then XZ → YZ for any Z,
   - e.g., AB → C ⇒ ABDE → CDE
3. Transitivity: If X → Y and Y → Z, then X → Z
   - e.g., AB → C and C → D ⇒ AB → D

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

Apply these rules on AB → C and check

- Additional rules that follow from Armstrong's Axioms

4. Union: If X → Y and X → Z, then X → YZ
5. Decomposition: If X → YZ, then X → Y and X → Z

A → B and A → C
⇒ A → BC

A → BC
⇒ A → B, A → C

Closure of a set of FDs

- Given some FDs, we can usually infer additional FDs:
  - SSN → DEPT, and DEPT → LOT implies SSN → LOT
- An FD f is implied by a set of FDs F if f holds whenever all FDs in F hold.
- F^+ = closure of FDs F is the set of all FDs that are implied by F
- S^+ = closure of attributes S is the set of all attributes that are implied by S according to F^+

Armstrong's Axioms are sound and complete inference rules for FDs
- sound: they only generate FDs in closure F^+ for F
- complete: by repeated application of these rules, all FDs in F^+ will be generated

Computing Attribute Closure

Algorithm:

- closure = X
- Repeat until no change
  - if there is an FD U → V in F such that U ⊆ closure, then closure = closure ∪ V

Let's do the example first, Then look at the algo yourself

Does F = {A → B, B → C, C D → E } imply

1. A → E? (i.e, is A → E in the closure F^+ , or E in A^+?)
2. AD → E?

On blackboard

Normal Forms

- Question: given a schema, how to decide whether any schema refinement is needed at all?
- If a relation is in a certain normal forms, it is known that certain kinds of problems are avoided/minimized
- Helps us decide whether decomposing the relation is something we want to do

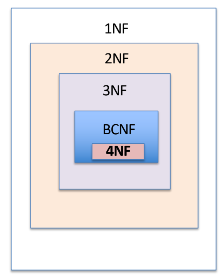
FDs play a role in detecting redundancy

Example

- Consider a relation R with 3 attributes, ABC
  - No FDs hold: There is no redundancy here – no decomposition needed
  - Given A → B: Several tuples could have the same A value, and if so, they'll all have the same B value ⇒ redundancy ⇒ decomposition may be needed if A is not a key
- Intuitive idea:
  - if there is any non-key dependency, e.g. A → B, decompose!

Normal Forms

R is in 4NF
⇒ R is in BCNF
⇒ R is in 3NF
⇒ R is in 2NF (a historical one)
⇒ R is in 1NF (every field has atomic values)



Only BCNF and 4NF are covered in the class

Boyce-Codd Normal Form (BCNF)

- Relation R with FDs *F* is in BCNF if, for all X → A in F
  - A ∊ X (called a trivial FD), or
  - X contains a key for R
    - i.e. X is a superkey

BCNF decomposition algorithm

- Find a BCNF violation
  - That is, a non-trivial FD $X \rightarrow Y$ in R where X is not a super key of R
- Decompose R into R_1 and R_2, where
  - R_1 has attributes $X \cup Y$
  - R_2 has attributes $X \cup Z$, where Z contains all attributes of R that are in neither X nor Y
- Repeat until all relations are in BCNF
- Also gives a lossless decomposition!
  - Check yourself

BCNF decomposition example - 1

On blackboard

- CSJDPQV, key C, F = {JP → C, SD → P, J → S}
  - To deal with SD → P, decompose into SDP, CSJDQV.
  - To deal with J → S, decompose CSJDQV into JS and CJDQV
- Is JP → C a violation of BCNF?
- Note:
  - several dependencies may cause violation of BCNF
  - The order in which we pick them may lead to very different sets of relations
  - there may be multiple correct decompositions (can pick J → S first)

BCNF decomposition example - 2

uid → uname, twitterid
twitterid → uid
uid, gid → fromDate

UserJoinsGroup (*uid, uname, twitterid, gid, fromDate*)
BCNF violation: *uid → uname, twitterid*

User (*uid, uname, twitterid*)
uid → uname, twitterid
twitterid → uid
BCNF

Member (*uid, gid, fromDate*)
uid, gid → fromDate
BCNF

BCNF decomposition example - 3

It is not enough to only look at given FDs! You need to Consider the closure!

uid → uname, twitterid
twitterid → uid
uid, gid → fromDate

UserJoinsGroup (*uid, uname, twitterid, gid, fromDate*)
BCNF violation: *twitterid → uid*

apply Armstrong's axioms and rules!

UserId (*twitterid, uid*)
BCNF

UserJoinsGroup' (*twitterid, uname, gid, fromDate*)
twitterid → uname
twitterid, gid → fromDate
BCNF violation: *twitterid → uname*

UserName (*twitterid, uname*)
BCNF

Member (*twitterid, gid, fromDate*)
BCNF

Recap

- Functional dependencies: a generalization of the key concept
- Non-key functional dependencies: a source of redundancy
- BCNF decomposition: a method for removing redundancies
  - BCNF decomposition is a lossless join decomposition
- BCNF: schema in this normal form has no redundancy due to FD's

BCNF = no redundancy?

- *User* (*uid, gid, place*)
  - A user can belong to multiple groups
  - A user can register places she's visited
  - Groups and places have nothing to do with other
  - FD's?
    - None
  - BCNF?
    - Yes
  - Redundancies?
    - Tons!

uid	*gid*	*place*
142	dps	Springfield
142	dps	Australia
456	abc	Springfield
456	abc	Morocco
456	gov	Springfield
456	gov	Morocco
…	…	…

Multivalued dependencies

- A multivalued dependency (MVD) has the form $X \twoheadrightarrow Y$, where X and Y are sets of attributes in a relation R
- $X \twoheadrightarrow Y$ means that whenever two rows in R agree on all the attributes of X, then we can swap their Y components and get two rows that are also in R

X	Y	Z
a	b_1	c_1
a	b_2	c_2
a	b_2	c_1
a	b_1	c_2
…	…	…

MVD examples

User (*uid*, *gid*, *place*)

- *uid* ↠ *gid*
- *uid* ↠ *place*
  - Intuition: given *uid*, attributes *gid* and *place* are "independent"
- *uid*, *gid* ↠ *place*
  - Trivial: LHS ∪ RHS = all attributes of R
- *uid*, *gid* ↠ *uid*
  - Trivial: LHS ⊇ RHS

Read this slide after looking at the examples

An elegant solution: "chase"

- Given a set of FD's and MVD's $\mathcal{D}$, does another dependency d (FD or MVD) follow from $\mathcal{D}$?
- Procedure
  - Start with the premise of d, and treat them as "seed" tuples in a relation
  - Apply the given dependencies in $\mathcal{D}$ repeatedly
    - If we apply an FD, we infer equality of two symbols
    - If we apply an MVD, we infer more tuples
  - If we infer the conclusion of d, we have a proof
  - Otherwise, if nothing more can be inferred, we have a counterexample

Proof by chase

- In $R(A, B, C, D)$, does $A \twoheadrightarrow B$ and $B \twoheadrightarrow C$ imply that $A \twoheadrightarrow C$?

Have:

	A	B	C	D
	a	b_1	c_1	d_1
	a	b_2	c_2	d_2
$A \twoheadrightarrow B$	a	b_2	c_1	d_1
	a	b_1	c_2	d_2
$B \twoheadrightarrow C$	a	b_2	c_1	d_2
	a	b_2	c_2	d_1
$B \twoheadrightarrow C$	a	b_1	c_2	d_1
	a	b_1	c_1	d_2

Need:

A	B	C	D	
a	b_1	c_2	d_1	✌
a	b_2	c_1	d_2	✌

Another proof by chase

- In $R(A, B, C, D)$, does $A \rightarrow B$ and $B \rightarrow C$ imply that $A \rightarrow C$?

Have:

A	B	C	D
a	b_1	c_1	d_1
a	b_2	c_2	d_2

Need: $c_1 = c_2$ ✌

$A \rightarrow B$ $\quad b_1 = b_2$

$B \rightarrow C$ $\quad c_1 = c_2$

In general, with both MVD's and FD's, chase can generate both new tuples and new equalities

Counterexample by chase

- In $R(A, B, C, D)$, does $A \twoheadrightarrow BC$ and $CD \rightarrow B$ imply that $A \rightarrow B$?

Have:

	A	B	C	D
	a	b_1	c_1	d_1
	a	b_2	c_2	d_2
$A \twoheadrightarrow BC$	a	b_2	c_2	d_1
	a	b_1	c_1	d_2

Counterexample!

Need: $b_1 = b_2$ 👎

4NF

- A relation R is in Fourth Normal Form (4NF) if
  - For every non-trivial MVD $X \twoheadrightarrow Y$ in R, X is a superkey
  - That is, all FD's and MVD's follow from "key → other attributes" (i.e., no MVD's and no FD's besides key functional dependencies)
- 4NF is stronger than BCNF
  - Because every FD is also a MVD

4NF decomposition algorithm

- Find a 4NF violation
  - A non-trivial MVD $X \twoheadrightarrow Y$ in R where X is not a superkey
- Decompose R into R_1 and R_2, where
  - R_1 has attributes $X \cup Y$
  - R_2 has attributes $X \cup Z$ (where Z contains R attributes not in X or Y)
- Repeat until all relations are in 4NF

- Almost identical to BCNF decomposition algorithm
- Any decomposition on a 4NF violation is lossless

4NF decomposition example

User (*uid*, *gid*, *place*)
4NF violation: *uid* ↠ *gid*

uid	*gid*	*place*
142	dps	Springfield
142	dps	Australia
456	abc	Springfield
456	abc	Morocco
456	gov	Springfield
456	gov	Morocco
…	…	…

Member (*uid*, *gid*)
4NF

uid	*gid*
142	dps
456	abc
456	gov
…	…

Visited (*uid*, *place*)
4NF

uid	*place*
142	Springfield
142	Australia
456	Springfield
456	Morocco
…	…

Other kinds of dependencies and normal forms

- Dependency preserving decompositions
- Join dependencies
- Inclusion dependencies
- 5NF, 3NF, 2NF
- See book if interested (not covered in class)

Summary

- Philosophy behind BCNF, 4NF:
  Data should depend on the key,
  the whole key,
  and nothing but the key!
  - You could have multiple keys though
- Redundancy is not desired typically
  - not always, mainly due to performance reasons
- Functional/multivalued dependencies – capture redundancy
- Decompositions – eliminate dependencies (should not be lossy!)
- Normal forms
  - Guarantees certain non-redundancy
  - BCNF, and 4NF
- How to decompose into BCNF, 4NF
- Chase