CMSC424: Normalization

Instructor: Amol Deshpande
amol@cs.umd.edu

Today's Class

- Review Reading Homework
  - Normalization overview; FDs
- More details
  - Normalization Theory
- Other things
  - iPython Notebook for Normalization
  - Project2: Let us know what help we can provide

Relational Database Design

- Where did we come up with the schema that we used ?
  - E.g. why not store the actor names with movies ?
- If from an E-R diagram, then:
  - Did we make the right decisions with the E-R diagram ?
- Goals:
  - Formal definition of what it means to be a "good" schema.
  - How to achieve it.

Movies Database Schema

Movie(*title, year*, length, inColor, studioName, producerC#)

StarsIn(movieTitle, movieYear, starName)

MovieStar(*name*, address, gender, birthdate)

MovieExec(name, address, *cert#,* netWorth)

Studio(*name*, address, presC#)

Changed to:

Movie(*title, year*, length, inColor, studioName, producerC#, starName)

<StarsIn merged into above>

MovieStar(*name*, address, gender, birthdate)

MovieExec(name, address, *cert#,* netWorth)

Studio(*name*, address, presC#)

Is this a good schema ???

Movie(*<u>title, year</u>*, length, inColor, studioName, producerC#, <u>starName</u>)

Title	Year	Length	inColor	StudioName	prodC#	StarName
Star wars	1977	121	Yes	Fox	128	Hamill
Star wars	1977	121	Yes	Fox	128	Fisher
Star wars	1977	121	Yes	Fox	128	H. Ford
King Kong	2005	187	Yes	Universal	150	Watts
King Kong	1933	100	no	RKO	20	Fay

<u>Issues:</u>

1. Redundancy ➔ higher storage, inconsistencies ("anomalies")

    update anomalies, insertion anamolies

2. Need nulls

    Unable to represent some information without using nulls

    How to store movies w/o actors (pre-productions etc) ?

Movie(*title, year*, length, inColor, studioName, producerC#, starNames)

Title	Year	Length	inColor	StudioName	prodC#	StarNames
Star wars	1977	121	Yes	Fox	128	{Hamill, Fisher, H. ford}
King Kong	2005	187	Yes	Universal	150	Watts
King Kong	1933	100	no	RKO	20	Fay

Issues:

3. Avoid sets

  - Hard to represent

  - Hard to query

Smaller schemas always good ????

Split Studio(*name*, address, presC#) into:

Studio1 (name, presC#)

Studio2(name, address)???

Name	presC#
Fox	101
Studio2	101
Universial	102

Name	Address
Fox	Address1
Studio2	Address1
Universial	Address2

This process is also called *"decomposition"*

Issues:

4. Requires more joins (w/o any obvious benefits)
5. Hard to check for some dependencies

What if the "address" is actually the presC#'s address ?

No easy way to ensure that constraint (w/o a join).

Smaller schemas always good ????

Decompose StarsIn(movieTitle, movieYear, starName) into:

StarsIn1(movieTitle, movieYear) StarsIn2(movieTitle, starName) ???

movieTitle	movieYear
Star wars	1977
King Kong	1933
King Kong	2005

movieTitle	starName
Star Wars	Hamill
King Kong	Watts
King Kong	Faye

Issues:

6. "joining" them back results in more tuples than what we started with

(King Kong, 1933, Watts) & (King Kong, 2005, Faye)

This is a "lossy" decomposition

We lost some constraints/information

The previous example was a "lossless" decomposition.

Desiderata

- No sets
- Correct and faithful to the original design
  - Avoid lossy decompositions
- As little redundancy as possible
  - To avoid potential anomalies
- No "inability to represent information"
  - Nulls shouldn't be required to store information
- Dependency preservation
  - Should be possible to check for constraints

Not always possible.

We sometimes relax these for:

simpler schemas, and *fewer joins during queries.*

Some of Your Questions

- Atomicity
  - It depends primarily on how you use it
  - A String is not really atomic (can be split into letters), but do you want to query the letters directly? Or would your queries operate on the strings?
- Which NF to use?
  - Your choice – Normalization theory is a tool to help you understand the tradeoffs
- Normal forms higher than 3NF?
  - Actually we always use 4NF – we will discuss later
- Trivial FDs
  - Just means that: RHS is contained in LHS – that's all

Approach

1. We will encode and list all our knowledge about the schema
   - Functional dependencies (FDs)

     SSN → name (means: SSN "implies" length)

   - If two tuples have the same "SSN", they must have the same "name"

     movietitle → length ???? Not true.

   - But, (movietitle, movieYear) → length --- True.
2. We will define a set of rules that the schema must follow to be considered good
   - "Normal forms": 1NF, 2NF, 3NF, BCNF, 4NF, …
   - A normal form specifies constraints on the schemas and FDs
3. If not in a "normal form", we modify the schema

FDs: Example 1

Title	Year	Length	StarName	Birthdate	producerC#	Producer -address	Prdocuer -name	netWorth
Plane Crazy	1927	6	NULL	NULL	WD100	Mickey Rd	Walt Disney	100000
Star Wars	1977	121	H. Ford	7/13/42	GL102	Tatooine	George Lucas	10^9
Star Wars	1977	121	M. Hamill	9/25/51	GL102	Tatooine	George Lucas	10^9
Star Wars	1977	121	C. Fisher	10/21/56	GL102	Tatooine	George Lucas	10^9
King Kong	1933	100	F. Wray	9/15/07	MC100	…	…	…
King Kong	2005	187	N. Watts	9/28/68	PJ100	Middle Earth	Peter Jackson	10^8

FDs: Example 2

State Name	State Code	State Population	County Name	County Population	Senator Name	Senator Elected	Senator Born	Senator Affiliation
Alabama	AL	4779736	Autauga	54571	Jeff Sessions	1997	1946	‘R’
Alabama	AL	4779736	Baldwin	182265	Jeff Sessions	1997	1946	‘R’
Alabama	AL	4779736	Barbour	27457	Jeff Sessions	1997	1946	‘R’
Alabama	AL	4779736	Autauga	54571	Richard Shelby	1987	1934	‘R’
Alabama	AL	4779736	Baldwin	182265	Richard Shelby	1987	1934	‘R’
Alabama	AL	4779736	Barbour	27457	Richard Shelby	1987	1934	‘R’

FDs: Example 3

Course ID	Course Name	Dept Name	Credits	Section ID	Semester	Year	Building	Room No.	Capacity	Time Slot ID

Functional dependencies

course_id → title, dept_name, credits
building, room_number → capacity
course_id, section_id, semester, year → building, room_number, time_slot_id

Examples from Quiz

- advisor(s_id, i_id, s_name, s_dept_name, i_name, i_dept_name)

Functional Dependencies

- Let R be a relation schema and

  $$\alpha \subseteq R \ \ and \ \ \beta \subseteq R$$

- The *functional dependency*

  $$\alpha \rightarrow \beta$$

  holds on R iff for any *legal* relations r(R), whenever two tuples t_1 and t_2 of r have same values for α, they have same values for β.

  $$t_1[\alpha] = t_2[\alpha] \ \Rightarrow \ t_1[\beta] = t_2[\beta]$$

- Example:

A	B
1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

Functional Dependencies

<u>Difference between holding on an *instance* and holding on *all legal relation*</u>

Title	Year	Length	inColor	StudioName	prodC#	StarName
Star wars	1977	121	Yes	Fox	128	Hamill
Star wars	1977	121	Yes	Fox	128	Fisher
Star wars	1977	121	Yes	Fox	128	H. Ford
King Kong	1933	100	no	RKO	20	Fay

Title → Year *holds on this instance*

Is this a true functional dependency ? ***No**.*

Two movies in different years can have the same name.

Can't draw conclusions based on a *single instance*

Need to use domain knowledge to decide which FDs hold

FDs and Redundancy

- Consider a table: R(A, B, C):
  - With FDs: B → C, and A → BC
  - So "A" is a Key, but "B" is not
- So: there is a FD whose left hand side is not a key
  - **Leads to redundancy**

Since B is not unique, it may be duplicated
Every time B is duplicated, so is C

Not a problem with A → BC
A can never be duplicated

A	B	C
a1	b1	c1
a2	b1	c1
a3	b1	c1
a4	b2	c2
a5	b2	c2
a6	b3	c3
a7	b4	c1

Not a duplication → Two different tuples just happen to have the same value for C

FDs and Redundancy

- Better to split it up

A	B
a1	b1
a2	b1
a3	b1
a4	b2
a5	b2
a6	b3
a7	b4

B	C
b1	c1
b2	c2
b3	c3
b4	c1

Not a duplication → Two different tuples just happen to have the same value for C

BCNF: Boyce-Codd Normal Form

- A relation schema *R* is "in BCNF" if:
  - Every functional dependency *A* → *B* that holds on it is *EITHER*:
    1. Trivial *OR*
    2. *A* is a *superkey* of *R*

- ***Why is BCNF good ?***
  - Guarantees that there can be no redundancy because of a functional dependency
  - Consider a relation *r(A, B, C, D)* with functional dependency *A* → *B* and two tuples: (a1, b1, c1, d1), and (a1, b1, c2, d2)
    - *b1* is repeated because of the functional dependency
    - BUT this relation is not in BCNF
      - *A* → *B* is neither trivial nor is *A* a superkey for the relation

Functional Dependencies

- Functional dependencies and *keys*
  - A *key* constraint is a specific form of a FD.
  - E.g. if *A* is a superkey for *R,* then:

    $$A \rightarrow R$$

  - Similarly for *candidate keys and primary keys.*

- Deriving FDs
  - A set of FDs may imply other FDs
  - *e.g. If $A \rightarrow B$, and $B \rightarrow C$, then clearly $A \rightarrow C$*
  - *We will see a formal method for inferring this later*

Definitions

1. A relation instance *r* *satisfies* a set of functional dependencies, *F*, if the FDs hold on the relation

2. F holds on a relation schema *R* if no legal (allowable) relation instance of *R* violates it

3. A functional dependency, *A → B*, is called *trivial* if:
   - B is a subset of A
   - e.g. Movieyear, length → length

4. Given a set of functional dependencies, *F*, its *closure*, F^+, is all the FDs that are implied by FDs in *F*.

Approach

1. We will encode and list all our knowledge about the schema
    - Functional dependencies (FDs)
    - Also:
        - Multi-valued dependencies (briefly discuss later)
        - Join dependencies etc…
2. We will define a set of rules that the schema must follow to be considered good
    - "Normal forms": 1NF, 2NF, 3NF, BCNF, 4NF, …
    - A normal form specifies constraints on the schemas and FDs
3. If not in a "normal form", we modify the schema

BCNF: Boyce-Codd Normal Form

- A relation schema *R* is "in BCNF" if:
  - Every functional dependency $A \rightarrow B$ that holds on it is *EITHER*:
    1. Trivial *OR*
    2. *A* is a *superkey* of *R*

- ***Why is BCNF good ?***
  - Guarantees that there can be no redundancy because of a functional dependency
  - Consider a relation *r(A, B, C, D)* with functional dependency $A \rightarrow B$ and two tuples: (a1, b1, c1, d1), and (a1, b1, c2, d2)
    - *b1* is repeated because of the functional dependency
    - BUT this relation is not in BCNF
      - $A \rightarrow B$ is neither trivial nor is *A* a superkey for the relation

BCNF and Redundancy

- *<u>Why does redundancy arise ?</u>*
  - Given a FD, A → B, if A is repeated (B – A) has to be repeated
  1. If rule 1 is satisfied, (B – A) is empty, so not a problem.
  2. If rule 2 is satisfied, then A can't be repeated, so this doesn't happen either

- Hence no redundancy because of FDs
  - Redundancy may exist because of other types of dependencies
    - Higher normal forms used for that (specifically, 4NF)
  - Data may naturally have duplicated/redundant data
    - We can't control that unless a FD or some other dependency is defined

Approach

1. We will encode and list all our knowledge about the schema
   - Functional dependencies (FDs); Multi-valued dependencies; Join dependencies etc…
2. We will define a set of rules that the schema must follow to be considered good
   - "Normal forms": 1NF, 2NF, 3NF, BCNF, 4NF, …
   - A normal form specifies constraints on the schemas and FDs
3. If not in a "normal form", we modify the schema
   - Through lossless decomposition (splitting)
   - Or direct construction using the dependencies information

BCNF

- What if the schema is not in BCNF ?
  - *Decompose (split) the schema into two pieces.*

- From the previous example: split the schema into:
  - *r1(A, B), r2(A, C, D)*
  - The first schema is in BCNF, the second one may not be (and may require further decomposition)
  - No repetition now: *r1* contains *(a1, b1)*, but *b1* will not be repeated

- Careful: you want the decomposition to be lossless
  - *No information should be lost*
    - The above decomposition is lossless
  - We will define this more formally later

Outline

- Mechanisms and definitions to work with FDs
  - Closures, candidate keys, canonical covers etc…
  - Armstrong axioms
- Decompositions
  - Loss-less decompositions, Dependency-preserving decompositions
- BCNF
  - How to achieve a BCNF schema
- BCNF may not preserve dependencies
- 3NF: Solves the above problem
- BCNF allows for redundancy
- 4NF: Solves the above problem

1. Closure

- Given a set of functional dependencies, *F,* its *closure, F^+ ,* is all FDs that are implied by FDs in *F.*
  - *e.g. If A → B, and B → C, then clearly A → C*
- We can find F+ by applying Armstrong's Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
  - if $\alpha \rightarrow \beta$, then $\gamma\, \alpha \rightarrow \gamma\, \beta$ **(augmentation)**
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
  - sound (generate only functional dependencies that actually hold)
  - complete (generate all functional dependencies that hold)

Additional rules

- If $\alpha \rightarrow \beta$ *and* $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta\gamma$ **(union)**
- If $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ **(decomposition)**
- If $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$ **(pseudotransitivity)**

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

Example

- $R = (A, B, C, G, H, I)$
  $F = \{\ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,
      and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,
      and then transitivity

2. Closure of an attribute set

- Given a set of attributes *A* and a set of FDs *F*, *closure of A under F* is the set of all attributes implied by *A*

- In other words, the largest *B* such that: $A \rightarrow B$

- Redefining *super keys:*
  - *The closure of a super key is the entire relation schema*

- Redefining *candidate keys:*

    1. It is a super key

    2. No subset of it is a super key

Computing the closure for *A*

- Simple algorithm

- 1. Start with $B = A$.
- 2. Go over all functional dependencies, $\beta \rightarrow \gamma$, in F^+
- 3. If $\beta \subseteq B$, *then*

  Add γ to B
- 4. Repeat till B changes

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$

- $(AG)^+$?
  - *1.* result = AG
  - 2.result = ABCG (A → C and A → B)
  - 3.result = ABCGH (CG → H and CG ⊆ AGBC)
  - 4.result = ABCGHI (CG → I and CG ⊆ AGBCH

- Is (AG) a candidate key ?
  1. It is a super key.
  2. (A+) = BCH, (G+) = G.

  YES.

Uses of attribute set closures

- Determining *superkeys and candidate keys*

- Determining if *A* → *B* is a valid FD
  - Check if A+ contains B

- Can be used to compute *F+*

3. Extraneous Attributes

- Consider *F*, and a functional dependency, $A \rightarrow B$.
- "Extraneous": Are there any attributes in *A or B* that can be safely removed ?

  Without changing the constraints implied by F

- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - C is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting C
  - ie., given: $A \rightarrow C$, and $AB \rightarrow D$, we can use Armstrong Axioms to infer $AB \rightarrow CD$

4. Canonical Cover

- A *canonical cover* for *F* is a set of dependencies F_c such that
  - F logically implies all dependencies in $F_{c,}$ and
  - F_c logically implies all dependencies in F, and
  - No functional dependency in F_c contains an extraneous attribute, and
  - Each left side of functional dependency in F_c is unique
- In some (vague) sense, it is a *minimal* version of *F*
- Read up algorithms to compute F_c

Outline

- Mechanisms and definitions to work with FDs
  - Closures, candidate keys, canonical covers etc…
  - Armstrong axioms
- Decompositions
  - Loss-less decompositions, Dependency-preserving decompositions
- BCNF
  - How to achieve a BCNF schema
- BCNF may not preserve dependencies
- 3NF: Solves the above problem
- BCNF allows for redundancy
- 4NF: Solves the above problem

Loss-less Decompositions

- Definition: A decomposition of *R* into *(R1, R2)* is called *lossless* if, for all legal instance of *r(R):*

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r)$$

- In other words, projecting on *R1 and R2,* and *joining back,* results in the relation you started with

- Rule: A decomposition of *R* into *(R1, R2)* is *lossless,* iff:

$$R1 \cap R2 \rightarrow R1 \qquad or \qquad R1 \cap R2 \rightarrow R2$$

in *F+.*

Dependency-preserving Decompositions

Is it easy to check if the dependencies in F hold ?

Okay as long as the dependencies can be checked in the same table.

Consider $R = (A, B, C)$, and $F = \{A \rightarrow B, B \rightarrow C\}$

1. Decompose into R1 = (A, B), and R2 = (A, C)

   Lossless ? Yes.

   But, makes it hard to check for $B \rightarrow C$

   The data is in multiple tables.

2. On the other hand, *R1 = (A, B), and R2 = (B, C),*

   is both lossless and dependency-preserving

   Really ? What about $A \rightarrow C$ *?*

   If we can check $A \rightarrow B$, and $B \rightarrow C$, $A \rightarrow C$ is implied.

Dependency-preserving Decompositions

- Definition:
  - Consider decomposition of *R into R1, …, Rn.*
  - Let F_i be the set of dependencies F^+ that include only attributes in R_i.

- The decomposition is dependency preserving, if

$$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$

Outline

- Mechanisms and definitions to work with FDs
  - Closures, candidate keys, canonical covers etc…
  - Armstrong axioms
- Decompositions
  - Loss-less decompositions, Dependency-preserving decompositions
- BCNF
  - How to achieve a BCNF schema
- BCNF may not preserve dependencies
- 3NF: Solves the above problem
- BCNF allows for redundancy
- 4NF: Solves the above problem

BCNF

- Given a relation schema *R,* and a set of functional dependencies *F,* if every FD, *A* → *B*, is either:
  1. Trivial
  2. *A* is a *superkey* of *R*

  Then, *R* is in *BCNF (Boyce-Codd Normal Form)*

- What if the schema is not in BCNF ?
  - *Decompose (split) the schema into two pieces.*
  - Careful: you want the decomposition to be lossless

Achieving BCNF Schemas

For all dependencies *A* → *B* in *F+,* check if *A* is a superkey

By using attribute closure

If not, then

Choose a dependency in F+ that breaks the BCNF rules, say A → B

Create R1 = A B

Create R2 = A (R – B – A)

Note that: R1 ∩ R2 = A and A → AB (= R1), so this is lossless decomposition

Repeat for *R1, and R2*

By defining F1+ to be all dependencies in F that contain only attributes in R1

Similarly F2+

Example 1

R = (A, B, C)

F = {A → B, B → C}

Candidate keys = {A}

BCNF = No. B → C violates.

B → C

R1 = (B, C)

F1 = {B → C}

Candidate keys = {B}

BCNF = true

R2 = (A, B)

F2 = {A → B}

Candidate keys = {A}

BCNF = true

Example 2-1

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

F = {$A \rightarrow B$, $BC \rightarrow D$}

Candidate keys = {ACE}

BCNF = Violated by {$A \rightarrow B$, $BC \rightarrow D$} etc…

$A \rightarrow B$

From $A \rightarrow B$ and $BC \rightarrow D$ by pseudo-transitivity

R1 = (A, B)

F1 = {$A \rightarrow B$}

Candidate keys = {A}

BCNF = true

R2 = (A, C, D, E)

F2 = {$AC \rightarrow D$}

Candidate keys = {ACE}

BCNF = false ($AC \rightarrow D$)

$AC \rightarrow D$

R3 = (A, C, D)

F3 = {$AC \rightarrow D$}

Candidate keys = {AC}

BCNF = true

R4 = (A, C, E)

F4 = {} [[only trivial]]

Candidate keys = {ACE}

BCNF = true

Dependency preservation ???

We can check:

$A \rightarrow B$ (R1), $AC \rightarrow D$ (R3),

but we lost $BC \rightarrow D$

So this is not a dependency-preserving decomposition

Example 2-2

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

F = {A → B, BC → D}

Candidate keys = {ACE}

BCNF = Violated by {A → B, BC → D} etc…

BC → D

R1 = (B, C, D)

F1 = {BC → D}

Candidate keys = {BC}

BCNF = true

R2 = (B, C, A, E)

F2 = {A → B}

Candidate keys = {ACE}

BCNF = false (A → B)

A → B

R3 = (A, B)

F3 = {A → B}

Candidate keys = {A}

BCNF = true

R4 = (A, C, E)

F4 = {} [[only trivial]]

Candidate keys = {ACE}

BCNF = true

Dependency preservation ???

We can check:

BC → D (R1), A → B (R3),

Dependency-preserving decomposition

Example 3

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

F = {A → BC, E → HA}

Candidate keys = {DE}

BCNF = Violated by {A → BC} etc…

A → BC

R1 = (A, B, C)

F1 = {A → BC}

Candidate keys = {A}

BCNF = true

R2 = (A, D, E, H)

F2 = {E → HA}

Candidate keys = {DE}

BCNF = false (E → HA)

E → HA

R3 = (E, H, A)

F3 = {E → HA}

Candidate keys = {E}

BCNF = true

R4 = (ED)

F4 = {} [[only trivial]]

Candidate keys = {DE}

BCNF = true

Dependency preservation ???

We can check:

A → BC (R1), E → HA (R3),

Dependency-preserving decomposition

Outline

- Mechanisms and definitions to work with FDs
  - Closures, candidate keys, canonical covers etc…
  - Armstrong axioms
- Decompositions
  - Loss-less decompositions, Dependency-preserving decompositions
- BCNF
  - How to achieve a BCNF schema
- BCNF may not preserve dependencies
- 3NF: Solves the above problem
- BCNF allows for redundancy
- 4NF: Solves the above problem

BCNF may not preserve dependencies

- $R = (J, K, L\}$
- $F = \{JK \rightarrow L, L \rightarrow K\}$

- Two candidate keys = JK and JL

- R is not in BCNF

- Any decomposition of R will fail to preserve

$$JK \rightarrow L$$

- This implies that testing for $JK \rightarrow L$ requires a join

BCNF may not preserve dependencies

- Not always possible to find a dependency-preserving decomposition that is in BCNF.

- PTIME to determine if there exists a dependency-preserving decomposition in BCNF
  - in size of F

- NP-Hard to find one if it exists

- Better results exist if F satisfies certain properties

Outline

- Mechanisms and definitions to work with FDs
  - Closures, candidate keys, canonical covers etc…
  - Armstrong axioms
- Decompositions
  - Loss-less decompositions, Dependency-preserving decompositions
- BCNF
  - How to achieve a BCNF schema
- BCNF may not preserve dependencies
- 3NF: Solves the above problem
- BCNF allows for redundancy
- 4NF: Solves the above problem

3NF

- Definition: *Prime* attributes
  An attribute that is contained in a candidate key for R

- Example 1:
  - R = (A, B, C, D, E, H}, F = {A → BC, E → HA},
  - Candidate keys = {ED}
  - Prime attributes: D, E

- Example 2:
  - R = (J, K, L), F = {JK → L, L → K},
  - Candidate keys = {JL, JK}
  - Prime attributes: J, K, L

- Observation/Intuition:
  1. A *key* has no redundancy (is not repeated in a relation)
  2. A *prime attribute* has limited redundancy

3NF

- Given a relation schema *R,* and a set of functional dependencies *F,* if every FD, $A \rightarrow B$, is either:

  1. Trivial, or
  2. *A* is a *superkey* of *R, or*
  3. *All attributes in (B – A) are prime*

  Then, *R* is in *3NF (3rd Normal Form)*

- *Why is 3NF good ?*

3NF and Redundancy

- ***Why does redundancy arise ?***
  - Given a FD, A → B, if A is repeated (B – A) has to be repeated
  1. If rule 1 is satisfied, (B – A) is empty, so not a problem.
  2. If rule 2 is satisfied, then A can't be repeated, so this doesn't happen either
  3. If not, rule 3 says (B – A) must contain only *prime attributes*

     This limits the redundancy somewhat.

- So 3NF relaxes BCNF somewhat by allowing for some (hopefully limited) redundancy
- Why ?
  - *There always exists a dependency-preserving lossless decomposition in 3NF.*

Decomposing into 3NF

- A *synthesis* algorithm
- Start with the canonical cover, and construct the 3NF schema directly
- Homework assignment.

Outline

- Mechanisms and definitions to work with FDs
  - Closures, candidate keys, canonical covers etc…
  - Armstrong axioms
- Decompositions
  - Loss-less decompositions, Dependency-preserving decompositions
- BCNF
  - How to achieve a BCNF schema
- BCNF may not preserve dependencies
- 3NF: Solves the above problem
- BCNF allows for redundancy
- 4NF: Solves the above problem

BCNF and redundancy

MovieTitle	MovieYear	StarName	Address
Star wars	1977	Harrison Ford	Address 1, LA
Star wars	1977	Harrison Ford	Address 2, FL
Indiana Jones	198x	Harrison Ford	Address 1, LA
Indiana Jones	198x	Harrison Ford	Address 2, FL
Witness	19xx	Harrison Ford	Address 1, LA
Witness	19xx	Harrison Ford	Address 2, FL
…	…	…	…

Lot of redundancy

FDs ? No non-trivial FDs.

So the schema is trivially in BCNF (and 3NF)

What went wrong ?

Multi-valued Dependencies

- The redundancy is because of *multi-valued dependencies*
- *Denoted:*

  $$starname \twoheadrightarrow address$$

  $$starname \twoheadrightarrow movietitle, movieyear$$

- Should not happen if the schema is constructed from an E/R diagram
- Functional dependencies are a special case of multi-valued dependencies

Outline

- Mechanisms and definitions to work with FDs
  - Closures, candidate keys, canonical covers etc…
  - Armstrong axioms
- Decompositions
  - Loss-less decompositions, Dependency-preserving decompositions
- BCNF
  - How to achieve a BCNF schema
- BCNF may not preserve dependencies
- 3NF: Solves the above problem
- BCNF allows for redundancy
- 4NF: Solves the above problem

4NF

- Similar to BCNF, except with MVDs instead of FDs.

- Given a relation schema *R,* and a set of multi-valued dependencies *F,* if every MVD, $A \rightarrow\rightarrow B$, is either:

  1. Trivial, or
  2. *A* is a *superkey* of *R*

  Then, *R* is in *4NF (4th Normal Form)*

- 4NF → BCNF → 3NF → 2NF → 1NF:
  - If a schema is in 4NF, it is in BCNF.
  - If a schema is in BCNF, it is in 3NF.
- Other way round is untrue.

Comparing the normal forms

	3NF	**BCNF**	**4NF**
Eliminates redundancy because of FD's	Mostly	Yes	Yes
Eliminates redundancy because of MVD's	No	No	Yes
Preserves FDs	Yes.	Maybe	Maybe
Preserves MVDs	Maybe	Maybe	Maybe

4NF is typically desired and achieved.

A good E/R diagram won't generate non-4NF relations at all

Choice between 3NF and BCNF is up to the designer

Database design process

- Three ways to come up with a schema

1. Using E/R diagram
   - If good, then little normalization is needed
   - Tends to generate 4NF designs
2. A universal relation *R* that contains all attributes.
   - Called universal relation approach
   - Note that MVDs will be needed in this case
3. An *ad hoc* schema that is then normalized
   - MVDs may be needed in this case

Recap

- What about 1st and 2nd normal forms ?
- 1NF:
  - Essentially says that no set-valued attributes allowed
  - Formally, a domain is called *atomic* if the elements of the domain are considered indivisible
  - A schema is in 1NF if the domains of all attributes are atomic
  - We assumed 1NF throughout the discussion
    - Non 1NF is just not a good idea
- 2NF:
  - Mainly historic interest
  - See Exercise 7.15 in the book

Recap

- We would like our relation schemas to:
  - Not allow potential redundancy because of FDs or MVDs
  - Be *dependency-preserving:*
    - Make it easy to check for dependencies
    - Since they are a form of integrity constraints
- Functional Dependencies/Multi-valued Dependencies
  - Domain knowledge about the data properties
- Normal forms
  - Defines the rules that schemas must follow
  - 4NF is preferred, but 3NF is sometimes used instead

Recap

- Denormalization
  - After doing the normalization, we may have too many tables
  - We may *denormalize* for performance reasons
    - Too many tables → too many joins during queries
  - A better option is to use *views* instead
    - So if a specific set of tables is joined often, create a view on the join

- More advanced normal forms
  - project-join normal form (PJNF or 5NF)
  - domain-key normal form
  - Rarely used in practice