# CS411
# Database Systems

## 05: Relational Schema Design

# Why Do We Learn This?

# Motivation

- We have designed ER diagram, and translated it into a relational db schema R = set of R1, R2, ...
- Now what?
- We can do the following
  – specify all relevant constraints over R
  – implement R in SQL
  – start using it, making sure the constraints always remain valid
- However, R may not be well-designed, thus causing us a lot of problems

# Q: Is this a good design?

Persons with several phones:

| Address | SSN | Phone Number |
|---|---|---|
| 10 Green | 123-321-99 | (201) 555-1234 |
| 10 Green | 123-321-99 | (206) 572-4312 |
| 431 Purple | 909-438-44 | (908) 464-0028 |
| 431 Purple | 909-438-44 | (212) 555-4000 |

# Potential Problems

- Redundancy

- Update anomalies

- Deletion anomalies

# Better Designs Exist

**Break the relation into two:**

| SSN | Address |
| --- | --- |
| 123-321-99 | 10 Green |
| 909-438-44 | 431 Purple |

| SSN | Phone Number |
| --- | --- |
| 123-321-99 | (201) 555-1234 |
| 123-321-99 | (206) 572-4312 |
| 909-438-44 | (908) 464-0028 |
| 909-438-44 | (212) 555-4000 |

# How do We Obtain a Good Design?

- Start with the original db schema R
- Transform it until we get a good design R*
- Some desirable properties for R*
  - must preserve the information of R
  - must have minimal amount of redundancy
  - must be "dependency preserving" (we'll come to this later)
  - (must also give good query performance)

# How do We Obtain a Good Design?

# Normal Forms

- DB gurus have developed many "normal forms"
- Most important ones
  - Boyce-Codd, 3rd, and 4th normal forms
- If R* is in one of these forms, then R* is guaranteed to achieve certain good properties
  - e.g., if R* is in Boyce-Codd NF, it is guaranteed to not have certain types of redundancy
- DB gurus have also developed algorithms to transform R into R* that is in some of these normal forms

# Normal Forms (cont.)

- DB gurus have also discussed trade-offs among normal forms
- Thus, all we have to do is
  - learn these forms
  - transform R into R* in one of these forms
  - carefully evaluate the trade-offs
- To understand these normal forms we'll need to understand certain types of constraints
  - functional dependencies and keys

# Functional Dependencies and Keys

# Functional Dependencies

- A form of constraint (hence, part of the schema)
- Finding them is part of the database design
- Used heavily in schema refinement

Definition:

If two tuples agree on the attributes

$A_1, A_2, \ldots A_n$

then they must also agree on the attributes

$B_1, B_2, \ldots B_m$

Formally: $A_1, A_2, \ldots A_n \longrightarrow B_1, B_2, \ldots B_m$

# Examples

| EmpID | Name | Phone | Position |
|---|---|---|---|
| E0045 | Smith | 1234 | Clerk |
| E1847 | John | 9876 | Salesrep |
| E1111 | Smith | 9876 | Salesrep |
| E9999 | Mary | 1234 | Lawyer |

- EmpID $\longrightarrow$ Name, Phone, Position
- Position $\longrightarrow$ Phone
- but Phone $\not\longrightarrow$ Position

# More examples

Product: name, manufacturer $\longrightarrow$ price
Person: ssn $\longrightarrow$ name, age
Company: name $\longrightarrow$ stock price, president

# Q: From this, can you conclude phone → SSN?

| SSN | Phone Number |
|---|---|
| 123-321-99 | (201) 555-1234 |
| 123-321-99 | (206) 572-4312 |
| 909-438-44 | (908) 464-0028 |
| 909-438-44 | (212) 555-4000 |

# Keys are a type of FD

- After defining FDs, we can now define keys
- Key of a relation R is a set of attributes that
  - functionally determines all attributes of R
  - none of its subsets determines all attributes of R
- Superkey
  - a set of attributes that contains a key

# Rules about Functional Dependencies

# The Splitting/Combining Rule

- $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$

- Equivalent to:

  $A_1 A_2 \ldots A_n \rightarrow B_1$;

  $A_1 A_2 \ldots A_n \rightarrow B_2$;

  …

  $A_1 A_2 \ldots A_n \rightarrow B_m$

- Can replace one for the other.

# Trivial Functional Dependencies

- $A_1A_2...A_n \rightarrow A_1$

- In general,

  $A_1A_2...A_n \rightarrow B_1B_2\ldots B_m$

  if $\{B_1B_2\ldots B_m\} \subseteq \{A_1A_2...A_n\}$

- Always holds.

# Closure of a Set of Attributes

Given a set of attributes $\{A1, \ldots, An\}$ and a set of dependencies S.
Problem: find all attributes $B$ such that:

any relation which satisfies S also satisfies:

$A1, \ldots, An \rightarrow B$

The **closure** of $\{A1, \ldots, An\}$, denoted $\{A1, \ldots, An\}^+$, is the set of all such attributes $B$

We will discuss the motivations for attribute closures soon

# Algorithm to Compute Closure

Split the FDs in S so that every FD has a single attribute on the right.

Start with $X=\{A_1A_2\ldots A_n\}$.

**Repeat until** X doesn't change **do**:

If ($B_1B_2\ldots B_m \rightarrow C$) is in S, such that $B_1,B_2,\ldots B_m$ are in X and C is not in X:

add C to X.

X is now the correct value of $\{A_1A_2\ldots A_n\}^+$

# Example

$$
\begin{aligned}
A\ B &\longrightarrow C \\
A\ D &\longrightarrow E \\
B &\longrightarrow D \\
A\ F &\longrightarrow B
\end{aligned}
$$

Closure of {A,B}: X = {A, B, C, D, E}

Closure of {A, F}: X = {A, F, B, D, C, E}

# Is this algorithm correct?

- Yes. See Text (Section 3.2.5) for proof.

# Usage for Attribute Closure

- Test if X is a superkey
  - compute X+, and check if X+ contains all attrs of R

- Check if X $\rightarrow$ Y holds
  - by checking if Y is contained in X+

# Reviewing Quiz 1

- Show that each of the following are not valid rules about FD's, by giving example relations that satisfy the given FDs (following the "If"), but not the FD that allegedly follows (after the "then").

  (1) If **A --> B then B --> A**

  (2) If **AB --> C and A --> C then B --> C.**
- **(1) A = SSN, B = Name**
- **(2) A = SSN, B = Phone, C = Name**

# Closure of a set of FDs

- Given a relation schema R & a set S of FDs
  - is the FD f logically implied by S?
- Example
  - R = {A,B,C,G,H,I}
  - S = A ➔B, A ➔C, CG ➔ H, CG ➔ I, B ➔ H
  - would A ➔ H be logically implied?
  - yes (you can prove this, using the definition of FD)
- Closure of S: S+ = all FDs logically implied by S
- How to compute S+?

# Computing $S^+$

- To check if A ➔ B is true, we can compute $A^+$
- To compute all A ➔ B implied by S, i.e., to compute the closure of S, we can use a particular algorithm.
- This algorithm depends on the so-called "Armstrong's axioms".

# Armstrong's Axioms

- Reflexivity rule
  - A1A2...An ➔ a subset of A1A2...An
- Augmentation rule
  - A1A2...An ➔ B1B2...Bm, then
    A1A2...An C1C2..Ck ➔ B1B2...Bm C1C2...Ck
- Transitivity rule
  - A1A2...An ➔ B1B2...Bm and
    B1B2...Bm ➔ C1C2...Ck, then
    A1A2...An ➔ C1C2...Ck

# Inferring S+ using Armstrong's Axioms

- S+ = S
- Loop
  - For each f in S+, apply reflexivity and augmentn rules
  - add the new FDs to S+
  - For each pair of FDs in S+, apply the transitivity rule
  - add the new FD to S+
- Until S+ does not change any further

# Too many rules? Not really.

- The "combining rule" can be derived from Armstrong's axioms
  - $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - (X, Y, Z are sets of attributes)

- The "splitting rule" can also be derived from Armstrong's axioms
  - $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

# More on FDs

- Given a set of FDs: S.

- Let’s say the set S’ is *equivalent* to S, in the sense that S’ can be inferred from S and vice versa.

- Any such S’ is said to be a *basis* for S.

- “Minimal basis”: see Section 3.2.7.

# But we were talking about schema design

- Armed with the concepts of “functional dependency” and “superkey” (and tools to reason about them), we will now define what a “good schema” is.

# Desirable Properties of Schema Refinement

1) minimize redundancy
2) avoid info loss
3) preserve dependency
4) ensure good query performance

# Normal Forms

**First Normal Form** = all attributes are atomic

**Second Normal Form** (2NF) = old and obsolete

**Boyce Codd Normal Form** (BCNF)

**Third Normal Form** (3NF)

**Fourth Normal Form** (4NF)

Others...

# Boyce-Codd Normal Form

A simple condition for removing anomalies from relations:

A relation R is in BCNF if and only if:

Whenever there is a nontrivial FD $A_1A_2\ldots A_n \rightarrow B$
for R , it is the case that $A_1A_2\ldots A_n$
is a super-key for R.

In English (though a bit vague):

Whenever a set of attributes of *R* is determining another attribute,
it should determine ***all*** attributes of *R*.

# Example

| Name | SSN | Phone Number |
| --- | --- | --- |
| Fred | 123-321-99 | (201) 555-1234 |
| Fred | 123-321-99 | (206) 572-4312 |
| Joe | 909-438-44 | (908) 464-0028 |
| Joe | 909-438-44 | (212) 555-4000 |

What are the dependencies?

SSN → Name

Is the left side a superkey?

Is it in BCNF?

# Decompose it into BCNF

| SSN | Name |
|---|---|
| 123-321-99 | Fred |
| 909-438-44 | Joe |

SSN → Name

| SSN | Phone Number |
|---|---|
| 123-321-99 | (201) 555-1234 |
| 123-321-99 | (206) 572-4312 |
| 909-438-44 | (908) 464-0028 |
| 909-438-44 | (212) 555-4000 |

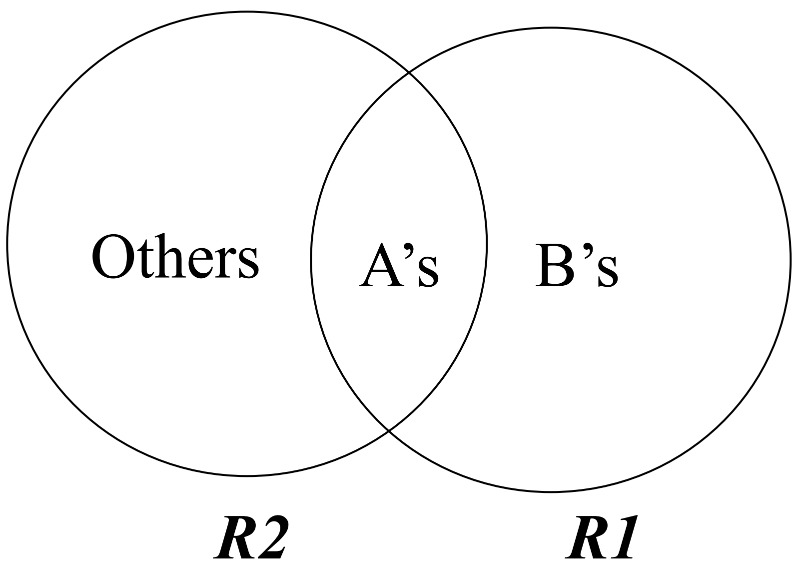
# BCNF Decomposition

Find a dependency that violates the BCNF condition:

$$A_1, A_2, \ldots A_n \longrightarrow B_1, B_2, \ldots B_m$$

Heuristic : choose $B_1, B_2, \ldots B_m$ "as large as possible"

Decompose:




Continue until there are no BCNF violations left.

# Example Decomposition

Person:

| Name | SSN | Age | EyeColor | PhoneNumber |
|---|---|---|---|---|
| | | | | |

Functional dependencies:

SSN ➔ Name, Age, Eye Color

BCNF: Person1(SSN, Name, Age, EyeColor),
Person2(SSN, PhoneNumber)

What if we also had an attribute Draft-worthy, and the FD:

Age ➔ Draft-worthy

# Example

- Person (Name, SSN, Age, EyeColor, Phone, Draftworthy)
- FD: SSN $\rightarrow$ Name, Age, EyeColor
- FD: Age $\rightarrow$Draftworthy

# Example

- Movie (title, yr, length, genre, studioName, starName)
- (Title, year, starName) is a key
- FD: Title, year ➔ length, genre, studioName

# Example

- Movie (title, yr, studioName, President, PresAddr)
- FD: Title, yr → studioName
- FD: studioName → President
- FD: President → PresAddr

# Two-attribute relations

- Let A and B be the only two attributes of R
- Claim: R is in BCNF. (See Example 3.17.)
- If $A \rightarrow B$ is true, $B \rightarrow A$ is not:
- If $B \rightarrow A$ is true, $A \rightarrow B$ is not:
- If $A \rightarrow B$ is true, $B \rightarrow A$ is true:

# BCNF Decomposition: The Algorithm

- Input: relation R, set S of FDs over R

1) Check if R is in BCNF, if not:

    a) pick a violation FD f: A ➔ B

    b) compute A+

    c) create R1 = A+, R2 = A union (R – A+)

    d) compute all FDs over R1, using R and S. Repeat similarly for R2. (See Algorithm 3.12)

    e) Repeat Step 1 for R1 and R2

4) Stop when all relations are BCNF, or are two-attributes

# Q: Is BCNF Decomposition unique?

# Q: Does BCNF Decompos. always exist?

# Properties of BCNF

- BCNF removes certain types of redundancy
  - those caused by adding many-many or one-many relationships

- For examples of redundancy that it cannot remove, see "multi-valued redundancy”
  - But this is not in curriculum

# Properties of BCNF

- BCNF Decomposition avoids information loss
  - You can construct the original relation instance from the decomposed relations' instances.

# Desirable Properties of Schema Refinement (again)

1) minimize redundancy
2) avoid info loss
3) preserve dependency
4) ensure good query performance

# An easy decomposition?

- We saw that two-attribute relations are in BCNF.

- Why don’t we break any R(A,B,C,D,E) into R1 (A,B); R2(B,C); R3(C,D); R4(D,E)? Why bother with finding BCNF violations etc.?

# Example of the "easy decomposition"

- R = (A,B,C); decomposed into R1(A,B); R2(B,C)

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

| A | B |
|---|---|
| 1 | 2 |
| 4 | 5 |

| B | C |
|---|---|
| 2 | 3 |
| 5 | 6 |

# Example of the "easy decomposition"

- R = (A,B,C); decomposed into R1(A,B); R2(B,C)

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

| A | B |
|---|---|
| 1 | 2 |
| 4 | 2 |

| B | C |
|---|---|
| 2 | 3 |
| 2 | 5 |

Join

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 5 |
| 1 | 2 | 5 |
| 4 | 2 | 3 |

- We get back some "bogus tuples" !

# Lossless Decompositions

A decomposition is *lossless* if we can recover:

R(A,B,C)

$\downarrow$ Decompose

{ R1(A,B) , R2(A,C) }

$\downarrow$ Recover

R'(A,B,C) = R(A,B,C)

R' is in general larger than R. Must ensure R' = R

# BCNF Decomposition is Lossless

R(A, B, C),       $A \rightarrow C$

BCNF: R1(A,B), R2(A,C)

Some tuple (a,b,c) in R
decomposes into (a,b) in R1
and (a,c) in R2

Recover tuples in R: (a,b,c),

# BCNF Decomposition is Lossless

R(A, B, C),   $A \rightarrow C$

BCNF: R1(A,B), R2(A,C)

Some tuple (a,b,c) in R   (a,b',c') also in R
decomposes into (a,b) in R1   (a,b') also in R1
and (a,c) in R2   (a,c') also in R2

Recover tuples in R: (a,b,c),

# BCNF Decomposition is Lossless

$R(A, B, C)$,  $A \rightarrow C$

BCNF: $R1(A,B)$, $R2(A,C)$

Some tuple (a,b,c) in R  (a,b',c') also in R
decomposes into (a,b) in R1  (a,b') also in R1
and (a,c) in R2  (a,c') also in R2

Recover tuples in R: (a,b,c),  (a,b,c'), (a,b',c), (a,b',c') also in R

Is any of these a "bogus tuple" (not present in R)?

# BCNF Decomposition is Lossless

R(A, B, C), $A \rightarrow C$

BCNF: R1(A,B), R2(A,C)

Some tuple (a,b,c) in R (a,b',c') also in R
decomposes into (a,b) in R1 (a,b') also in R1
and (a,c) in R2 (a,c') also in R2

Recover tuples in R: (a,b,c), (a,b,c'), (a,b',c), (a,b',c') also in R

Is any of these a "bogus tuple" (not present in R)?

No! Also see text 3.4.1 for proof. (But skip 3.4.2 – 3.4.3.)

# Desirable Properties of Schema Refinement (again)

1) **minimize redundancy**
2) **avoid info loss**
3) preserve dependency
4) ensure good query performance

BCNF

# However,

- BCNF is not always dependency preserving
- In fact, some times we cannot find a BCNF decomposition that is dependency preserving
- Can handle this situation using 3NF
- But what is "dependency preserving"?

# Normal Forms

**First Normal Form** = all attributes are atomic

**Second Normal Form** (2NF) = old and obsolete

**Boyce Codd Normal Form** (BCNF)

**Third Normal Form** (3NF)

**Fourth Normal Form** (4NF)

Others...

# 3NF: A Problem with BCNF

| Phone | Address | Name |
|---|---|---|
| | | |

FD's: Phone → Address; Address, Name → Phone

So, there is a BCNF violation (Phone ➔ Address), and we decompose.

| Phone | Address |
|---|---|
| | |

Phone → Address

| Phone | Name |
|---|---|
| | |

No FDs

# So where's the problem?

| Phone | Address |
|---|---|
| 1234 | 10 Downing |
| 5678 | 10 Downing |

| Phone | Name |
|---|---|
| 1234 | John |
| 5678 | John |

No problem so far. All *local* FD's are satisfied.

Let's put all the data into a single table:

| Phone | Address | Name |
|---|---|---|
| 1234 | 10 Downing | John |
| 5678 | 10 Downing | John |

**Violates the dependency: Address, Name $\rightarrow$ Phone**

# Preserving FDs

- What if, when a relation is decomposed, the X of an $X \rightarrow Y$ ends up only in one of the new relations and the Y ends up only in another?
- Such a decomposition is not "dependency-preserving."
- Sometimes it is not possible to decompose a relation into BCNF relations that have both lossless-join and dependency preservation properties. May need to make a tradeoff.

# An alternative: 3rd Normal Form (3NF)

A simple condition for removing anomalies from relations:

A relation R is in 3rd normal form if :

Whenever there is a nontrivial dependency $A_1, A_2, ..., A_n \rightarrow B$ for R , then $\{A_1, A_2, ..., A_n\}$ is a super-key for R,
or B is part of a key.

"Excuse" the "Phone ➔ Address" FD from causing a decomposition

# 3NF vs. BCNF

- R is in BCNF if whenever $X \rightarrow A$ holds, then X is a superkey.
- Slightly stronger than 3NF.
- Example: $R(A,B,C)$ with $\{A,B\} \rightarrow C$, $C \rightarrow A$
  - 3NF but not BCNF

# Decomposing R into 3NF

- The algorithm is complicated
- 1. Get a "minimal basis" *G* of given FDs (Section 3.2.7)
- 2. For each FD X → A in the minimal basis *G*, use XA as the schema of a new relation.
- 3. If none of the schemas from Step 2 is a superkey, add another relation whose schema is a key for the original relation.
- Result will be lossless, will be dependency-preserving, 3NF; might not be BCNF
- *Example 3.27 in textbook.*
- But skip Section 3.5.3.

# Desirable Properties of Schema Refinement (again)

1) minimize redundancy
2) **avoid info loss**
3) **preserve dependency**
4) ensure good query performance

3NF

# Fact of life...

***Finding a decomposition which is both lossless and dependency-preserving is not always possible.***

***Guideline: Aim for BCNF and settle for 3NF***

# Multi-valued Dependencies and 4NF

we will not cover this.

# Caveat

- Normalization is not the be-all and end-all of DB design
- Example: suppose attributes A and B are always used together, but normalization theory says they should be in different tables.
  - decomposition might produce unacceptable performance loss (extra disk reads)