Logical Database Design: Entity-Relation Models

Functional Dependencies
Schema Normalization

Alvin Cheung
Fall 2022

Reading: R & G Chapter 19

Steps in Database Design, cont

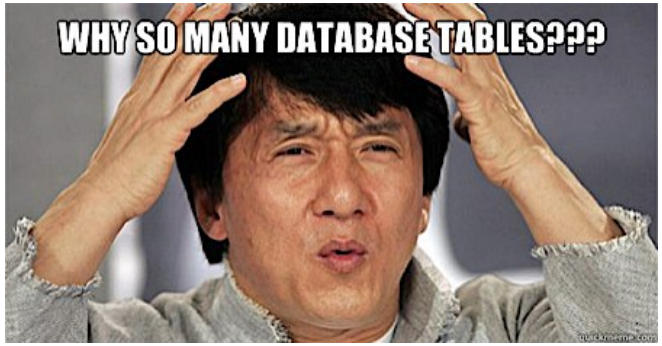
- Requirements Analysis
  - user needs; what must database do?
- Conceptual Design
  - *high level description (often done w/ER model)*
  - ORM encourages you to program here

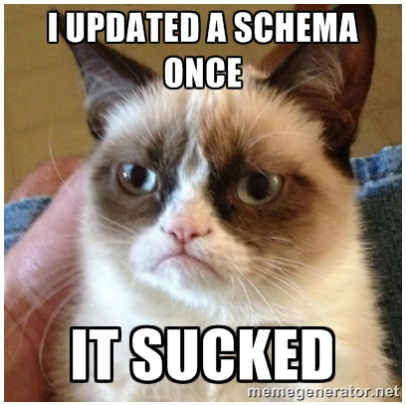
  Completed
- Logical Design
  - translate ER into DBMS data model
  - ORMs often require you to help here too

  Completed
- **Schema Refinement**
  - **consistency, normalization**

  You are here
- Physical Design - indexes, disk layout
- Security Design - who accesses what, and how

What makes good schemas?





Relational Schema Design

Name	SSN	PhoneNumber	City
Fred	123-45-6789	510-555-1234	Berkeley
Fred	123-45-6789	510-555-6543	Berkeley
Joe	987-65-4321	908-555-2121	San Jose

One person may have multiple phones, but lives in only one city

Primary key is thus (SSN, PhoneNumber)

What is the problem with this schema?

Relational Schema Design

Name	SSN	PhoneNumber	City
Fred	123-45-6789	510-555-1234	Berkeley
Fred	123-45-6789	510-555-6543	Berkeley
Joe	987-65-4321	908-555-2121	San Jose

Anomalies:

- Redundancy = repeat data
- Update anomalies = what if Fred moves to "Oakland"?
- Deletion anomalies = what if Joe deletes his phone number?

update all tables in relationship set?

delete in all table in relationship set

Relation Decomposition

Break the relation into two:

Name	SSN	PhoneNumber	City
Fred	123-45-6789	510-555-1234	Berkeley
Fred	123-45-6789	510-555-6543	Berkeley
Joe	987-65-4321	908-555-2121	San Jose

Name	SSN	City
Fred	123-45-6789	Berkeley
Joe	987-65-4321	San Jose

SSN	PhoneNumber
123-45-6789	510-555-1234
123-45-6789	510-555-6543
987-65-4321	908-555-2121

Anomalies have gone:

- No more repeated data
- Easy to move Fred to “Oakland” (how?)
- Easy to delete all Joe’s phone numbers (how?)

Relational Schema Design (or Logical Design)

How do we do this systematically?

- Start with some relational schema
- Find out its ***functional dependencies*** (FDs)
- Use FDs to ***normalize*** the relational schema

Functional Dependencies (FDs)

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:

$A_1 \ldots A_n$ **determines** $B_1..B_m$

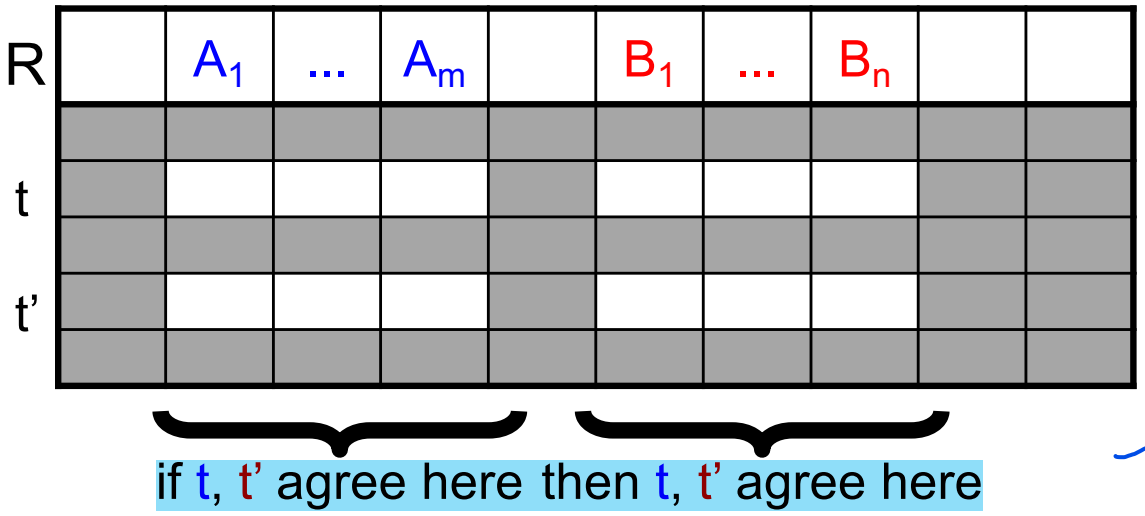
$A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m$

Functional Dependencies (FDs)

Definition $A_1, ..., A_m \rightarrow B_1, ..., B_n$ **holds** in R if:

for all t, t' ∈ R,

$t.A_1 = t'.A_1 \wedge ... \wedge t.A_m = t'.A_m \rightarrow t.B_1 = t'.B_1 \wedge ... \wedge t.B_n = t'.B_n$



Example

An FD holds, or does not hold on an instance:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

EmpID → Name, Phone, Position

Position → Phone

but not Phone → Position

Example

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876 ←	Salesrep
E1111	Smith	9876 ←	Salesrep
E9999	Mary	1234	Lawyer

Position → Phone

FDs can be broken w/ a new insert

Example

EmpID	Name	Phone	Position
E0045	Smith	1234 →	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234 →	Lawyer

But not Phone → Position

Example

- name → color
- category → department
- color, category → price
- department → price

name	category	color	department	price
Gizmo	Gadget	Green	Toys	49
Tweaker	Gadget	Red	Toys	49
Gizmo	Stationary	Green	Office-supp.	59

Which FD's hold?

Buzzwords

- FD **holds** or **does not hold** on an instance
- If we can be sure that *every instance of R* will be one in which a given FD is true, then we say that **R satisfies the FD**
- If we say that R satisfies an FD, we are **stating a constraint on R**

Why bother with FDs?

Name	SSN	PhoneNumber	City
Fred	123-45-6789	510-555-1234	Berkeley
Fred	123-45-6789	510-555-6543	Berkeley
Joe	987-65-4321	908-555-2121	San Jose

Anomalies:

- Redundancy = repeat data
- Update anomalies = what if Fred moves to “Oakland”?
- Deletion anomalies = what if Joe deletes his phone number?

An Interesting Observation

If all these FDs are true:

name → color
category → department
color, category → price

Then this FD also holds:

name, category → price

If we find out from application domain that a relation satisfies some FDs, it doesn't mean that we found all the FDs that it satisfies!
There could be more FDs implied by the ones we have.

Finding New FDs: Armstrong's Axioms

- Suppose X, Y, Z are sets of attributes, then:
  - ***Reflexivity*****:** If $X \supseteq Y$, then $X \rightarrow Y$
  - ***Augmentation***: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - ***Transitivity***: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- Sound and complete inference rules for FDs!

- Some additional rules (that follow from AA):
  - ***Union***: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - ***Decomposition***: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
  - See if you can prove these!

Closure of a set of Attributes

Given a set of attributes $A_1, \ldots, A_n$

The **closure** is the set of attributes B, notated $\{A_1, \ldots, A_n\}^+$, s.t. $A_1, \ldots, A_n \rightarrow B$

remember augmentation axiom

Example:

1. name → color
2. category → department
3. color, category → price

Closures:

$name^+ = \{name, color\}$

$\{name, category\}^+ = \{name, category, color, department, price\}$

$color^+ = \{color\}$

Closure Algorithm

$X=\{A_1, \ldots, A_n\}$.

Repeat until X doesn't change **do**:
  if $B_1, \ldots, B_n \rightarrow C$ is a FD **and**
    $B_1, \ldots, B_n$ are all in X
  then add C to X.

Example:

1. name → color
2. category → department
3. color, category → price

$\{name, category\}^+ =$
  { name, category, color, department, price }

Hence: name, category → color, department, price

Keys

- A **superkey** is a set of attributes A_1, ..., A_n s.t. for any other attribute B, we have A_1, ..., A_n → B

- A **candidate key** (or sometimes just key) is a minimal superkey
  - A superkey and for which no subset is a superkey

Computing (Super)Keys

- For all sets X, compute X^+
- If X^+ = [all attributes], then X is a superkey
- Try reducing to the minimal X's to get the candidate key

Example

Product(name, price, category, color)

name, category → price
category → color

What is the candidate key ?

Example

Product(name, price, category, color)

name, category → price
category → color

What is the candidate key ?

(name, category)+ = { name, category, price, color }

Hence (name, category) is a candidate key

Key or Keys ?

We can we have more than one candidate key!

What are the candidate keys here ?

A → B
B → C
C → A

Key or Keys ?

We can we have more than one candidate key!

What are the candidate keys here ?

A → B
B → C
C → A

AB→C
BC→A

Eliminating Anomalies

Main idea:

- X → A is OK if X is a (super)key for the relation
  - A = all attributes in the relation
  - OK = no need to further decompose the relation
- X → A is not OK otherwise
  - Need to decompose the table, but how?
  - That's where *normalization* comes in!

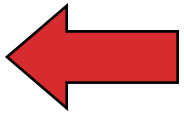
Thanks for that…

- So we know a lot about FDs
- So what?

- Can they help with removing redundancy, update and deletion anomalies?
- Yes! We use **normalization** to cast schemas into **Normal Forms** (aka good schemas)

Normal Forms

First Normal Form = all attributes are atomic
Second Normal Form (2NF) = old and obsolete
Third Normal Form (3NF) = rarely preferred over BCNF
Fourth Normal Form (4NF) = unnecessary/complex

Boyce Codd Normal Form (BCNF) 

Boyce-Codd Normal Form

A simple condition for removing redundancy/anomalies from relations:

A relation R is in BCNF if and only if:

Whenever there is a nontrivial FD: $A_1A_2\ldots A_n \rightarrow B$,
then $A_1A_2\ldots A_n$ is a super-key for R.

- Non-trivial means RHS is not a subset of LHS
- "Whenever a set of attributes of R is determining another attribute, it should determine all attributes of R."

Why does this make sense?

Say R(A, B, C) with AB as the key has an FD: A→C.
Then C is being repeated for multiple Bs

Example

Name	SSN	Phone Number
Jia	123-32-9931	(201) 555-1234
Jia	123-32-9931	(206) 572-4312
Marco	909-43-4486	(908) 464-0028
Marco	909-43-4486	(212) 555-4000

What are the dependencies?

SSN ➔ Name

Is the left side a superkey?

No

Is it in BCNF?

No.

Decompose it into BCNF

SSN	Name
123-32-9931	Jia
909-43-4486	Marco

SSN	Phone Number
123-32-9931	(201) 555-1234
123-32-9931	(206) 572-4312
909-43-4486	(908) 464-0028
909-43-4486	(212) 555-4000

SSN ➔ Name

Now is it in BCNF?

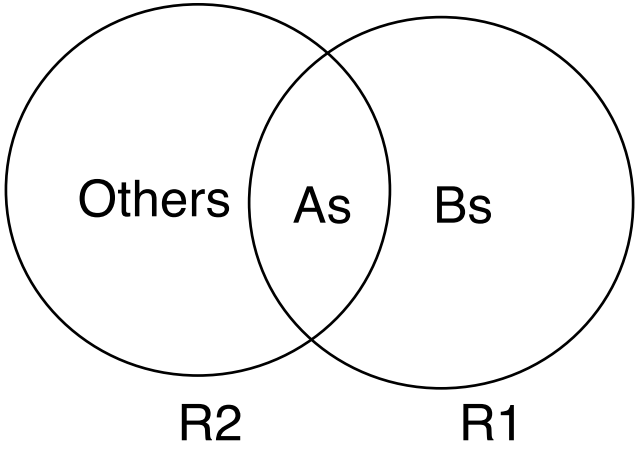
BCNF Decomposition

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"
helps avoid unnecessarily fine-grained decomposition

Decompose:



Continue until there are no BCNF violations left.

Example Decomposition

Person:

Name	SSN	Age	EyeColor	PhoneNum

Functional dependencies:

SSN ➔ Name, Age, Eye Color

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

Example

Same example, slightly more complex.

Person (Name, SSN, Age, EyeColor, PhoneNum, Draftworthy)

- FD 1: SSN → Name, Age, EyeColor
- FD 2: Age → Draftworthy

Example

- Person (Name, SSN, Age, EyeColor, PhoneNum, Draftworthy)
- FD 1: SSN → Name, Age, EyeColor
- FD 2: Age →Draftworthy

- FD 1 and 2 imply SSN → Name, Age, EyeColor, Draftworthy
- Split based on this
  - (SSN, Name, Age, EyeColor, Draftworthy)
  - (SSN, PhoneNum)
- Split based on Age → Draftworthy
  - (SSN, Name, Age, EyeColor)
  - (Age, Draftworthy)
  - (SSN, Phone Number)
- Will get same result if you apply in a different order (but not always!)

Two-attribute relations

- Let A and B be the only two attributes of R
- Claim: R is in BCNF.
- If A ➔ B is true, B ➔ A is not:
  - A → B does not violate BCNF
- If B ➔ A is true, A ➔ B is not:
  - Symmetric
- If A ➔ B is true, B ➔ A is true:
  - Both are keys, therefore neither violate BCNF

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 and R2, using R and S.

   e) repeat Step 1 for R1 and R2

2) Stop when all relations are BCNF or are two attributes

(Remember, two attribute relations are always in BCNF)

Q: Is BCNF Decomposition unique?

- R(SSN, netid, phone).
- FD1: SSN -> netid
- FD2: netid -> SSN
- Each of these two FDs violates BCNF.

Can you tell me two different BCNF decomp for R?

- Pick FD1 and decompose, you get:
  - (SSN, netid); (SSN, phone).
- Pick FD2 and you get
  - (netid, SSN); (netid, phone).

Properties of BCNF

- BCNF removes certain types of redundancies
  - for examples of redundancy that it cannot remove, see “multi-valued redundancy” (Addressed by 4NF, see textbook)
- BCNF decomposition **avoids information loss**
  - You can construct the original relation instance from the decomposed relations’ instances.
  - How? What would the relational algebra exp look like?
    - R(A, B, C) from R(A, B), R(B, C)
  - Ans: Natural join
  - Proof: in the textbook

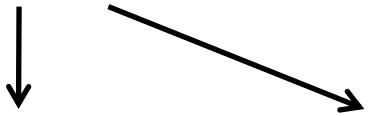
Can we cheat?

- 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.?
- Turns out, this leads to information loss …

Example of the "easy decomposition"

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

A	B	C
1	2	3
4	2	6



A	B
1	2
4	2

B	C
2	3
2	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	2	6

Nat.Join

A	B	C
1	2	3
4	2	6
1	2	6
4	2	3

A	B
1	2
4	2

B	C
2	3
2	6

We get back some "bogus tuples"!

Lossless decompositions (like BCNF) don't give bogus tuples.

Summary of Schema Refinement

- BCNF: each field contains data that cannot be inferred via FDs.
  - ensuring BCNF is a good heuristic.
- Not in BCNF? Try decomposing into BCNF relations.
- Downside of BCNF: not all dependencies are preserved (some are split across relations)
  - TINSTAFL! If you want to preserve dependencies, you will have redundancy
  - Take a look at the textbook for these tradeoffs