Database Design: Functional Dependencies and Normal Forms

Abdu Alawini

University of Illinois at Urbana-Champaign

CS411: Database Systems

July 4, 2020

Overview of Database Design

- Conceptual design: (ER & UML Models are used for this.)
  - What are the **entities** and **relationships** we need?
- Logical design:
  - Transform ER design to Relational Schema
- Schema Refinement: (Normalization)
  - Check relational schema for redundancies and related anomalies.
- Physical Database Design and Tuning:
  - Consider typical workloads; (sometimes) modify the database design; select file types and indexes.

We're here

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
  - implement R in SQL
  - start using it
- However, R may not be well-designed, thus causing us a lot of problems
- OR: people may start without an ER diagram, and you need to reformat the schema R
  - Either way you may need to **improve** the schema

Q: Is this a good design?

Individuals 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

Potential Problems

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

- Redundancy
- Update anomalies
  - maybe we'll update the address of the person with phone number '(206) 572-4312' to something other than '10 Green'. Then there will be two addresses for that person.
- Deletion anomalies
  - delete the phone number of a person; if not careful then the address can also disappear with it.

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

Unfortunately, this is not something you will detect even if you did principled ER design and translation

How do We Obtain a Good Design?

- Start with the original db schema R
  - From ER translation or otherwise
- Identify its ***functional dependencies***
- Use them to transform R until we get a good design R*

Desirable Properties of R*

1. must preserve the information of R
2. must have minimal amount of redundancy
3. must be "dependency preserving"
    - (we'll come to this later)

- must also give good query performance

Normal Forms

- DB researchers have developed many "**normal forms**"
- These are basically schemas obeying certain rules
  - Converting a schema that doesn't obey rules to one that does is called "**normalization**"
  - This typically involves some kind of decomposition into smaller tables, just like we saw earlier.
  - (the opposite: grouping tables together, is called "**denormalization**")

First Normal Form (1NF)

A database schema is in 1NF, if all tables are flat:

Likes

Customer	drink
Abdu	Mocha Latte Macchiato
Jonathan	Milk Orange Juice
Olivia	8-shots Latte Red Bull

Likes

Customer	drink
Abdu	Mocha
Abdu	Latte
Abdu	Macchiato
Jonathan	Milk
Jonathan	Orange Juice
Olivia	8-shots Latte
Olivia	Red Bull

Normal Forms

- Most important Normal Forms
  - 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 redundancies
- DB gurus have also developed algorithms to transform R into R* in these normal forms
- To understand these normal forms we'll need to understand ***functional dependencies***

Functional Dependencies

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

Functional Dependencies

Definition:

If two tuples agree on the attributes

$\boxed{A_1, A_2, \ldots A_n}$

then they must also agree on the attributes

$\boxed{B_1, B_2, \ldots B_m}$

Where have we seen this before?

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

What a FD actually means

- Knowing FD: $A \rightarrow B$ holds in R(A, B, C) means that
  - For ALL valid instances R(A, B, C):
    - A determines B
    - Or, if two tuples share A, then they share the same B
  - This is the property of the "world"
- Conversely, if: $A \not\rightarrow B$, then there is no guarantee that the "A determines B" property holds in a given instance (though it might).
  - Trivially, it holds when you have only one tuple.

More examples

Product: name, manufacturer ⟶ price

Person: ssn ⟶ name, age

Company: name ⟶ 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

- **No**, you cannot.
- This a property of the world, not of the current data
- **In general, you cannot conclude from a given instance of a table that an FD is true. An FD is an assertion that must always be respected.**
- You can, however, check if a given FD ***is violated*** by the table instance.

Keys are a type of FD

- 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
- There could be many keys of a relation

Student (UIN, email, dept, age)
UIN → UIN, email, dept, age
email → UIN, email, dept, age

- Superkey
  - "Superset" of key
  - a set of attributes that contains a key
  - *Any examples for student?*

Many many FDs…

- MovieInfo (name, year, actor, director, studio)
  - Same movie can be remade multiple years, but a name, year pair uniquely determines a movie
  - A movie has a single director/studio but many actors
    - Name, year → director, studio
      - Name, year → director
      - Name, year → studio
    - Name, year /→ actor
  - A director works only with a single studio
    - Director → studio
  - An actor works on a given movie only once (never for remakes), but may work for many movies in a year
    - Actor, name → year; actor, year /→ name

Many many FDs...

- MovieInfo (name, year, actor, director, studio)
  - Name, year → director, studio
  - Name, year → director
  - Name, year → studio
  - Director → studio
  - Actor, name → year
  - …
  - Actor, name, year → director, studio
  - Director, actor, name → studio, year
  - Director, name, year → studio
  - Studio, actor, name → year
  - …

Any missing FDs?

Goal: Find ALL Functional Dependencies

- Anomalies occur when certain “bad” FDs hold
- We can identify some FDs
- But we need to find *all* FDs, and then look for the bad ones.

Inference Rules for FDs

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

Equivalent to:

$A_1A_2 \ldots A_n \rightarrow B_1$;

$A_1A_2 \ldots A_n \rightarrow B_2$;

…

$A_1A_2 \ldots A_n \rightarrow B_m$

Splitting/Combining Rule

Inference Rules for FDs

- $A_1A_2 \ldots A_n \rightarrow A_1$
- In general,

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

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

  Example: name, UIN → UIN

  Why does this make sense?

Trivial Functional Dependencies Rule

Inference Rules for FDs

IF	A1A2...An → B1B2...Bm
AND	B1B2...Bm → C1C2...Ck
THEN	A1A2...An → C1C2...Ck

Transitive Closure Rule

Armstrong's Axioms

- The previous three rules are all we need to derive all possible FDs
- Formally called "Armstrong's Axioms"
  - Reflexivity rule (Trivial FDs)
  - Augmentation rule
    - A1A2...An ➔ B1B2...Bm, then
      A1A2...An C1C2..Ck ➔ B1B2...Bm C1C2...Ck
  - Transitivity rule (Transitive Closure)

Armstrong's Axioms (cont.)

- But Armstrong axioms are hard to use in practice
  - Ask students from previous semesters ☺

- Better to use "closure" of a set of attributes

Closure of a Set of Attributes

Given a set of attributes **{*A1*, …, *An*}** and a set of FDs F.

Problem: find all attributes *B* such that:

for all relations that satisfy F, they also satisfy:

$A1, \ldots, An \rightarrow B$

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

Example

- Set of attributes A,B,C,D,E,F.
  - Functional Dependencies:

$A\ B \longrightarrow C$

$A\ D \longrightarrow E$

$B \longrightarrow D$

$A\ F \longrightarrow B$

Closure of $\{A,B\}^+$:

Closure of $\{A,F\}^+$:

Example

- Set of attributes A,B,C,D,E,F.
  - Functional Dependencies:

$$A\ B \longrightarrow C$$

$$A\ D \longrightarrow E$$

$$B \longrightarrow D$$

$$A\ F \longrightarrow B$$

Closure of $\{A,B\}^+ = \{A, B, C, D, E\}$

Closure of $\{A,F\}^+ = \{A, F, B, D, C, E\}$

Algorithm to Compute Closure

Split the FDs in F so that every FD has a single attribute on the right. (Simplify the FDs)

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 F,
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 $\{A_1A_2\ldots A_n\}^+$

Why does this algorithm converge?

Uses for Attribute Closure

- Use 1: To test if X is a (super)key
  - **How?** By computing X+, and check if X+ contains all attrs of R
  - We can also use it to find candidate keys
    - Compute X+ for all sets X where X+ = all attributes
    - Then list only the minimal X's

- Use 2: To check if X → Y holds
  - **How?** By checking if Y is contained in X+

Finding Keys Example

Person(SSN, name, age, hair color)

SSN → name, age

age → hair color

What are the candidate keys?

Closure of a set of FDs

- Given a relation schema R & a set F of FDs
  - Closure of F: F+ = all FDs logically implied by F
  - Allows us to answer all questions of the type
    - is the FD f logically implied by F?
- Example
  - R = {A,B,C,G,H,I}
  - F = 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)
- How to compute F+?

Using Attribute Closure to Infer ALL FDs

Example:

Given R(A, B, C, D) and

F = {AB → C; AD → B; B → D}

Compute the set of all inferred FDs of F

Step 1: Computer X+, for every X:

A+ = A, B+ = BD, C+ = C, D+ = D

AB+ = ABCD, AC+ = AC, AD+ = ABCD, BC+=BCD, BD+=BD, CD+= CD

ABC + = ABD + = ACD + = ABCD (no need to compute, why?)

BCD+ = BCD, ABCD+ = ABCD

Step 2: Enumerate all FD's X → Y, s.t. $Y \subseteq X+$ and $X \cap Y = \emptyset$

B → D, AB → CD, AD → BC, BC → D, ABC → D, ABD → C, ACD → B

Eliminating Anomalies

Main idea:

- X → A is OK, if X is a (super)key

- X → A is NOT OK, otherwise
  - Need to decompose the table, but how?

Boyce-Codd Normal Form (BCNF)

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)

Others...

Boyce-Codd Normal Form

Definition. 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 superkey for R.

There are no "bad" FDs: whenever there is a nontrivial FD, its left side must be a superkey

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?

A: Yes **B: No**

Is it in BCNF?

No.

Decompose it into BCNF

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

SSN → Name

Now is it in BCNF?

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

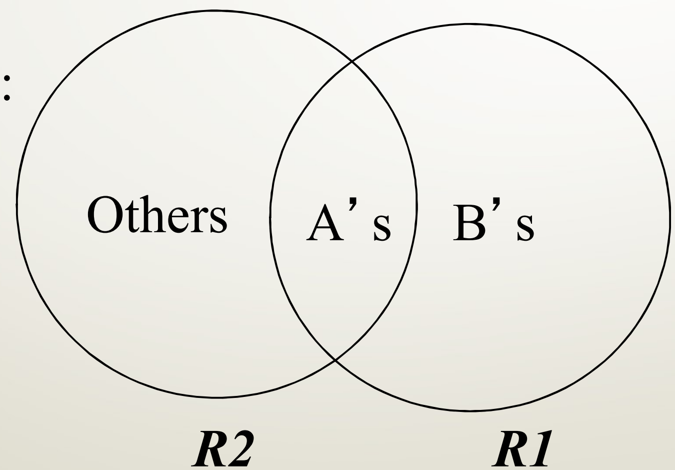
BCNF Decomposition

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	Phone

Functional dependencies:

SSN ➔ Name, Age, Eye Color

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

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, next slide)
   e) Repeat Step 1 for R1 and R2
2) Stop when all relations are BCNF, or are two-
  attributes
```

(Two attribute relations are always in BCNF, see E.g. 3.17 (pg. 89) for proof and examples)

FD Projection Algorithm

Input: relation R, set F of FDs over R, and relation R1 (projected from R)

Output: **t** (the set of FDs that holds in R1)

1) Compute X+ for every subset of R1 attributes
   - calculate the closure based on the FD in R (F)

2) Add to **t** all non-trivial FDs in R1

3) Modify **t** by computing its minimal basis

FD Projection Example

- Given a relation R(A,B,C,D,E,F), F = {AD->BC, B->A, C->E, E->BF}, and R1(A,D,E,F) projected from R, compute the set of FDs in R1.

Step 1: Compute X+ for every subset of R1 attributes

- calculate the closure based on the FD in R

{A}+={A}, {D}+={D}, {E}+={AEF}, {F}+={F}

{AD}+={ADEF}, {AE}+={AEF}, {AF}+={AF}, {DE}+={ADEF}, {EF}+={AEF}

{ADE}+={ADEF}, {ADF}+={ADEF}, {AEF}+={AEF}, {DEF}+={ADEF}

Step 2: Add to **t** all non-trivial FDs in R1

t= {E->AF, AD->EF, AE->F, DE->AF, EF->S ADE->F, ADF->E, DEF->A}

Step 3: Modify **t** by computing its minimal basis

{E->F, AD->E, E->A}

Another Example

- Person (Name, SSN, Age, EyeColor, Phone, HairColor)
- FD 1: SSN → Name, Age, EyeColor
- FD 2: Age → HairColor

FD 1 and 2 imply: SSN → Name, Age, EyeColor, HairColor

Iteration 1: Split based on SSN → Name, Age, EyeColor, HairColor

- Person(SSN, Name, Age, EyeColor, HairColor)
- Phone(SSN, Phone)

Iteration 2: Split based on Age → HairColor

- Person(SSN, Name, Age, EyeColor)
- Hair(Age, HairColor)
- Phone(SSN, Phone)

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

R(SSN, netid, phone)

(SSN, netid) (SSN, phone)

Pick FD2

R(SSN, netid, phone)

(netid, SSN) (netid, phone)

Properties of BCNF

- BCNF removes certain types of redundancies
  - All redundancies based on FDs are removed.
- BCNF Decomposition avoids information loss
  - You can construct the original relation instance from the decomposed relations' instances.

How would get R(A, B, C) from R(A, B), R(B, C)?

A: Cross Product

B: Natural Join

C: Group BY

An easy decomposition?

Since two-attribute relations are always 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

A decomposition is *lossless* if we can recover:

R(A,B,C)

↓ Decompose

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

↓ Recover

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

R' is in general larger than R. Why?

Must ensure R' = R

Desirable Properties of Schema Refinement

✓ 1) minimize redundancy

✓ 2) avoid info loss

3) preserve dependency

4) ensure good query performance

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)

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

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

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 → Phone

Preserving FDs

- Thus, if the X and Y of a FD X->Y do not both end up in the same decomposed relation:
  - Such a decomposition is not "dependency-preserving."
  - No way to force BCNF to preserve dependencies

- Thus, while BCNF gives us **lossless join** and **less redundancy**, it doesn't give us **dependency preservation**

An alternative: 3rd Normal Form (3NF)

Definition. A relation R is in 3rd normal form if :

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

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

Textbook uses rule with many B_i on the RHS, if so, then each one must be part of some key.

3NF vs. BCNF

- R is in BCNF if whenever X→ A holds, then X is a superkey.
  - Slightly stricter than 3NF.
    - Doesn't let R get away with it if A is part of some key
  - Thus, BCNF "more aggressive" in splitting
- Example: R(A,B,C) with AB → C; C → A
  - 3NF but not BCNF

Decomposing R into 3NF Preliminaries: Minimal basis

Given a set of FDs: F.

Say the set F' is *equivalent* to F, in the sense that F' can be inferred from F and v. versa.

- Any such F' is said to be a *basis* for F.

- "Minimal basis"
  - A basis with all RHS singletons, where any modifications lead to no longer a basis, including:
    - Dropping attribute from LHS of a rule: compact rules
    - Dropping a rule: small # of rules

Example of minimal basis

- R(A, B, C) with FDs:
  - A → BC; B → AC; C → AB
- A basis:
  - A→ B; A → C; B → A; B → C; C → A; C → B
- One minimal basis:
  - A → B
  - B → C
  - C → A

Conversion into minimal basis

- "Algorithm for converting F to a minimal basis
  - R = F with all RHS singletons:
  - Repeat until convergence:
    - If a rule minus an attribute from LHS is inferred from F, replace rule with rule minus attribute from LHS
    - If a rule is inferred from rest, drop it

Minimal basis example

Given R (A B C D E) and

F = { **A->D, BC->AD, C->B, E->A, E->D**}

Find F', the minimal basis for F.

Algorithm:

1- Only singleton in RHS

2- Remove unnecessary att. from LHS

3- Remove FDs that can be inferred from the rest

① BC->A ; BC->D

② BC->A ⇒ C->A

BC->D ⇒ C->D

③ F = { A->D, C->A, ~~C->D~~, C->B, E->A, ~~E->D~~ }

E^+ = { E, A, D }

E->D can be inferred

F' = { A->D, C->A, C->B, E->A }

Decomposing R into 3NF

1. Get a "minimal basis" *G* of given FDs
2. For each FD A ➔ B in the minimal basis *G*, use AB 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

Decomposing R into 3NF

1. Get a "minimal basis" *G* of given FDs
2. For each FD A → B in the minimal basis *G*,
   use AB 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.

Implicitly this is connecting all the LHSs with the remaining attributes

Result will be lossless, will be dependency-preserving,

Basically every minimal FD is preserved somewhere

Example

- R(A, B, C) with FDs:
  - A → BC; B → AC; C → AB

Minimal Basis: A → B; B → C; C → A

So, first cut:

R_1(A, B), R_2(B, C), R_3(C, A)

Any attributes left? Nope ➔ done

Example

- R(A, B, C, D, E) with FDs:
  - A → B; CD → B; DA → C

  BCNF Decomp:

  (AB), (ACD), (ADE) or:

  (BCD), (ACD), (ADE)

  Which FDs do each of these not preserve?

Minimal Basis:

A → B; CD → B; DA → C

3NF Decomp: (AB), (BCD), (ACD), (**ADE**)

Desirable Properties of Schema Refinement

1) minimize redundancy

✓ **2) avoid info loss**

✓ **3) preserve dependency**

4) ensure good query performance

3NF

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)

Overview of Database Design

- Conceptual design: (ER & UML Models are used for this.)
  - What are the **entities** and **relationships** we need?
- Logical design:
  - Transform ER design to Relational Schema
- Schema Refinement: (Normalization)
  - Check relational schema for redundancies and related anomalies.
- Physical Database Design and Tuning:
  - Consider typical workloads; (sometimes) modify the database design; select file types and indexes.

We'll discuss indexing next.