Chapter 3 Design Theory for Relational Databases

Contents

- Functional Dependencies
- Decompositions
- Normal Forms (BCNF, 3NF)
- Multivalued Dependencies (and 4NF)
- Reasoning About FD's + MVD's

Our example of chapter 2

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

A theory : “dependencies” will be talked first

Some questions:

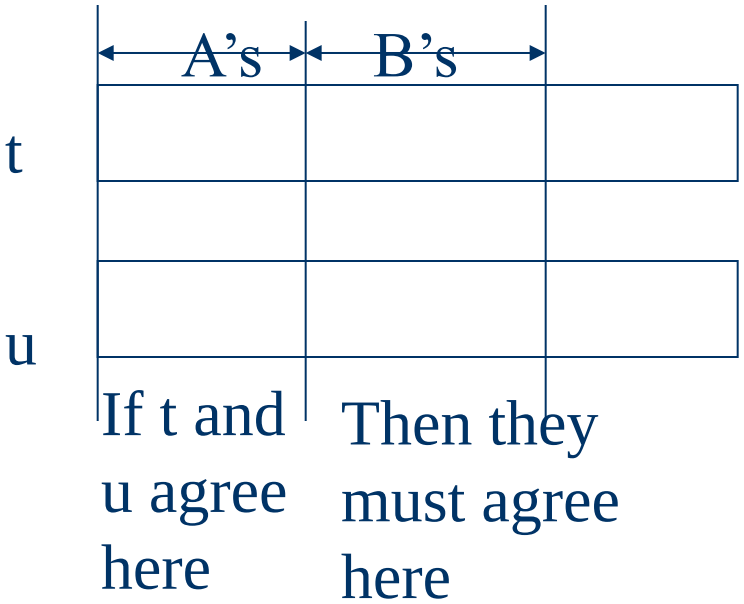
1. Why do we design relations like the example?
2. What makes a good relational database schema?
3. what we can do if it has flaws?

Functional Dependencies

- *X* ->*Y* is an assertion about a relation *R* that whenever two tuples of *R* agree on all the attributes of *X*, then they must also agree on all attributes in set *Y*.
  - Say "*X* ->*Y* holds in *R*."
  - Convention: ..., *X*, *Y*, *Z* represent **sets of attributes**; *A*, *B*, *C*,... represent **single attributes**.
  - Convention: no set formers in sets of attributes, just *ABC*, rather than {*A*,*B*,*C* }.

Functional Dependency (cont.)

- Exist in a relational schema as a constraint.
- Agree for all instances of the schema (t and u are any two tuples)



We have functional dependency like this $A_1A_2\ldots \rightarrow B_1B_2\ldots$

Why we call "functional" dependency?

Functional Dependency (cont.)

- Some examples

Beers(name, manf)

name→manf manf→name ?

Sells(bar, beer, price)

Bar,beer → price

Splitting Right Sides of FD's

- $X \rightarrow A_1A_2...A_n$ holds for R exactly when each of $X \rightarrow A_1$, $X \rightarrow A_2$,..., $X \rightarrow A_n$ hold for R.
- Example: $A \rightarrow BC$ is equivalent to $A \rightarrow B$ and $A \rightarrow C$.
- There is no splitting rule for left sides.
- We'll generally express FD's with singleton right sides.

Trivial Functional Dependencies

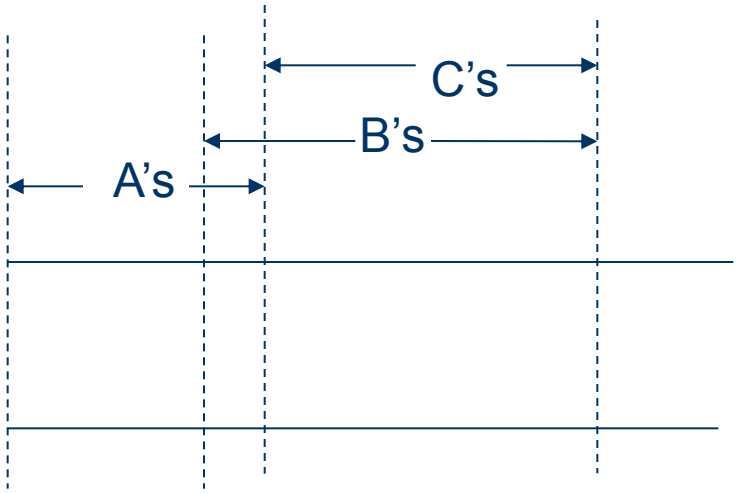
Sells(bar, beer, price)

Bar, Beer → bar (trivial functional dependencies)

Bar, beer → price (nontrivial function dependencies)

A's → B's

A's → C's

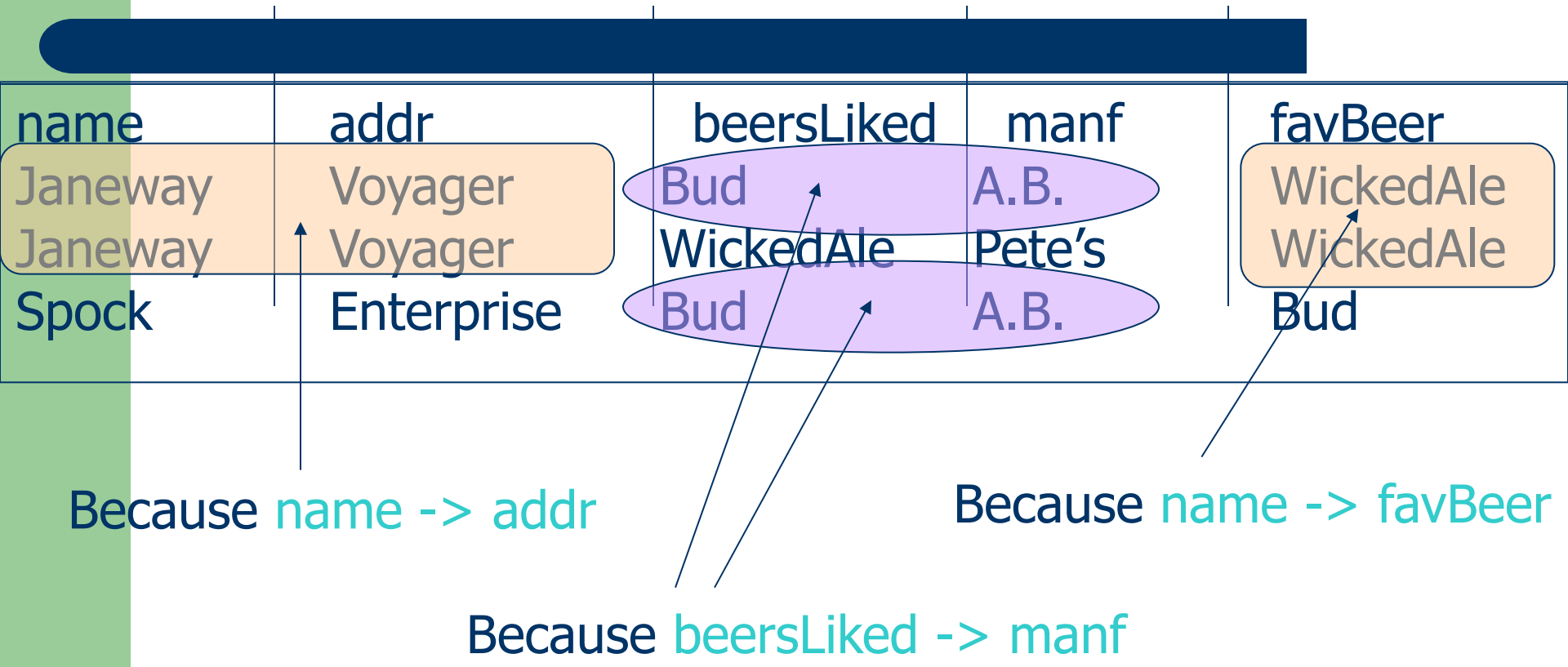


Example: FD's

Drinkers(name, addr, beersLiked, manf, favBeer)

- Reasonable FD's to assert:
  1. name -> addr favBeer (combining rule)
     - Note this FD is the same as name -> addr and name -> favBeer. (splitting rule)
  2. beersLiked -> manf

Example: Possible Data



Keys of Relations

- *K* is a *superkey* for relation *R* if *K* functionally determines all of *R*.
- *K* is a *key* for *R* if *K* is a superkey, but no proper subset of *K* is a superkey. (minimality)

Example: Superkey

Drinkers(name, addr, beersLiked, manf, favBeer)

- {name, beersLiked} is a superkey because together these attributes determine all the other attributes.
  - name -> addr favBeer
  - beersLiked -> manf

Example: Key

- {name, beersLiked} is a key because neither {name} nor {beersLiked} is a superkey.
  - name doesn't -> manf; beersLiked doesn't -> addr.
- There are no other keys, but lots of superkeys.
  - Any superset of {name, beersLiked}.

Where Do Keys Come From?

1. Just assert a key *K*.
   - The only FD's are *K* -> *A* for all attributes *A.*
2. Assert FD's and deduce the keys by systematic exploration.
3. More FD's From "Physics"

   - Example: "no two courses can meet in the same room at the same time" tells us:

   hour room -> course.

Inferring FD's

Three ways:
1. A simple test for it
2. Use FD to deduce
3. Calculate closure of y

- We are given FD's X_1 -> A_1, X_2 -> A_2,..., X_n -> A_n , and we want to know whether an FD Y -> B must hold in any relation (instance) that satisfies the given FD's.
  - Example: If A -> B and B -> C hold, surely A -> C holds, even if we don't say so.
- Important for design of good relation schemas.

1: Inference Test

- To test if *Y* -> *B*, start by assuming two tuples agree in all attributes of *Y*.

← *Y* →

0000000. . . 0

00000?? . . . ?

2: Inference Test

- Use the given FD's to infer that these tuples must also agree in certain other attributes.
  - If B is one of these attributes, then *Y -> B* is true.
  - Otherwise, the two tuples, with any forced equalities, form a two-tuple relation that proves *Y -> B* does not follow from the given FD's.

2: Inference Test : example

- R(A,B,C) with FD's: A→B, B→C
- To prove A→C?

A	B	C
a	b1	c1
a	b2	c2

Inference steps:

1) Assume two tuples that agree on A
2) Because A→B, b1=b2
3) Because B→C

   c1=c2

Inference rules

- Reflexivity:

If $\{B_1B_2\ldots B_m\} \subseteq \{A_1,A_2,\ldots A_n\}$ then

$A_1,A_2,\ldots A_n \rightarrow B_1B_2\ldots B_m$ called trivial FD's

- Augmentation:

If $A_1,A_2,\ldots A_n \rightarrow B_1B_2\ldots B_m$ then,

$A_1,A_2,\ldots A_n\ C_1,C_2..C_k \rightarrow B_1B_2\ldots B_m\ C_1,C_2..C_k$

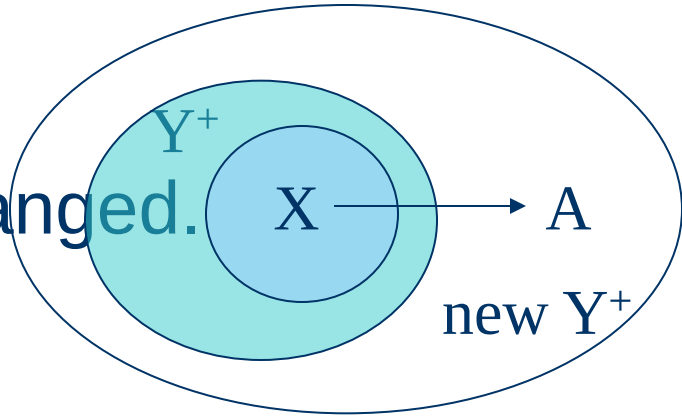
- Transitivity:

If $A_1,A_2,\ldots A_n \rightarrow B_1B_2\ldots B_m$,and $B_1B_2\ldots B_m \rightarrow C_1,C_2..C_k$

then, $A_1,A_2,\ldots A_n \rightarrow C_1,C_2..C_k$

3: Closure Test

- An easier way to test is to compute the *closure* of Y, denoted Y^+.
- Basis: $Y^+ = Y$.
- Induction: Look for an FD's left side X that is a subset of the current Y^+. If the FD is $X \rightarrow A$, add A to Y^+.
- End: when Y^+ can not be changed.



3: Closure Test: example

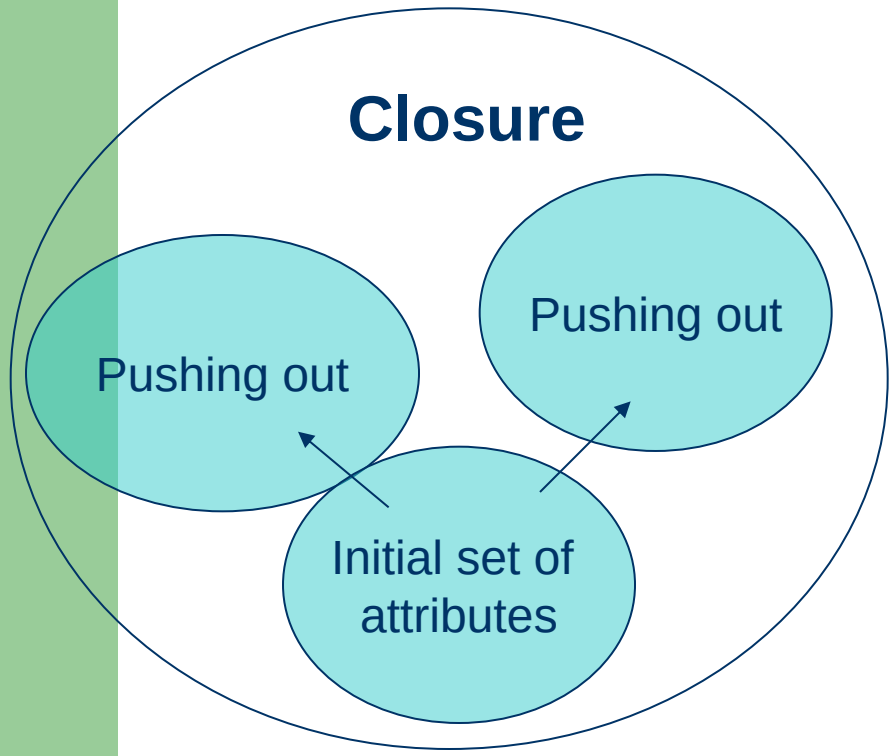
- R(A,B,C) with FD's: A→B, B→C
- To prove A→C?

- Calculating steps for A^+ :

1. $A^+ = A$
2. $A^+ = A, B$
3. $A^+ = A, B, C$ ⟶ A→C

Closure and Keys: if the closure of X is all attributes of a relation, then X is a key /superkey.

Computing the closure of a set of attributes



- The closure algorithm 3.7 (pp.76) can discovers all true FD's.
- We need a FD's (minimal basis) to represent the full set of FD's for a relation.

Closing sets of Functional dependencies

- Example: R(A,B,C) with all FD's:

A→B, A→C, B→A, B→C, C→A, C→B, AB→C, AC→B, BC→A,…

- We are free to choose any **basis** for the FD's of R, a set of FD's that imply all the FD's that hold for R :

FD1: A→B, B→A, B→C, C→B

FD2: A→B, B→C, C→A

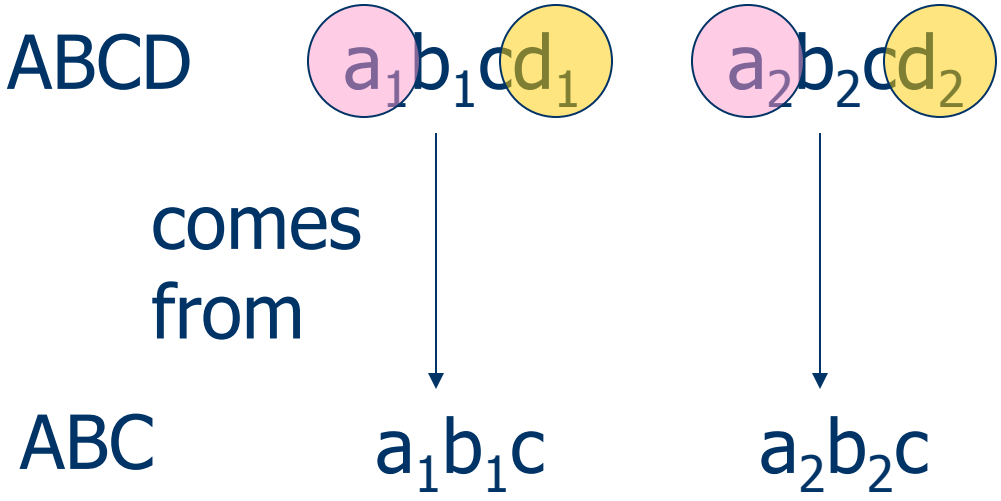
Given Versus Implied FD's

- Typically, we state a few FD's that are known to hold for a relation R
- Other FD's may follow logically from the given FD's; these are implied FD's.
- Example: R(A,B,C) with FD's: A→B, B→C, C→A
- A→ C is implied FD

Finding All Implied FD's

- Motivation: "normalization," the process where we break a relation schema into two or more schemas.
- Example: *ABCD* with FD's *AB ->C*, *C ->D*, and *D ->A*.
  - Decompose into *ABC*, *AD*. What FD's hold in *ABC* ?
  - Not only *AB ->C*, but also *C ->A* !

Why?



$d_1=d_2$ because
C -> D

$a_1=a_2$ because
D -> A

Thus, tuples in the projection with equal C's have equal A's;
C -> A.

Basic Idea for projecting functional dependencies

1. Start with given FD's and find all *nontrivial* FD's that follow from the given FD's.
   - Nontrivial = right side not contained in the left.
2. Restrict to those FD's that involve only attributes of the projected schema.

Simple, Exponential Algorithm

1. For each set of attributes X, compute X^+.
2. Add $X \rightarrow A$ for all A in $X^+ - X$.
3. However, drop $XY \rightarrow A$ whenever we discover $X \rightarrow A$.
   - Because $XY \rightarrow A$ follows from $X \rightarrow A$ in any projection.
4. Finally, use only FD's involving projected attributes.

A Few Tricks

- No need to compute the closure of the empty set or of the set of all attributes.
- If we find X^+ = all attributes, so is the closure of any superset of X.

Example: Projecting FD's

- *ABC* with FD's *A* ->*B* and *B* ->*C*. Project onto *AC*.
  - $A^+ = ABC$; yields *A* ->*B*, *A* ->*C*.
    - We do not need to compute AB^+ or AC^+.
  - $B^+ = BC$; yields *B* ->*C*.
  - $C^+ = C$; yields nothing.
  - $BC^+ = BC$; yields nothing.

Example -- Continued

- Resulting FD's: *A* ->*B*, *A* ->*C*, and *B* ->*C*.
- Projection onto *AC* : *A* ->*C*.
  - Only FD that involves a subset of {*A*,*C* }.

Relational Schema Design

- Goal of relational schema design is to avoid anomalies and redundancy.
  - *Update anomaly* : one occurrence of a fact is changed, but not all occurrences.
  - *Deletion anomaly* : valid fact is lost when a tuple is deleted.

Example of Bad Design

Drinkers(name, addr, beersLiked, manf, favBeer)

name	addr	beersLiked	manf	favBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	???	WickedAle	Pete's	???
Spock	Enterprise	Bud	???	Bud

Data is redundant, because each of the ???'s can be figured out by using the FD's name -> addr favBeer and beersLiked -> manf.

This Bad Design Also Exhibits Anomalies

name	addr	beersLiked	manf	favBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

- Update anomaly: if Janeway is transferred to *Intrepid*, will we remember to change each of her tuples?
- Deletion anomaly: If nobody likes Bud, we lose track of the fact that Anheuser-Busch manufactures Bud.

Solve the problem

Analysis result : Problems caused by FD's

Drinkers(name, addr, beersLiked, manf, favBeer) → decompose into smaller relations :

Drinker= projection (name,addr, favBeer) (Drinkers)
Likes= projection (name, beersLiked) (Drinkers)
Beer = projection (beersliked, manf) (Drinkers)

Drinkers = Drinker join Likes join beer not more, not less

Solve the problem (cont.)

name	addr	beersLiked	manf	favBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

Janeway	voyager	WickedAle
Spock	Enterprise	Bud

Janeway	Bud
Janeway	WickedAle
Spock	Bud

Bud	A.B.
WickedAle	Peter's

Any anomalies?

Remember our questions:

- Why do we design relations like the example? -- good design
- What makes a good relational database schema? -- no redundancy, no Update/delete anomalies,
- what we can do if it has flaws? -- decomposition

New Question:

- **any standards for a good design?**

  → Normal forms: a condition on a relation schema that will eliminate problems
- **any standards or methods for a decomposition?**