Design Theory for Relational Databases

Functional Dependencies

Decompositions

Normal Forms

Introduction

- There are always many different schemas for a given set of data.
- E.g., you could combine or divide tables.
- How do you pick a schema? Which is better? What does "better" mean?
- Fortunately, there are some principles to guide us.

Avoid redundancy

This table has redundant data, and that can lead to 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.

Database Design Theory

- It allows us to improve a schema systematically.
- General idea:
  - Express constraints on the data
  - Use these to decompose the relations
- Ultimately, get a schema that is in a “normal form” that guarantees good properties, such as no anomalies.
- “Normal” in the sense of conforming to a standard.
- The process of converting a schema to a normal form is called normalization.

Part I:
Functional Dependency Theory

Functional Dependencies

- *X* -> *Y* is an assertion about a relation *R* that if two tuples of *R* agree on all the attributes in set *X*, they must also agree on all the attributes in set *Y*.
  - Say "*X* -> *Y* holds in *R*."
    "*X* functionally determines *Y*."
  - Convention: Use …, *X*, *Y*, *Z* to represent sets of attributes; *A*, *B*, *C*,… to represent single attributes.
  - Convention: Omit braces for sets of attributes.
    E.g., just *ABC*, rather than {*A*,*B*,*C* }.

Why "functional dependency"?

- "dependency" because the value of *Y* depends on the value of *X*.
- "functional" because there is a mathematical function that takes a value for X and gives a *unique* value for Y.
- (It's not a typical function; just a lookup.)

Splitting Right Sides of FD's

- $X \text{->} A_1 A_2 \ldots A_n$ holds for R iff each of $X \text{->} A_1$, $X \text{->} A_2$,…, $X \text{->} A_n$ hold for R.
- Example: $A \text{->} BC$ is equivalent to $A \text{->} B$ and $A \text{->} C$.
- There is no splitting rule for left sides.
- We'll generally express FD's with singleton right sides.

Example: FD's

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

- "Reasonable" FD's to assert:
  name -> addr favBeer
  beersLiked -> manf
- Note: The first FD is the same as
  name -> addr and name -> favBeer.

Example: Possible Data

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

Because name -> addr

Because name -> favBeer

Because beersLiked -> manf

Coincidence or FD?

ID	Email	City	Country	Surname
1983	tom@gmail.com	Toronto	Canada	Fairgrieve
8624	mar@bell.com	London	Canada	Samways
9141	scotty@gmail.com	Winnipeg	Canada	Samways
1204	birds@gmail.com	Aachen	Germany	Lakemeyer

- In this instance:
  - Surname determines Country
  - City determines Country
- Are these FDs?

- We have an FD only if it holds for *every* instance of the relation.
- You can’t know this just by looking at one instance.
- You can only determine this based on knowledge of the domain.

Keys and FDs

- *K* is a *key* for *R* if
  K functionally determines all of *R*, and
  no proper subset of *K* does.
- *K* is a *superkey* for relation *R* if
  K contains a key for *R*.
  ("superkey" is short for "superset of key".)
- (Beware: terminology to do with keys varies across authors.)

Example: Superkey

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

- Suppose we have the FD's from before:
  name -> addr favBeer
  beersLiked -> manf
- {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}.

FDs are a generalization of keys

- Functional dependency:
  X -> Y
- Superkey:
  X -> R
- A superkey must include all the attributes of the relation on the RHS.
- An FD can involve just a subset of them.

Inferring FDs

- Given a set of FDs, we can often infer further FDs.
- This will come in handy when we apply FDs to the problem of database design.

The Problem

- Given:
  FD's $X_1 \text{ -> } A_1$, $X_2 \text{ -> } A_2$, ..., $X_n \text{ -> } A_n$.
- Determine:
  Whether the FD $Y \text{ -> } B$ also hold in any relation that satisfies the given FD's.
- Example:
  If $A \text{ -> } B$ and $B \text{ -> } C$ hold,
  $A \text{ -> } C$ must also hold.

Method 1: Prove it from first principles

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

$\leftarrow Y \rightarrow$

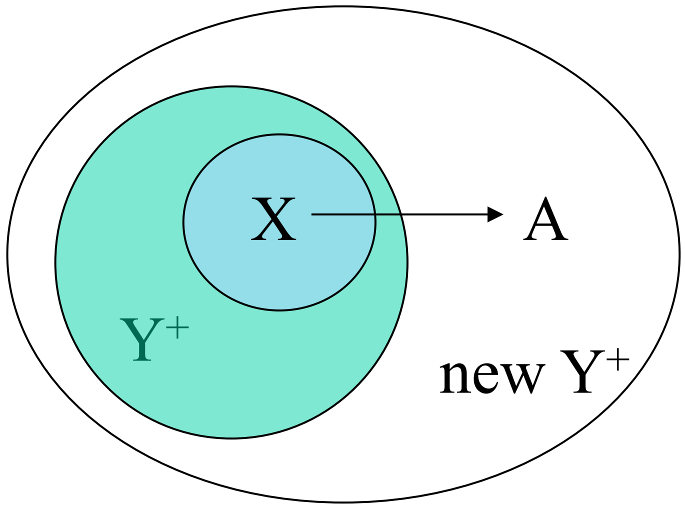
aaaaabb. . . b

aaaaa?? . . . ?

Method 2: Closure Test

- Easier: compute the *closure* of the attributes Y, denoted Y^+:
  - Basis: $Y^+ = Y$.
  - Induction:
    1. Look for an FD whose left side X is a subset of the current Y^+.
       Suppose that FD is $X \to A$.
    2. Add A to Y^+.
- If B is in Y^+, then $Y \to B$ holds.

Computing the closure Y^+ of a set of attributes Y



Projecting FDs

- Later, we will learn how to normalize a schema by decomposing relations.
- We will need to be aware of what FDs hold in the new, smaller, relations.
- In other words, we must project our FDs onto the attributes of our new relations.

Projecting FDs

- Given:
  - a relation R
  - the set S of FDs that hold in R
  - A relation $R_1 = \Pi_L(R)$
- Determine the set of all FDs that
  - Follow from S and
  - Involve only attributes of R_1

Projection Algorithm

Initialize result set T to {}.

For each subset X of the attributes of R_1:

    Compute X^+

    For every attribute A in X^+:

        If A is in L,

            add the FD X -> A to T.

Final step: make T a minimal basis

Repeat until no more changes result:

Remove FDs that are implied by the rest.

For each FD with 2^+ attributes on the left:

- If you can remove one attribute from the LHS and get an FD that follows from the rest:
  - Do so! (It's a stronger FD.)

A Few Tricks

- No need to add X -> A if A is in X itself.
- No need to compute the closure of the empty set or of the set of all attributes (even though they are subsets of X).
- If we find X^+ = all attributes, we can ignore any superset of X. It can only give use "weaker" FDs (with more on the LHS).

Projection is expensive

- Even with these tricks, projection is still expensive.
- Suppose R_1 has n attributes.
  How many subsets of R_1 are there?

Example: Projecting FD's

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

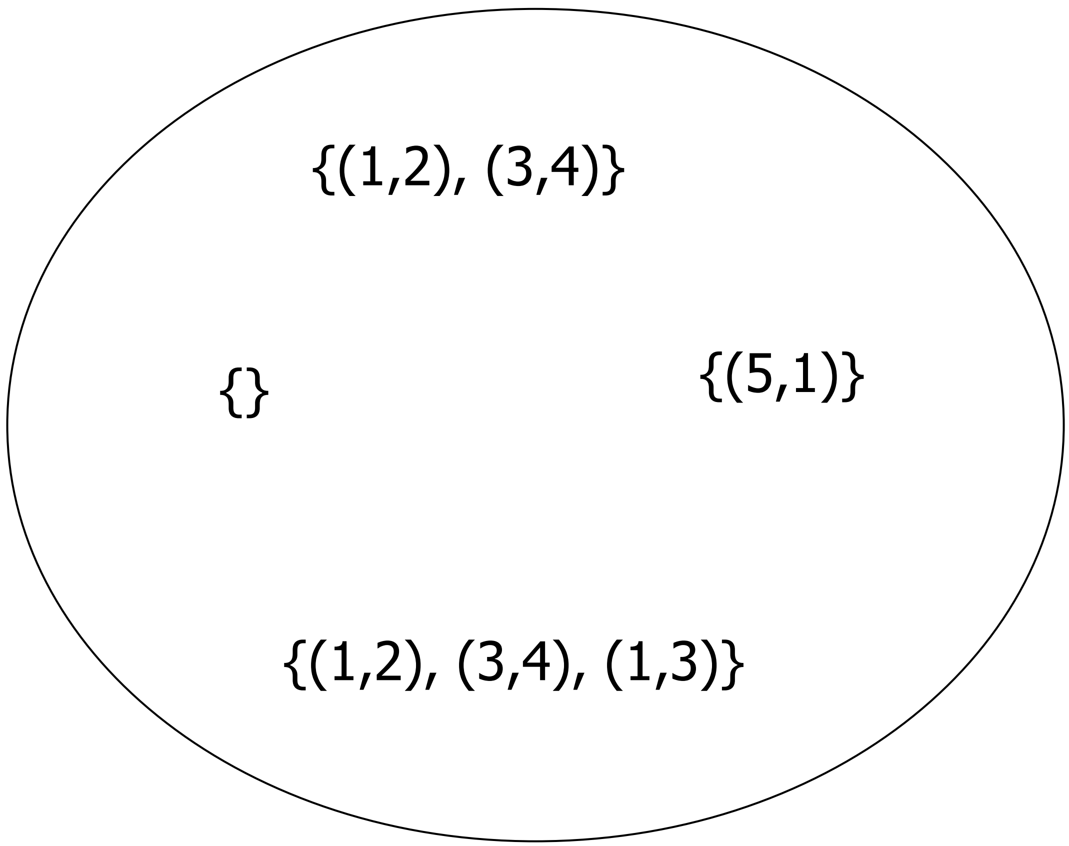
Example -- Continued

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

A Geometric View of FD's

- Imagine the set of all *instances* of a particular relation.
- That is, all finite sets of tuples that have the proper number of components.
- Each instance is a point in this space.

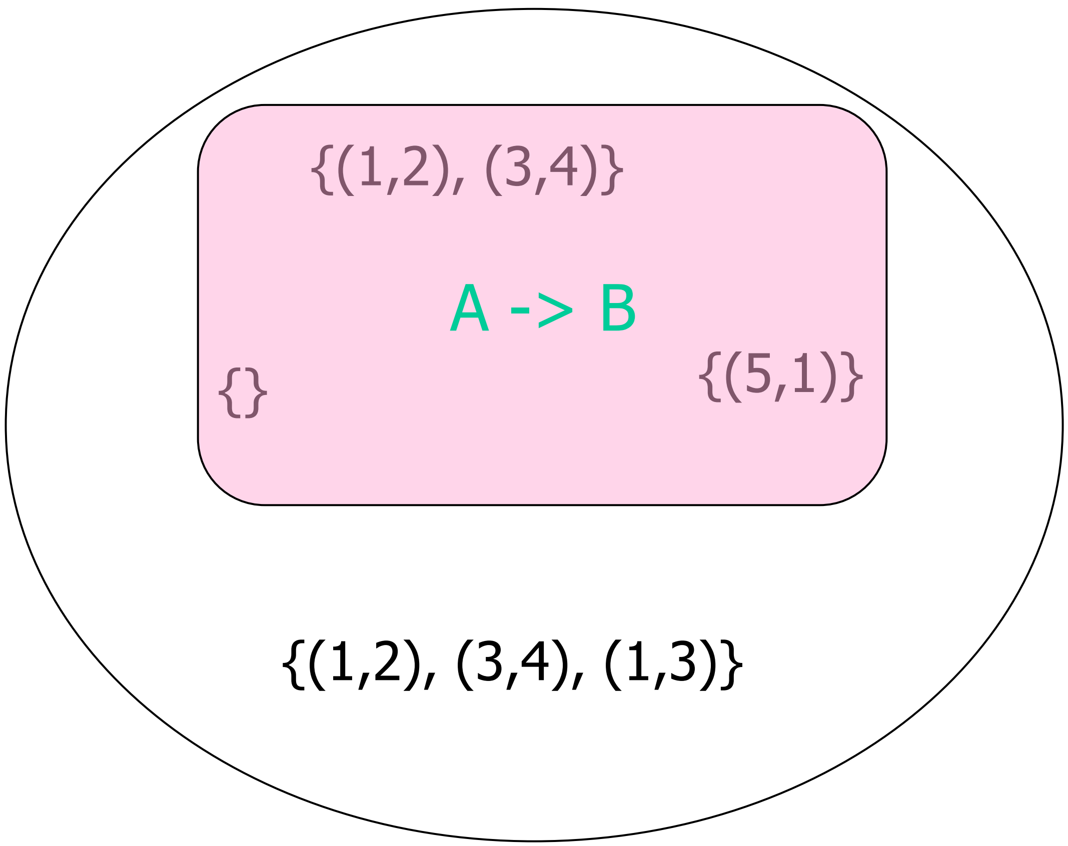
Example: R(A,B)



An FD is a Subset of Instances

- For each FD X -> A there is a subset of all instances that satisfy the FD.
- We can represent an FD by a region in the space.
- Trivial FD = an FD that is represented by the entire space.
  - Example: A -> A.

Example: A -> B for R(A,B)

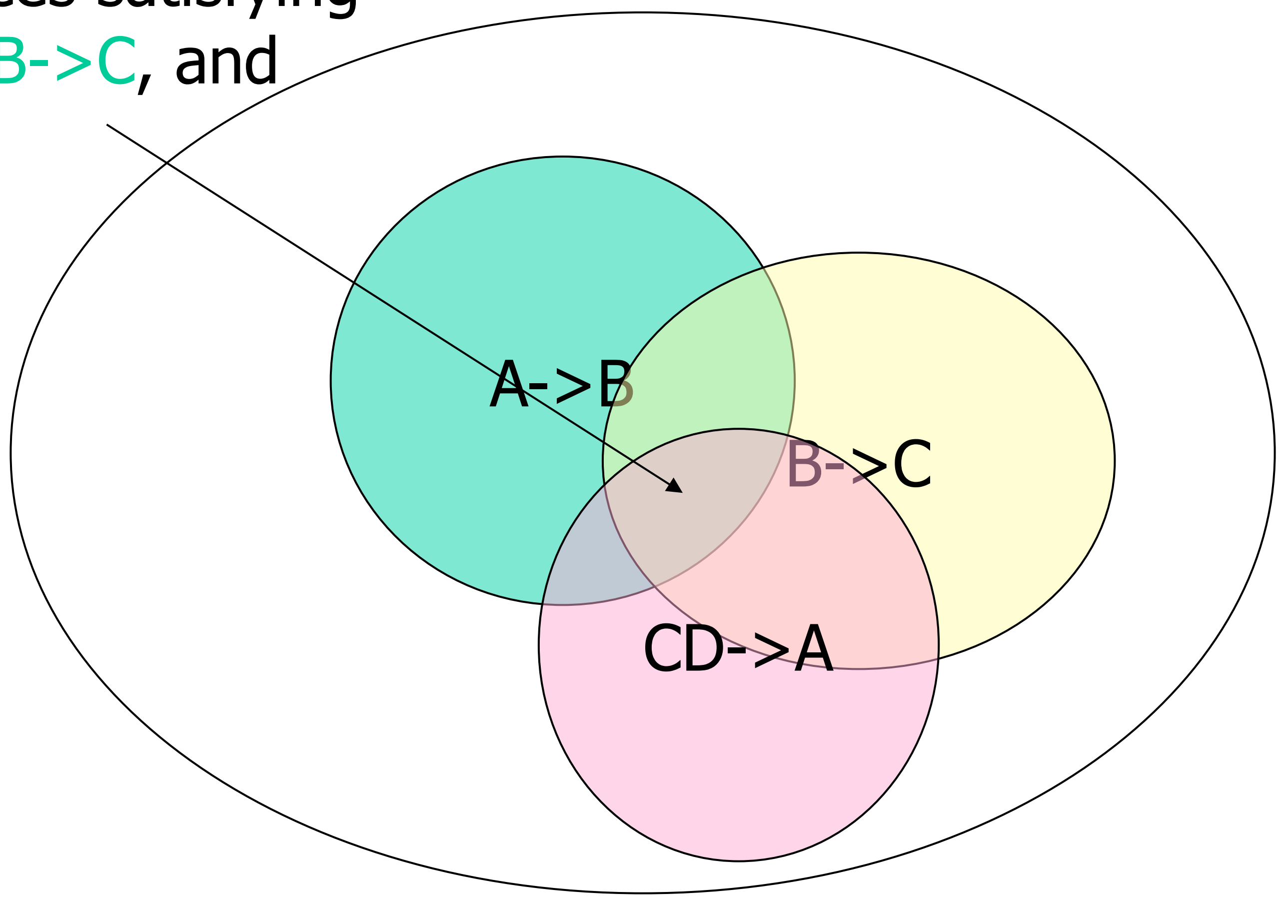


Representing *Sets* of FD's

- If each FD is a set of relation instances, then a collection of FD's corresponds to the intersection of those sets.
  - Intersection = all instances that satisfy all of the FD's.

Example

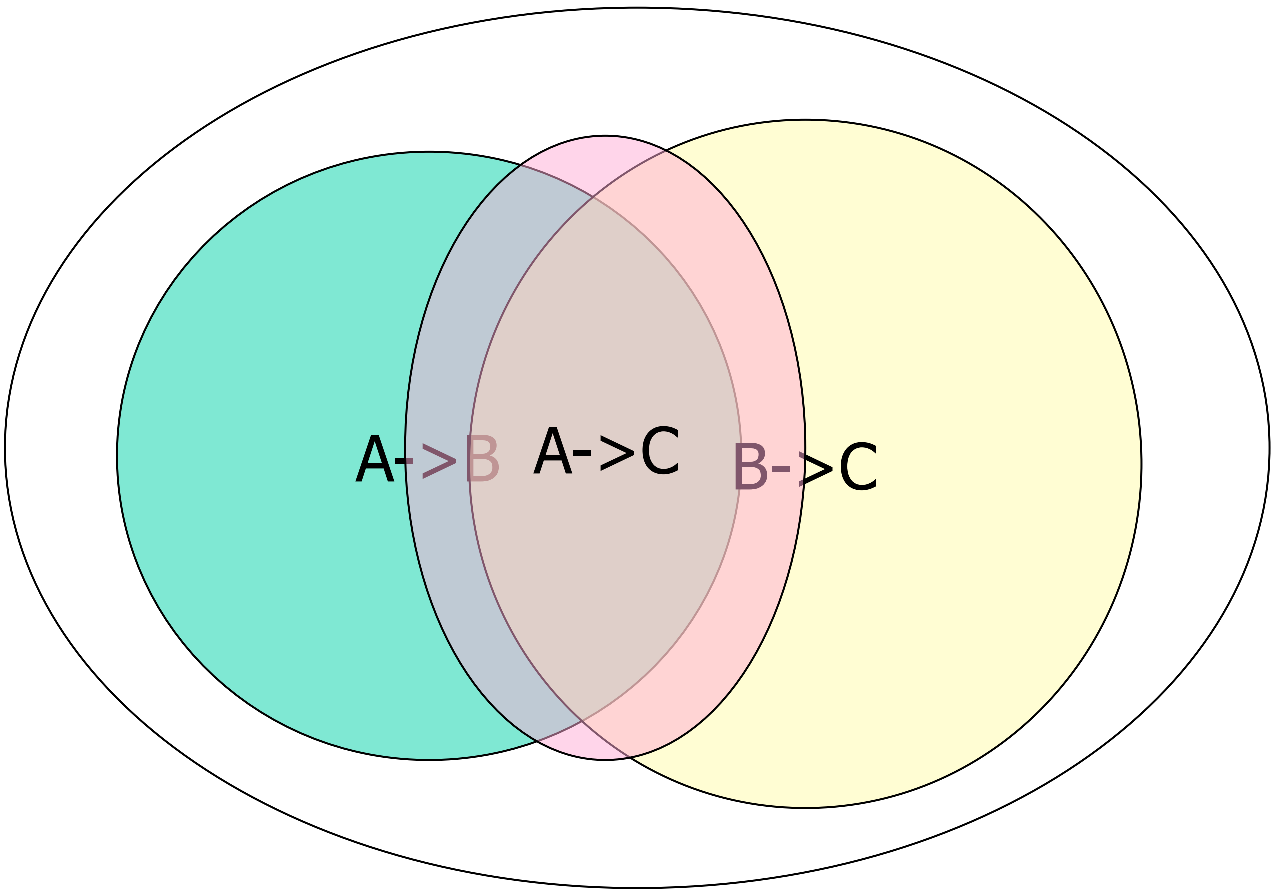
Instances satisfying
A->B, B->C, and
CD->A



Implication of FD's

- If an FD Y -> B follows from FD's X_1 -> A_1,…,X_n -> A_n, then the region in the space of instances for Y -> B must include the intersection of the regions for the FD's X_i -> A_i.
  - That is, every instance satisfying all the FD's X_i -> A_i surely satisfies Y -> B.
  - But an instance could satisfy Y -> B, yet not be in this intersection.

Example



Part II: Using FD Theory to do Database Design

Relational Schema Design

- Goal of relational schema design is to avoid redundancy, and the anomalies it enables.
  - *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

Suppose we have FD's name -> addr favBeer and beersLiked -> manf. This design is bad:

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.

Result of bad design: 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.

Decomposition

- To improve a badly-designed schema $R(A_1, A_2, \ldots, A_n)$, we will decompose it into smaller relations $S(B_1, B_2, \ldots, B_m)$ and $T(C_1, C_2, \ldots C_k)$ such that:
  - $S = \pi_{B1,\ B2,\ \ldots,\ Bn}(R)$
  - $T = \pi_{C1,\ C2,\ \ldots\ Ck}(R)$
  - $\{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots C_k\}$

Example: Decomposition can improve a schema

- See figures 3.6 and 3.7 of the text.

But which decomposition?

- Decomposition can definitely improve a schema.
- But which decomposition? How can we be sure a new schema doesn't exhibit other anomalies?
- Boyce-Codd Normal Form guarantees it.

Boyce-Codd Normal Form

- We say a relation *R* is in *BCNF* if whenever *X* -> *Y* is a nontrivial FD that holds in *R*, *X* is a superkey.
  - Remember: *nontrivial* means *Y* is not contained in *X*.
  - Remember, a *superkey* is any superset of a key (not necessarily a proper superset).

Example: a relation not in BCNF

Drinkers(<u>name</u>, addr, <u>beersLiked</u>, manf, favBeer)

FD's: name->addr favBeer, beersLiked->manf

- Only key is {name, beersLiked}.
- In each FD, the left side is *not* a superkey.
- Any one of these FD's shows *Drinkers* is not in BCNF

Another Example

Beers(name, manf, manfAddr)

FD's: name->manf, manf->manfAddr

- Only key is {name} .
- name->manf does not violate BCNF, but manf->manfAddr does.

Intuition

In other words, BCNF requires that:

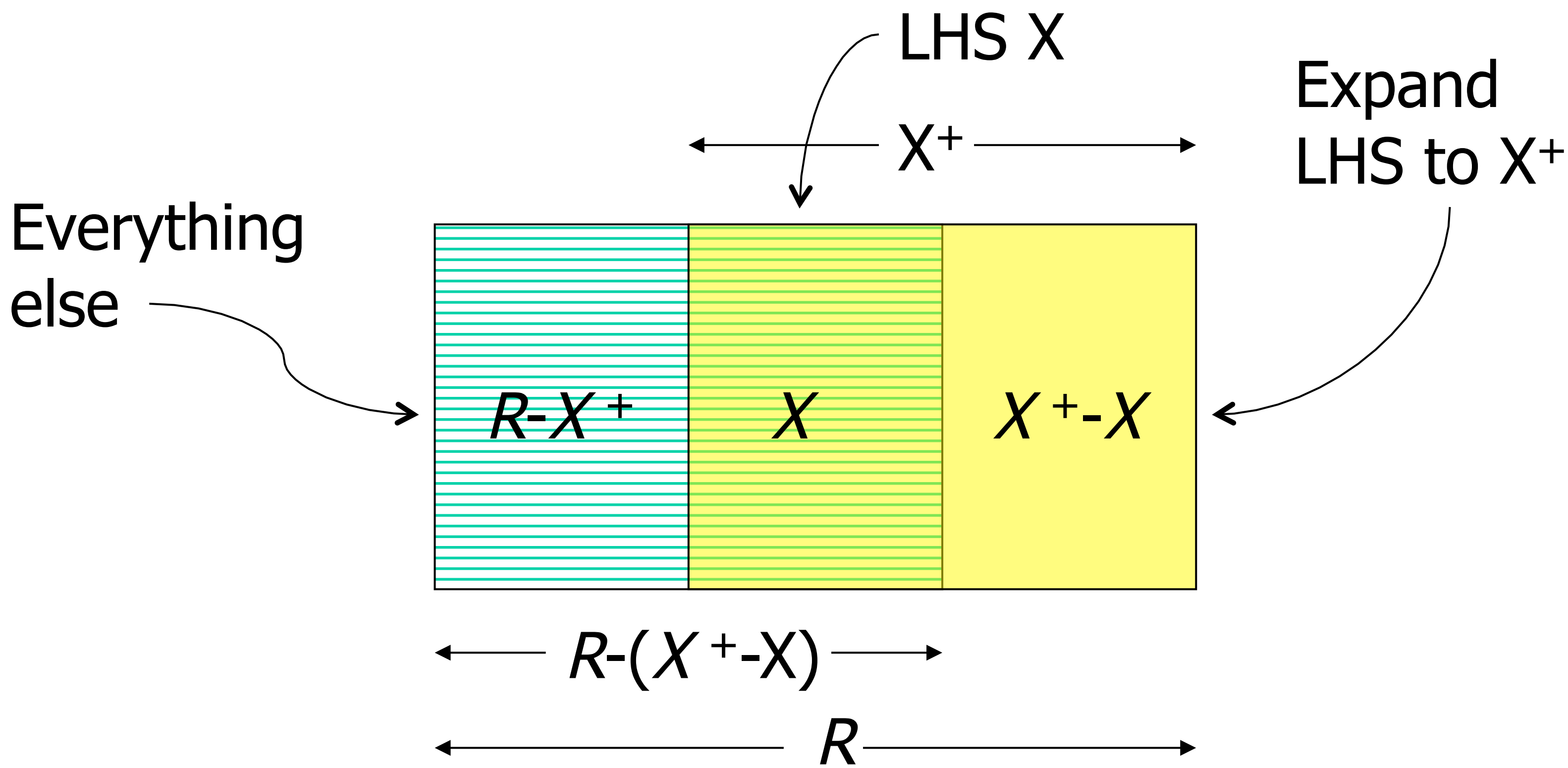
Only things that FD *everything* can FD anything.

Why does that help?

Decomposition into BCNF

- Given: relation R with FD's F.
- Look for an FD $X \text{->} Y$ in F that violates BCNF.
- Compute X^+.
  - != all attributes, because X is not a superkey.
- Replace R by relations with schemas:
  1. $R_1 = X^+$.
  2. $R_2 = R - (X^+ - X)$.
- *Project* given FD's F onto R_1 and R_2.

Decomposition Picture



Need to repeat

- The new relations are not guaranteed to be in BCNF themselves.
- We must repeat the entire process, recursively, on the new relations.
- We only stop when all relations are in BCNF.

Example: BCNF Decomposition

Drinkers(<u>name</u>, addr, <u>beersLiked</u>, manf, favBeer)

F = name->addr, name -> favBeer, beersLiked->manf

- Pick BCNF violation name->addr.
- Close the left side: $\{name\}^+ = \{name, addr, favBeer\}$.
- Decomposed relations:
  1. Drinkers1(<u>name</u>, addr, favBeer)
  2. Drinkers2(<u>name</u>, <u>beersLiked</u>, manf)

Example -- Continued

- We are not done; we need to check Drinkers1 and Drinkers2 for BCNF.
- Projecting FD's is easy here.
- For Drinkers1(<u>name</u>, addr, favBeer), relevant FD's are name->addr and name->favBeer.
  - Thus, {name} is the only key and Drinkers1 is in BCNF.

Example -- Continued

- For Drinkers2(<u>name</u>, <u>beersLiked</u>, manf), the only FD is beersLiked->manf, and the only key is {name, beersLiked}.
  - Violation of BCNF.
- $beersLiked^+$ = {beersLiked, manf}, so we decompose *Drinkers2* into:

1. Drinkers3(<u>beersLiked</u>, manf)
2. Drinkers4(<u>name</u>, <u>beersLiked</u>)

Example -- Concluded

- The resulting decomposition of *Drinkers* :
  1. Drinkers1(<u>name</u>, addr, favBeer)
  2. Drinkers3(<u>beersLiked</u>, manf)
  3. Drinkers4(<u>name</u>, <u>beersLiked</u>)
- Notice: *Drinkers1* tells us about drinkers, *Drinkers3* tells us about beers, and *Drinkers4* tells us the relationship between drinkers and the beers they like.

Tricks for checking whether a relation is in BCNF

- Don't need to know any keys;
  only superkeys matter.
- Don't need to know all superkeys.
- Only need to check whether LHS of each FD is a superkey.
  Use the closure test (simple and fast!)

Tricks for BCNF decomposition

- When projecting FDs onto a new relation, check each new FD: Does the new relation violate BCNF because of this FD?
- If so, abort the projection; you are about to discard this relation anyway (and decompose further).

What we want from a decomposition

1. *No anomalies*.
2. *Lossless Join* : it should be possible to project the original relations onto the decomposed schema, and then reconstruct the original, i.e., get back exactly the original tuples.
3. *Dependency Preservation* : All the original FD's should be satisfied.

What we get from a BCNF decomposition

1. *No anomalies* : ✓
2. *Lossless Join* : ✓ (Section 3.4.1 argues this)
3. *Dependency Preservation* : ✗

What is "lossy" join?

- For any decomposition, it is the case that:
  - $r \subseteq r_1 \bowtie \ldots \bowtie r_n$
  - I.e., we get back every tuple.
- But it may *not* be the case that:
  - r supset $r_1 \bowtie \ldots \bowtie r$
  - I.e., we can get spurious tuples.

r

ID	Name	Addr
11	Pat	1 Main
12	Jen	2 Pine
13	Jen	3 Oak

$r_1 = \Pi_{R1}(r)$

ID	Name
11	Pat
12	Jen
13	Jen

$r_2 = \Pi_{R2}(r)$

Name	Addr
Pat	1 Main
Jen	2 Pine
Jen	3 Oak

$r_1 \bowtie r_2$

ID	Name	Addr
11	Pat	1 Main
12	Jen	2 Pine
13	Jen	3 Oak
12	Jen	3 Oak
13	Jen	2 Pine

What is “lost” in a lossy join?

- Our lossy decomposition loses the fact that 12 lives at 2 Pine (not 3 Oak).
- Lossy decompositions yield ***more*** tuples than they should.
- Tuples aren’t lost; information is.
- Remember that BCNF doesn’t have this problem; it guarantees lossless join.

Example: Failure to preserve dependencies

- Suppose we start with R = *ABC* and FDs *AB* -> *C* and *C* -> *B*.
  - Example: *A* = street address, *B* = city, *C* = zip code.
- There are two keys, {*A*,*B* } and {*A*,*C* }.
- *C* -> *B* is a BCNF violation, so we must decompose into *AC*, *BC*.

We can't enforce the FD $AB \text{->} C$

- The problem is that if we use AC and BC as our database schema, we cannot enforce the FD $AB \text{->} C$ by checking FD's in these decomposed relations.
- Example with A = street, B = city, and C = zip on the next slide.

An Unenforceable FD

street	zip
545 Tech Sq.	02138
545 Tech Sq.	02139

city	zip
Cambridge	02138
Cambridge	02139

Join tuples with equal zip codes.

street	city	zip
545 Tech Sq.	Cambridge	02138
545 Tech Sq.	Cambridge	02139

Although no FD's were violated in the decomposed relations, FD street city -> zip is violated by the database as a whole.

3NF Let's Us Avoid This Problem

- *3rd Normal Form* (3NF) modifies the BCNF condition so we do not have to decompose in this problem situation.
- An attribute is *prime* if it is a member of any key.
- $X \rightarrow A$ violates 3NF if and only if X is not a superkey, and also A is not prime.
- I.e., it's ok if X is not a superkey as long as A is prime.

Example: 3NF

- In our problem situation with FD's *AB* -> *C* and *C* -> *B*, we have keys *AB* and *AC*.
- Thus *A*, *B*, and *C* are each prime.
- Although *C* -> *B* violates BCNF, it does not violate 3NF.

What we get from a 3NF decomposition

1. *No anomalies* : ✗
2. *Lossless Join* : ✓
3. *Dependency Preservation* : ✓

Unfortunately, neither BCNF nor 3NF can guarantee all three properties we want.

Creating a schema in 3NF: The 3NF Synthesis algorithm

- Construct a minimal basis for the FDs.
- Define one relation for each FD in the minimal basis.
  - Schema is the union of the left and right sides.
- If no key is contained in an FD, then add one relation whose schema is some key.

Example: 3NF Synthesis

- Relation R = ABCD.
- FD's *A->B* and *A->C*.
- Keys: Just one key, AD.
- Decomposition: AB and AC from the FD's, plus AD for a key.

Why "synthesis"

- 3NF synthesis:
  - We build up the relations in the schema from nothing.
- BCNF decomposition:
  - We start with a bad relation schema and break it down.

Why 3NF synthesis works

- 3NF: hard part – a property of minimal bases.
- Preserves dependencies: each FD from a minimal basis is contained in a relation, thus preserved.
- Lossless Join: we'll return to this once we've learned how to test for a lossless join.

Testing for a Lossless Join

- If we project R onto $R_1, R_2, \ldots, R_k$, can we recover R by rejoining?
- Any tuple in R can be recovered from its projected fragments.
- So the only question is: when we rejoin, do we ever get back something we didn't have originally?

The Chase Test

- Suppose tuple *t* comes back in the join.
- Then *t* is the join of projections of some tuples of *R*, one for each R_i of the decomposition.
- Can we use the given FD's to show that one of these tuples must be *t* ?

- Start by assuming $t = abc\ldots$.
- For each i, there is a tuple s_i of R that has a, b, c,… in the attributes of R_i.
- s_i can have any values in other attributes.
- We'll use the same letter as in t, but with a subscript, for these components.

Example: Chase test succeeds = lossless join

- Let $R = ABCD$, and the decomposition be AB, BC, and CD.
- Let the given FD's be $C \text{->} D$ and $B \text{->} A$.
- Suppose the tuple t = abcd is the join of tuples projected onto AB, BC, CD.

The tuples of R projected onto AB, BC, CD.

The *Tableau*

A	B	C	D
a	b	c_1	d_1
$\cancel{a_2}$ a	b	c	$\cancel{d_2}$ d
a_3	b_3	c	d

Use $B \to A$

Use $C \to D$

We've proved the second tuple must be *t*.

Example: Chase test fails = lossy Join

- Same relation $R = ABCD$ and same decomposition.
- But with only the FD $C \rightarrow D$.

The *Tableau*

These projections rejoin to form *abcd*.

A	B	C	D
a	b	c_1	d_1
a_2	b	c	$\cancel{d_2}$ d
a_3	b_3	c	d

Use C->D

These three tuples are an example R that shows the join lossy. *abcd* is not in R, but we can project and rejoin to get *abcd*.

Summary of the Chase

1. If two rows agree in the left side of a FD, make their right sides agree too.
2. Always replace a subscripted symbol by the corresponding unsubscripted one, if possible.
3. If we ever get an unsubscripted row, we know any tuple in the project-join is in the original (the join is lossless).
4. Otherwise, the final tableau is a counterexample.

Why 3NF synthesis works

- 3NF: hard part – a property of minimal bases.
- Preserves dependencies: each FD from a minimal basis is contained in a relation, thus preserved.
- Lossless Join: use the "chase test" to show that the row for the relation that contains a key can be made all-unsubscripted variables.

When we don't need to test for lossless Join

- Both BCNF decomposition and 3NF synthesis guarantee lossless join.
- So we never need to test for lossless join if we have done BCNF decomposition or 3NF synthesis.