DESIGN THEORY FOR RELATIONAL DATABASES

MICHAEL LIUT (LIUTM@MCMASTER.CA)
DEPARTMENT OF COMPUTING AND SOFTWARE
MCMASTER UNIVERSITY

SE 3DB3 (Slides adapted from Dr. Fei Chiang) Fall 2016

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.

Schemas and Constraints

- Consider the following sets of schemas:

  Students(macid, name, email)

  vs.

  Students(macid, name)
  Emails(macid, address)

- Consider also:

  House(street, city, value, owner, propertyTax)

  vs.

  House(street, city, value, owner)

  TaxRates(city, value, propertyTax)

Constraints are domain-dependent

Acknowledgements: R. J. Miller, M. Papagelis

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

Keys

- *K* is a *key* for *R* if
  K uniquely 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".)

Example

RegNum	Surname	FirstName	BirthDate	DegreeProg
284328	Smith	Luigi	29/04/59	Computing
296328	Smith	John	29/04/59	Computing
587614	Smith	Lucy	01/05/61	Engineering
934856	Black	Lucy	01/05/61	Fine Art
965536	Black	Lucy	05/03/58	Fine Art

- **`RegNum`** is a key: i.e., **`RegNum`** is a superkey and it contains a sole attribute, so it is minimal.
- **`{Surname, Firstname, BirthDate}`** is another key

Functional Dependencies

- Need a special type of constraint to help us with normalization
- $X \rightarrow Y$ is an assertion about a relation R that whenever two tuples of R agree on all the attributes in set X, they must also agree on all attributes in set Y.
- E.g., suppose X = {AB}, Y = {C}

R

A	B	C
x1	y1	c2
x1	y1	c2
x2	y2	c3
x2	y2	c3

Functional Dependencies

- Say "$X \rightarrow Y$ holds in R."
  "X functionally determines Y."
- Convention: …, X, Y, Z represent sets of attributes; A, B, C,… represent single attributes.
- Convention: no braces used for sets of attributes, 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.

Properties about FDs

- Rules
  - Splitting/combining
  - Trivial FDs
  - Armstrong's Axioms
- Algorithms related to FDs
  - the closure of a set of attributes of a relation
  - a minimal basis of a relation

Splitting Right Sides of FDs

- $X \rightarrow A_1A_2...A_n$ 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 $A \rightarrow C$.
- Combining: if $A \rightarrow F$ and $A \rightarrow G$, then $A \rightarrow FG$

- There is no splitting rule for the left side
  - $ABC \rightarrow DEF$ is NOT the same as $AB \rightarrow DEF$ and $C \rightarrow DEF$!
- We'll generally express FDs with singleton right sides.

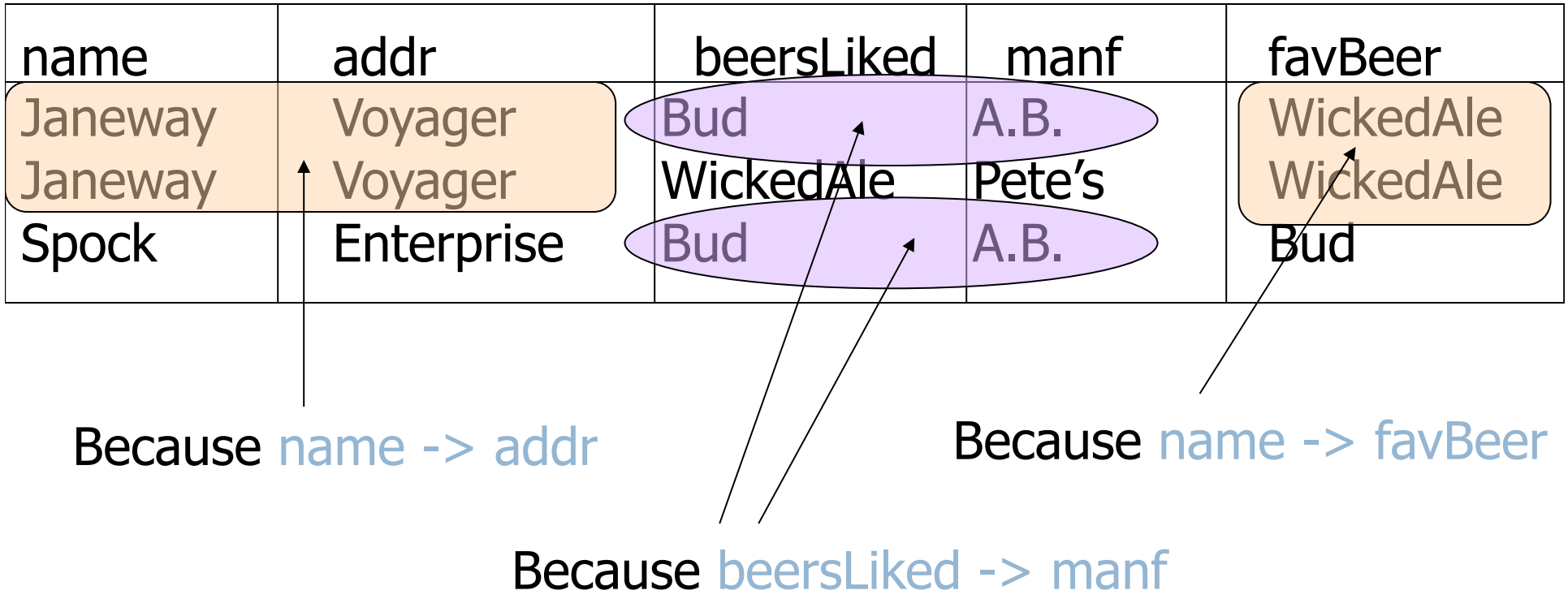
Example: FDs

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

Reasonable FDs to assert:

- name → addr, favBeer.
  - Note this FD is the same as: name → addr and name → favBeer.
- beersLiked → manf

Example: Possible Data



Trivial FDs

- Not all functional dependencies are useful
  - A → A always holds
  - ABC → A also always holds (right side is subset of left side)
- FD with an attribute on both sides
  - ABC → AD becomes ABC → D
  - Or, in singleton form, delete trivial FDs

    ABC → A and ABC → D becomes just ABC → D

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

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

Example: Key

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

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

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
  - Example:

Houses (street, city, value, owner, tax)

- street,city -> value,owner,tax *(both FD and key)*
- city,value -> tax *(FD only)*

Identifying functional dependencies

- FDs are domain knowledge
  - Intrinsic features of the data you're dealing with
  - Something you know (or assume) about the data
- Database engine cannot identify FDs for you
  - Designer must specify them as part of schema
  - DBMS can only enforce FDs when told to
- DBMS cannot "optimize" FDs either
  - It has only a finite sample of the data
  - An FD constrains the entire domain

Coincidence or FD?

ID	Email	City	Country	Surname
1983	tom@gmail.com	Bern	Switzerland	Mendes
8624	jones@bell.com	London	Canada	Jones
9141	scotty@gmail.com	Winnipeg	Canada	Jones
1204	birds@gmail.com	Aachen	Germany	Lakemeyer

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

Coincidence or FD

- 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.

Armstrong's Axioms

X, Y, Z are sets of attributes

1. **Reflexivity:** If $Y \subseteq X$, then $X \rightarrow Y$
2. **Augmentation:** If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
3. **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
4. **Union:** If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
5. **Decomposition:** If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

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.

Dependency Inference

- Suppose we are given FDs
  $X_1 \rightarrow A_1$,
  $X_2 \rightarrow A_2$,
  …,
  $X_n \rightarrow A_n$.
- Does the FD $Y \rightarrow B$ also hold in any relation that satisfies the given FDs?
- Example: If $A \rightarrow B$ and $B \rightarrow C$ hold, surely $A \rightarrow C$ holds, even if we don't say so.

  $A \rightarrow C$ is ***entailed (implied)*** by $\{A \rightarrow B, B \rightarrow C\}$

Transitive Property

The transitive property holds for FDs

- Consider the FDs: $A \rightarrow B$ and $B \rightarrow C$; then $A \rightarrow C$ holds
- Consider the FDs: $AD \rightarrow B$ and $B \rightarrow CD$; then $AD \rightarrow CD$ holds or just $AD \rightarrow C$ (because of trivial FDs)

Method 1: Prove it from first principles

- To test if $Y \rightarrow B$, start by assuming two tuples agree on all attributes of Y.

$\leftarrow Y \rightarrow$

t1: aaaaa bb. . . b

t2: aaaaa ?? . . . ?

Example

ClientID	Income	OtherProd	Rate	Country	City	State
225	High	A	2.1%	USA	San Francisco	MD
420	High	A	2.1%	USA	San Francisco	CA
333	High	B	3.0%	USA	San Francisco	CA
576	High	B	3.0%	USA	San Francisco	CA
128	Low	C	4.5%	UK	Reading	Berkshire
193	Low	C	4.5%	UK	London	London
550	Low	B	3.5%	UK	London	London

F1: [Income, OtherProd] → [Rate]

F2: [Country, City] → [State]

How to prove it in the general case?

Closure test for FDs

- Given attribute set Y and FD set F
  - Denote Y_F^+ or Y^+ the closure of Y relative to F

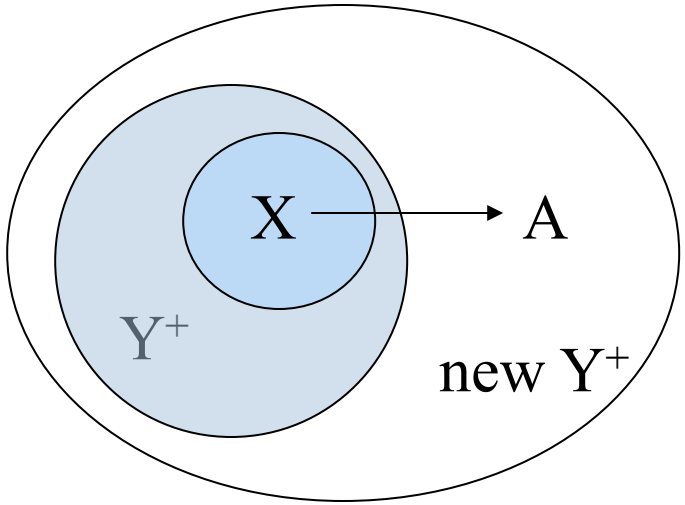
    Y_F^+ = set of all FDs given or implied by Y
- Computing the closure of Y
  - Start: $Y_F^+ = Y$, $F' = F$
  - While there exists an $f \in F'$ s.t. LHS(f) $\subseteq Y_F^+$:

    $Y_F^+ = Y_F^+ \cup RHS(f)$

    $F' = F' - f$
  - At end: $Y \rightarrow B$ for all $B \in Y_F^+$

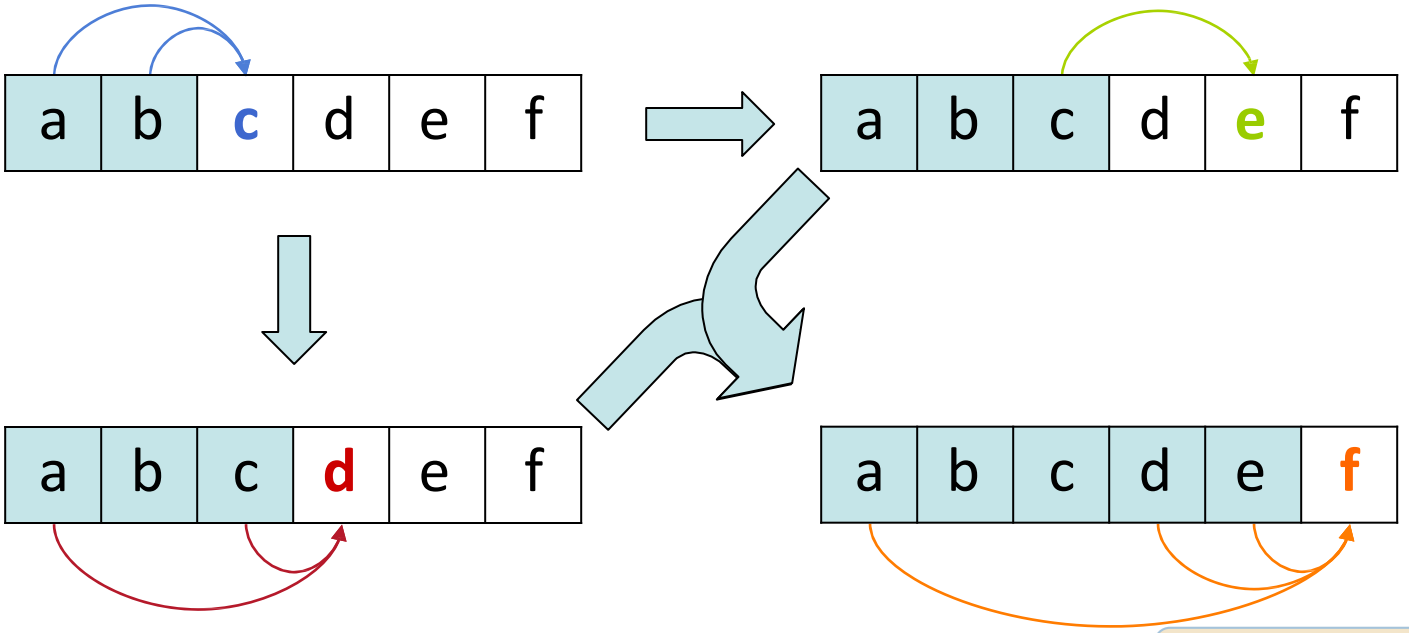
Acknwledgements: M. Papagelis

Computing the closure Y^+ of a set of attributes Y
Given FDs F:



Example: Closure Test

- Consider R(a,b,c,d,e,f)
  with FDs ab -> c, ac -> d, c -> e, ade -> f
- Find Y^+ if Y = ab or find $\{a,b\}^+$



$\{a,b\}^+=\{a,b,c,d,e,f\}$ or *ab -> cdef*

ab is a candidate key!

Example : Closure Test

F: *AB -> C*
A -> D
D -> E
AC -> B

X	X_F^+
A	*{A, D, E}*
AB	*{A, B, C, D, E}*
AC	*{A, C, B, D, E}*
B	*{B}*
D	*{D, E}*

Is *AB -> E* entailed by ***F***? *Yes*
Is *D -> C* entailed by ***F***? *No*

Result: X_F^+ allows us to determine all FDs of the form
X -> Y entailed by ***F***

Example

- R(A, B, C, D, E and F). Suppose that this relation satisfies the FDs:

  AB→C,

  BC→AD,

  D→E,

  CF→B.

  Does AB→ F hold?

- Iterations:
  - X^+= {A,B} Use: AB→C
  - X^+= {A,B,C} Use: BC→AD
  - X^+= {A,B,C,D} Use: D→E
  - X^+= {A,B,C,D,E} No more changes to X are possible

- The FD: CF→B cannot be used because its left side is never contained in X^+.

Discarding redundant FDs

- Minimal basis: opposite from closure
- Given a set of FDs F, want a minimal F' s.t.
  - F' $\subseteq$ F
  - F' entails f for all f $\in$ F
- Properties of a minimal basis F'
  - RHS is always singleton
  - If any FD is removed from F', F' is no longer a minimal basis
  - If for any FD in F' we remove one or more attributes from the LHS of f $\in$ F', the result is no longer a minimal basis

In other words …

- *Minimal basis also referred to as minimal cover*
- *Minimal basis* for FDs:
  - Right sides are single attributes.
  - No FD can be removed.
  - No attribute can be removed from a left side.

- Constructing a minimal cover
  - Decompose RHS to single attributes
  - Repeatedly try to remove an FD and see if remaining FDs are equivalent to original set. That is, does the closure of the LHS attributes (of the removed FD) include the RHS attribute?
  - Repeatedly try to remove an attribute from a LHS and see if the removed attribute can be derived from the remaining FDs.

Constructing a minimal basis

Straightforward but time-consuming

1. Split all RHS into singletons
2. For all f in F', test whether J = $(F'-f)^+$ is still equivalent to F^+

=> Might make F' too small

3. For all i $\in$ LHS(f), for all f$\in$ F', let LHS(f')=LHS(f)-i
   Test whether $(F'-f+f')^+$ is still equivalent to F^+
4. Repeat (2) and (3) until neither makes progress

Minimal Basis: Example

- Relation R: R(A, B, C, D)
- Defined FDs:
  - F = {A->AC, B->ABC, D->ABC}

Find the minimal basis M of F

Minimal Basis: Example (cont.)

F = {A->AC, B->ABC, D->ABC}

1st Step

- H = {A->A, A->C, B->A, B->B, B->C, D->A, D->B, D->C}

2nd Step

- A->A, B->B: can be removed as trivial
- A->C: can't be removed, as there is no other LHS with A
- B->A: can't be removed, because for J=H-{B->A} is B^+=BC
- B->C: can be removed, because for J=H-{B->C} is B^+=ABC
- D->A: can be removed, because for J=H-{D->A} is D^+=DBA
- D->B: can't be removed, because for J=H-{D->B} is D^+=DC
- D->C: can be removed, because for J=H-{D->C} is D^+=DBAC

Step outcome => H = {A->C, B->A, D->B}

Minimal Basis: Example (cont.)

3rd Step

- H doesn't change as all LHS in H are single attributes

4th Step

- H doesn't change

Minimal Basis: M = H = {A->C, B->A, D->B}

Example

F = {A->BC, B->CE, A->E AC->H, D->B}

1st Step

- H = {A->B, A->C, B->C, B->E, A->E, AC->H, D->B}

2nd Step

- A->B: cannot be removed, A^+ = ACEH
- A->C: can be removed, A^+ = ABC
- B->C, B->E: cannot be removed, B^+ =BE, B^+ = BC, respectively
- A->E: can be removed, A^+ = ABCE
- AC->H: cannot be removed, AC^+ = AC
- D->B: cannot be removed, D^+ = D

Step outcome => H = {A->B, B->CE, AC->H, D->B}

Minimal Basis: Example (cont.)

3rd Step

H = {A->B, B->C, B->E, C->H D->B}

$A^+ = \{A,B,C\}$

Since C can be derived, it is redundant.

$C^+ = \{C\}$

Since A cannot be derived, it is not redundant.

4th Step

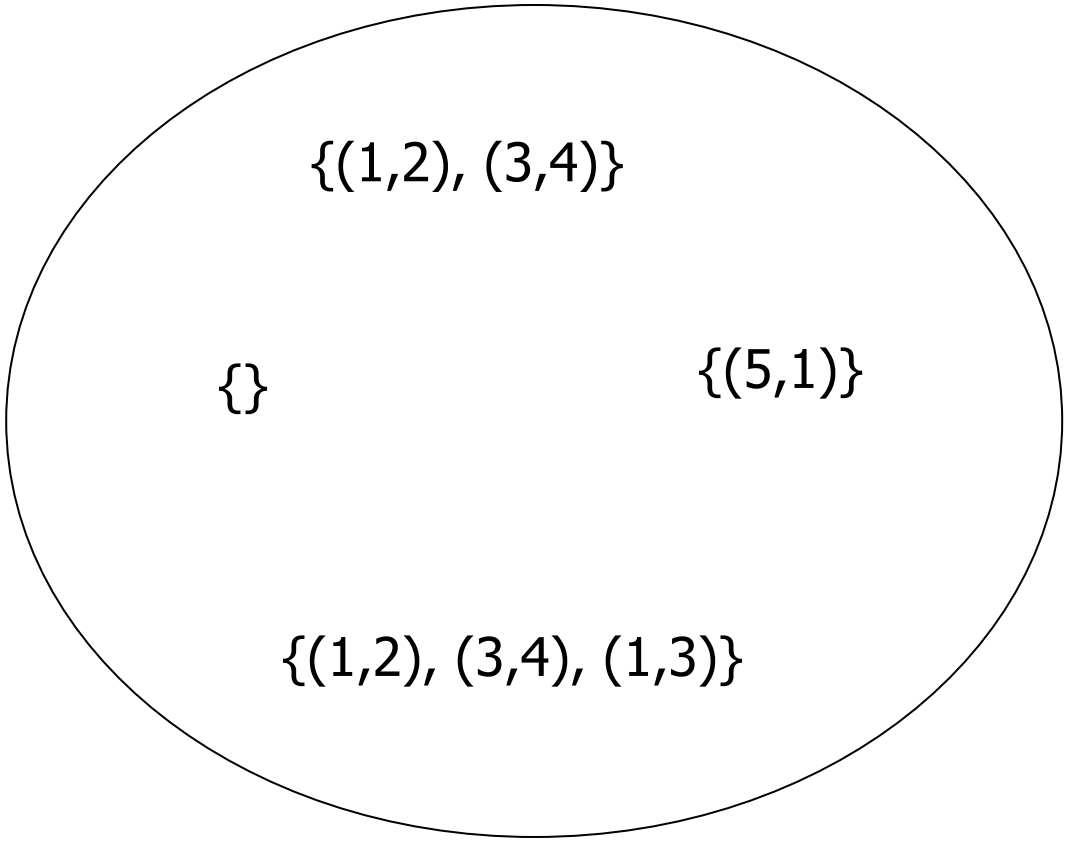
- H doesn't change

Minimal Basis: M = H = {A->BH, B->CE, D->B}

A Geometric View of FDs

- 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.

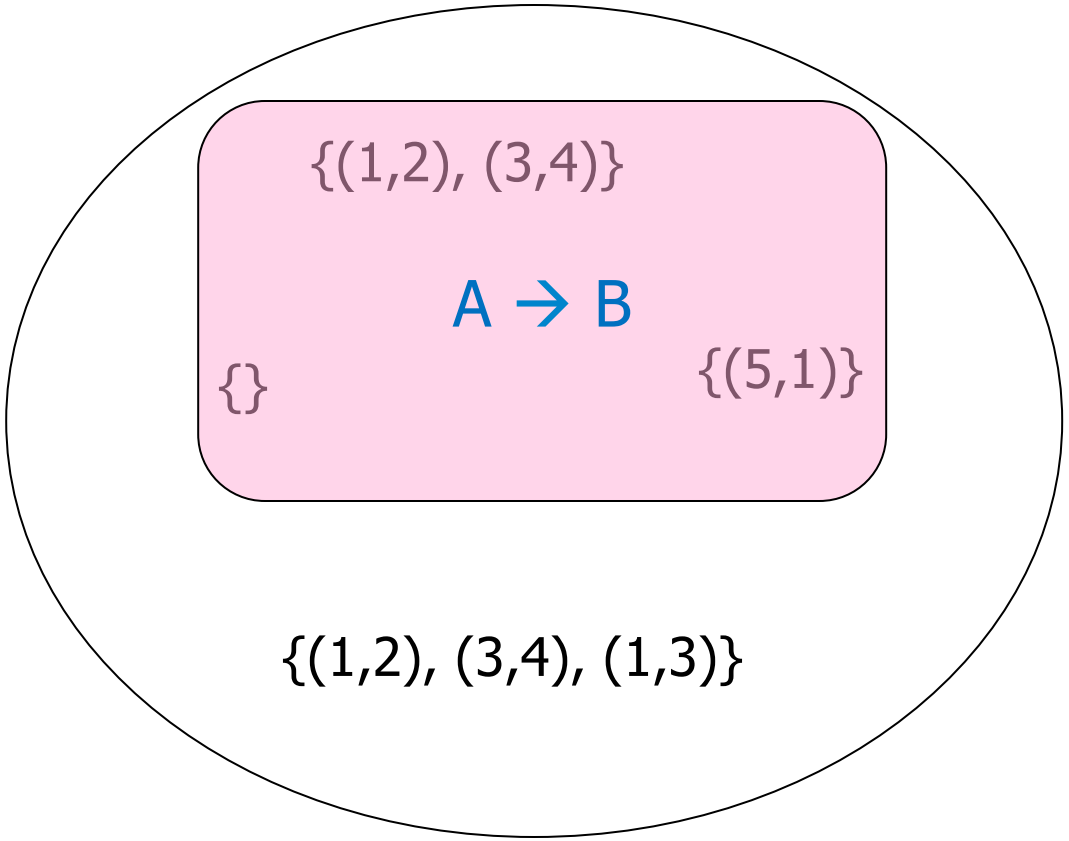
Example: R(A,B)



An FD is a Subset of Instances

- For each FD $X \rightarrow A$ there is a subset of all instances that satisfy the FD.
- We can represent an FD by a region in the space.

Example: A → B for R(A,B)

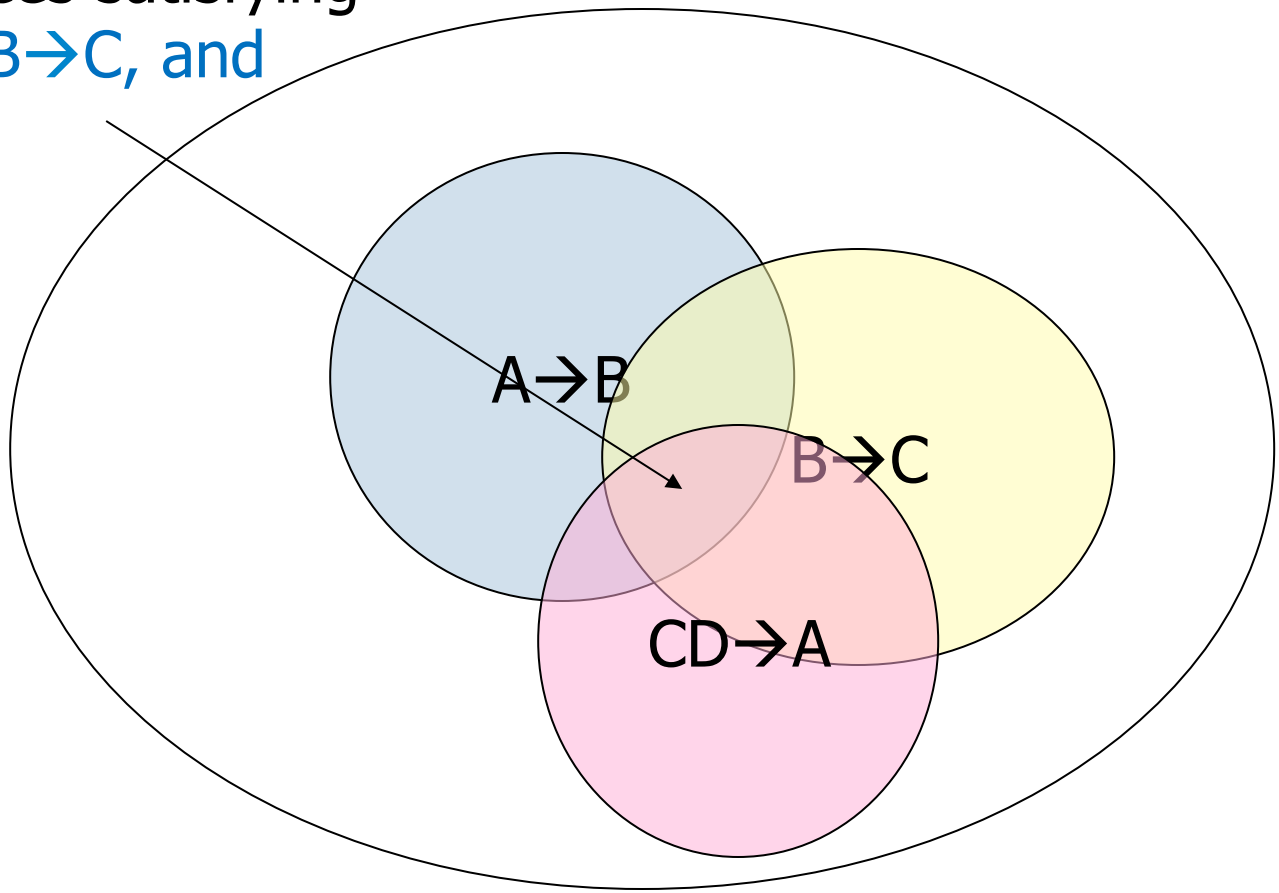


Representing *Sets* of FDs

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

Example

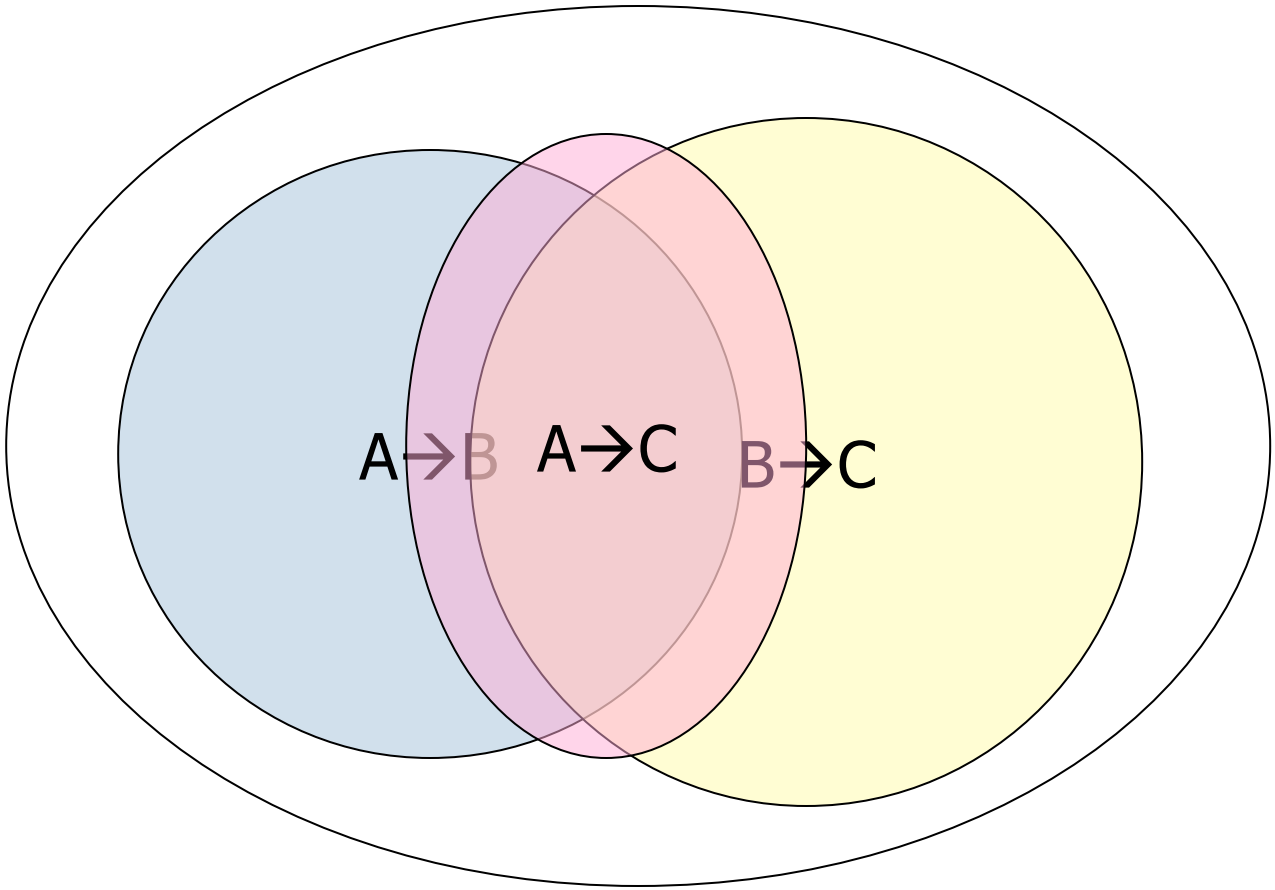
Instances satisfying
A→B, B→C, and
CD→A



Implication of FDs

- If an FD $Y \rightarrow B$ follows from FDs $X_1 \rightarrow A_1,\ldots,X_n \rightarrow A_n$, then the region in the space of instances for $Y \rightarrow B$ must include the intersection of the regions for the FDs $X_i \rightarrow A_i$.
  - That is, every instance satisfying all the FD's $X_i \rightarrow A_i$ surely satisfies $Y \rightarrow B$.
  - But an instance could satisfy $Y \rightarrow B$, yet not be in this intersection.
- For a set of FDs ***F***, F^+ (the closure of ***F***) is the set of all FDs implied by ***F***

Example



Part II:
Schema Decomposition

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 have been updated.
  - *Deletion anomaly* : valid fact is lost when a tuple is deleted.

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.

Example of Bad Design

Suppose we have FDs 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 FDs.

Goal of Decomposition

- Eliminate redundancy by decomposing a relation into several relations

- Check that a decomposition does not lead to bad design

FDs and redundancy

Given relation R and FDs F

- R often exhibits anomalies due to redundancy
- F identifies many (not all) of the underlying problems

Idea

- Use F to identify "good" ways to split relations
- Split R into 2+ smaller relations having less redundancy
- Split F into subsets which apply to the new relations (compute the projection of functional dependencies)

Schema decomposition

- Given relation R and FDs F
  - Split R into R_i s.t. for all i $R_i \subset R$ (no new attributes)
  - Split F into F_i s.t. for all i, F entails F_i (no new FDs)
  - F_i involves only attributes in R_i
- Caveat: entirely possible to lose information
  - F^+ may entail FD f which is not in $(\cup_i F_i)^+$

  => Decomposition lost some FDs
  - Possible to have $R \subset \bowtie_i R_i$

  => Decomposition lost some relationships
- Goal: minimize anomalies without losing info

Good Properties of Decomposition

1) Lossless Join Decomposition
   - When we join decomposed relations we should get ***exactly*** what we started with
2) Avoid anomalies
   - Avoid redundant data
3) Dependency Preservation
   - $(F_1 \cup \ldots \cup F_n)^+ = F^+$

Problem with Decomposition

Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation – information loss

Example: Splitting Relations

Student_Name	Student_Email	Course	Instructor
Alice	alice@gmail	SE 4DB3	Chiang
Alice	alice@gmail	CS 3SH3	Zheng
Bob	bob@mcmaster	SE 3RA3	Janicki
Laura	laura@gmail	SE 4DB3	Jones

Students (email, name)

Courses (code, instructor)

Taking (email, courseCode)

Students ⋈ Taking ⋈ Courses has additional tuples!

- (Alice, alice@gmail, SE4DB3, Jones), but Alice is not in Jones' section of SE 4DB3
- (Laura, laura@gmail, SE4DB3, Chiang), but Laura is not in Chiang's section of SE 4DB3

Why did this happen? How to prevent it?

Information loss with decomposition

- Decompose R into S and T
  - Consider FD A->B, with A only in S and B only in T
- FD loss
  - Attributes A and B no longer in same relation

  => Must join T and S to enforce A->B (expensive)
- Join loss
  - Neither (S ∩ T) -> S nor (S ∩ T) -> T in F^+

  => Joining T and S produces extraneous tuples

Lossless Join Decomposition

- A decomposition should not lose information
- A decomposition (**R**$_1$,…,**R**$_n$) of a schema, **R**, is **lossless** if every valid instance, r, of **R** can be reconstructed from its components:
  - $r = r_1 \bowtie \ldots \bowtie r_n$ where $r_i = \Pi_{Ri}(r)$

Lossy Decomposition

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?

- Lossy decomposition
  - Loses the fact that (12, Jen) lives at 2 Pine (not 3 Oak)
  - Loses the fact that (13, Jen) lives at 3 Oak
- Remember: lossy decompositions yield ***more*** tuples than they should when relations are joined together

Example 2

R

Model Name	Price	Category
a11	100	Canon
s20	200	Nikon
a70	150	Canon

R1

Model Name	Category
a11	Canon
s20	Nikon
a70	Canon

R2

Price	Category
100	Canon
200	Nikon
150	Canon

Ack: S.M. Lee

Example 2 (cont'd)

R1 ⋈ R2

Model Name	Price	Category
a11	100	Canon
a11	150	Canon
s20	200	Nikon
a70	100	Canon
a70	150	Canon

R

Model Name	Price	Category
a11	100	Canon
s20	200	Nikon
a70	150	Canon

Lossy decomposition

- Additional tuples are obtained along with original tuples
- Although there are more tuples, this leads to less information
- Due to the loss of information, the decomposition for the previous example is called lossy decomposition or lossy-join decomposition

Testing for Losslessness

- A (binary) decomposition of **R** = (R, **F**) into **R**1 = (R1, **F**1) and **R**2 = (R2, **F**2) is lossless if and only if:
  - either the FD (R1 ∩ R2) → R1 is in **F**+
  - or the FD (R1 ∩ R2) → R2 is in **F**+

  - all attributes common to both R1 and R2 functionally determine ALL the attributes in R1 OR
  - all attributes common to both R1 and R2 functionally determine ALL the attributes in R2

Decomposition Property

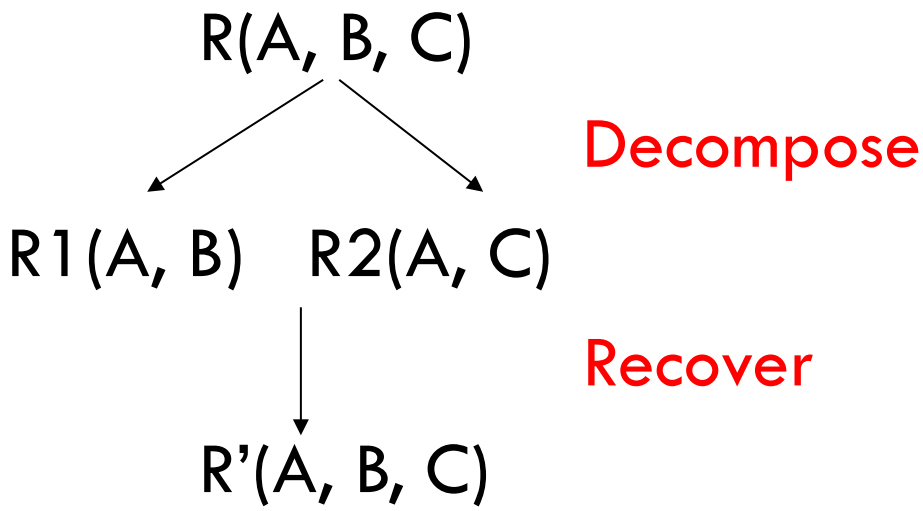
- In our example
  - *Name* $\not\rightarrow$ ID,*Name*
  - *Name* $\not\rightarrow$ *Name, Addr*
- *A lossless decomposition*
  - *[ID,Name] and [ID,Addr]*

- Example 2:
  - *Category* $\not\rightarrow$ ModelName, Category
  - *Category* $\not\rightarrow$ Price, Category
  - Better to use [MN, Category] and [MN, Price]

- In other words, if R1 $\cap$ R2 forms a superkey of either R1 or R2, the decomposition of R is a lossless decomposition

Lossless Decomposition

A decomposition is lossless if we can recover:



Thus, $R' = R$

Example : Lossless Decomposition

Given:

Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount*)

FDs:

branch-name $\longrightarrow$ *branch-city, assets*

loan-number $\longrightarrow$ *amount, branch-name*

Decompose Lending-schema into two schemas:

Branch-schema = (*branch-name, branch-city, assets*)

Loan-info-schema = (*branch-name, customer-name, loan-number, amount*)

Example : Lossless Decomposition

Show that the decomposition is a Lossless Decomposition

Branch-schema = (*branch-name, branch-city, assets*)

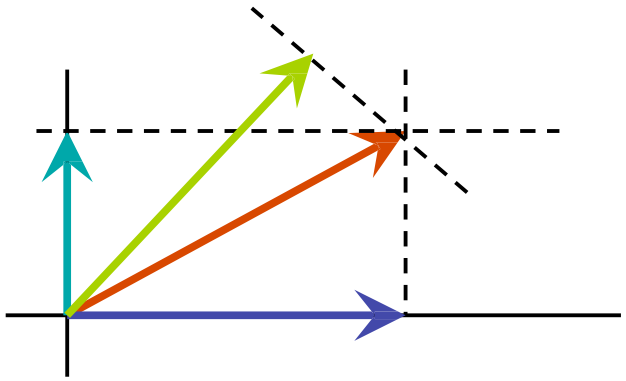
Loan-info-schema = (*branch-name, customer-name, loan-number, amount*)

- Since *Branch-schema* $\cap$ *Loan-info-schema* = {*branch-name*}
- We are given: *branch-name* $\longrightarrow$ *branch-city, assets*

Thus, this decomposition is lossless.

Projecting FDs

- Once we've split a relation, we have to re-factor our FDs to match
  - Each FDs must only mention attributes from one relation
- Similar to geometric projection
  - Many possible projections (depends on how we slice it)
  - Keep only the ones we need (minimal basis)



Projecting FDs

- Given:
  - a relation *R*
  - the set *F* of FDs that hold in *R*
  - a relation $R_i \subset R$
- Determine the set of all FDs F_i that
  - Follow from *F* and
  - Involve only attributes of R_i

FD Projection Algorithm

- Start with $F_i = \emptyset$
- For each subset X of R_i
  - Compute X^+
  - For each attribute A in X^+
    - If A is in R_i
      - add X -> A to F_i
- Compute the minimal basis of F_i

Making projection more efficient

- Ignore trivial dependencies
  - No need to add *X* -> *A* if *A* is in *X* itself
- Ignore trivial subsets
  - The empty set or the set of all attributes (both are subsets of X)
- Ignore supersets of X if $X^+ = R$
  - They can only give us "weaker" FDs (with more on the LHS)

Example: Projecting FDs

- *ABC* with FDs *A->B* and *B->C*
  - $A^+ = ABC$; yields *A->B, A->C*
    - We ignore A->A as trivial
    - We ignore the supersets of A, AB^+ and AC^+, because they can only give us "weaker" FDs (with more on the LHS)
  - $B^+ = BC$; yields *B->C*
  - $C^+ = C$; yields nothing.
  - $BC^+ = BC$; yields nothing.

Example cont'd

- Resulting FDs: *A->B*, *A->C*, and *B->C*
- Projection onto *AC* : *A->C*
  - Only FD that involves a subset of {*A*,*C*}
- Projection on BC: *B->C*
  - Only FD that involves subset of {B, C}

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?

Part III:
Normal Forms

Database Design Theory

- 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.

Acknowledgements: M. Papagelis, R. Johnson

Motivation for normal forms

- Identify a "good" schema
  - For some definition of "good"
  - Avoid anomalies, redundancy, etc.
- Many normal forms
  - 1st
  - 2nd
  - 3rd
  - Boyce-Codd
  - ... and several more we won't discuss…

$$BCNF \subseteq 3NF \subseteq 2NF \subseteq 1NF$$

1st Normal Form (1NF)

- No multi-valued attributes allowed
  - Imagine storing a list of values in an attribute
- Counter example
  - Course(name, instructor, [student,email]*)

Name	Instructor	Student Name	Student Email
CS 3DB3	Chiang	Alice	alice@gmail
		Mary	mary@mac
		Mary	mary@mac
SE 3SH3	Miller	Nilesh	nilesh@gmail

2nd normal form (2NF)

- Non-key attributes depend on candidate keys
  - Consider non-key attribute A
  - Then there exists an FD X s.t. X -> A, and X is a candidate key
- Counter-example
  - Movies(title, year, star, studio, studioAddress, salary)
  - FD: title, year -> studio; studio -> studioAddress; star->salary

Title	Year	Star	Studio	StudioAddr	Salary
Star Wars	1977	Hamill	Lucasfilm	1 Lucas Way	$100,000
Star Wars	1977	Ford	Lucasfilm	1 Lucas Way	$100,000
Star Wars	1977	Fisher	Lucasfilm	1 Lucas Way	$100,000
Patriot Games	1992	Ford	Paramount	Cloud 9	$2,000,000
Last Crusade	1989	Ford	Lucasfilm	1 Lucas Way	$1,000,000

3rd normal form (3NF)

- Non-prime attr. depend *only* on candidate keys
  - Consider FD X -> A
  - Either X is a superkey OR A is *prime* (part of a key)
- Counter-example:
  - studio -> studioAddr

    (*studioAddr* depends on *studio* which is not a candidate key)

Title	Year	Studio	StudioAddr
Star Wars	1977	Lucasfilm	1 Lucas Way
Patriot Games	1992	Paramount	Cloud 9
Last Crusade	1989	Lucasfilm	1 Lucas Way

3NF, dependencies, and join loss

- Theorem: always possible to convert a schema to lossless join, dependency-preserving 3NF
- Caveat: always *possible* to create schemas in 3NF for which these properties do not hold
- FD loss example 1:
  - MovieInfo(title, year, studioName)
  - StudioAddress(title, year, studioAddress)

  => Cannot enforce studioName -> studioAddress
- Join loss example 2:
  - Movies(title, year, star)
  - StarSalary(star, salary)

  => Movies ⋈ StarSalary yields additional tuples

Boyce-Codd normal form (BCNF)

- One additional restriction over 3NF
  - All non-trivial FDs have superkey LHS
- Counterexample
  - CanadianAddress(<u>street</u>, <u>city</u>, <u>province</u>, postalCode)
  - Candidate keys: {street, postalCode}, {street, city, province}
  - FD: postalCode -> city, province
  - Satisfies 3NF: city, province both prime
  - Violates BCNF: postalCode is not a superkey

=> Possible anomalies involving postalCode

Boyce-Codd Normal Form

- We say a relation *R* is in *BCNF* if whenever $X \rightarrow A$ is a nontrivial FD that holds in *R*, *X* is a superkey.
  - Remember: *nontrivial* means A is not contained in *X*.

Example: a relation not in BCNF

Drinkers(name, addr, beersLiked, 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 FDs shows *Drinkers* is not in BCNF

Another Example

Beers(name, manf, manfAddr)

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

- Beers w.r.t. name->manf does not violate BCNF, but manf->manfAddr does.

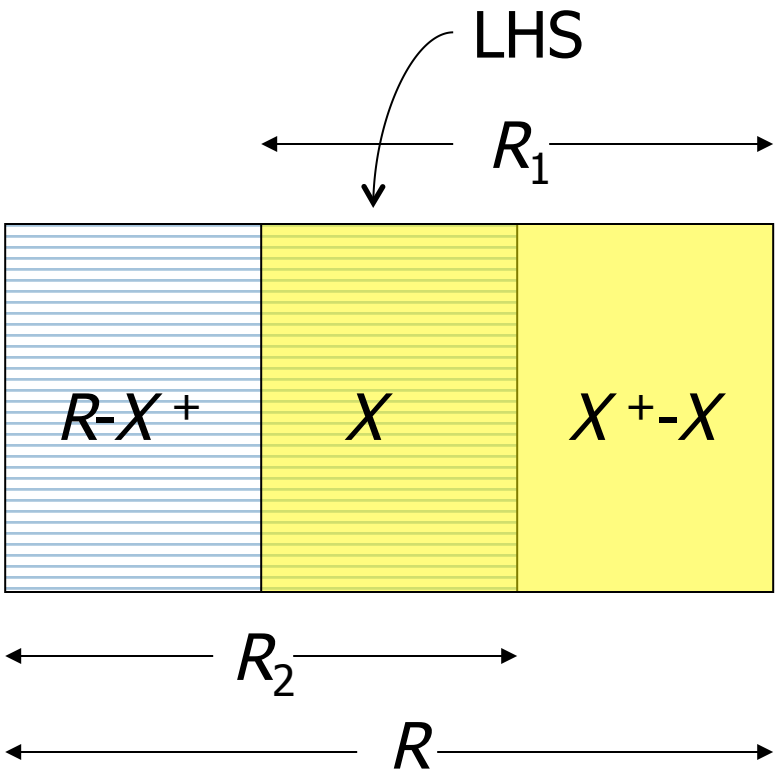
In other words, BCNF requires that:
the only FDs that hold are the result of key(s).
Why does that help?

Decomposition into BCNF

- Given: relation *R* with FDs *F*
- Look among the given FDs for a BCNF violation $X \rightarrow Y$ (i.e., X is not a superkey)

- Compute X^+.
  - Find $X^+ \neq X \neq$ all attributes, (o.w. X is a superkey)
- Replace *R* by relations with:
  - $R_1 = X^+$.
  - $R_2 = R - (X^+ - X) = R - X^+ \cup X$
- Continue to recursively decompose the two new relations
- *Project* given FDs *F* onto the two new relations.

Decomposition on X → Y



Example: BCNF Decomposition

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

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

Key = name, beersLiked

- Pick BCNF violation name->addr.
- Closure: {name}$^+$ = {name, addr, favBeer}.
- Decomposed relations:
  - Drinkers1(name, addr, favBeer)
  - Drinkers2(name, beersLiked, manf)

Example -- Continued

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

Example -- Continued

- For Drinkers2(name, beersLiked, 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:
  - Drinkers3(beersLiked, manf)
  - Drinkers4(name, beersLiked)

Example -- Concluded

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

What we want from a decomposition

- *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.
- *No anomalies*
- *Dependency Preservation* : All the original FDs should be satisfied.

What we get from a BCNF decomposition

- *Lossless Join* : ✓
- *No anomalies* : ✓
- *Dependency Preservation* : ✗

Example: Failure to preserve dependencies

- Suppose we start with R(*A,B,C*) and FDs
  - *AB* ->*C* and *C* ->*B*.
- There are two keys, {*A,B* } and {*A,C* }.
- *C* ->*B* is a BCNF violation, so we must decompose into *AC*, *BC*.

The problem is that if we use *AC* and *BC* as our database schema, we cannot enforce the FD *AB* →*C* in these decomposed relations.

3NF Let's Us Avoid This Problem

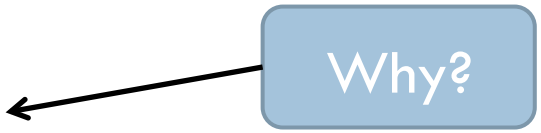
- *3rd Normal Form* (3NF) modifies the BCNF condition so we do not have to decompose in this problem situation.
- An attribute is *prime* if it is a member of any key.
- $X \rightarrow A$ violates 3NF if and only if X is not a superkey, and also A is not prime.
- i.e., it's ok if X is not a superkey,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

Example: 3NF

- In our problem situation with FDs *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

- *Lossless Join* : ✓
- *No anomalies* : ✗
- *Dependency Preservation* : ✓



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

3NF Synthesis Algorithm

- We can always construct a decomposition into 3NF relations with a lossless join and dependency preservation.
- Need *minimal basis* for the FDs (same as used in projection)
  - Right sides are single attributes.
  - No FD can be removed.
  - No attribute can be removed from a left side.

3NF Synthesis – (2)

- 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.
- FDs *A->B* and *A->C*.
- Decomposition: AB and AC from the FDs, plus AD for a key.

Limits of decomposition

- Pick two…
  - Lossless join
  - Dependency preservation
  - Anomaly-free
- 3NF
  - Provides lossless join and dependency preserving
  - May allow some anomalies
- BCNF
  - Anomaly-free, lossless join
  - Sacrifice dependency preservation

Use domain knowledge to choose 3NF vs. BCNF