Lectures 5 & 7: Design Theory

Announcements

Homework #1 due today! Homework was not easy

- You learned a new, ***declarative way of programming!***
  - **Hope**: you got the concept, so you can pick up a book on SQL whenever you need.

- Some issues with iPython+SQLite versions. Ugh!
  - You were **great** about this issues! Constructive questions and feedback! Thanks!
  - You will never **need** iPython to submit homework
  - *Setting Expectations*: cf. simply text assignments with no expected output.

- **Searching** on piazza is problem, we'll try to do a better job aggregating posts…
  - Lots of stuff was there, but hard to find!
  - Thank you to those who aggregated posts! (Candy?)

Announcements #2

Office hours last night was filled with nice people!

- Queue management had a hiccup (our mistake?) Resulted in “fake” long queue times... *Don’t be scared away!*
- We need a better way to group and identify problems (Luke is on it! Thanks!)
- We will be better about triaging issues to get help and make groups quickly.
- **Thanks to CAs who came in for extra time!**

Announcements #3

- **How to use activity time:** Review slides, ask questions, **or** do the activity!
  - We designed these to cope with different backgrounds and pace requirements. If you don't get the activity in class **YOU ARE DOING FINE!**

- **Thank you**! We appreciate the "thank you"s for new material!
  - Be aggressive about giving feedback, *we want you to have best class possible!*
  - *There will be hiccups in new material, please start early.*
  - We're trying to make class more fun and `teach you more material!!!

- Guest lecture on Thursday from GOOGLE. HAVE FUN!

Lecture 5:
Design Theory I

Today's Lecture

1. Normal forms & functional dependencies
   - ACTIVITY: Finding FDs
2. Finding functional dependencies
   - ACTIVITY: Compute the closures
3. Closures, superkeys & keys
   - ACTIVITY: The key or a key?

1. Normal forms & functional dependencies

What you will learn about in this section

1. Overview of design theory & normal forms
2. Data anomalies & constraints
3. Functional dependencies
4. ACTIVITY: Finding FDs

Design Theory

- Design theory is about how to represent your data to avoid ***anomalies***.

- It is a mostly mechanical process
  - Tools can carry out routine portions

- *We have a notebook implementing all algorithms!*
  - *We'll play with it in the activities!*

Normal Forms

- 1st Normal Form (1NF) = All tables are flat

- *2nd Normal Form = disused*

- **Boyce-Codd Normal Form (BCNF)**
- **3rd Normal Form (3NF)**

DB designs based on *functional dependencies*, intended to prevent data ***anomalies***

Our focus in this lecture + next one

- *4th and 5th Normal Forms = see text books*

1st Normal Form (1NF)

Student	Courses
Mary	{CS145,CS229}
Joe	{CS145,CS106}
…	…

Violates 1NF.

Student	Courses
Mary	CS145
Mary	CS229
Joe	CS145
Joe	CS106

In 1st NF

1NF Constraint: Types must be atomic!

Data Anomalies & Constraints

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

If every course is in only one room, contains ***redundant*** information!

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
Mary	CS145	B01
Joe	CS145	C12
Sam	CS145	B01
..	..	..

If we update the room number for one tuple, we get inconsistent data = an ***update*** anomaly

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

Student	Course	Room
..	..	..

If everyone drops the class, we lose what room the class is in! = a ***delete*** anomaly

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes ***anomalies***:

…	CS229	C12

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Similarly, we can't reserve a room without students = an ***insert*** anomaly

Constraints Prevent (some) Anomalies in the Data

Student	Course
Mary	CS145
Joe	CS145
Sam	CS145
..	..

Course	Room
CS145	B01
CS229	C12

Is this form better?

- Redundancy?
- Update anomaly?
- Delete anomaly?
- Insert anomaly?

Today: develop theory to understand why this design may be better **and** how to find this *decomposition*…

Functional Dependencies

Functional Dependency

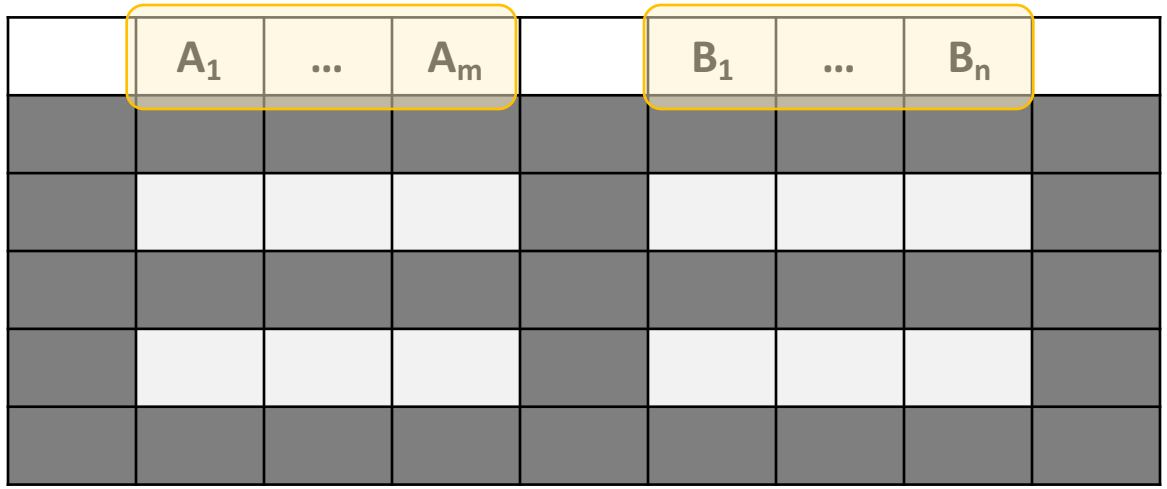
Def: Let A,B be *sets* of attributes
We write A → B or say A ***functionally determines*** B if, for any tuples t_1 and t_2:

$$t_1[A] = t_2[A] \text{ implies } t_1[B] = t_2[B]$$

and we call A → B a **functional dependency**

A->B means that
"whenever two tuples agree on A then they agree on B."

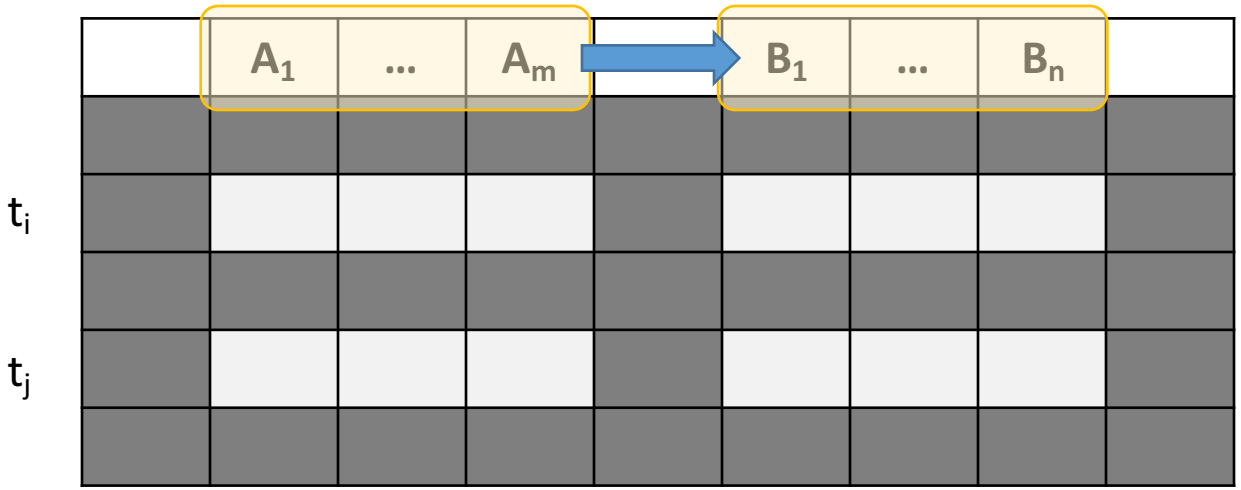
A Picture Of FDs



Defn (again):

Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots B_n\}$ in R,

A Picture Of FDs

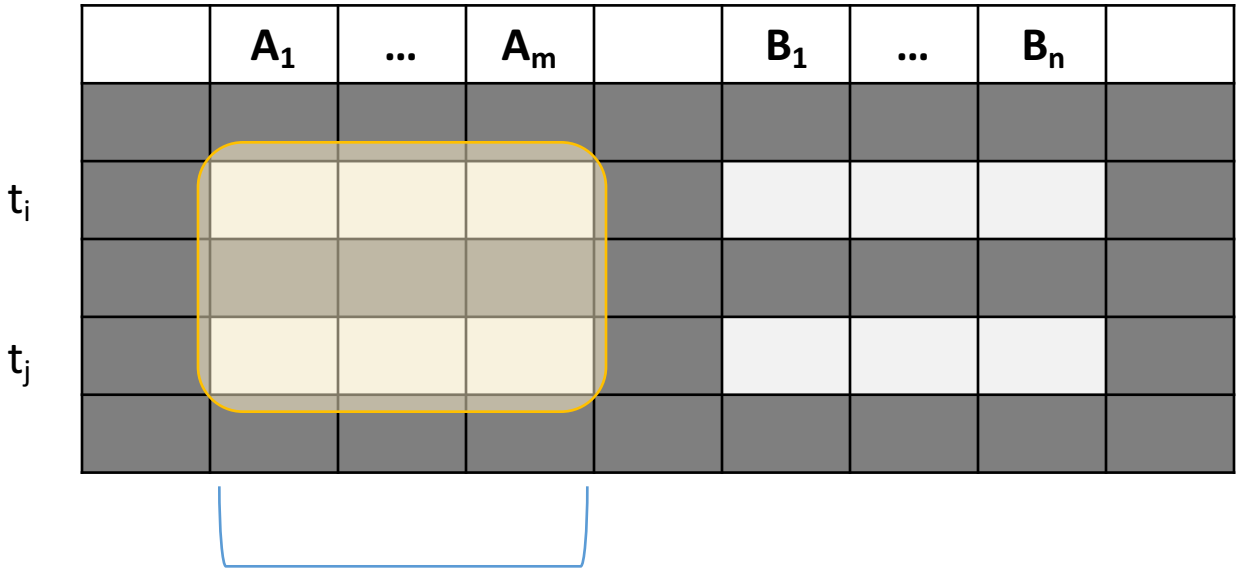


Defn (again):

Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots B_n\}$ in R,

The *functional dependency* A→ B on R holds if for ***any*** t_i,t_j in R:

A Picture Of FDs



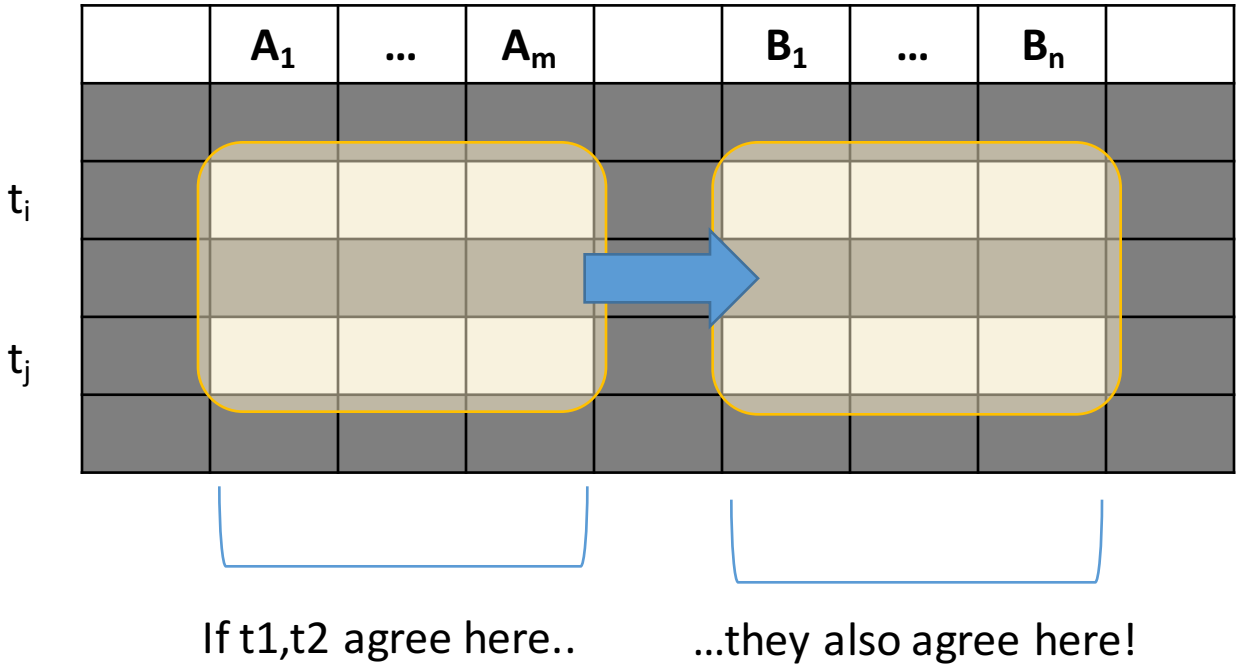
Defn (again):

Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots B_n\}$ in R,

The *functional dependency* A→ B on R holds if for ***any*** t_i,t_j in R:

$t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

A Picture Of FDs



Defn (again):

Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots B_n\}$ in R,

The *functional dependency* A→ B on R holds if for ***any*** t_i,t_j in R:

if $t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

then $t_i[B_1] = t_j[B_1]$ AND $t_i[B_2]=t_j[B_2]$ AND … AND $t_i[B_n] = t_j[B_n]$

FDs for Relational Schema Design

- High-level idea: **why do we care about FDs?**

    1. Start with some relational *schema*

    2. Find out its *functional dependencies (FDs)*

    3. Use these to *design a better schema*
        1. One which minimizes the possibility of anomalies

Functional Dependencies as Constraints

A **functional dependency** is a form of **constraint**

- *Holds* on some instances not others.
- Part of the schema, helps define a valid *instance*.

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Note: The FD {Course} -> {Room} ***holds on this instance***

*Recall: an **instance** of a schema is a multiset of tuples conforming to that schema, **i.e. a table***

Functional Dependencies as Constraints

Note that:

- You can check if an FD is **violated** by examining a single instance;

- However, you **cannot prove** that an FD is part of the schema by examining a single instance.
  - *This would require checking every valid instance*

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

However, cannot *prove* that the FD {Course} -> {Room} is ***part of the schema***

More Examples

An FD is a constraint which 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

More Examples

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

{Position} → {Phone}

More Examples

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

but *not* {Phone} → {Position}

ACTIVITY

A	B	C	D	E
1	2	4	3	6
3	2	5	1	8
1	4	4	5	7
1	2	4	3	6
3	2	5	1	8

Find at least *three* FDs which hold on this instance:

{ } → { }
{ } → { }
{ } → { }

2. Finding functional dependencies

What you will learn about in this section

1. “Good” vs. “Bad” FDs: Intuition
2. Finding FDs
3. Closures
4. ACTIVITY: Compute the closures

"Good" vs. "Bad" FDs

We can start to develop a notion of **good** vs. **bad** FDs:

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

Intuitively:

EmpID -> Name, Phone, Position is *"good FD"*

- ***Minimal redundancy, less possibility of anomalies***

"Good" vs. "Bad" FDs

We can start to develop a notion of **good** vs. **bad** FDs:

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

Intuitively:

EmpID -> Name, Phone, Position is *"good FD"*

But Position -> Phone *is a "bad FD"*

- ***Redundancy! Possibility of data anomalies***

“Good” vs. “Bad” FDs

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Returning to our original example… can you see how the “bad FD” {Course} -> {Room} could lead to an:

- Update Anomaly
- Insert Anomaly
- Delete Anomaly
- …

Given a set of FDs (from user) our goal is to:

1. **Find all FDs, and**
2. **Eliminate the “Bad Ones".**

FDs for Relational Schema Design

- High-level idea: **why do we care about FDs?**

1. Start with some relational *schema*
2. Find out its *functional dependencies (FDs)*
3. Use these to *design a better schema*
    1. One which minimizes possibility of anomalies

This part can be tricky!

Finding Functional Dependencies

- There can be a very **large number** of FDs…
  - *How to find them all efficiently?*
- We can't necessarily show that any FD will hold **on all instances…**
  - *How to do this?*

We will start with this problem:
Given a set of FDs, F, what other FDs ***must*** hold?

Finding Functional Dependencies

Equivalent to asking: Given a set of FDs, F = $\{f_1,\ldots f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Finding Functional Dependencies

Example:

Products

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Department}
3. {Color, Category} → {Price}

Given the provided FDs, we can see that {Name, Category} → {Price} must also hold on **any instance**…

Which / how many other FDs do?!?

Finding Functional Dependencies

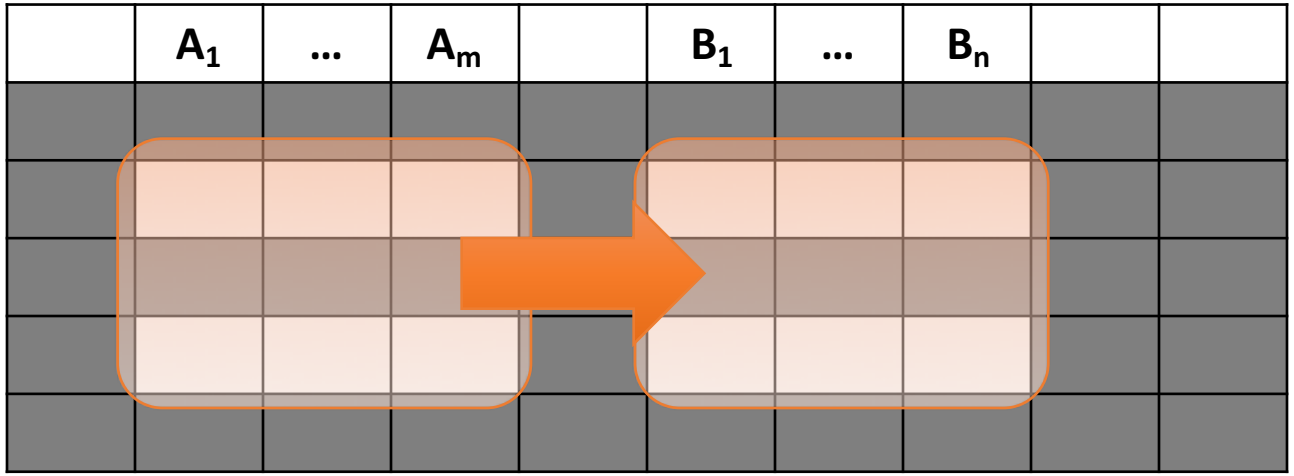
Equivalent to asking: Given a set of FDs, F = $\{f_1,\ldots f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Answer: Three simple rules called **Armstrong's Rules.**

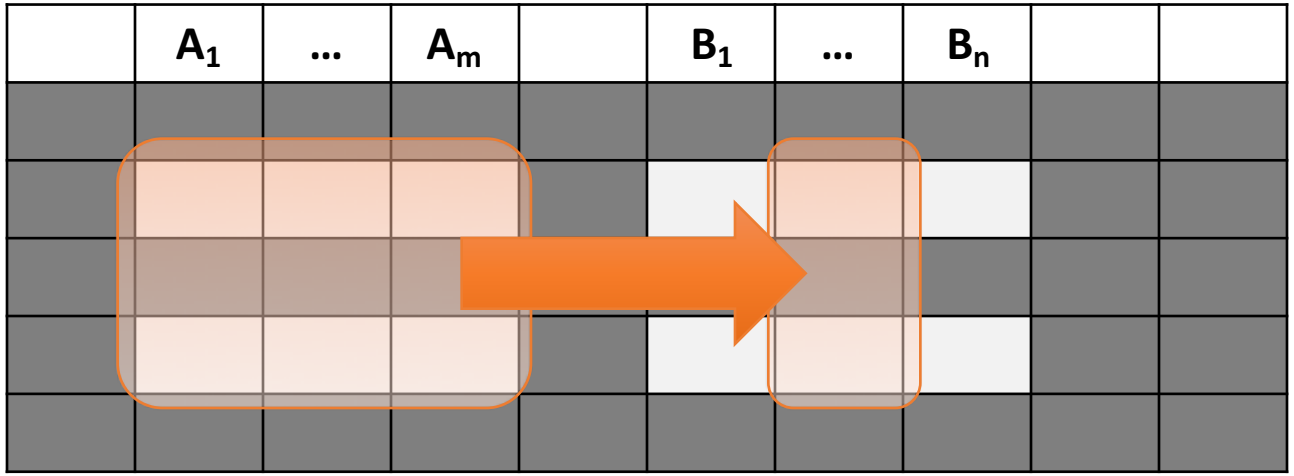
1. **Split/Combine,**
2. **Reduction, and**
3. **Transitivity…** *ideas by picture*

1. Split/Combine



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

1. Split/Combine

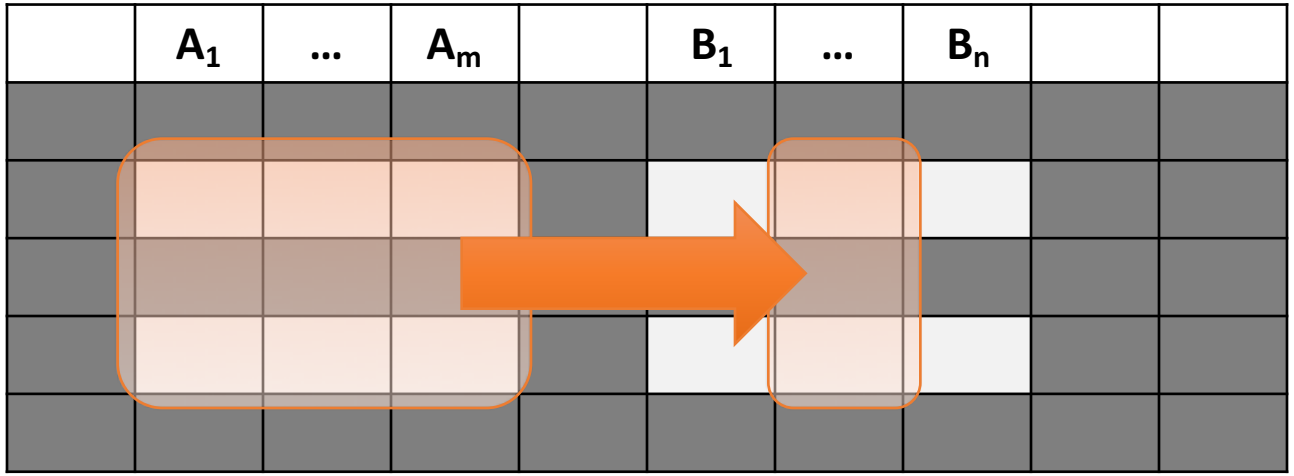


$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

… is equivalent to the following *n* FDs…

$A_1,\ldots,A_m \rightarrow B_i$ for i=1,…,n

1. Split/Combine

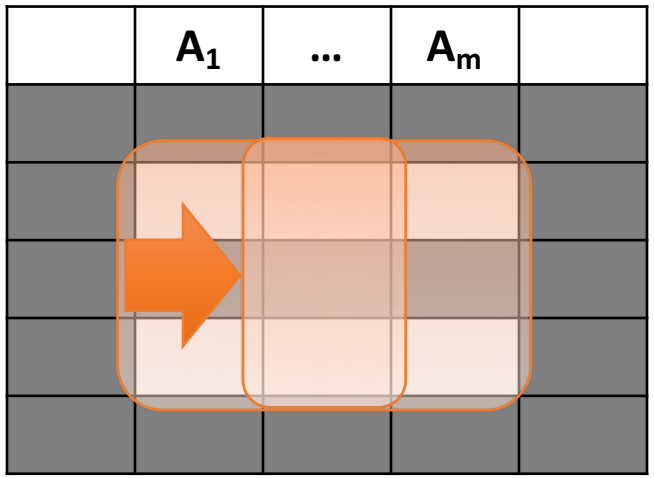


And vice-versa, $A_1,\ldots,A_m \rightarrow B_i$ for i=1,…,n

… is equivalent to …

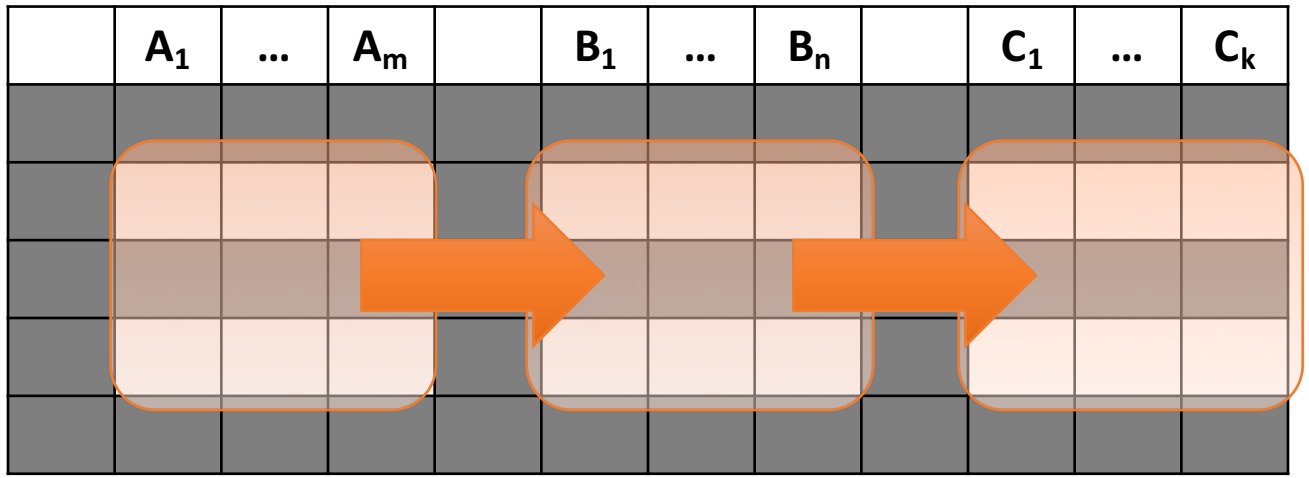
$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

Reduction/Trivial



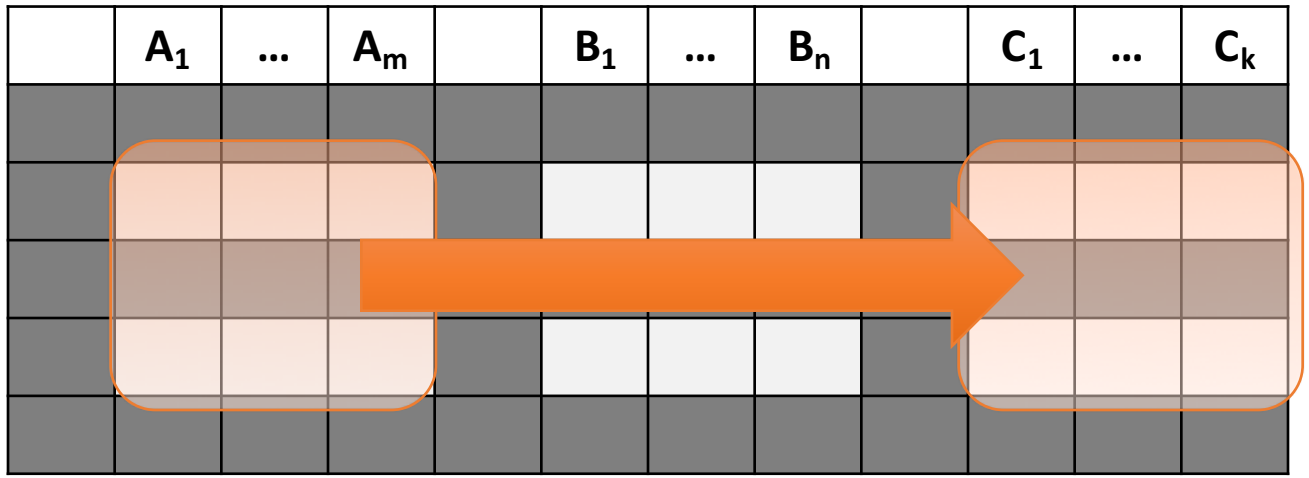
$A_1,\ldots,A_m \rightarrow A_j$ for any $j=1,\ldots,m$

3. Transitive Closure



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and
$B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

3. Transitive Closure



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and
$B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

implies

$A_1,\ldots,A_m \rightarrow C_1,\ldots,C_k$

Finding Functional Dependencies

Example:

Products

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Department}
3. {Color, Category} → {Price}

Which / how many other FDs hold?

Finding Functional Dependencies

Example:

Inferred FDs:

Inferred FD	Rule used
4. {Name, Category} -> {Name}	?
5. {Name, Category} -> {Color}	?
6. {Name, Category} -> {Category}	?
7. {Name, Category -> {Color, Category}	?
8. {Name, Category} -> {Price}	?

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

Which / how many other FDs hold?

Finding Functional Dependencies

Example:

Inferred FDs:

Inferred FD	Rule used
4. {Name, Category} -> {Name}	Trivial
5. {Name, Category} -> {Color}	Transitive (4 -> 1)
6. {Name, Category} -> {Category}	Trivial
7. {Name, Category -> {Color, Category}	Split/combine (5 + 6)
8. {Name, Category} -> {Price}	Transitive (7 -> 3)

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

Can we find an algorithmic way to do this?

Closures

Closure of a set of Attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs **F:**
Then the **closure**, $\{A_1, \ldots, A_n\}^+$ is the set of attributes **B** s.t. $\{A_1, \ldots, A_n\} \rightarrow B$

Example: F =

```
{name} → {color}
{category} → {department}
{color, category} → {price}
```

Example Closures:

```
{name}+ = {name, color}
{name, category}+ =
{name, category, color, dept, price}
{color}+ = {color}
```

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$ and set of FDs F.

Repeat until X doesn't change; **do**:

if $\{B_1, \ldots, B_n\} \rightarrow C$ is entailed by F

and $\{B_1, \ldots, B_n\} \subseteq X$

then add C to X.

Return X as X^+

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

`{name, category}`$^+$ = `{name, category}`

F =

`{name}` → `{color}`

`{category}` → `{dept}`

`{color, category}` → `{price}`

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

{name, category}$^+$ =
{name, category}

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

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

{name, category}+ =
{name, category}

{name, category}+ =
{name, category, color}

{name, category}+ =
{name, category, color, dept}

Closure Algorithm

Start with X = $\{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

{name, category}+ =
{name, category}

{name, category}+ =
{name, category, color}

{name, category}+ =
{name, category, color, dept}

{name, category}+ =
{name, category, color, dept, price}

Example

R(A,B,C,D,E,F)

{A,B} → {C}
{A,D} → {E}
{B} → {D}
{A,F} → {B}

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

Compute $\{A, F\}^+ = \{A, F,$ $\}$

Example

R(A,B,C,D,E,F)

{A,B} → {C}
{A,D} → {E}
{B} → {D}
{A,F} → {B}

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

Compute $\{A, F\}^+ = \{A, F, B$ $\}$

Example

R(A,B,C,D,E,F)

{A,B} → {C}
{A,D} → {E}
{B} → {D}
{A,F} → {B}

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

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

Activity-5-2.ipynb

3. Closures, Superkeys & Keys

What you will learn about in this section

1. Closures Pt. II
2. Superkeys & Keys
3. ACTIVITY: The key or a key?

Why Do We Need the Closure?

- With closure we can find all FD's easily

- To check if $X \rightarrow A$

  1. Compute X^+
  2. Check if $A \in X^+$

Note here that **X** is a *set* of attributes, but **A** is a *single* attribute. Why does considering FDs of this form suffice?

Recall the **Split/combine** rule:
$X \rightarrow A_1, \ldots, X \rightarrow A_n$
implies
$X \rightarrow \{A_1, \ldots, A_n\}$

Using Closure to Infer ALL FDs

Step 1: Compute X^+, for every set of attributes X:

Example:

Given F =

{A,B} → C
{A,D} → B
{B} → D

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

No need to compute these- why?

Using Closure to Infer ALL FDs

Example:

Given F =

```
{A,B} → C
{A,D} → B
{B}   → D
```

Step 1: Compute X^+, for every set of attributes X:

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

Step 2: Enumerate all FDs X → Y, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

$\{A,B\} \rightarrow \{C,D\}$, $\{A,D\} \rightarrow \{B,C\}$, $\{A,B,C\} \rightarrow \{D\}$, $\{A,B,D\} \rightarrow \{C\}$, $\{A,C,D\} \rightarrow \{B\}$

Using Closure to Infer ALL FDs

Example:

Given F =

{A,B} → C
{A,D} → B
{B} → D

Step 1: Compute X^+, for every set of attributes X:

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

Step 2: Enumerate all FDs X → Y, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

"Y is in the closure of X"

{A,B} → {C,D}, {A,D} → {B,C},
{A,B,C} → {D}, {A,B,D} → {C},
{A,C,D} → {B}

Using Closure to Infer ALL FDs

Example:

Given F =

```
{A,B} → C
{A,D} → B
{B}   → D
```

Step 1: Compute X^+, for every set of attributes X:

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

Step 2: Enumerate all FDs X → Y, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

$\{A,B\} \rightarrow \{C,D\}$, $\{A,D\} \rightarrow \{B,C\}$, $\{A,B,C\} \rightarrow \{D\}$, $\{A,B,D\} \rightarrow \{C\}$, $\{A,C,D\} \rightarrow \{B\}$

The FD X → Y is non-trivial

Superkeys and Keys

Keys and Superkeys

A **superkey** is a set of attributes $A_1, \ldots, A_n$ s.t. for *any other* attribute **B** in R, we have $\{A_1, \ldots, A_n\} \rightarrow B$

I.e. all attributes are *functionally determined* by a superkey

A **key** is a *minimal* superkey

Meaning that no subset of a key is also a superkey

Finding Keys and Superkeys

- For each set of attributes X

  1. Compute X^+
  2. If X^+ = set of all attributes then X is a **superkey**
  3. If X is minimal, then it is a **key**

Do we need to check all sets of attributes? Which sets?

Example of Finding Keys

```
Product(name, price, category, color)
```

```
{name, category} → price
{category} → color
```

What is a key?

Example of Keys

Product(name, price, category, color)

{name, category} → price
{category} → color

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

= the set of all attributes

$\Longrightarrow$ this is a **superkey**

$\Longrightarrow$ this is a **key**, since neither name nor category alone is a superkey

Activity-5-3.ipynb

Lecture 7:
Design Theory II

Today's Lecture

1. PS#1 Review (***via AJ RATNER!!!!***)

2. Decompositions
   - ACTIVITY

3. Online Course Feedback
   - Give you some time in class to fill it out.
   - Want feedback about new format (20+10/notebooks/heavy ps feedback). Whatever else we're doing wrong ☺

PS1: What you learned

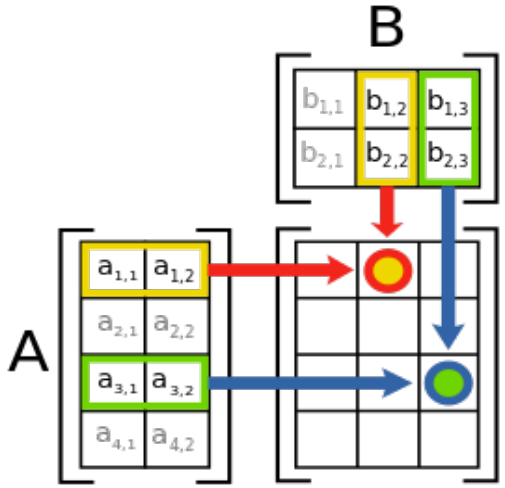
- This was a **tough** problem set- congratulations on doing so well!

- You used a **declarative** programming language (SQL) to
  - do *linear algebra*
  - answer *questions* about data
  - do *graph* operations
    - Cool stuff! However the point is not these specific applications…

- **Less tricky** versions of these same types of queries will be fair game for exams

Linear Algebra, Declaratively

- Matrix multiplication & other operations = just **joins**!
- The shift from **procedural** to **declarative** programming

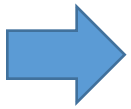


$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

```
C = [[0]*p for i in range(n)]

for i in range(n):
  for j in range(p):
    for k in range(m):
      C[i][j] += A[i][k] * B[k][j]
```

Proceed through a series of instructions



```
SELECT A.i, B.j, SUM(A.x * B.x)
FROM A, B
WHERE A.j = B.i
GROUP BY A.i, B.j;
```

Declare a desired output set

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

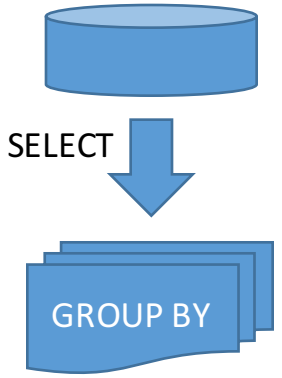
```
SELECT station_id,
       COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

Think about **order*!**

**of the semantics, not the actual execution*

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries



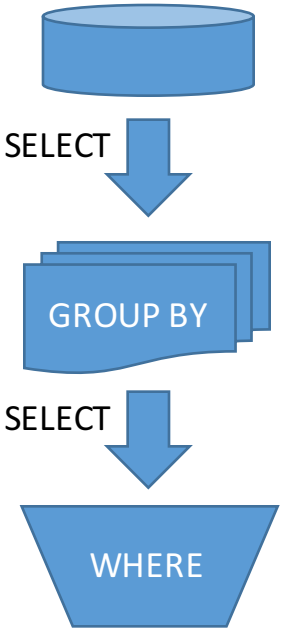
Get the max precipitation **by day**

```
SELECT station_id,
       COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
       COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```



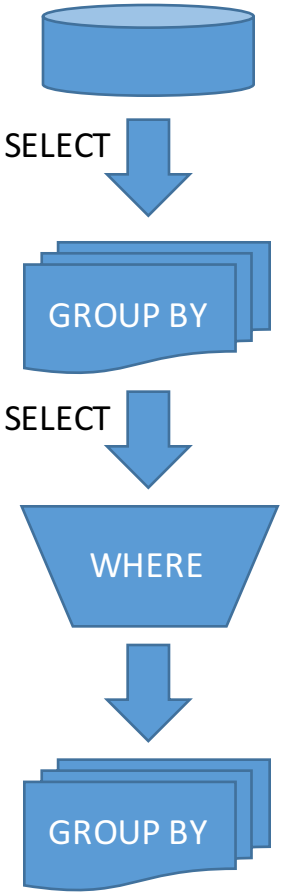
Get the max precipitation **by day**

Get the station, day pairs where / when this happened

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
       COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```



Get the max precipitation **by day**

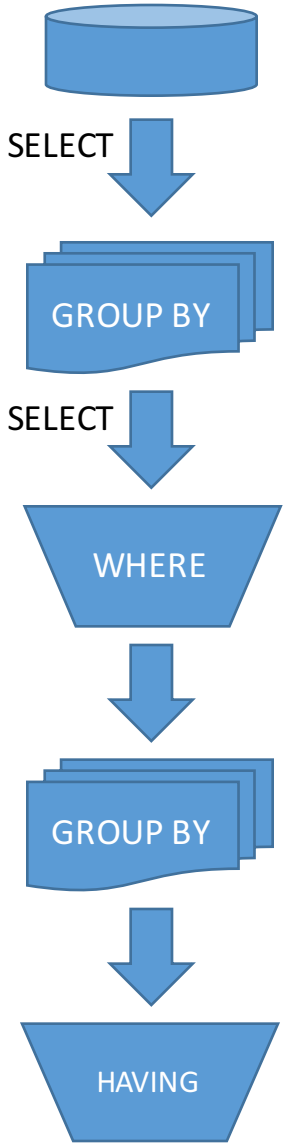
Get the station, day pairs where / when this happened

Group by stations

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,
       COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```



Get the max precipitation **by day**

Get the station, day pairs where / when this happened

Group by stations

Having > 1 such day

Common SQL Query Paradigms

Complex correlated queries

```
SELECT x1.p AS median
FROM x AS x1
WHERE
      (SELECT COUNT(*)
       FROM X AS x2
       WHERE x2.p > x1.p)
      =
      (SELECT COUNT(*)
       FROM X AS x2
       WHERE x2.p < x1.p);
```

This was a tricky problem- but good practice in thinking about things declaratively

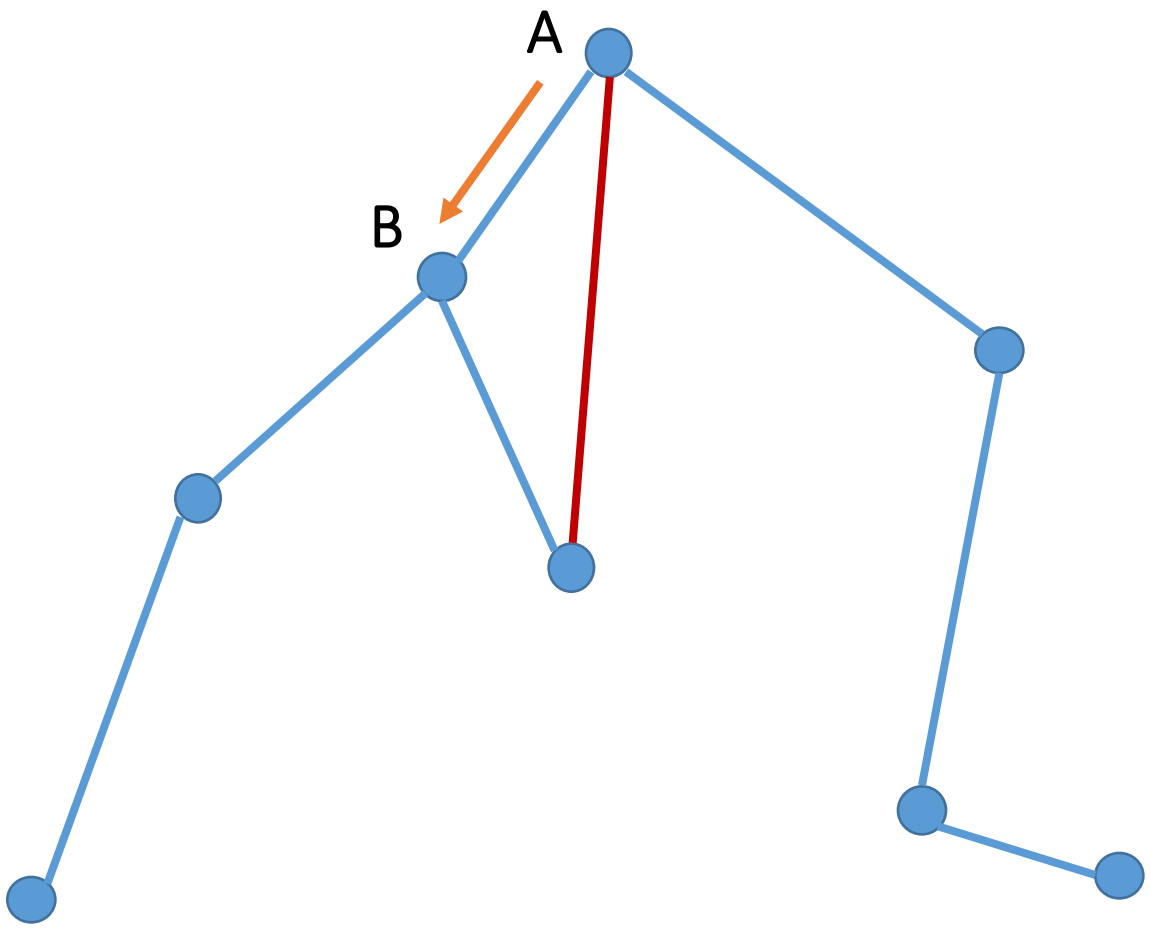
Common SQL Query Paradigms

Nesting + EXISTS / ANY / ALL

```
SELECT sid, p3.precip
FROM (
      SELECT sid, precip
      FROM precipitation AS p1
      WHERE precip > 0 AND NOT EXISTS (
            SELECT p2.precip
            FROM precipitation AS p2
            WHERE p2.sid = p1.sid
              AND p2.precip > 0
              AND p2.precip < p1.precip)) AS p3
WHERE NOT EXISTS (
      SELECT p4.precip
      FROM precipitation AS p4
      WHERE p4.precip – 400 > p3.precip);
```

More complex, but again just think about **order!**

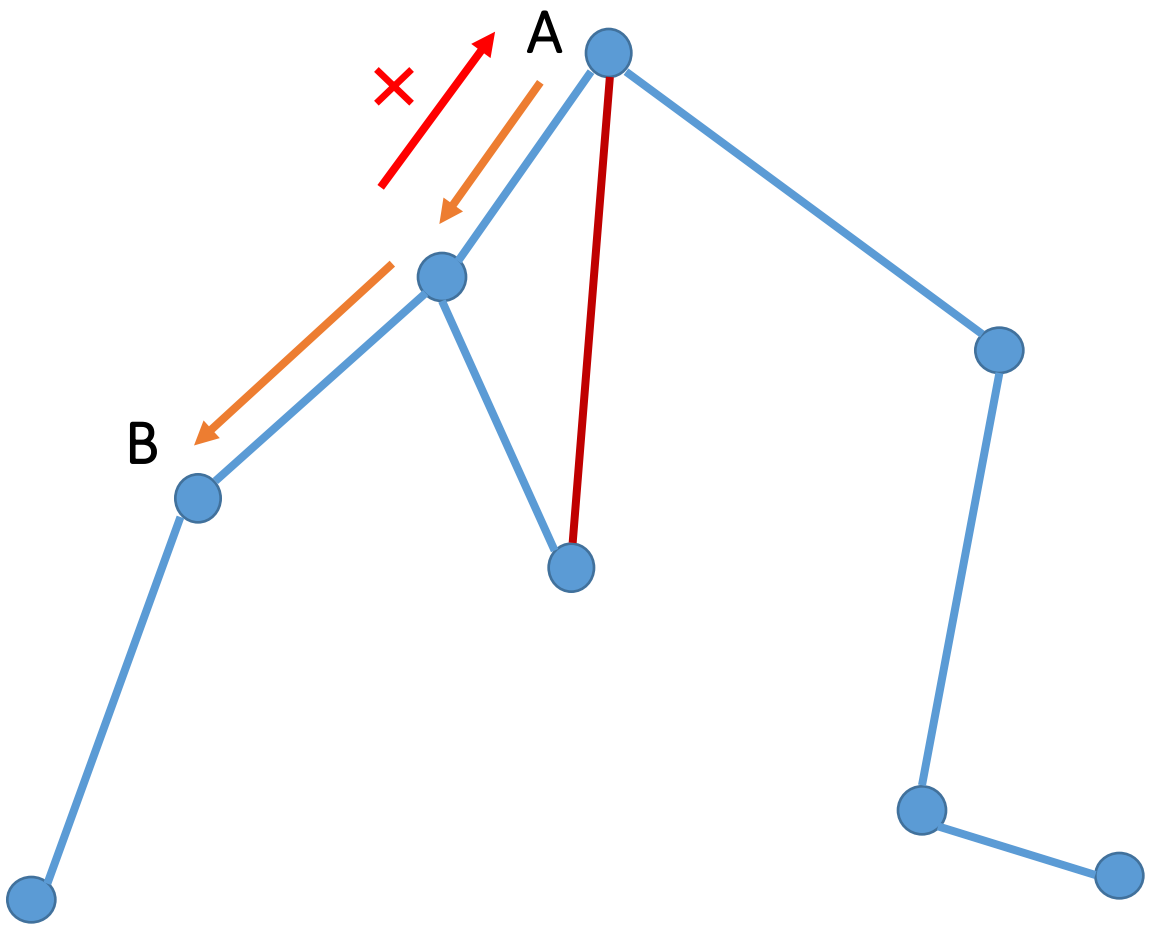
Graph traversal & recursion



For fixed-length paths

```
SELECT A, B, d
FROM edges
```

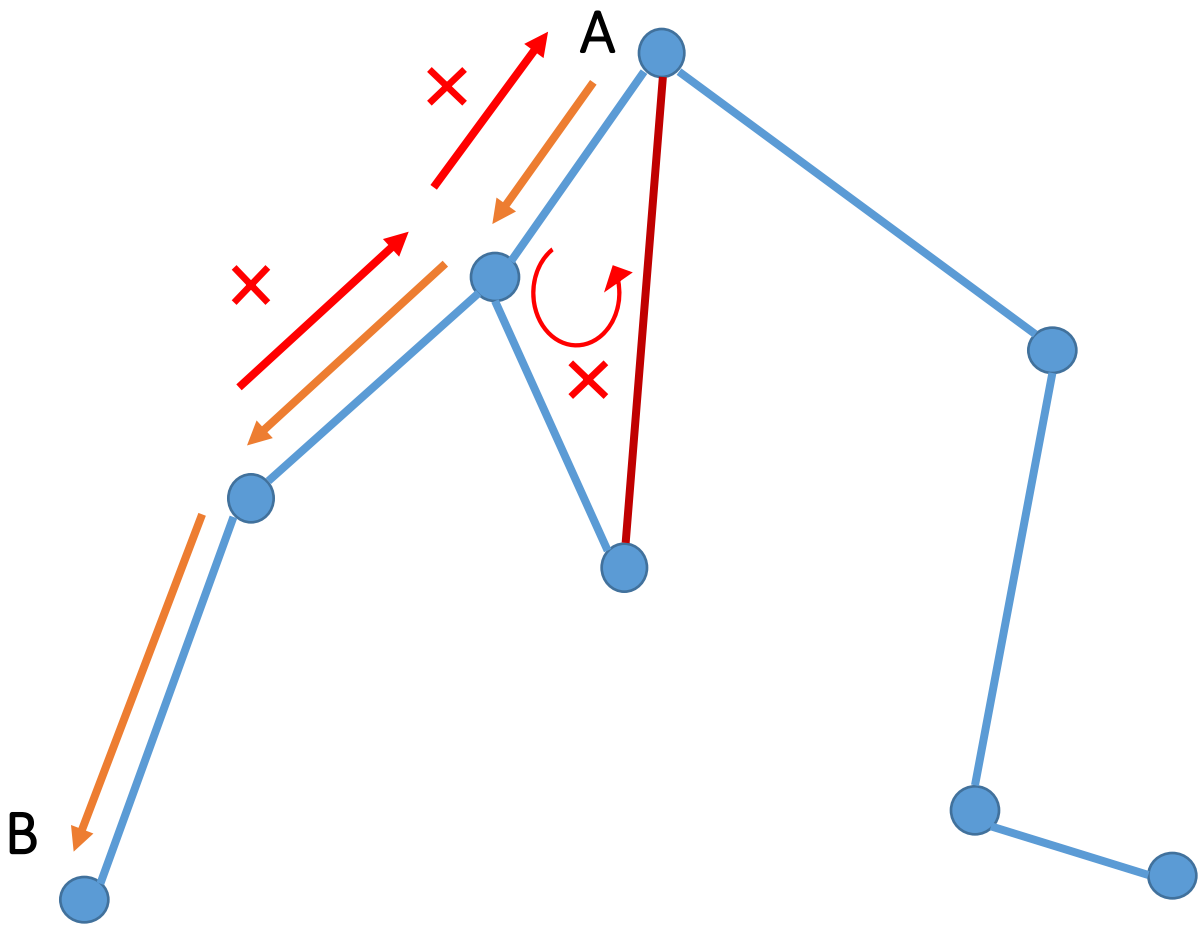
Graph traversal & recursion



For fixed-length paths

```
SELECT A, B, d
FROM edges
UNION
SELECT e1.A, e2.B,
        e1.d + e2.d AS d
FROM edges e1, edges e2
WHERE e1.B = e2.A
  AND e2.B <> e1.A
```

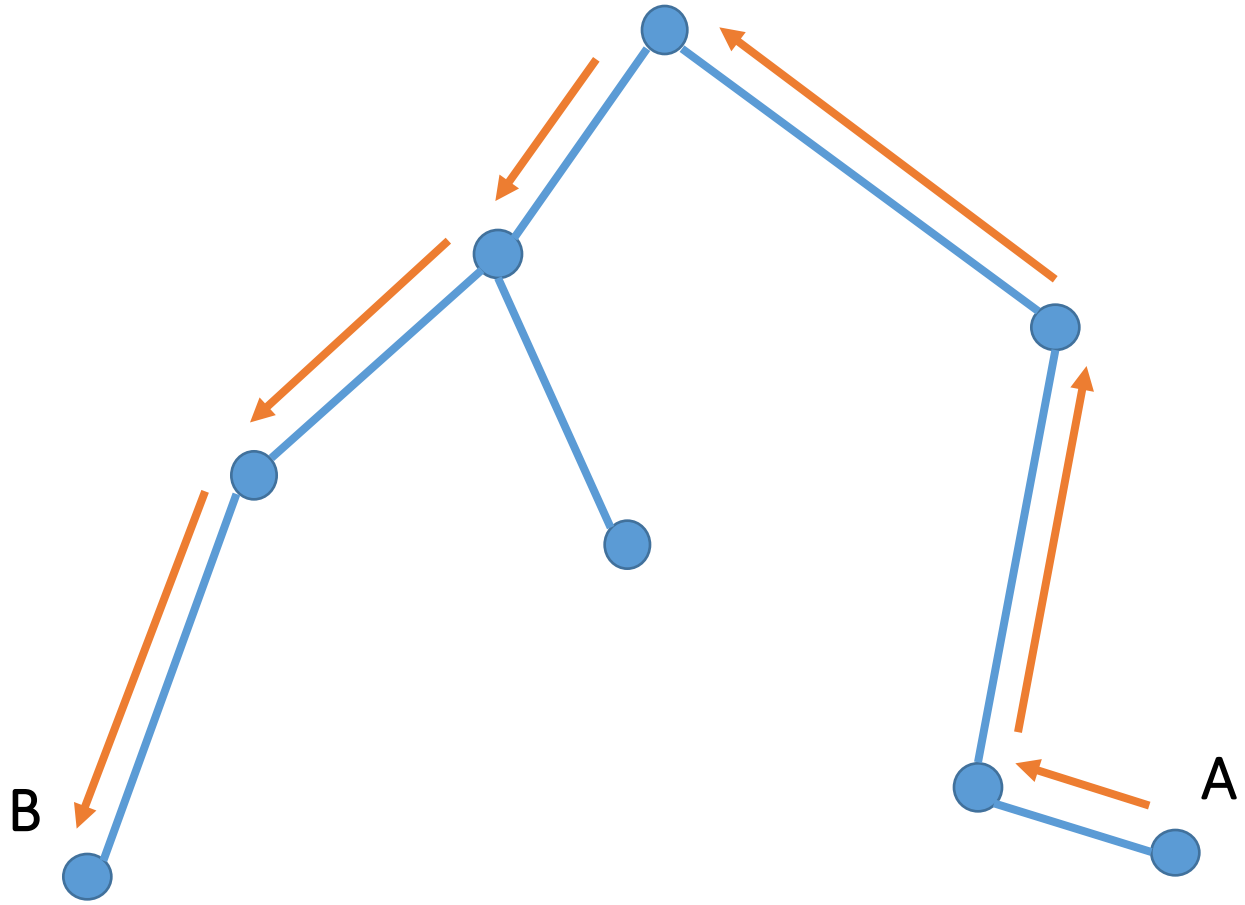
Graph traversal & recursion



For fixed-length paths

```
SELECT A, B, d
FROM edges
UNION
SELECT e1.A, e2.B,
        e1.d + e2.d AS d
FROM edges e1, edges e2
WHERE e1.B = e2.A
  AND e2.B <> e1.A
UNION
SELECT e1.A, e3.B,
        e1.d + e2.d + e3.d AS d
FROM edges e1, edges e2, edges e3
WHERE e1.B = e2.A
  AND e2.B = e3.A
  AND e2.B <> e1.A
  AND e3.B <> e2.A
  AMD e3.B <> e1.A
```

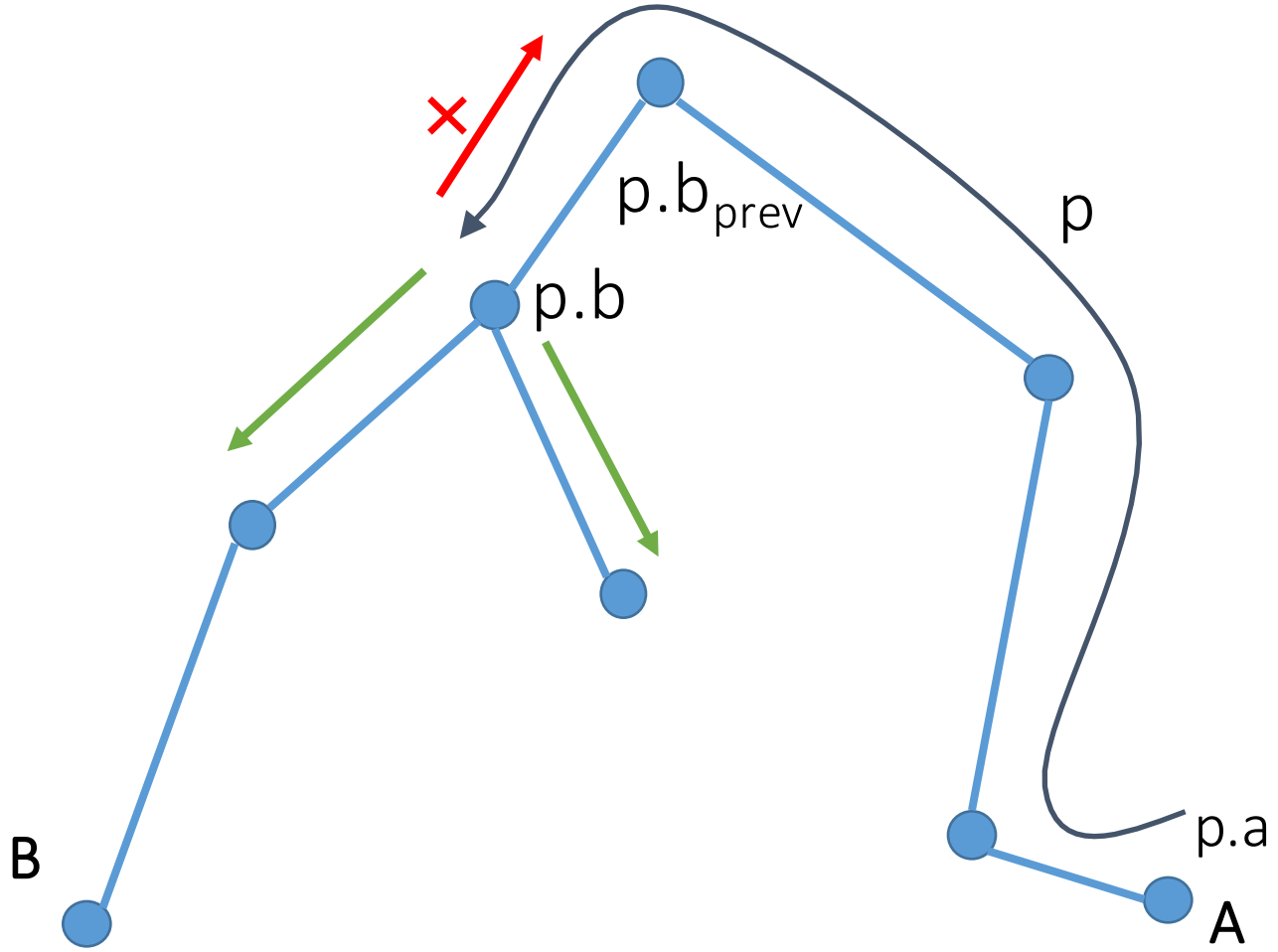
Graph traversal & recursion



For variable-length paths on trees

```
WITH RECURSIVE
paths(a, b, b_prev, d) AS (
       SELECT A, B, A
       FROM edges
       UNION
       SELECT p.a, e.B, e.A,
              p.d + e.d
       FROM paths p, edges e
       WHERE p.b = e.A
         AND s.B <> p.b_prev)
SELECT a, b, MAX(d)
FROM paths;
```

Graph traversal & recursion



For variable-length paths on trees

```
WITH RECURSIVE
paths(a, b, b_prev, d) AS (
       SELECT A, B, A
       FROM edges
       UNION
       SELECT p.a, e.B, e.A,
              p.d + e.d
       FROM paths p, edges e
       WHERE p.b = e.A
         AND e.B <> p.b_prev)
SELECT a, b, MAX(d)
FROM paths;
```

Today's Lecture

1. Boyce-Codd Normal Form
   - ACTIVITY
2. Decompositions & 3NF
   - ACTIVITY
3. MVDs
   - ACTIVITY

1. Boyce-Codd Normal Form

What you will learn about in this section

1. Conceptual Design
2. Boyce-Codd Normal Form
3. The BCNF Decomposition Algorithm
4. ACTIVITY

Conceptual Design

Back to Conceptual Design

Now that we know how to find FDs, it's a straight-forward process:

1. Search for "bad" FDs
2. If there are any, then *keep decomposing the table into sub-tables* until no more bad FDs
3. When done, the database schema is *normalized*

Recall: there are several normal forms…

Boyce-Codd Normal Form (BCNF)

- Main idea is that we define "good" and "bad" FDs as follows:
  - X → A is a *"good FD" if X is a (super)key*
    - In other words, if A is the set of all attributes
  - X → A is a *"bad FD"* otherwise
- We will try to eliminate the "bad" FDs!

Boyce-Codd Normal Form (BCNF)

- Why does this definition of "good" and "bad" FDs make sense?

- If X is *not* a (super)key, it functionally determines *some* of the attributes

  - Recall: this means there is redundancy
  - And redundancy like this can lead to data anomalies!

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

Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation R is **in BCNF** if:

if $\{A_1, ..., A_n\} \rightarrow B$ is a *non-trivial* FD in R

then $\{A_1, ..., A_n\}$ **is a superkey** for R

Equivalently: ∀ sets of attributes X, either ($X^+ = X$) or (X^+ = all attributes)

In other words: there are no "bad" FDs

Example

Name	SSN	PhoneNumber	City
Fred	123-45-6789	206-555-1234	Seattle
Fred	123-45-6789	206-555-6543	Seattle
Joe	987-65-4321	908-555-2121	Westfield
Joe	987-65-4321	908-555-1234	Westfield

{SSN} → {Name,City}

This FD is *bad* because it is **not** a superkey

⟹ **Not** in BCNF

What is the key?
{SSN, PhoneNumber}

Example

Name	SSN	City
Fred	123-45-6789	Seattle
Joe	987-65-4321	Madison

SSN	PhoneNumber
123-45-6789	206-555-1234
123-45-6789	206-555-6543
987-65-4321	908-555-2121
987-65-4321	908-555-1234

Now in BCNF!

{SSN} → {Name,City}

This FD is now *good* because it is the key

Let's check anomalies:

- Redundancy ?
- Update ?
- Delete ?

BCNF Decomposition Algorithm

BCNFDecomp(R):

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find *a set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

Find a set of attributes X which has non-trivial "bad" FDs, i.e. is not a superkey, using closures

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) **then Return** R

If no "bad" FDs found, in BCNF!

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) **then Return** R

let $Y = X^+ - X$, $Z = (X^+)^C$

Let Y be the attributes that ***X functionally determines*** (+ that are not in X)

And let Z be **the other attributes that it *doesn't***

BCNF Decomposition Algorithm

BCNFDecomp(R):

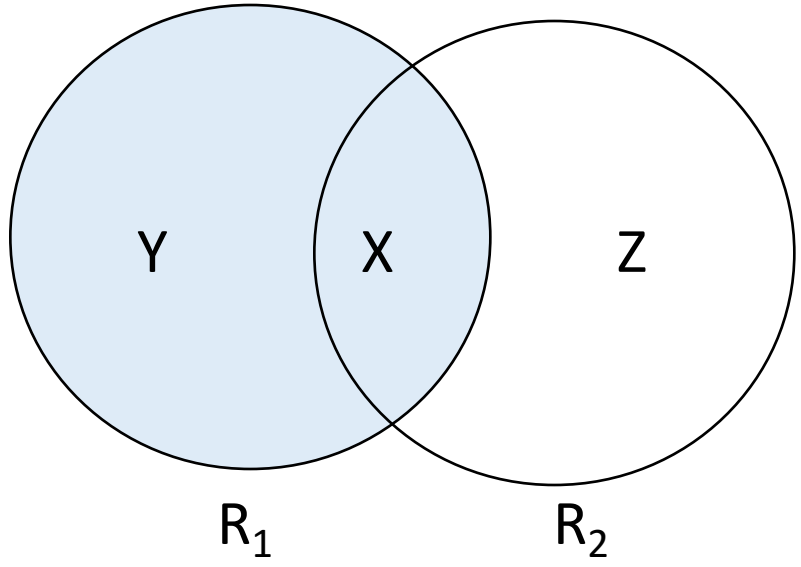
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) **then Return** R

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

Split into one relation (table) with X plus the attributes that X determines (Y)…



BCNF Decomposition Algorithm

BCNFDecomp(R):

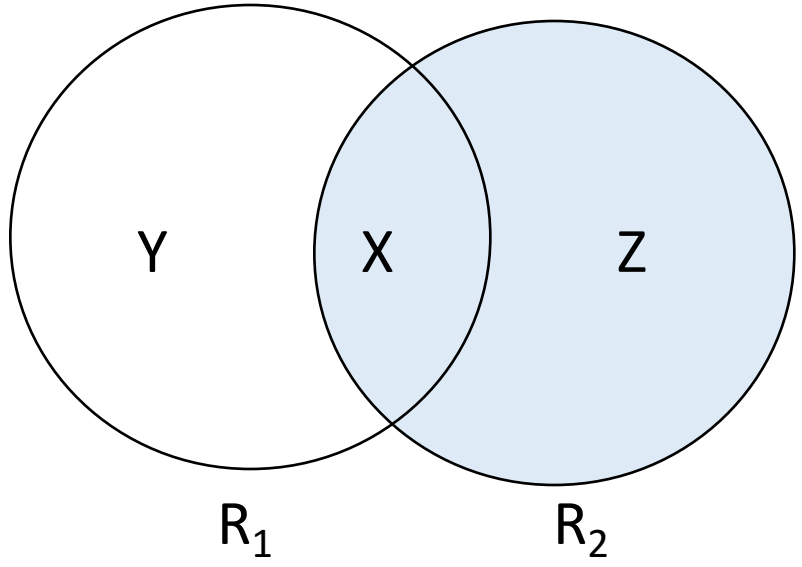
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) **then Return** R

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

And one relation with X plus the attributes it *does not* determine (Z)



BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) **then Return** R

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

Proceed recursively until no more "bad" FDs!

Example

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) **then Return** R

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

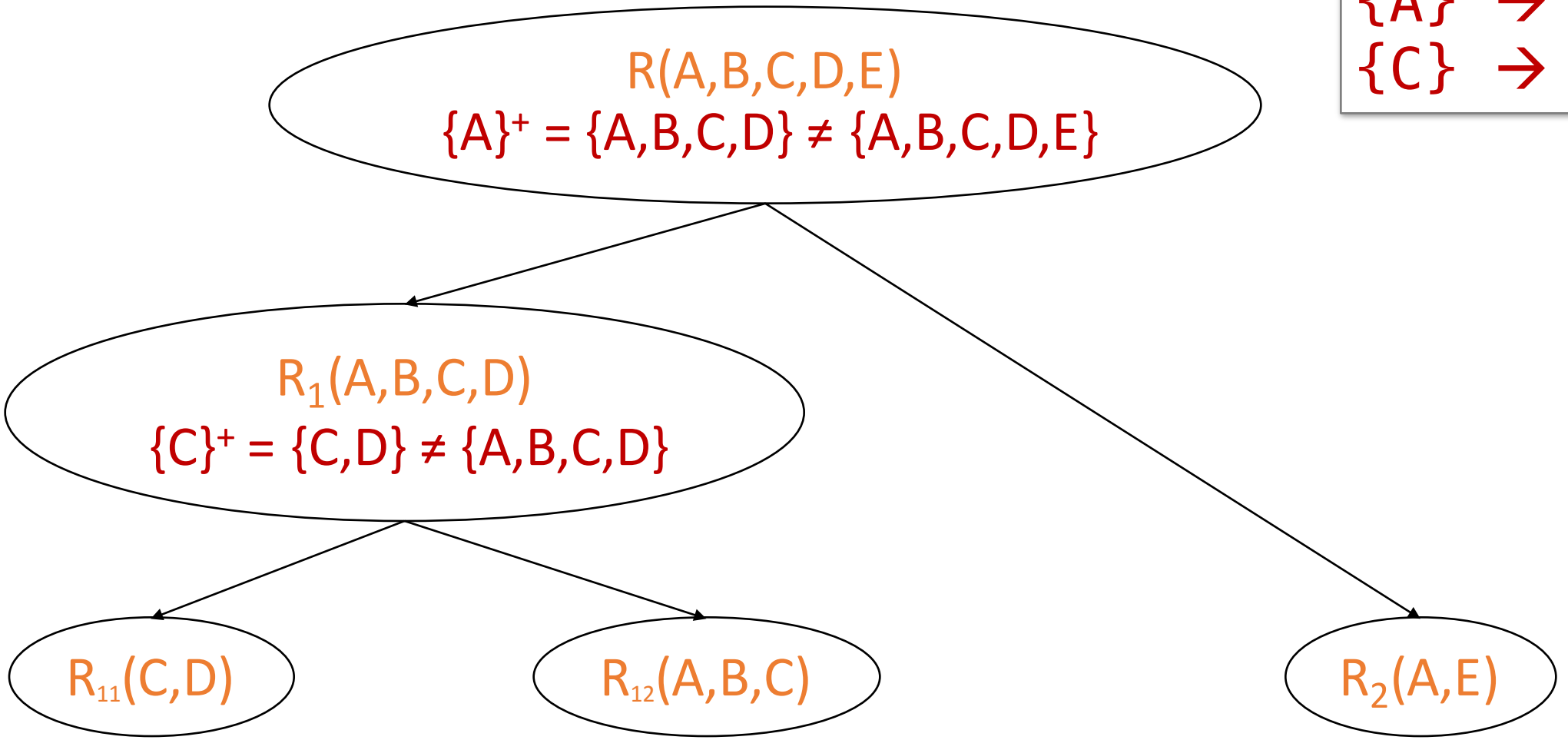
R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}

Example

R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}



Activity-7-1.ipynb

2. Decompositions

What you will learn about in this section

1. Lossy & Lossless Decompositions

2. ACTIVITY

Recap: Decompose to remove redundancies

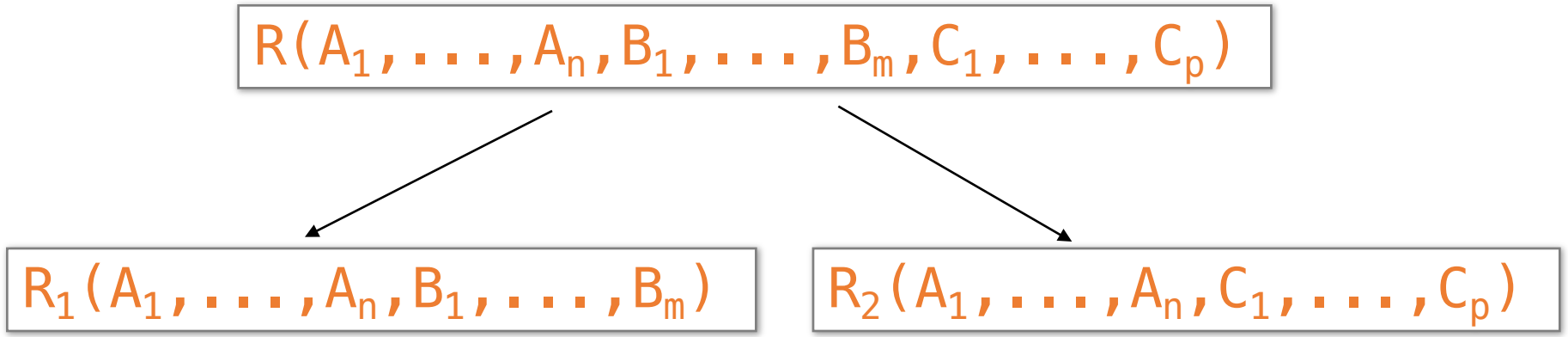
1. We saw that **redundancies** in the data ("bad FDs") can lead to data anomalies

2. We developed mechanisms to **detect and remove redundancies by decomposing tables into BCNF**
    1. BCNF decomposition is *standard practice*- very powerful & widely used!

3. However, sometimes decompositions can lead to **more subtle unwanted effects…**

When does this happen?

Decompositions in General



R_1 = the *projection* of R on A_1, ..., A_n, B_1, ..., B_m

R_2 = the *projection* of R on A_1, ..., A_n, C_1, ..., C_p

Theory of Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Price
Gizmo	19.99
OneClick	24.99
~~Gizmo~~	~~19.99~~

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Sometimes a decomposition is "correct"

I.e. it is a **Lossless decomposition**

Lossy Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

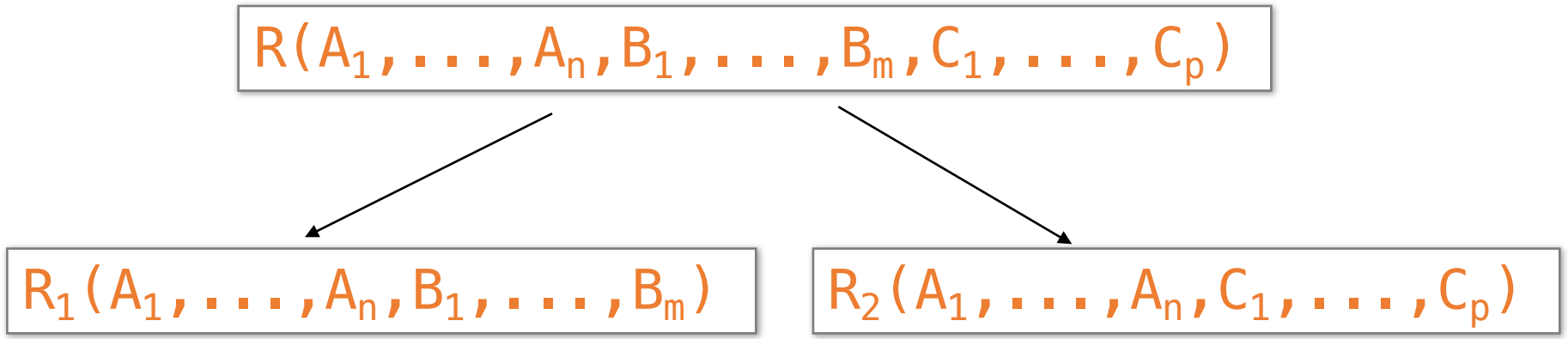
Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Price	Category
19.99	Gadget
24.99	Camera
19.99	Camera

However sometimes it isn't

What's wrong here?

Lossless Decompositions

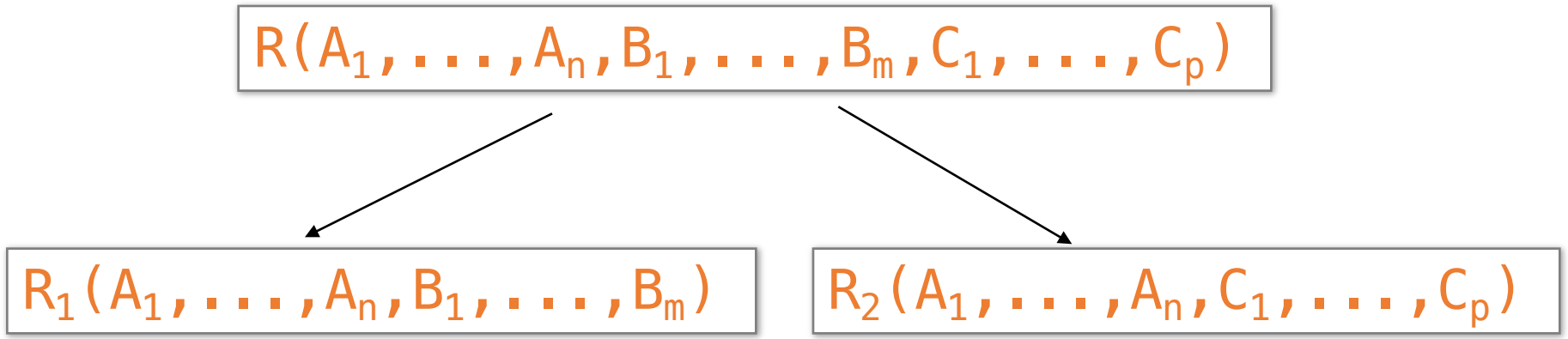


What (set) relationship holds between R1 Join R2 and R if lossless?

Hint: Which tuples of R will be present?

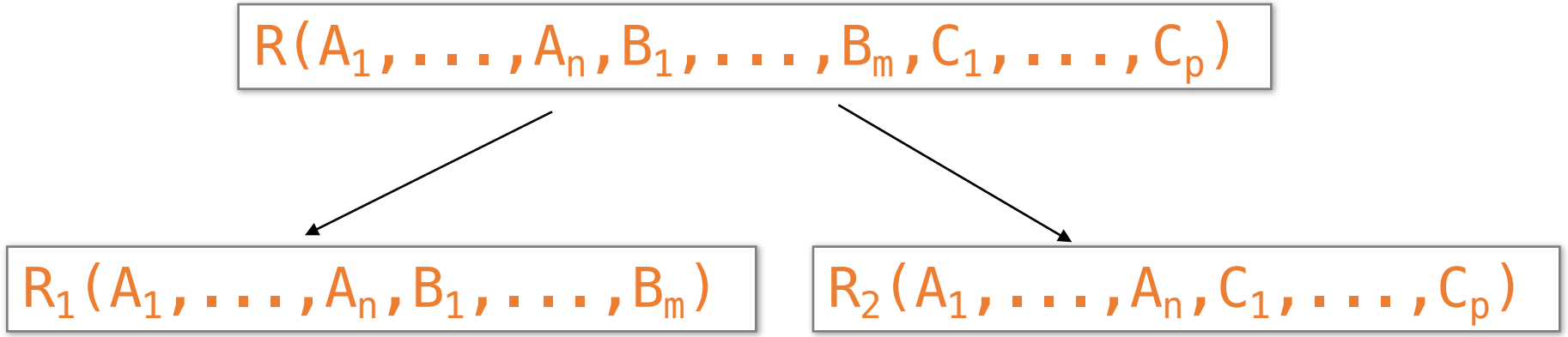
It's lossless if we have equality!

Lossless Decompositions



A decomposition R to (R1, R2) is **lossless** if R = R1 Join R2

Lossless Decompositions



If $\{A_1, ..., A_n\} \rightarrow \{B_1, ..., B_m\}$
Then the decomposition is lossless

Note: don't need
$\{A_1, ..., A_n\} \rightarrow \{C_1, ..., C_p\}$

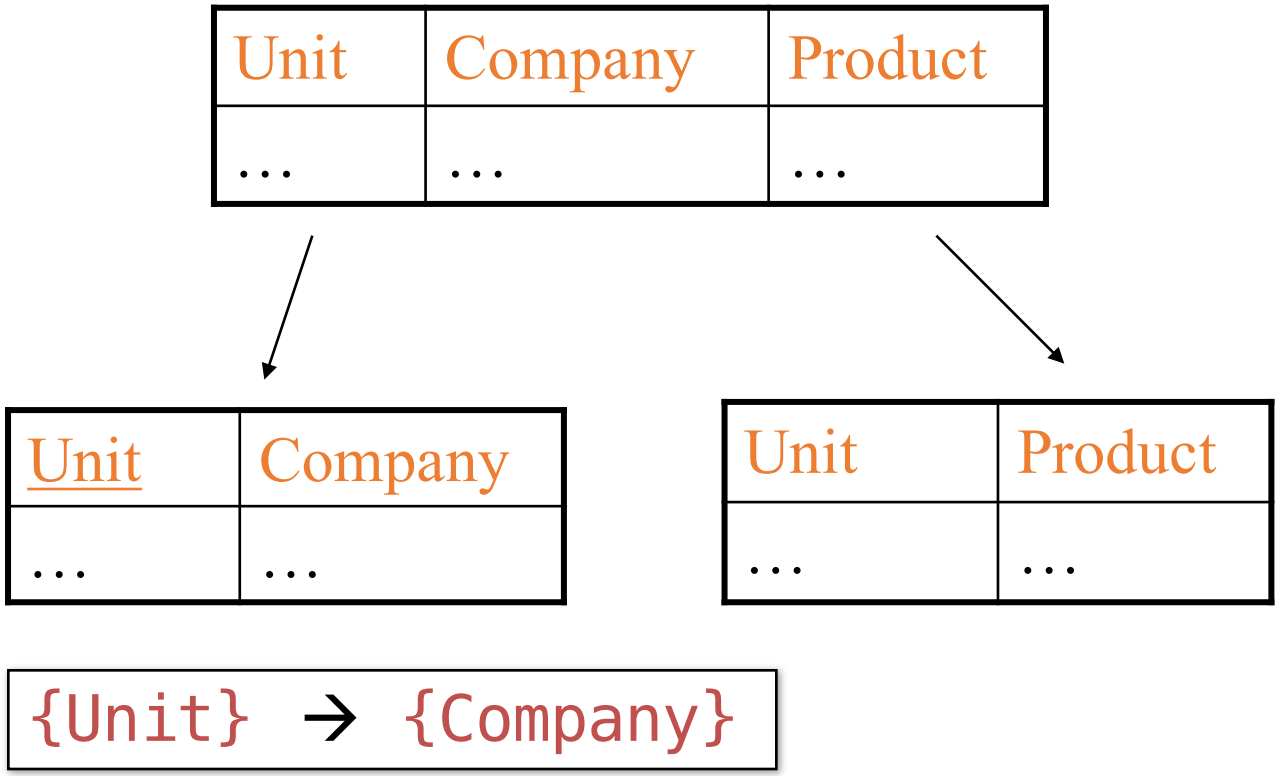
BCNF decomposition is always lossless. Why?

A problem with BCNF

Problem: To enforce a FD, must reconstruct original relation—*on each insert!*

Note: This is historically inaccurate, but it makes it easier to explain

A Problem with BCNF



{Unit} → {Company}

{Unit} → {Company}
{Company,Product} → {Unit}

We do a BCNF decomposition on a "bad" FD:

$\{Unit\}^+ = \{Unit, Company\}$

We lose the FD {Company,Product} → {Unit}!!

So Why is that a Problem?

Unit	Company
Galaga99	UW
Bingo	UW

Unit	Product
Galaga99	Databases
Bingo	Databases

No problem so far. All *local* FD's are satisfied.

{Unit} → {Company}

Unit	Company	Product
Galaga99	UW	Databases
Bingo	UW	Databases

Let's put all the data back into a single table again:

Violates the FD {Company,Product} → {Unit}!!

The Problem

- We started with a table R and FDs F
- We decomposed R into BCNF tables R_1, R_2, … with their own FDs F_1, F_2, …
- We insert some tuples into each of the relations—which satisfy their local FDs but when reconstruct it violates some FD **across** tables!

Practical Problem: To enforce FD, must reconstruct R—*on each insert!*

Possible Solutions

- Various ways to handle so that decompositions are all lossless / no FDs lost
  - For example 3NF- stop short of full BCNF decompositions. See Bonus Activity!
- Usually a tradeoff between redundancy / data anomalies and FD preservation…

BCNF still most common- with additional steps to keep track of lost FDs…

Activity-7-2.ipynb

3. MVDs

What you will learn about in this section

1. MVDs
2. ACTIVITY

Multiple Value Dependencies (MVDs)

Course	Staff	Student
CS949	Amy	Bob
CS145	Chris	Deb
CS145	Chris	Eli
CS145	Firas	Deb
CS145	Firas	Eli

MVD Ex: For each fixed course (e.g. CS145),
Every staff member in that course **and** every student in that course occur in a tuple in that table.

Write: Course ↠ Staff or Course ↠ Student

Formal Definition of MVD

Course ↠ Staff

	Course	Staff	Student
	CS949	Amy	Bob
t_1	CS145	Chris	Deb
t_3	CS145	Chris	Eli
	CS145	Firas	Deb
t_2	CS145	Firas	Eli

We write **A↠ B** if for any tuples t_1,t_2 s.t.
$t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[C] = t_2[C]$

where $C = (A \cup B)^C$, i.e. the attributes of R not in A or B.

Formal Definition of MVD

Course ↠ Staff

	Course	Staff	Student
	CS949	Amy	Bob
t_1	CS145	Chris	Deb
	CS145	Chris	Eli
	CS145	Firas	Deb
t_2	CS145	Firas	Eli

A

We write **A↠ B** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$

Formal Definition of MVD

Course ↠ Staff

	Course	Staff	Student
	CS949	Amy	Bob
t_1	CS145	Chris	Deb
t_3	CS145	Chris	Eli
	CS145	Firas	Deb
t_2	CS145	Firas	Eli

A

We write **A↠ B** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$

Formal Definition of MVD

Course ↠ Staff

	Course	Staff	Student
	CS949	Amy	Bob
t_1	CS145	Chris	Deb
t_3	CS145	Chris	Eli
	CS145	Firas	Deb
t_2	CS145	Firas	Eli

A B

We write **A↠ B** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$

Formal Definition of MVD

Course ↠ Staff

	Course	Staff	Student
	CS949	Amy	Bob
t_1	CS145	Chris	Deb
t_3	CS145	Chris	Eli
	CS145	Firas	Deb
t_2	CS145	Firas	Eli

A B C

We write **A↠ B** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[C] = t_2[C]$

where $C = (A \cup B)^C$, i.e. the attributes of R not in A or B

Does Course ↠ Staff hold now?

Course	Staff	Student
CS949	Amy	Bob
CS145	Chris	Deb
CS145	Chris	Eli
CS145	Firas	Eli

Connection to FDs

If A → B does A↠ B ?

Hint: sort of like multiplying by one…

Comments on MVDs

- MVDs have “rules” too!
  - **Experts**: Axiomatizable
- 4th Normal Form is “non-trivial MVD”
- *For AI nerds*: MVD is conditional independence in graphical models!

Activity-7-3.ipynb

Summary

- Constraints allow one to reason about **redundancy** in the data
- Normal forms describe how to **remove** this redundancy by **decomposing** relations
  - Elegant—by representing data appropriately certain errors are essentially impossible
  - For FDs, BCNF is the normal form.
- A tradeoff for insert performance: 3NF

Feedback!

- https://ctleval.stanford.edu/auth/evaluation.php?id=9451
- We're trying a new format (the notebooks, the 20+activity lecture format).
- Constructive feedback (of any aspect) is *really appreciated*!
  - We've tried to be responsive (when we can get out of our own way!)