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

• 1\textsuperscript{st} Normal Form (1NF) = All tables are flat

• 2\textsuperscript{nd} Normal Form = disused

• **Boyce-Codd Normal Form (BCNF)**

• 3\textsuperscript{rd} Normal Form (3NF)

• 4\textsuperscript{th} and 5\textsuperscript{th} Normal Forms = see text books

Our focus in this lecture + next one
1\textsuperscript{st} Normal Form (1NF)

<table>
<thead>
<tr>
<th>Student</th>
<th>Courses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>{CS145,CS229}</td>
</tr>
<tr>
<td>Joe</td>
<td>{CS145,CS106}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\textit{Violates 1NF.}

<table>
<thead>
<tr>
<th>Student</th>
<th>Courses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
</tr>
<tr>
<td>Mary</td>
<td>CS229</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
</tr>
<tr>
<td>Joe</td>
<td>CS106</td>
</tr>
</tbody>
</table>

\textit{In 1\textsuperscript{st} NF}

1\text{NF Constraint:} Types must be atomic!
Data Anomalies & Constraints
Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes *anomalies*:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

If every course is in only one room, contains *redundant* information!
## Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes anomalies:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
<td>C12</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

Similarly, we can’t reserve a room without students = an **insert** anomaly
Constraints Prevent (some) Anomalies in the Data

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>CS229</td>
<td>C12</td>
</tr>
</tbody>
</table>

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 $A \rightarrow B$ or say $A$ functionally determines $B$ if, for any tuples $t_1$ and $t_2$:

$$t_1[A] = t_2[A] \implies t_1[B] = t_2[B]$$

and we call $A \rightarrow B$ 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 $R$, 
Defn (again):
Given attribute sets $A=\{A_1,\ldots,A_m\}$ and $B = \{B_1,\ldots,B_n\}$ in $R$,

The *functional dependency* $A \rightarrow B$ on $R$ holds if for *any* $t_i, t_j$ in $R$:
Defn (again):
Given attribute sets $A = \{A_1, \ldots, A_m\}$ and $B = \{B_1, \ldots, B_n\}$ in $R$,

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

$t_i[A_1] = t_j[A_1] \text{ AND } t_i[A_2] = t_j[A_2] \text{ AND } \ldots \text{ AND } t_i[A_m] = t_j[A_m]$
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 $A \rightarrow\!\!\!\!\!\!\!\rightarrow B$ on $R$ holds if for any $t_i, t_j$ in $R$:

if $t_i[A_1] = t_j[A_1] \text{ AND } t_i[A_2] = t_j[A_2] \text{ AND } 
\ldots \text{ AND } t_i[A_m] = t_j[A_m]

then $t_i[B_1] = t_j[B_1] \text{ AND } t_i[B_2] = t_j[B_2] \text{ AND } 
\ldots \text{ AND } t_i[B_n] = t_j[B_n]$
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*.

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

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

Note: The FD \{Course\} \(\rightarrow\) \{Room\} holds on this *instance*
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

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E3542</td>
<td>Mike</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>Lawyer</td>
</tr>
</tbody>
</table>
## More Examples

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E3542</td>
<td>Mike</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>Lawyer</td>
</tr>
</tbody>
</table>

\{Position\} \rightarrow \{Phone\}
More Examples

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E3542</td>
<td>Mike</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>Lawyer</td>
</tr>
</tbody>
</table>

but not {Phone} → {Position}
ACTIVITY

Find at least three FDs which hold on this instance:

\[
\left\{ \begin{array}{c}
A \\
B \\
C \\
D \\
E \\
\end{array} \right. \rightarrow \left\{ \begin{array}{c}
1 \\
2 \\
4 \\
3 \\
6 \\
\end{array} \right. \\
\left\{ \begin{array}{c}
1 \\
2 \\
2 \\
5 \\
1 \\
8 \\
\end{array} \right. \rightarrow \left\{ \begin{array}{c}
4 \\
5 \\
4 \\
7 \\
\end{array} \right. \\
\left\{ \begin{array}{c}
1 \\
4 \\
2 \\
3 \\
6 \\
\end{array} \right. \rightarrow \left\{ \begin{array}{c}
4 \\
5 \\
2 \\
1 \\
8 \\
\end{array} \right. \\
\right. 
\]
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:

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E3542</td>
<td>Mike</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>Lawyer</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E3542</td>
<td>Mike</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>Lawyer</td>
</tr>
</tbody>
</table>

Intuitively:

EmpID -> Name, Phone, Position is “good FD”

But Position -> Phone is a “bad FD”

- **Redundancy!**
- **Possibility of data anomalies**
“Good” vs. “Bad” FDs

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Products</th>
<th>Name</th>
<th>Color</th>
<th>Category</th>
<th>Dep</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gizmo</td>
<td>Green</td>
<td>Gadget</td>
<td>Toys</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Widget</td>
<td>Black</td>
<td>Gadget</td>
<td>Toys</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Gizmo</td>
<td>Green</td>
<td>Whatsit</td>
<td>Garden</td>
<td>99</td>
</tr>
</tbody>
</table>

Given the provided FDs, we can see that \{Name, Category\} \rightarrow \{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...

\[ A_1, \ldots, A_m \rightarrow B_i \text{ for } i=1,\ldots,n \]
1. Split/Combine

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

... is equivalent to ...

\( A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n \)
Reduction/Trivial

\[
A_1, \ldots, A_m \rightarrow A_j \text{ for any } j=1,\ldots,m
\]
3. Transitive Closure

<table>
<thead>
<tr>
<th></th>
<th>$A_1$</th>
<th>...</th>
<th>$A_m$</th>
<th>$B_1$</th>
<th>...</th>
<th>$B_n$</th>
<th>$C_1$</th>
<th>...</th>
<th>$C_k$</th>
</tr>
</thead>
</table>

$A_1, ..., A_m \rightarrow B_1, ..., B_n$ and $B_1, ..., B_n \rightarrow C_1, ..., C_k$
3. Transitive Closure

\[
\begin{array}{cccccccc}
A_1 & \ldots & A_m & B_1 & \ldots & B_n & C_1 & \ldots & C_k \\
\hline
\vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\
\end{array}
\]

\[A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n \text{ and } B_1, \ldots, B_n \rightarrow C_1, \ldots, C_k\]

implies
\[A_1, \ldots, A_m \rightarrow C_1, \ldots, C_k\]
Finding Functional Dependencies

Example:

<table>
<thead>
<tr>
<th>Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Gizmo</td>
</tr>
<tr>
<td>Widget</td>
</tr>
<tr>
<td>Gizmo</td>
</tr>
</tbody>
</table>

Provided FDs:
1. {Name} \( \rightarrow \) {Color}
2. {Category} \( \rightarrow \) {Department}
3. {Color, Category} \( \rightarrow \) {Price}

Which / how many other FDs hold?
Finding Functional Dependencies

Example:

**Inferred FDs:**

<table>
<thead>
<tr>
<th>Inferred FD</th>
<th>Rule used</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. {Name, Category} -&gt; {Name}</td>
<td>?</td>
</tr>
<tr>
<td>5. {Name, Category} -&gt; {Color}</td>
<td>?</td>
</tr>
<tr>
<td>6. {Name, Category} -&gt; {Category}</td>
<td>?</td>
</tr>
<tr>
<td>7. {Name, Category} -&gt; {Color, Category}</td>
<td>?</td>
</tr>
<tr>
<td>8. {Name, Category} -&gt; {Price}</td>
<td>?</td>
</tr>
</tbody>
</table>

**Provided FDs:**

1. \{Name\} -> \{Color\}
2. \{Category\} -> \{Dept.\}
3. \{Color, Category\} -> \{Price\}

Which / how many other FDs hold?
Finding Functional Dependencies

Example:

Inferred FDs:

<table>
<thead>
<tr>
<th>Inferred FD</th>
<th>Rule used</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. {Name, Category} -&gt; {Name}</td>
<td>Trivial</td>
</tr>
<tr>
<td>5. {Name, Category} -&gt; {Color}</td>
<td>Transitive (4 -&gt; 1)</td>
</tr>
<tr>
<td>6. {Name, Category} -&gt; {Category}</td>
<td>Trivial</td>
</tr>
<tr>
<td>7. {Name, Category} -&gt; {Color, Category}</td>
<td>Split/combine (5 + 6)</td>
</tr>
<tr>
<td>8. {Name, Category} -&gt; {Price}</td>
<td>Transitive (7 -&gt; 3)</td>
</tr>
</tbody>
</table>

Provided FDs:

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

Can we find an algorithmic way to do this?
Closures
Closure of a set of Attributes

**Given** a set of attributes \( A_1, ..., A_n \) and a set of FDs \( F \):

Then the **closure** \( \{ A_1, ..., A_n \}^+ \) is the set of attributes \( B \) s.t. \( \{ A_1, ..., A_n \} \rightarrow B \)

**Example:**

\[
F = \begin{align*}
\{\text{name}\} & \rightarrow \{\text{color}\} \\
\{\text{category}\} & \rightarrow \{\text{department}\} \\
\{\text{color, category}\} & \rightarrow \{\text{price}\}
\end{align*}
\]

**Example Closures:**

\[
\begin{align*}
\{\text{name}\}^+ &= \{\text{name, color}\} \\
\{\text{name, category}\}^+ &= \{\text{name, category, color, dept, price}\} \\
\{\text{color}\}^+ &= \{\text{color}\}
\end{align*}
\]
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 $X^+$

$F = \{name\} \rightarrow \{color\}$

$\{category\} \rightarrow \{dept\}$

$\{color, category\} \rightarrow \{price\}$
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 $X^+$

$F = \{\text{name} \rightarrow \text{color}\}$

$\{\text{category} \rightarrow \text{dept}\}$

$\{\text{color, category} \rightarrow \text{price}\}$

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

$\{\text{name, category}\}^+ = \{\text{name, category, color}\}$
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 $X^+$

$F =$

$\{name\} \rightarrow \{color\}$

$\{category\} \rightarrow \{dept\}$

$\{color, category\} \rightarrow \{price\}$

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

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

$\{name, category\}^+ = \{name, category, color, dept\}$
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 \(X^+\)

\[
F = \\
\{\text{name} \rightarrow \{\text{color}\}\}\\
\{\text{category} \rightarrow \{\text{dept}\}\}\\
\{\text{color, category} \rightarrow \{\text{price}\}\}
\]

\[
\{\text{name, category}\}^+ = \\
\{\text{name, category}\}
\]

\[
\{\text{name, category}\}^+ = \\
\{\text{name, category, color}\}
\]

\[
\{\text{name, category}\}^+ = \\
\{\text{name, category, color, dept}\}
\]

\[
\{\text{name, category}\}^+ = \\
\{\text{name, category, color, dept, price}\}
\]
Example

\[ R(A, B, C, D, E, F) \]

Compute \( \{A, B\}^+ = \{A, B, \} \)

Compute \( \{A, F\}^+ = \{A, F, \} \)
Example

\[ R(A, B, C, D, E, F) \]

\[
\begin{align*}
\{A, B\} & \rightarrow \{C\} \\
\{A, D\} & \rightarrow \{E\} \\
\{B\} & \rightarrow \{D\} \\
\{A, F\} & \rightarrow \{B\}
\end{align*}
\]

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

Compute \( \{A, F\}^+ = \{A, F, B\} \)
Example

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:

$X \rightarrow A_1, ..., X \rightarrow A_n$

implies

$X \rightarrow \{A_1, ..., A_n\}$
Using Closure to Infer ALL FDs

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

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

Example:

Given $F = \{
\{A, B\} \rightarrow C
\{A, D\} \rightarrow B
\{B\} \rightarrow D
\}$

No need to compute these—why?
Using Closure to Infer ALL FDs

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


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

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

Example:

Given $F =$

- $\{A,B\} \rightarrow C$
- $\{A,D\} \rightarrow B$
- $\{B\} \rightarrow D$
Using Closure to Infer ALL FDs

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


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

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

Example:

Given $F = \{\{A,B\} \rightarrow C, \{A,D\} \rightarrow B, \{B\} \rightarrow D\}$
Using Closure to Infer ALL FDs

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

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

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

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

Example:

Given $F = \{\{A,B\} \rightarrow C, \{A,D\} \rightarrow B, \{B\} \rightarrow D\}$

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

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

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

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

The FD $X \rightarrow Y$ is non-trivial
Superkeys and Keys
Keys and Superkeys

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

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

I.e. all attributes are *functionally determined* by a 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} $\rightarrow$ price
{category} $\rightarrow$ color

What is a key?
Example of Keys

Product(name, price, category, color)

{name, category} \rightarrow price
{category} \rightarrow color

\{name, category\}^+ = \{name, price, category, color\}
= the set of all attributes
\Rightarrow this is a superkey
\Rightarrow 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

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

\[
C = [[0]*p \text{ for } i \text{ in range}(n)]
\]

```python
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

```sql
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

SELECT station_id, COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip) AS m
     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
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

```sql
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

```sql
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

```sql
SELECT A, B, d
FROM edges
UNION
FROM edges e1, edges e2
WHERE e1.B = e2.A
    AND e2.B <> e1.A
```
For fixed-length paths

```sql
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
    AND 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:

  • $X \rightarrow A$ is a “good FD” if $X$ is a (super)key
    • In other words, if $A$ is the set of all attributes

  • $X \rightarrow A$ 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!

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E3542</td>
<td>Mike</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>Lawyer</td>
</tr>
</tbody>
</table>
Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation $R$ is in BCNF if:

- If $\{A_1, \ldots, A_n\} \rightarrow B$ is a non-trivial FD in $R$
- Then $\{A_1, \ldots, A_n\}$ is a superkey for $R$

Equivalently: $\forall$ sets of attributes $X$, either ($X^+ = X$) or ($X^+ = \text{all attributes}$)

In other words: there are no “bad” FDs
Example

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>PhoneNumber</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>206-555-1234</td>
<td>Seattle</td>
</tr>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>206-555-6543</td>
<td>Seattle</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>908-555-2121</td>
<td>Westfield</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>908-555-1234</td>
<td>Westfield</td>
</tr>
</tbody>
</table>

\{SSN\} \rightarrow \{Name, City\}

This FD is bad because it is not a superkey.

What is the key? \{SSN, PhoneNumber\}

\Rightarrow \textbf{Not} in BCNF
Example

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>Seattle</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>Madison</td>
</tr>
</tbody>
</table>

\[
\{\text{SSN}\} \rightarrow \{\text{Name}, \text{City}\}
\]

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

Let's check anomalies:
- Redundancy?
- Update?
- Delete?

Now in BCNF!
BCNF Decomposition Algorithm

BCNFDcomp(R):

\[
\text{Find } X \text{ s.t. } X^+ \neq X \text{ and } X^+ \neq \text{all attributes}
\] if (not found) then Return R
let Y = X^+ - X, Z = (X^+ \cap C)

decompose R into R1(X \cup Y) and R2(X \cup Z)

Return BCNFDcomp(R1), BCNFDcomp(R2)
BCNF Decomposition Algorithm

BCNFDcomp(R):

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

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

BCNFDcomp(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

BCNFDcomp(R):
  Find a set of attributes X s.t.: X⁺ ≠ X and X⁺ ≠ [all attributes]

  if (not found) then Return R

  let Y = X⁺ - X, Z = (X⁺)ᶜ
  decompose R into R₁(X ∪ Y) and R₂(X ∪ 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** \( Y = X^+ \setminus X, \ Z = (X^+)^C \)

**decompose** \( R \) into \( R_1(X \cup Y) \) and \( R_2(X \cup Z) \)

And one relation with X plus the attributes it does not determine (Z)
BCNF Decomposition Algorithm

BCNFDcomp(R):
   Find a set of attributes X s.t.: X⁺ ≠ X and X⁺ ≠ [all attributes]

   if (not found) then Return R

   let Y = X⁺ - X, Z = (X⁺)ᶜ
   decompose R into R₁(X U Y) and R₂(X U Z)

   Return BCNFDcomp(R₁), BCNFDcomp(R₂)

Proceed recursively until no more “bad” FDs!
Example

BCNFD Decomp(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 $R_1(X \cup Y)$ and $R_2(X \cup Z)$

  Return BCNFD Decomp($R_1$), BCNFD Decomp($R_2$)

Example

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

$\{C\} \rightarrow \{D\}$
Example

\[ R(A,B,C,D,E) \]
\[ \{A\}^+ = \{A,B,C,D\} \neq \{A,B,C,D,E\} \]

\[ R_1(A,B,C,D) \]
\[ \{C\}^+ = \{C,D\} \neq \{A,B,C,D\} \]

\[ R_{11}(C,D) \]
\[ R_{12}(A,B,C) \]
\[ R_2(A,E) \]

\{A\} \rightarrow \{B,C\} \quad \{C\} \rightarrow \{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

\[ R(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_p) \]

\[ R_1(A_1, \ldots, A_n, B_1, \ldots, B_m) \]

\[ R_2(A_1, \ldots, A_n, C_1, \ldots, C_p) \]

\( R_1 = \text{the projection of } R \text{ on } A_1, \ldots, A_n, B_1, \ldots, B_m \)

\( R_2 = \text{the projection of } R \text{ on } A_1, \ldots, A_n, C_1, \ldots, C_p \)
### Theory of Decomposition

#### Example Table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>

#### Diagram:

- **Lecture 7 > Section 2 > Decompositions**

Sometimes a decomposition is "correct".

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>

However sometimes it isn’t

What’s wrong here?
Lossless Decompositions

What (set) relationship holds between $R_1$ Join $R_2$ and $R$ if lossless?

*Hint: Which tuples of $R$ will be present?*

It’s lossless if we have equality!
A decomposition $R$ to $(R_1, R_2)$ is **lossless** if $R = R_1 \text{ Join } R_2$
Lossless Decompositions

If \( \{A_1, ..., A_n\} \rightarrow \{B_1, ..., B_m\} \)

Then the decomposition is lossless

BCNF decomposition is always lossless. Why?

Note: don’t need \( \{A_1, ..., A_n\} \rightarrow \{C_1, ..., C_p\} \)
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

We do a BCNF decomposition on a “bad” FD:
\{Unit\}+ = \{Unit, Company\}

We lose the FD \{Company, Product\} \rightarrow \{Unit\}!!
So Why is that a Problem?

No problem so far. All local FD’s are satisfied.

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

Violates the FD \( \{\text{Company, Product}\} \rightarrow \{\text{Unit}\}!! \)
The Problem

• We started with a table R and FDs F

• We decomposed R into BCNF tables R₁, R₂, ... with their own FDs F₁, F₂, ...

• 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)

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
We write $A \leadsto B$ 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**

<table>
<thead>
<tr>
<th>Course</th>
<th>Staff</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS949</td>
<td>Amy</td>
<td>Bob</td>
</tr>
<tr>
<td>CS145</td>
<td>Chris</td>
<td>Deb</td>
</tr>
<tr>
<td>CS145</td>
<td>Chris</td>
<td>Eli</td>
</tr>
<tr>
<td>CS145</td>
<td>Firas</td>
<td>Deb</td>
</tr>
<tr>
<td>CS145</td>
<td>Firas</td>
<td>Eli</td>
</tr>
</tbody>
</table>

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

A
Formal Definition of MVD

We write $A \rsa B$ 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

<table>
<thead>
<tr>
<th>Course</th>
<th>Staff</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS949</td>
<td>Amy</td>
<td>Bob</td>
</tr>
<tr>
<td>CS145</td>
<td>Chris</td>
<td>Deb</td>
</tr>
<tr>
<td>CS145</td>
<td>Chris</td>
<td>Eli</td>
</tr>
<tr>
<td>CS145</td>
<td>Firas</td>
<td>Deb</td>
</tr>
<tr>
<td>CS145</td>
<td>Firas</td>
<td>Eli</td>
</tr>
</tbody>
</table>

We write $A \Rightarrow B$ 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

We write $A \rightarrow\!ightarrow B$ 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?

<table>
<thead>
<tr>
<th>Course</th>
<th>Staff</th>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS949</td>
<td>Amy</td>
<td>Bob</td>
</tr>
<tr>
<td>CS145</td>
<td>Chris</td>
<td>Deb</td>
</tr>
<tr>
<td>CS145</td>
<td>Chris</td>
<td>Eli</td>
</tr>
<tr>
<td>CS145</td>
<td>Firas</td>
<td>Eli</td>
</tr>
</tbody>
</table>
Connection to FDs

If $A \rightarrow B$ does $A \rightarrow B$?

Hint: sort of like multiplying by one...
Comments on MVDs

• MVDs have “rules” too!
  • **Experts**: Axiomatizable

• 4\textsuperscript{th} 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!


• 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!)