Lectures 5 & 6: Design Theory
Lecture 5:
Design Theory I
Today’s Lecture

1. Normal forms & functional dependencies
   • ACTIVITY: Finding FDs

2. Finding functional dependencies

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

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

Violates 1NF.

\begin{tabular}{|c|c|}
  \hline
  Student & Courses \\
  \hline
  Mary     & CS145 \\
  Mary     & CS229 \\
  Joe      & CS145 \\
  Joe      & CS106 \\
  \hline
\end{tabular}

In 1\textsuperscript{st} NF

1NF 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 $t_2$:

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

and we call $A \rightarrow B$ a functional dependency.

A->B means that “whenever two tuples agree on A then they agree on B.”
Defn (again):
Given attribute sets $A = \{A_1, ..., A_m\}$ and $B = \{B_1, ..., B_n\}$ in $R$, 

| $A_1$ | $\ldots$ | $A_m$ | $B_1$ | $\ldots$ | $B_n$ |
### 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 $R$:

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 $R$:

$t_i[A_1] = t_j[A_1]$ AND $t_i[A_2] = t_j[A_2]$ AND ... AND $t_i[A_m] = t_j[A_m]$
Defn (again):
Given attribute sets $A = \{A_1, ..., A_m\}$ and $B = \{B_1, ..., B_n\}$ in $R$,

The **functional dependency** $A \rightarrow B$ 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 (but not others) – can check whether there are violations
- Part of the schema, helps define a *valid* instance

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

<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>

Note: The FD `{Course} -> {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>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
</tr>
<tr>
<td>Joe</td>
</tr>
<tr>
<td>Sam</td>
</tr>
<tr>
<td>..</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS145</td>
</tr>
<tr>
<td>CS145</td>
</tr>
<tr>
<td>CS145</td>
</tr>
<tr>
<td>..</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>B01</td>
</tr>
<tr>
<td>B01</td>
</tr>
<tr>
<td>B01</td>
</tr>
<tr>
<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\} ➔ \{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\} $\rightarrow$ \{Position\}
ACTIVITY

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Find at least *three* FDs which are violated on this instance:

\[
\{ \} \rightarrow \{ \}
\]

\[
\{ \} \rightarrow \{ \}
\]

\[
\{ \} \rightarrow \{ \}
\]
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></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Color</td>
<td>Category</td>
<td>Dep</td>
<td>Price</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>Green</td>
<td>Gadget</td>
<td>Toys</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Widget</td>
<td>Black</td>
<td>Gadget</td>
<td>Toys</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>Green</td>
<td>Whatsit</td>
<td>Garden</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

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, $F = \{f_1, ..., f_n\}$, 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

<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>
</tr>
</thead>
</table>

$A_1, ..., A_m \rightarrow B_1, ..., B_n$
1. Split/Combine

\[
\begin{array}{cccc}
A_1 & \ldots & A_m & B_1 & \ldots & B_n \\
\end{array}
\]

\[A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n\]

... is equivalent to the following \(n\) FDs...

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

<table>
<thead>
<tr>
<th>A₁</th>
<th>…</th>
<th>Aₘ</th>
<th>B₁</th>
<th>…</th>
<th>Bₙ</th>
</tr>
</thead>
</table>

And vice-versa, $A₁, ..., Aₘ \rightarrow B_i$ for $i=1, ..., n$

... is equivalent to ...

$A₁, ..., Aₘ \rightarrow B₁, ..., Bₙ$
Reduction/Trivial

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

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

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

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

Provided FDs:
1. \{Name\} → \{Color\}
2. \{Category\} → \{Department\}
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} \rightarrow {Name}</td>
<td>?</td>
</tr>
<tr>
<td>5. {Name, Category} \rightarrow {Color}</td>
<td>?</td>
</tr>
<tr>
<td>6. {Name, Category} \rightarrow {Category}</td>
<td>?</td>
</tr>
<tr>
<td>7. {Name, Category} \rightarrow {Color, Category}</td>
<td>?</td>
</tr>
<tr>
<td>8. {Name, Category} \rightarrow {Price}</td>
<td>?</td>
</tr>
</tbody>
</table>

Provided FDs:

1. \{Name\} \rightarrow \{Color\}
2. \{Category\} \rightarrow \{Dept.\}
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>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 \( A_1, \ldots, A_n \) 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: \( F = \{\text{name} \Rightarrow \{\text{color}\}, \text{category} \Rightarrow \{\text{department}\}, \text{color, category} \Rightarrow \{\text{price}\}\} \)

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 \( X^+ \)

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

\( \{\text{name, category}\}^+ = \{\text{name, category}\} \)
Closure Algorithm

Start with X = \{A_1, ..., A_n\}, FDs F.

Repeat until X doesn’t change; do:
  if \{B_1, ..., B_n\} \rightarrow C is in F and \{B_1, ..., 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 \( X = \{A_1, ..., A_n\} \), FDs \( F \).
Repeat until \( X \) doesn’t change; do:

\[
\text{if } \{B_1, ..., B_n\} \rightarrow C \text{ is in } F \text{ and } \{B_1, ..., B_n\} \subseteq X:
\]

\[
\text{then add } C \text{ to } X.
\]

Return \( X \) as \( X^+ \)

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

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

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

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

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

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

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

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

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

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

Compute \(\{A, F\}^+ = \{A, B, C, D, E, F\}\)
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 $X$:

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

Example:
Given $F = \{\{A, B\} \to C, \{A, D\} \to B, \{B\} \to D\}$

No need to compute all of these- why?
Using Closure to Infer ALL FDs

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

\[
\begin{align*}
\{A\}^+ &= \{A\}, \quad \{B\}^+ = \{B,D\}, \quad \{C\}^+ = \{C\}, \quad \{D\}^+ = \{D\}, \\
\{A,B\}^+ &= \{A,B,C,D\}, \quad \{A,C\}^+ = \{A,C\}, \\
\{A,D\}^+ &= \{A,B,C,D\}, \quad \{A,B,C\}^+ = \{A,B,D\}^+ = \{A,D\}^+ = \{A,B,C,D\}, \\
\{A,B,C,D\}^+ &= \{A,B,C,D\}, \quad \{B,C\}^+ = \{B,C,D\}, \\
\{A,B,C,D\}^+ &= \{A,B,C,D\}
\end{align*}
\]

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

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

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

Example:
Given $F =$

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

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

\[
\begin{align*}
\{A\}^+ &= \{A\}, \quad \{B\}^+ = \{B,D\}, \quad \{C\}^+ = \{C\}, \quad \{D\}^+ = \{D\}, \\
\{A,B\}^+ &= \{A,B,C,D\}, \quad \{A,C\}^+ = \{A,C\}, \\
\{A,D\}^+ &= \{A,B,C,D\}, \quad \{A,B,C\}^+ = \{A,B,D\}^+ = \{A,C,D\}^+ = \{A,B,C,D\}, \\
\{B,C,D\}^+ &= \{B,C,D\}, \quad \{A,B,C,D\}^+ = \{A,B,C,D\}
\end{align*}
\]

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

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

“$Y$ is in the closure of $X$”
Using Closure to Infer ALL FDs

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

$$
\begin{align*}
\{A\}^+ &= \{A\},
\{B\}^+ &= \{B,D\},
\{C\}^+ &= \{C\},
\{D\}^+ &= \{D\},
\{A,B\}^+ &= \{A,B,C,D\},
\{A,C\}^+ &= \{A,C\},
\{A,D\}^+ &= \{A,B,C,D\},
\{A,B,C\}^+ &= \{A,B,D\}^+ = \{A,B,C,D\},
\{A,C,D\}^+ &= \{A,B,C,D\},
\{A,B,C,D\}^+ &= \{A,B,C,D\}
\end{align*}
$$

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

$$
\begin{align*}
\{A,B\} &\rightarrow \{C,D\},
\{A,D\} &\rightarrow \{B,C\},
\{A,B,C\} &\rightarrow \{D\},
\{A,B,D\} &\rightarrow \{C\},
\{A,C,D\} &\rightarrow \{B\}
\end{align*}
$$

Example:

Given $F =$

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

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

This means that no subset of a key is also a superkey (i.e., dropping any attribute from the key makes it no longer a superkey).
Finding Keys and Superkeys

• For each set of attributes $X$

1. Compute $X^+$

2. If $X^+ = \text{set of all attributes}$ then $X$ is a superkey

3. If $X$ is minimal, then it is a key
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)**

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

\[
\{\text{name, category}\}^+ = \{\text{name, price, category, color}\} \\
= \text{the set of all attributes} \\
\Rightarrow \text{this is a superkey} \\
\Rightarrow \text{this is a key, since neither name nor category alone is a superkey}
\]
Activity-5-1.ipynb
Lecture 6: Design Theory II
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; therefore, those other attributes can be duplicated

  • 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}

$\Rightarrow$ **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>

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

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

### Let’s check anomalies:
- Redundancy?
- Update?
- Delete?

<table>
<thead>
<tr>
<th>SSN</th>
<th>PhoneNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-45-6789</td>
<td>206-555-1234</td>
</tr>
<tr>
<td>123-45-6789</td>
<td>206-555-6543</td>
</tr>
<tr>
<td>987-65-4321</td>
<td>908-555-2121</td>
</tr>
<tr>
<td>987-65-4321</td>
<td>908-555-1234</td>
</tr>
</tbody>
</table>
BCNF Decomposition Algorithm

BCNFDecomp(R):

1. Find $X$ such that $X + \neq X$ and $X + \neq \text{all attributes}$
2. If not found, return $R$
3. Let $Y = X + - X$, $Z = (X +) C$
4. Decompose $R$ into $R_1(X \cup Y)$ and $R_2(X \cup Z)$
5. Return $BCNFDecomp(R_1)$, $BCNFDecomp(R_2)$
BCNF Decomposition Algorithm

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

BCNFD Decomp(R):
   Find a set of attributes X s.t.: $X^+ \neq X$ and $X^+ \neq [\text{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 complement, the other attributes that it doesn’t
BCNF Decomposition Algorithm

BCNFDecomp(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)

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

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

BCNFD Decomp (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)

Return BCNFD Decomp (R₁), BCNFD Decomp (R₂)

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

Return BCNFDecomp($R_1$), BCNFDecomp($R_2$)

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

{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\}
\{C\} \rightarrow \{D\}
Activity-7-1.ipynb
2. Decompositions
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

Sometimes a decomposition is “correct”

I.e. it is a **Lossless 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>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>Camera</td>
</tr>
<tr>
<td>Gizmo</td>
<td>Camera</td>
</tr>
</tbody>
</table>
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?

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

<table>
<thead>
<tr>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.99</td>
<td>Gadget</td>
</tr>
<tr>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>19.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>
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 $R = R_1$ Join $R_2$. 

$Lecture 6 > Section 2 > Decompositions$
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

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

\{\text{Unit}\} \rightarrow \{\text{Company}\}

\{\text{Company, Product}\} \rightarrow \{\text{Unit}\}!!

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

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaga99</td>
<td>UW</td>
<td>Databases</td>
</tr>
<tr>
<td>Bingo</td>
<td>UW</td>
<td>Databases</td>
</tr>
</tbody>
</table>

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 \{Company, Product\} \rightarrow \{Unit\}!!
The Problem

• We started with a table $R$ and FDs $F$

• We decomposed $R$ into BCNF tables $R_1, R_2, \ldots$ with their own FDs $F_1, F_2, \ldots$

• 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...
3. MVDs
What you will learn about in this section

1. MVDs

2. ACTIVITY
Multi-Value Dependencies (MVDs)

• A multi-value dependency (MVD) is another type of dependency that could hold in our data, which is not captured by FDs

• Formal definition:
  • Given a relation $R$ having attribute set $A$, and two sets of attributes $X, Y \subseteq A$
  • The multi-value dependency (MVD) $X \rightarrow Y$ holds on $R$ if
  • for any tuples $t_1, t_2 \in R$ s.t. $t_1[X] = t_2[X]$, there exists a tuple $t_3$ s.t.:
    • $t_1[X] = t_2[X] = t_3[X]$
    • $t_1[Y] = t_3[Y]$
    • $t_2[A \setminus Y] = t_3[A \setminus Y]$
  • Where $A \setminus B$ means “elements of set $A$ not in set $B$”
Multi-Value Dependencies (MVDs)

• One less formal, literal way to phrase the definition of an MVD:

  • **The MVD** $X \rightarrow Y$ holds on $R$ if for any pair of tuples with the same $X$ values, the “swapped” pair of tuples with the same $X$ values, but the other permutations of $Y$ and $A \setminus Y$ values, is also in $R$

Ex: $X = \{x\}$, $Y = \{y\}$:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

For $X \rightarrow Y$ to hold must have...

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note the connection to a local *cross-product*...
Multi-Value Dependencies (MVDs)

• Another way to understand MVDs, in terms of *conditional independence*:

• **The MVD** \( X \rightarrow Y \)** holds on \( R \) if given \( X \), \( Y \) is conditionally independent of \( A \setminus Y \) and vice versa...

Here, given \( x = 1 \), we know for ex. that:
\[ y = 0 \rightarrow z = 1 \]

I.e. \( z \) is conditionally **dependent** on \( y \) given \( x \)

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Here, this is not the case!

I.e. \( z \) is conditionally **independent** of \( y \) given \( x \)

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Multiple Value Dependencies (MVDs)

A “real life” example...

Grad student CA thinks:
“Hmm… what is real life??
Watching a movie over the weekend?”
MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater</th>
<th>film_name</th>
<th>snack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

Are there any functional dependencies that might hold here?  
No...

And yet it seems like there is some pattern / dependency...
# MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater</th>
<th>film_name</th>
<th>snack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>
### MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater</th>
<th>film_name</th>
<th>snack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>
For a given movie theatre...

Given a set of movies and snacks...

Any movie / snack combination is possible!
### MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

More formally, we write \( \{A\} \rightarrow \{B\} \) if for any tuples \( t_1, t_2 \) s.t. \( t_1[A] = t_2[A] \).
### MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

More formally, 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.

- \( t_3[A] = t_1[A] \)
### MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

More formally, 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.:

- \( t_3[A] = t_1[A] \)
- \( t_3[B] = t_1[B] \)
More formally, 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.

- \( t_3[A] = t_1[A] \)
- \( t_3[B] = t_1[B] \)
- and \( t_3[R\backslash B] = t_2[R\backslash B] \)

Where \( R\backslash B \) is “\( R \) minus \( B \)” i.e. the attributes of \( R \) not in \( B \).
MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp;</td>
<td>Kale Chips</td>
</tr>
<tr>
<td></td>
<td>Extended Edition</td>
<td></td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp;</td>
<td>Burrito</td>
</tr>
<tr>
<td></td>
<td>Extended Edition</td>
<td></td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

Note this also works!

Remember, an MVD holds over a relation or an instance, so defn. must hold for every applicable pair...
MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn, Kale Chips</td>
</tr>
<tr>
<td></td>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn, Burrito</td>
</tr>
<tr>
<td>t&lt;sub&gt;3&lt;/sub&gt;</td>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition, Kale Chips</td>
</tr>
<tr>
<td>t&lt;sub&gt;1&lt;/sub&gt;</td>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition, Burrito</td>
</tr>
<tr>
<td></td>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel, Ramen</td>
</tr>
<tr>
<td></td>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel, Plain Pasta</td>
</tr>
</tbody>
</table>

This expresses a sort of dependency (= data redundancy) that we can’t express with FDs.

*Actually, it expresses conditional independence (between film and snack given movie theatre)!
Comments on MVDs

• *For AI nerds*: MVD is conditional independence in graphical models!

See the MVDs IPython notebook for more examples!
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