Subclasses → Relations

Three approaches:

1. ODL style: each object is in one class. Create a relation for each class, with all the attributes for that class.
   ✦ Don’t forget inherited attributes.

2. E/R style: an entity is in a network of classes related by isa. Create one relation for each E.S.; entities represented in all E.S. to which it belongs.
   ✦ Relation has only the attributes attached to that E.S. + key.

3. Use nulls. Create one relation for the root class or root E.S., with all attributes found anywhere in its network of subclasses.
   ✦ Put NULL in attributes not relevant to an object/entity.
Example

interface Beers (key name) {
    attribute string name;
    attribute string manf;
}

interface Ales: Beers {
    attribute string color;
}
<table>
<thead>
<tr>
<th>name</th>
<th>manf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>A.B.</td>
</tr>
</tbody>
</table>

Beers

<table>
<thead>
<tr>
<th>name</th>
<th>manf</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>SummerBrew</td>
<td>Pete’s</td>
<td>dark</td>
</tr>
</tbody>
</table>

Ales

<table>
<thead>
<tr>
<th>name</th>
<th>manf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>A.B.</td>
</tr>
<tr>
<td>SummerBrew</td>
<td>Pete’s</td>
</tr>
</tbody>
</table>

Beers

<table>
<thead>
<tr>
<th>name</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>SummerBrew</td>
<td>dark</td>
</tr>
</tbody>
</table>

Ales
Using Nulls

<table>
<thead>
<tr>
<th>name</th>
<th>manf</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bud</td>
<td>A.B.</td>
<td>NULL</td>
</tr>
<tr>
<td>SummerBrew</td>
<td>Pete’s</td>
<td>dark</td>
</tr>
</tbody>
</table>

Beers

Design Challenge

Remember the problem with local and express trains and local and express stations?

- Trains have numbers and engineers.
- Stations have names and addresses.
- There is a time for each train/station pair where a stop occurs.

How would we best represent this information as relations? Can we design the database schema so it is impossible for an express train to have a stop time for a local station? Should we?
Functional Dependencies

$X \rightarrow A =$ assertion about a relation $R$ that whenever two tuples agree on all the attributes of $X$, then they must also agree on attribute $A$.

- Important as a constraint on the data that may appear within a relation.
  - Schema-level control of data.
- Mathematical tool for explaining the process of “normalization” — vital for redesigning database schemas when original design has certain flaws.
Example

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

<table>
<thead>
<tr>
<th>name</th>
<th>addr</th>
<th>beersLiked</th>
<th>manf</th>
<th>favoriteBeer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Janeway</td>
<td>Voyager</td>
<td>Bud</td>
<td>A.B.</td>
<td>WickedAle</td>
</tr>
<tr>
<td>Janeway</td>
<td>Voyager</td>
<td>WickedAle</td>
<td>Pete’s</td>
<td>WickedAle</td>
</tr>
<tr>
<td>Spock</td>
<td>Enterprise</td>
<td>Bud</td>
<td>A.B.</td>
<td>Bud</td>
</tr>
</tbody>
</table>

- Reasonable FD’s to assert:

  1. name → addr
  2. name → favoriteBeer
  3. beersLiked → manf

- Note: These happen to imply the underlined key, but the FD’s give more detail than the mere assertion of a key.
• Key (in general) functionally determines all attributes. In our example:

name beersLiked → addr favoriteBeer beerManf

• Shorthand: combine FD’s with common left side by concatenating their right sides.

• When FD’s are not of the form Key → other attribute(s), then there is typically an attempt to “cram” too much into one relation.

• Sometimes, several attributes jointly determine another attribute, although neither does by itself. Example:

  beer bar → price
Formal Notion of Key

*K* is a *key* for relation *R* if:

1. *K* → all attributes of *R*.
2. For no proper subset of *K* is (1) true.
   - If *K* satisfies only (1), then *K* is a *superkey*.

FD Conventions

- *X*, etc., represent sets of attributes; *A* etc., represent single attributes.
- No set formers in FD’s, e.g., *ABC* instead of \{*A*, *B*, *C*\}.
Example

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

- \{name, beersLiked\} FD’s all attributes, as seen.
  - Shows \{name, beersLiked\} is a superkey.
- name → beersLiked is false, so name not a superkey.
- beersLiked → name also false, so beersLiked not a superkey.
- Thus, \{name, beersLiked\} is a key.
- No other keys in this example.

- Neither name nor beersLiked is on the right of any observed FD, so they must be part of any superkey.
Who Determines Keys/FD’s?

- We could define a relation schema by simply giving a single key \( K \).
  - Then the only FD’s asserted are that \( K \rightarrow A \) for every attribute \( A \).
  - No surprise: \( K \) is then the only key for those FD’s, according to the formal definition of “key.”

- Or, we could assert some FD’s and deduce one or more keys by the formal definition.

- Example where > 1 key: employees with SS# and employee ID.

- Rule of thumb: FD’s either come from keyness or from physics.
  - E.g., “no two courses can meet in the same room at the same time” yields room time \( \rightarrow \) course.
Inferring FD’s

And this is important because . . .

- When we talk about improving relational designs, we often need to ask “does this FD hold in this relation?”

Given FD’s $X_1 \rightarrow A_1$, $X_2 \rightarrow A_2$ $\cdots$ $X_n \rightarrow A_n$, does FD $Y \rightarrow B$ necessarily hold in the same relation?

- Start by assuming two tuples agree in $Y$. Use given FD’s to infer other attributes on which they must agree. If $B$ is among them, then yes, else no.
Algorithm
Define $Y^+ = \textit{closure}$ of $Y$:

- **Basis:** $Y^+ = Y$.

- **Induction:** If $X \subseteq Y^+$, and $X \rightarrow A$ is a given FD, then add $A$ to $Y^+$.

- **End when $Y^+$ cannot be changed.** Then $Y$ functionally determines all members of $Y^+$, and no other attributes.
Example

\( A \rightarrow B, \ BC \rightarrow D. \)

- \( A^+ = AB. \)
- \( C^+ = C. \)
- \( (AC')^+ = ABCD. \)

- Thus, \( AC' \) is a key.
Inference Rules

Some useful tricks that help us infer FD’s without resorting to the closure algorithm.

- But each is proved using the closure algorithm.

Armstrong’s Axioms

1. **Reflexivity**: If $Y \subseteq X$, then $X \rightarrow Y$.

2. **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$.

3. **Transitivity**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. 
Why?

- Reflexivity obvious from closure test.
- Augmentation:

\[
\begin{array}{c}
X \\
\rightarrow Y \\
Z
\end{array}
\]

- Transitivity:

\[
\begin{array}{c}
X \\
\rightarrow Y \\
\rightarrow Z
\end{array}
\]
Example

Prove $A \rightarrow B$ and $BC \rightarrow D$ imply $AC \rightarrow D$.

1. $A \rightarrow B$ (given).
2. $AC \rightarrow BC$ (augmentation).
3. $BC \rightarrow D$ (given).
4. $AC \rightarrow D$ (transitivity using 2 and 3).

Example

$A \rightarrow B$ and $A \rightarrow C$ imply $A \rightarrow BC$.

1. $A \rightarrow B$ (given).
2. $A \rightarrow AB$ (augmentation using $A$).
3. $A \rightarrow C$ (given).
4. $AB \rightarrow BC$ (augmentation).
5. $A \rightarrow BC$ (transitivity using 2 and 4).