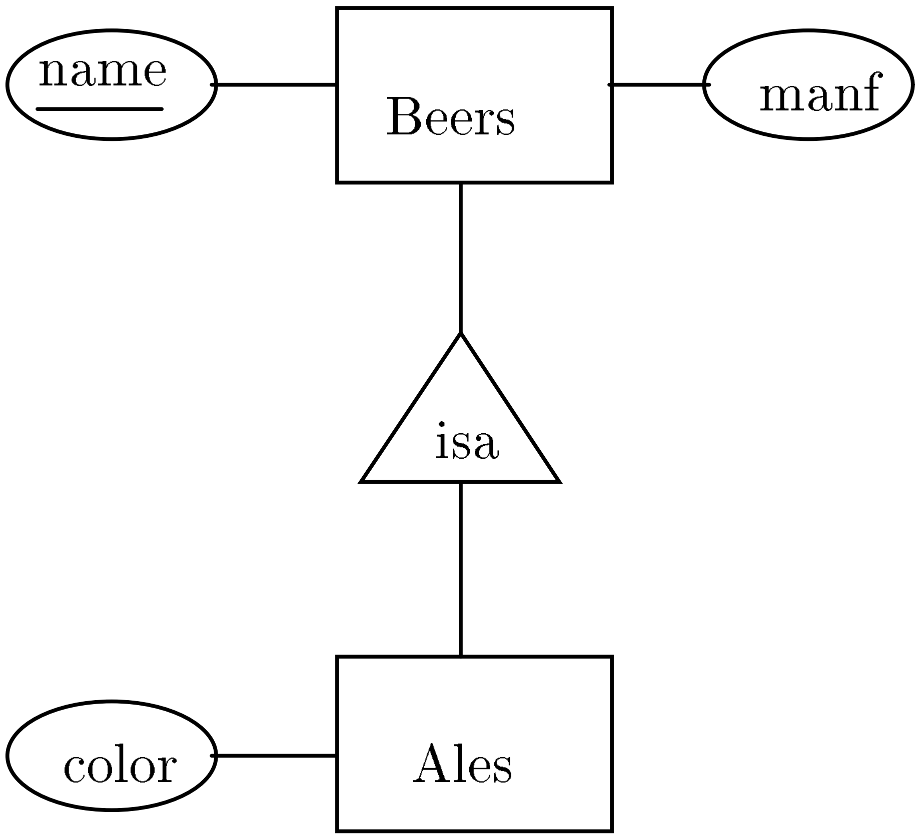
Subclasses → Relations

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;
}

interfaceAles:Beers {
    attribute string color;
}
```

ODL Style

name	manf
Bud	A.B.

Beers

name	manf	color
SummerBrew	Pete's	dark

Ales

E/R Style

name	manf
Bud	A.B.
SummerBrew	Pete's

Beers

name	color
SummerBrew	dark

Ales

Using Nulls

name	manf	color
Bud	A.B.	NULL
SummerBrew	Pete's	dark

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(`<u>`name`</u>`, addr, `<u>`beersLiked`</u>`, manf, favoriteBeer)`

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

- 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. $K \rightarrow$ 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(`<u>`name`</u>`, addr, `<u>`beersLiked`</u>`, manf, favoriteBeer)`

- {`name, beersLiked`} FD's all attributes, as seen.
  - Shows {`name, beersLiked`} is a superkey.
- `name` $\rightarrow$ `beersLiked` is false, so `name` not a superkey.
- `beersLiked` $\rightarrow$ `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?"

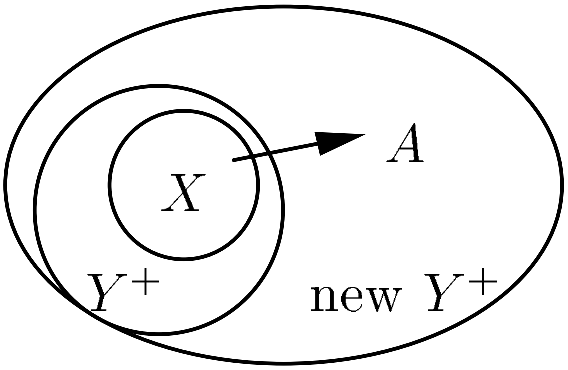
Given FD's $X1 \rightarrow A1$, $X2 \rightarrow A2 \cdots Xn \rightarrow An$, does FD $Y \rightarrow B$ necessarily hold in the same relation?

- 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^+ = *closure* of Y:

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

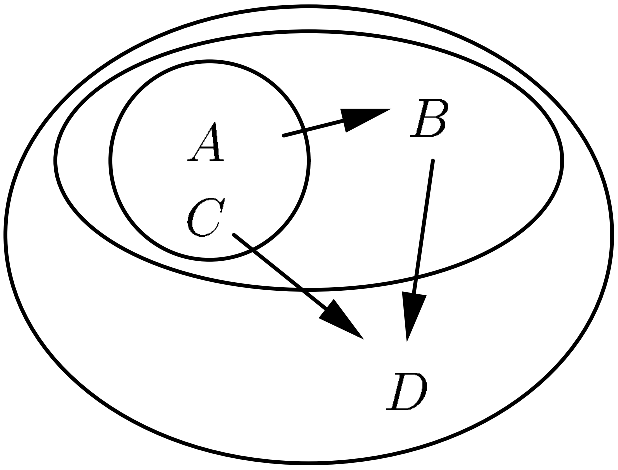


- End when Y^+ cannot be changed. Then Y functionally determines all members of Y^+, and no other attributes.

Example

$A \to B$, $BC \to D$.

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



- Thus, AC is a key.

Inference Rules

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

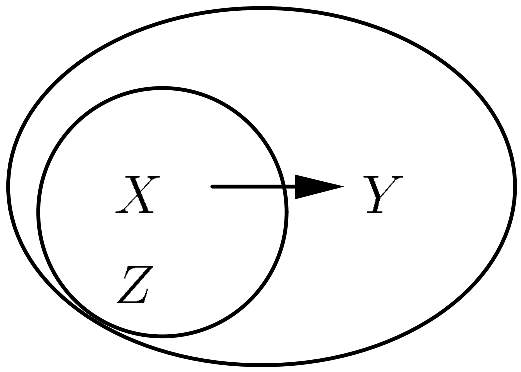
- But each is proved using the closure algorithm.

Armstrong's Axioms

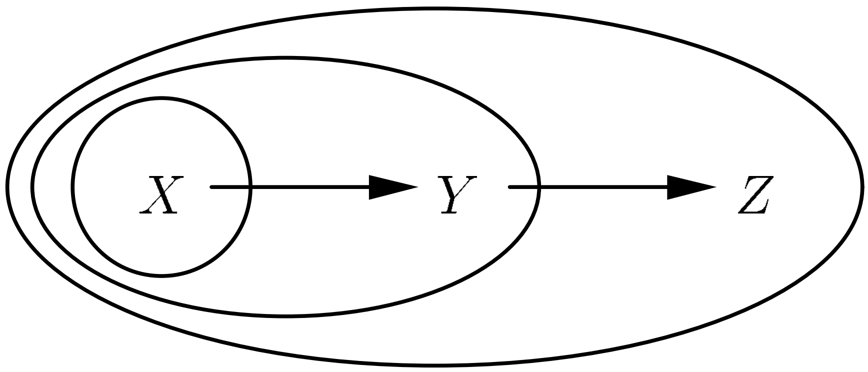
1. *Reflexivity*: If $Y \subseteq X$, then $X \rightarrow Y$.
2. *Augmentation*: If $X \rightarrow Y$, then $XZ \rightarrow YZ$.
3. *Transitivity*: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Why?

- Reflexivity obvious from closure test.
- Augmentation:



- Transitivity:



Example

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

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

Example

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

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