E/R Modelling:

A Conceptual Modelling Language for Schema

Table of Contents

Parke Godfrey

2016-09-19 initial
2016-10-03 [v5]
2018-01-23 [v6]

Acknowledgments

Thanks

- to *Jeffrey D. Ullman*
  for initial slidedeck
- to *Jarek Szlichta*
  for the slidedeck with significant refinements on which this is derived

Why a *conceptual* modelling language?

- So we can sketch out database-schema designs more easily, and “see” their *semantics* / *logic* more easily.
  - The relational model just has *relations*. It can be harder to see the schema's *semantics* / *logic*.
- Many conceptual modelling languages are *pictoral*.

Later, we'll *convert* E/R designs into relational schema.

Design flowchart

ideas / *requirements*

⇓

high-level design / *specification*

⇓

relational schema / *implementation*

⇓

relational database (in RDBMS) / *realization*

Different conceptual modelling languages

- the Entity / Relationship Model(ling language)
  - many *dialects*!
- UML (*Universal Modeling Language*)
- ODL (Object Definition Language)
- *Many more*…

We choose E/R here because it is in many ways the simplest.

Schema semantics

A schema design should specify

- **what** data the schema represents, and how the data are related, but
- *not* **how** the data is used or processed (*operational*).

In other words, our schema designs are *not* meant to be *process models*.

Again, this is about *data independence*.

Design work

Design is hard work! And serious business.

- The boss or client may know they need a database, but they do not know *what* they need in it.
- Sketching the key components is an efficient way to develop a working database.
  - A design can be explained to lay people.
  - A design can be *iteratively* refined.

In software engineering, this is called *requirements elicitation*.

The Entity / Relationship Model

- **entities** (*things*)
- **relationships**: how entities are *related*
- **attributes**: *properties* of an entity
  - attributes are simple values;
    e.g., integers & strings

    Not structs, sets, lists, etc.

Types (sets)

We assume we will have *types* of entities & of rel-ships.

- For instance, we would have many *students* (entities), many *classes* (entities), and many *enrolled* (rel-ships).
- So, we'll have
  - an *entity set* called *student*,
  - an *entity set* called *class*, and
  - a *rel-ship set* called *enrolled*.
- Entity and rel-ship *instances* of the same *type* (*set*) each has the same set of attributes.

E/R diagrams

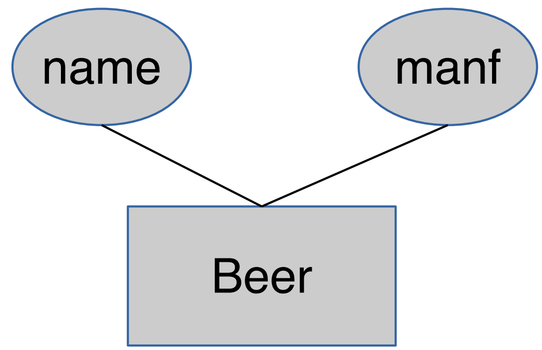
The E/R model is pictoral.

- **entity set**: a labelled *rectangle*
- **rel-ship set**: a labelled *diamond*
- **attribute**: a labelled *oval*, with a line to the entity set or the rel-ship set that *owns* it

Rel-ship sets connect entity sets together that they relate by lines.

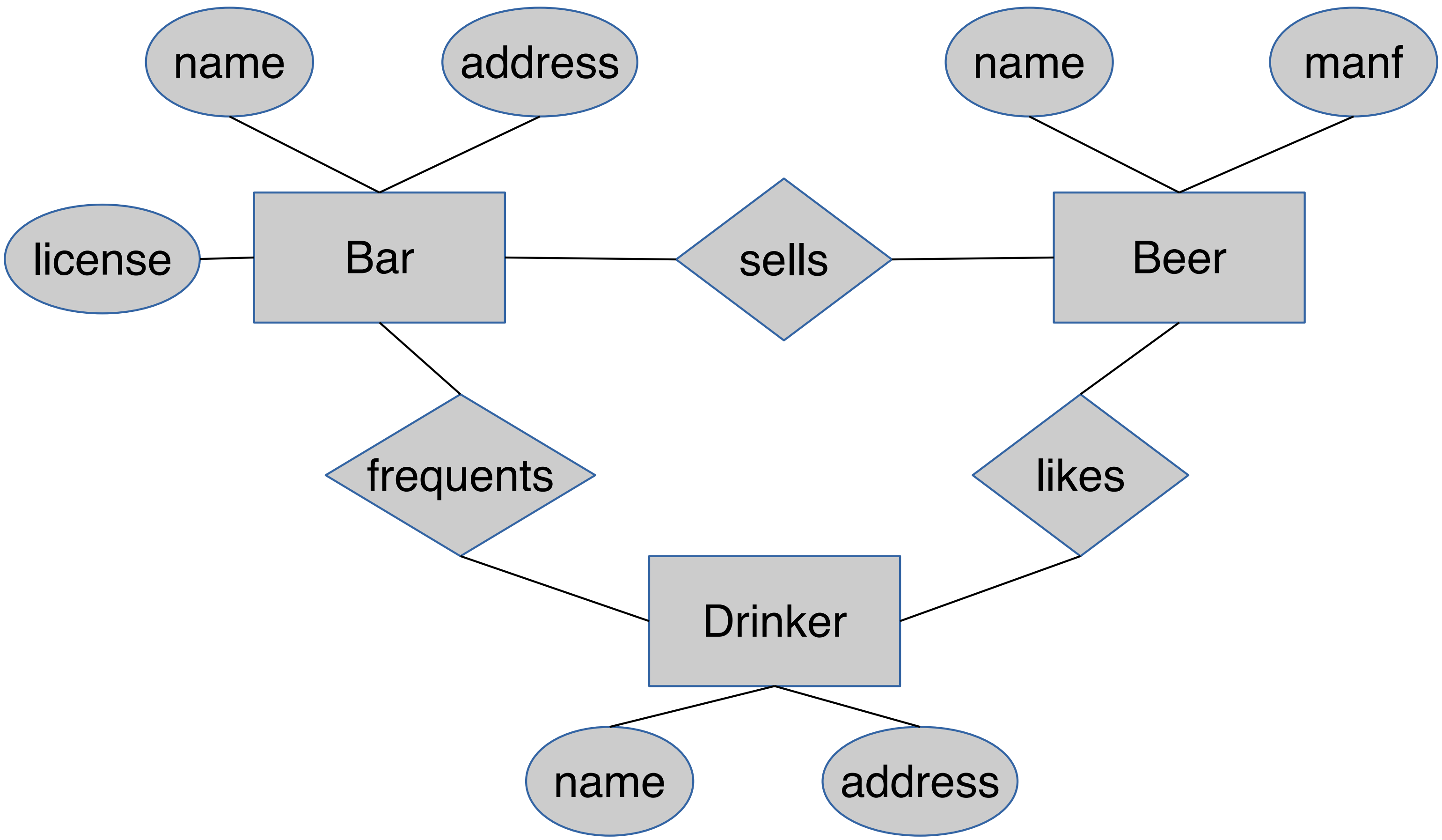
Thus, an E/R diagram is a type of bipartite graph between entity and rel-ship sets.

Example: entity set



- Entity set **Beer** has two attributes, *name* & *manf*.
- Each *Beer* entity (instance) has values for these two attributes.
  E.g., *Bud* & *Anheuser-Busch*

Example: relationship sets



How to read

- Certain **pubs** *sell* certain **beers**.
- Certain **drinkers** *like* certain **beers**.
- Certain **drinkers** *frequent* certain **pubs**.

Values for *license* are intended as *beer only*, *full*, and *none*.

Value of an entity set or rel-ship set

- The current "value" of an entity set is the set of entities that belong to it.
  - E.g., the set of all **pubs** in our database.
- The "value" of a relationship is the set of relationships that belong to it.
  - The "value" of a relationship is a tuple with one component for *each* related entity set.
  - E.g., ⟨*"Fox & Fiddle", "Molson"*⟩ is a *sells* relationship.

Example: relationship set value

Thus, an rel-ship value for *sells* might be

Pub	**Beer**
Fox & Fiddle	Molson
Fox & Fiddle	Bud
Clintons	Creemore
Clintons	Guiness
Clintons	Molson

Multi-way relationships

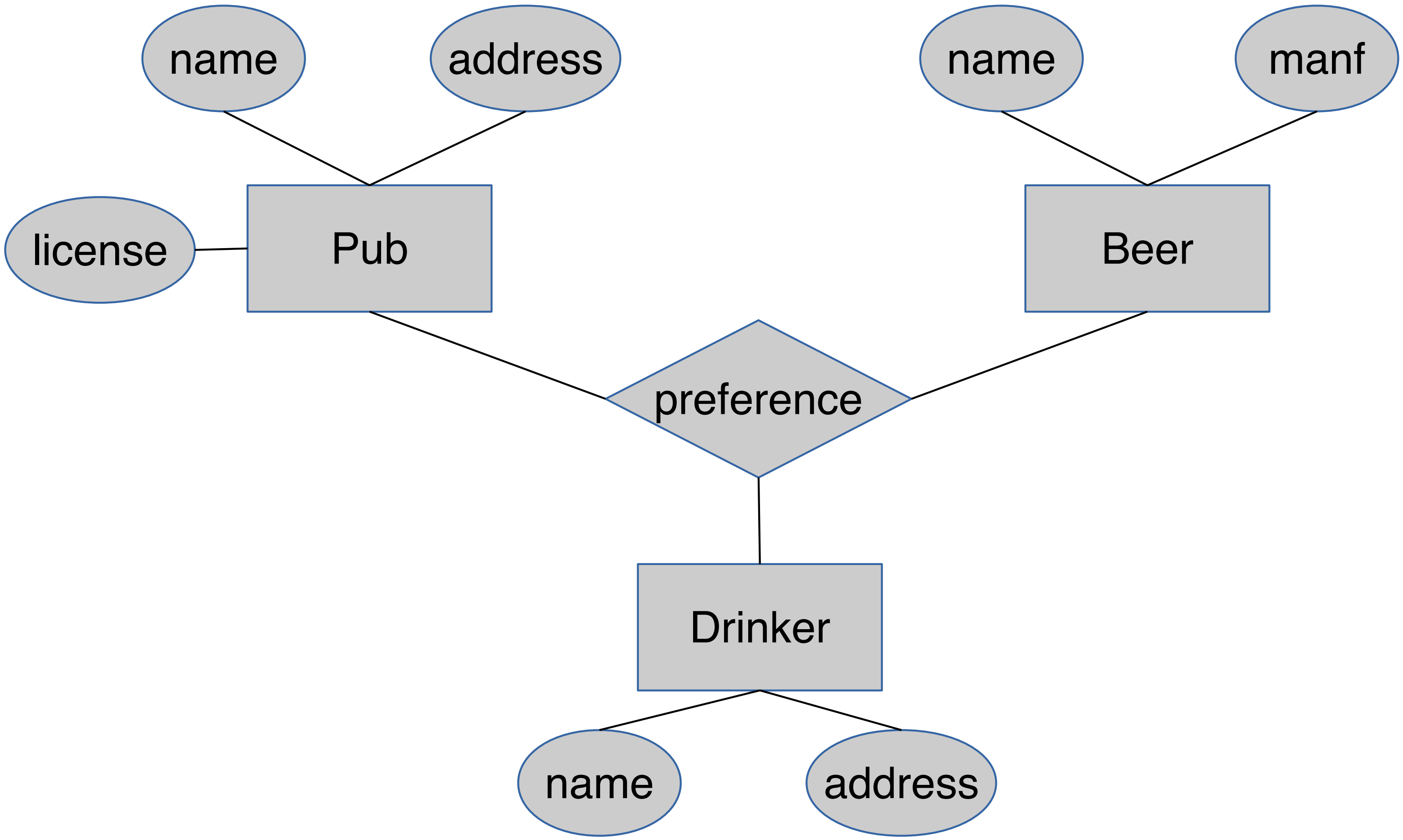
Sometimes, we need a relationship that connects more than two entity sets.

Suppose that drinkers will only drink certain beers at certain pubs.

Our three binary relationships *likes*, *sells*, and *frequents* do not allow us to make this distinction.

But a 3-way relationship would!

Example: 3-way rel-ship



Example: rel-ship set value

Pub	Drinker	Beer
Fox & Fiddle	Franck	Molson
Fox & Fiddle	Franck	Bud
Fox & Fiddle	Jeff	Bud
Clintons	Franck	Molson
Clintons	Parke	Creemore
Clintons	Jeff	Creemore
Clintons	Parke	Guiness

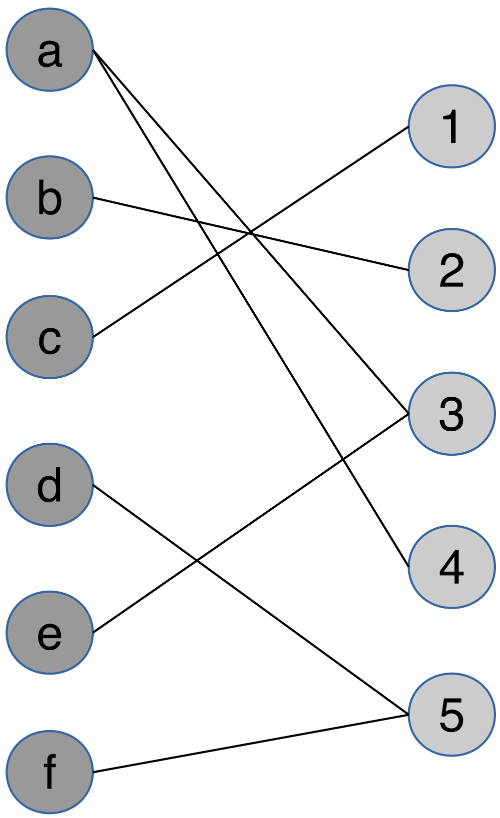
Many-many rel-ships

Focus: binary relationships, such as **sells** between **Pub** and **Beer**.

In a *many-many relationship*, an entity of either set can be connected to many entities of the other set. E.g.,

- a pub sells many beers; and
- a beer is sold by many pubs.

Many-many



Many-One rel-ships

Some binary relationships are *many-one* from one entity set to another.

Each entity of the first set is connected to *at most one* entity of the second set.

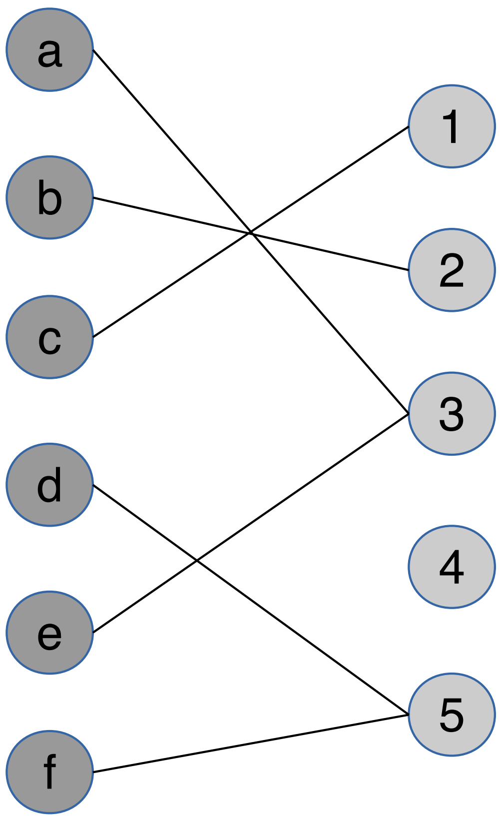
But an entity of the second set can be connected to *zero*, *one*, or *many* entities of the first set.

Example: many-one rel-ship

Consider a rel-ship **favourite**, from **Drinker** to **Beer**, and that it is *many-one*. That is,

- a drinker has at most one favourite beer, but
- a beer can be the favourite of any number of drinkers (including zero).

Many-one



One-one rel-ship

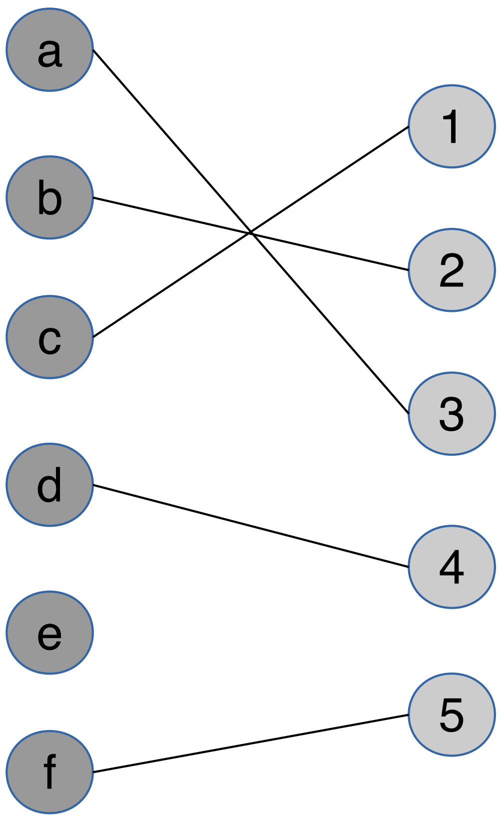
In a *one-one* rel-ship, each entity of either entity set is related to *at most one* entity of the other set.

Example

Rel-ship **best-seller** between entity sets **Manf** (*manufacturer*) and *Beer*.

- A beer cannot be made by more than one manufacturer, and
- no manufacturer can have more than one best-seller. (Assume no ties.)

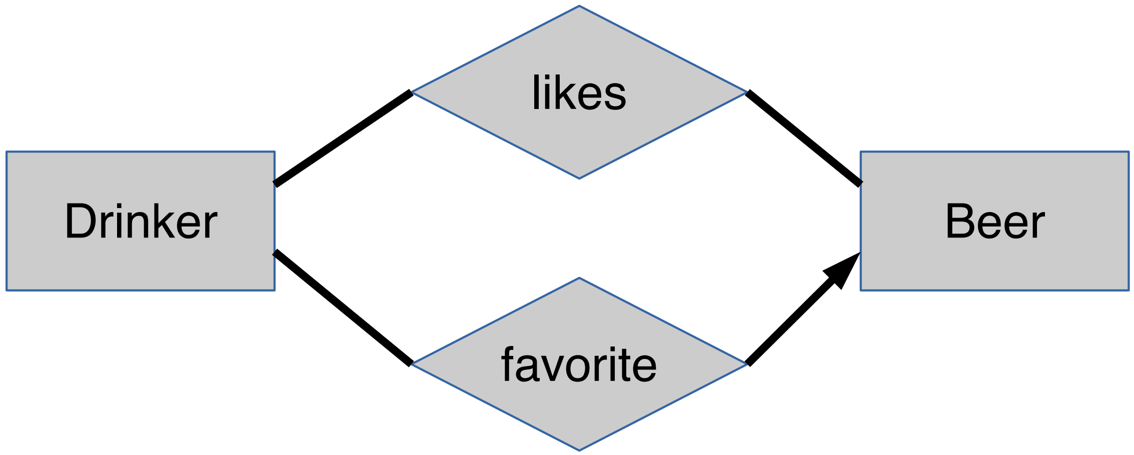
One-one



Representing multiplicity

- Show a *many-one* relationship by an arrowhead pointing to the “one” side.
- Show a *one-one* relationship by arrowheads pointing to both entity sets.
- Two types of arrowheads.
  - **plain arrowhead**: *zero* or *one*
    Each entity of the first set is related to *no* entity or *one* entity of the target set.
  - **rounded arrowhead**: *exactly one*
    Each entity of the first set is related to *exactly one* entity of the target set.

Favourite: Many-one rel-ship w/ multiplicity



Note that **Drinker** and **Beer** have *two* rel-ships between them. This is fine in E/R.

Thus, an E/R diagram is a *multi-graph*.

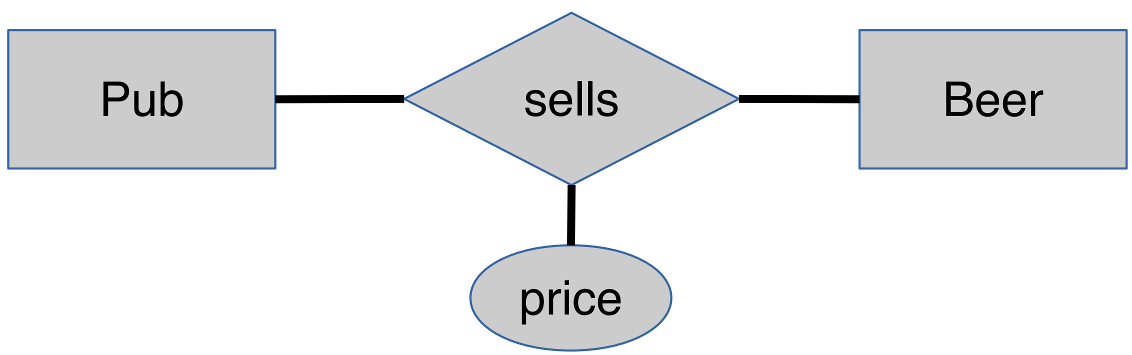
Q) How to ensure a *drinker's favourite beer* is actually a *beer* that he or she *likes*?

Best seller: One-one rel-ship w/ multiplicity

- A beer is the best-seller for 0 or 1 manufacturer.
- A manufacturer has *exactly* one best seller.

Attributes of rel-ships

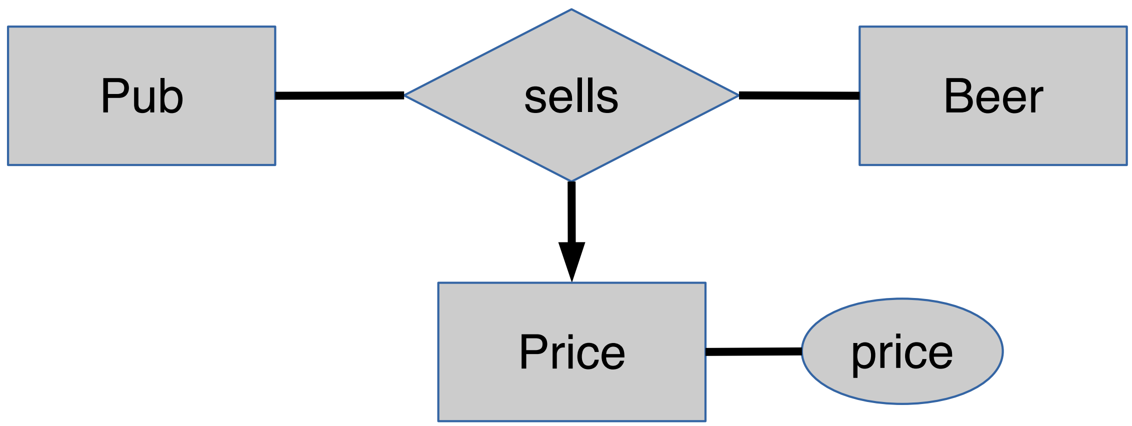
- Sometimes a rel-ship needs attributes.
- Such an attribute can be thought of as a property of "tuples" in the rel-ship set.



Price is a function of both the **Pub** and the **Beer**, not of just one or the other.

Without attributes on rel-ships

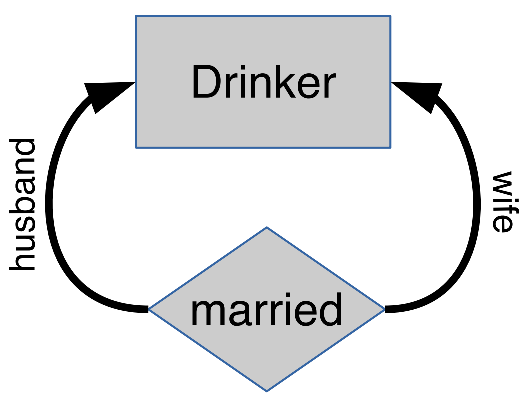
- Instead, could create an entity set representing values of the attribute.
- Then, make that entity set participate in the rel-ship.



Convention: Arrowhead from multiway rel-ship means that all *other* entity sets *together* determine a unique one of these.

"Recursive" rel-ships & roles

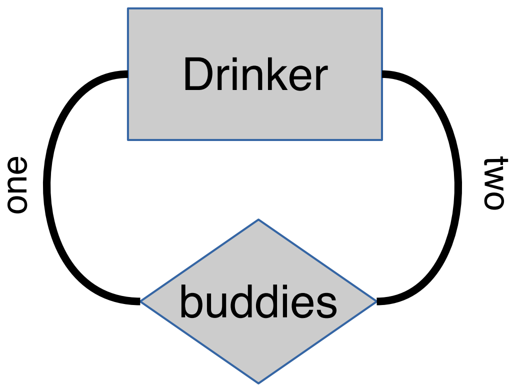
- A given entity set may appear more than once in a rel-ship!
- To disambiguate, we label the edges between the rel-ship and the entity set with names called *roles*.



husband	wife
Bob	Ann
Joe	Sue
…	…

Buddies

one	two
Joe	Moe
Ann	Sue
Ann	Moe
…	…

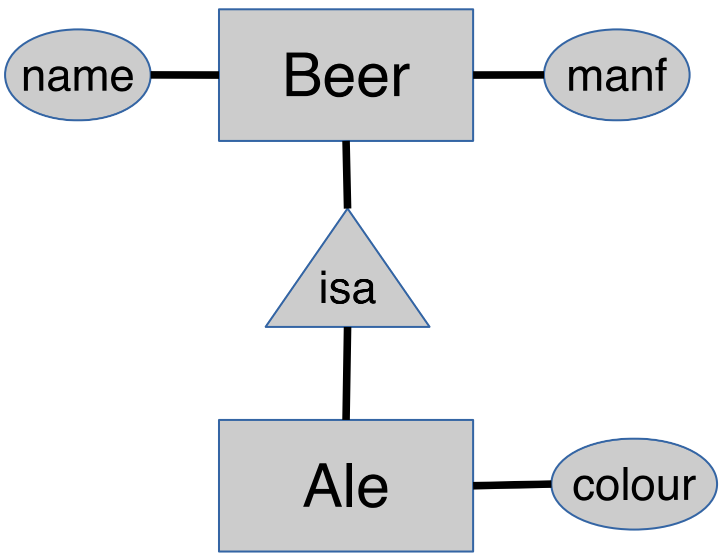


Subclasses

- A *subclass* is a special case that has more properties.
- E.g., An **Ale** is a kind of **Beer**.
- Not every beer is an ale, but some are.

Let us suppose that, in addition to the properties — attributes and rel-ships — of *beers*, *ales* also have the attribute *colour*.

Subclasses in E/R



- Assume subclasses form a tree.

  That is, no multiple inheritance.

- An **isa** triangle indicates the subclass “relationship”.

  The triangle “points” to the superclass.

E/R vs OO

- In OO, objects are in one class only.
  - Subclasses inherit from superclasses.
- In contrast, E/R entities have representatives in all subclasses to which they belong.

Rule. If entity *E* is represented in a subclass, then *E* is represented in the superclass. (And recursively up the tree.)

If “Pete's Ale” is an **Ale**, it is also a **Beer**.

Keys (déjà vu)

A *key* is a set of attributes for one entity set such that *no* two entities in this set agree on *all* the attributes's values of the key.

(Note that two entities may agree on *some*, but not *all*, of the key attributes's values.)

In a *complete* E/R diagram, a key must be designated for every entity set.

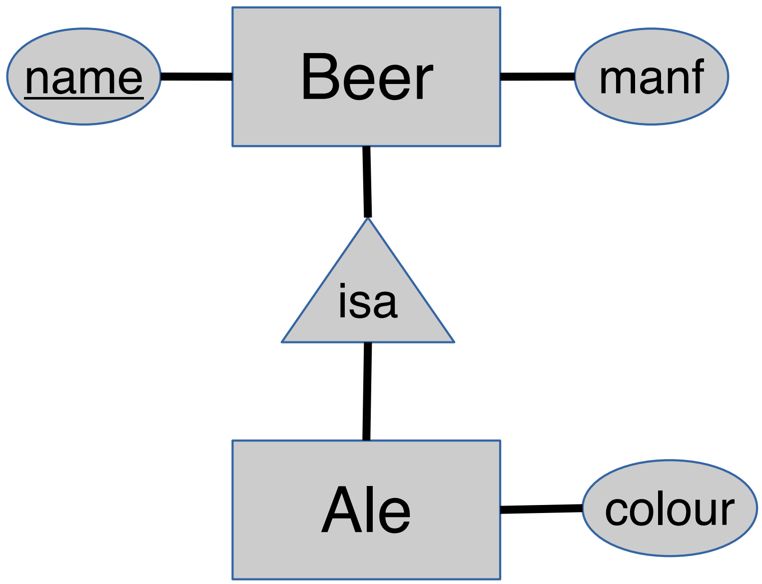
Keys in E/R diagrams

We underline the key attribute(s).

Note. In an **isa** hierarchy, only the root entity set has a key; this must serve as the key for all entities in the hierarchy.

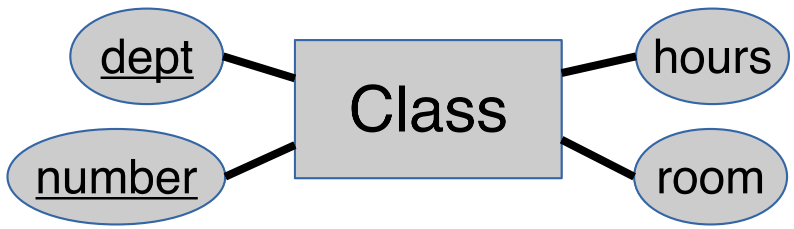
Entity sets have keys.
Rel-ships do *not!*

Keys in E/R



- *Name* is key for **Beer** *and* for **Ale**.

A multi-attribute key



- Note that *hours* and *room* could also serve as a key
- But let us limit ourselves to one key. For now.

Weak entity sets

Entities of entity set might need “help” to *identify* them uniquely.

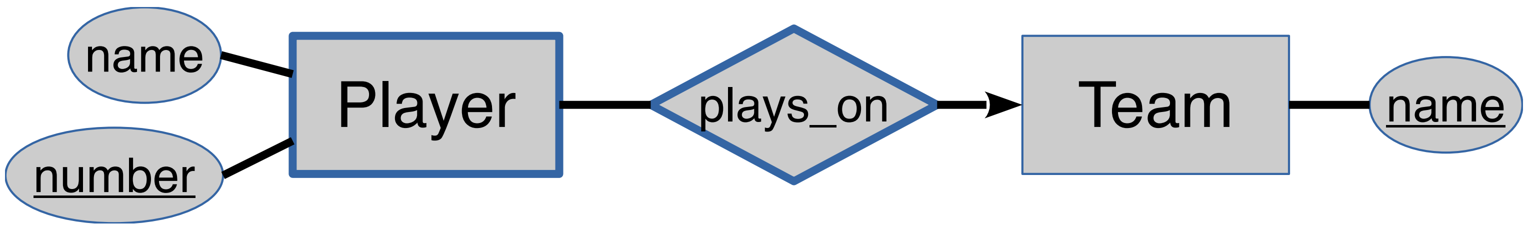
Entity set **E** is said to be *weak* if, in order to *identify* entities of **E** uniquely,

- we need to follow *one or more many-one* rel-ships from **E**, and
- include the *key* of each of the so connected entity sets as part of the weak entity's *key*.

Example of a weak entity set

- *Name* is “almost” a key for football players; but there can be two with the same name.
- *Number* is certainly not a key, since players on different teams can have the same number.
- But *number* together with the player's team's *name* — as related to the **Player**, say, by a **plays-on** many-one rel-ship — ought to be unique.

Weak entity sets in E/R: plays-on



- **Bolded** diamond for the supporting many-one rel-ships.
  (In textbook, double-framed.)
- **Bolded** rectangle for the weak entity set.
  (In textbook, double-framed.)

Note. The arrowhead must be *rounded* because each **Player** needs a **Team** to compose his or her *key* value.

Compositional rules for weak entity sets

- A weak entity set must have *one or more many-one* rel-ships to other *supporting* entity sets.
- Not *every* many-one rel-ship from the weak entity set needs to be *supporting*.
- Each supporting many-one rel-ship must have a *rounded* arrowhead; that is, the entity at the "one" end is guaranteed.
- The *key* for a weak entity set is its own underlined attributes *union* the keys from each of the supporting entity sets.
  - E.g., **Player**'s *number* and **Team**'s *name* is the key for *Player* in the previous example.

“Recursively” weak

- Can a weak entity set be supported by another weak entity set?
  - Of course.

We do not permit cyclic weak dependencies in E/R diagrams, though; they would not make logical sense.

When are weak entity sets needed?

The usual reason is that there is no global authority capable of creating unique ID's.

Example. It is unlikely that there could be an agreement to assign unique player numbers across all football teams in the world.

A very powerful modelling tool in E/R

- We can “promote” a rel-ship (set) by replacing it with a weak entity (set).
  - The weak entity's supporting entities are those the rel-ship was relating.
- Why is this useful?
  - The “rel-ship” (as a weak entity) can be related to *other* (non-supporting) entities!

A weak entity set used in this way is called a *connecting entity*.

A connecting entity does *not* contribute any of its *own* attr's to its key.

Overusing weak entity sets

- Beginning database designers often doubt that anything could be a key by itself.
  - They make all entity sets weak, supported by all other entity sets to which they are linked.
- In reality, we usually create unique ID's for entity sets. (These are called *surrogate* keys.)
  - Examples include social-security numbers, automobile VIN's etc.

E/R design principles

1. **fidelity**
2. **brevity**
3. **simplicity**
4. **naturalness**

These guidelines are in order of *precedence*. For example, *brevity* takes precedence over *simplicity*.

See §4.2 (*Design Principles*) in the textbook. Note that these slides is *my* take on design princples; but they align quite nicely with the textbook's.

1. Fidelity (to the domain)

faithfulness

- Be logically *true* to the real-world domain that we are modeling.
- Capture all of the real-world properties (*semantics*) of the domain as is possible and “reasonable”.

2. Brevity (of the data)

avoiding redundancy

- The design should *not* require that a piece of information be represented more than once.

  This is called the *principle* of *single source of truth*.

  - That is, it should not be possible that the same information appears many times in the same entity set.
  - If information can be logically inferred from other information in the design, it should *not* be kept.

3. Simplicity (of the schema)

- **Occam's razor**: Keep the design as *simple* as possible (but no simpler).
  - Should contain no elements — entities, rel-ships, attr's, etc. — that are not necessary.
  - Should have as few connections as possible.
  - **Precedence**: Use attributes before entities, and relationships before entities.

4. Naturalness (of the model)

- Model as *naturally* as possible the domain.
  - Entities ought to correspond to real things.
  - Rel-ships should be understandable.

Requirements to specifications

What is your *domain*?

- **queries**. What questions does the database need to be able to answer?
- **transactions**. What data-processing *activities* does the database need to support?

Modelling the domain

What is your design?

1. Choose your entities, attr's, and keys.

   What are the important rel-ships among them?

2. Any logical problems; anything missing?

   If so, then *refine* the design.

3. Can the design accommodate the questions (*queries*) and activities (*transactions*)?

   - If not, then *refine* the design.

4. Can we violate any real-world constraints?

   - If so, re-design — if *possible* and if *practical!* — so that we cannot.

5. Simplify. Repeat.

E/R to relational

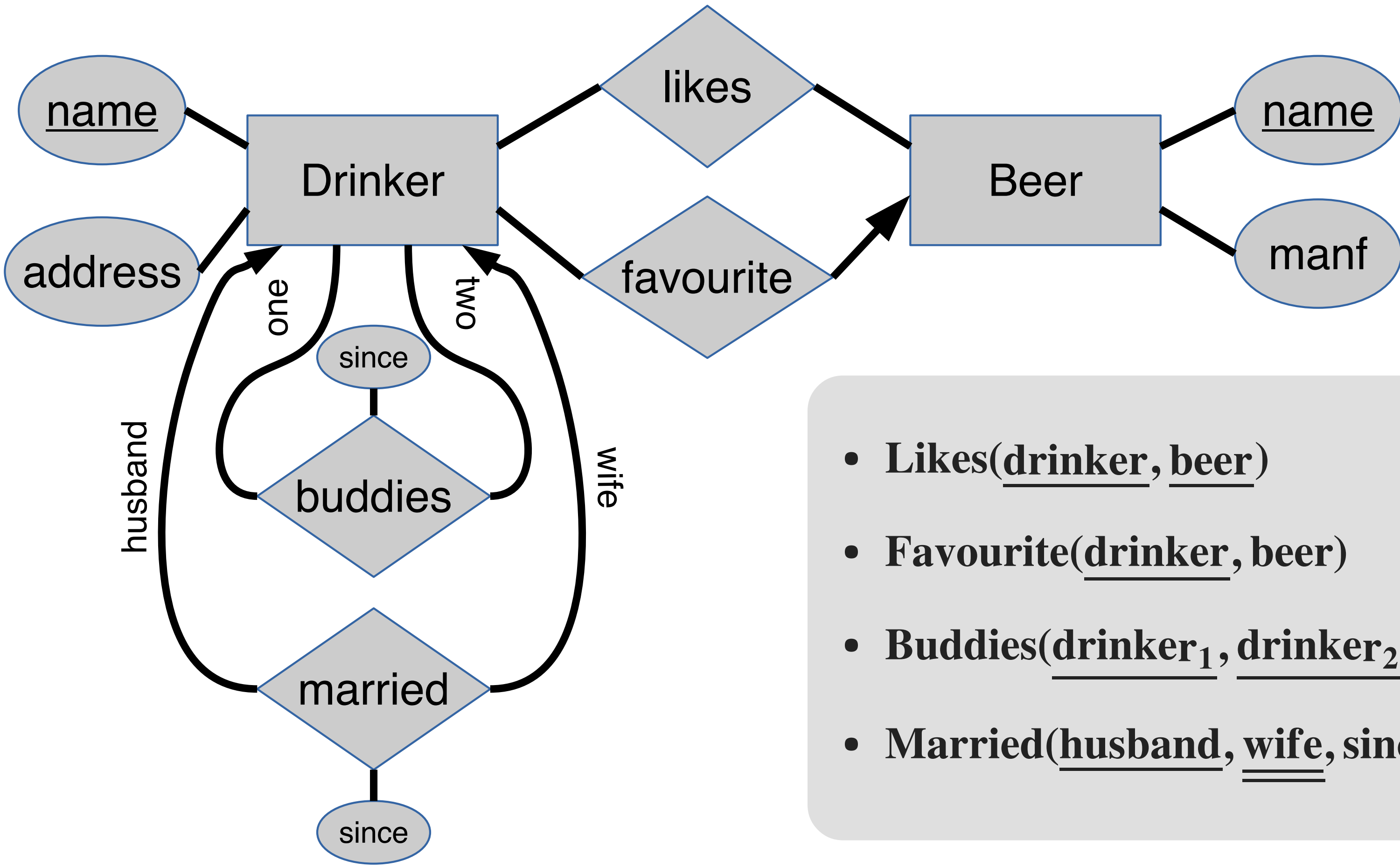
- *Entity set* ⇒ *relation* (table)
  - *Attributes* ⇒ *attributes* (columns)
  - key's *Attributes* ⇒ key's *attributes*
- *Rel-ship set -> relation* ⇒ for which the attr's are only
  - *Attributes* ⇒ attr's of the rel-ship itself.
  - key's *Attributes* ⇒ all the keys's attr's of the “many” connected entity sets (but not the “one”'s!)

We may *rename* attr's to disambiguate!

Entity set ⇒ relation

Relation: **Beer(name, manf)**

Rel-ship ⇒ relation



- **Likes(drinker, beer)**
- **Favourite(drinker, beer)**
- **Buddies(drinker$_1$, drinker$_2$, since)**
- **Married(husband, wife, since)**

Combining many-one relations

Okay to combine into one relation

- the relation for an entity-set **E** *and*
- the relations for *many-one* rel-ships for which **E** is the "many".

Example: **Drinker(<u>name</u>, addr)** and **Favourite(<u>drinker</u>, beer)** combine to make **$Drinker_1$(<u>name</u>, addr, favBeer)**.

Combining many-one relations

This is *compulsory* whenever the *many-one* is "mandatory participation"; that is, the "one" arrowhead is rounded, meaning *exactly* one (as opposed to zero or one).

Do not combine many-many rel-ships!

Combining **Drinker** and **Likes** would be a mistake.

It leads to *redundancy*. E.g.,

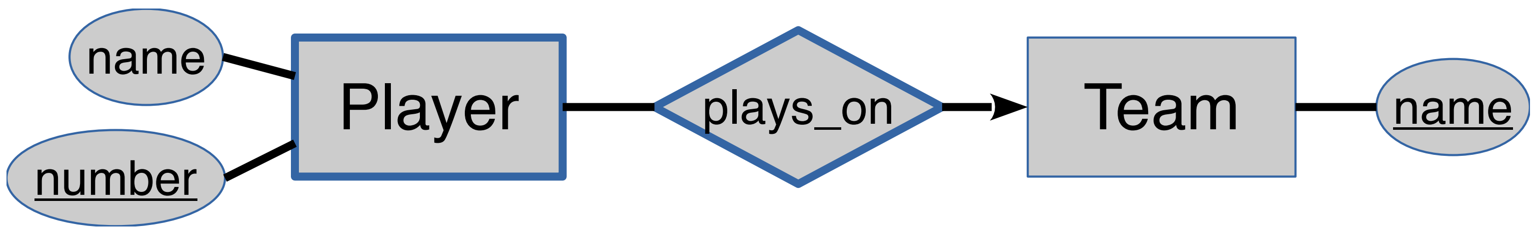
name	**addr**	**beer**
Sally	123 Maple	Bud
Sally	123 Maple	Miller

Plus, would we combine **Likes** with **Drinker** or with **Beer**? It would not make sense.

Handling weak entity sets

- Relation for a *weak entity set* must include attr's for its complete key — including those of the other entity sets upon which it is *weak* — as well as its own, non-key attr's.
- The same applies for *connecting entities*; these simply add no attr's of their own to the key.

Weak entity set ⇒ relation



- **Team(<u>name</u>)**
- **Player(<u>team</u>, <u>number</u>, name)**

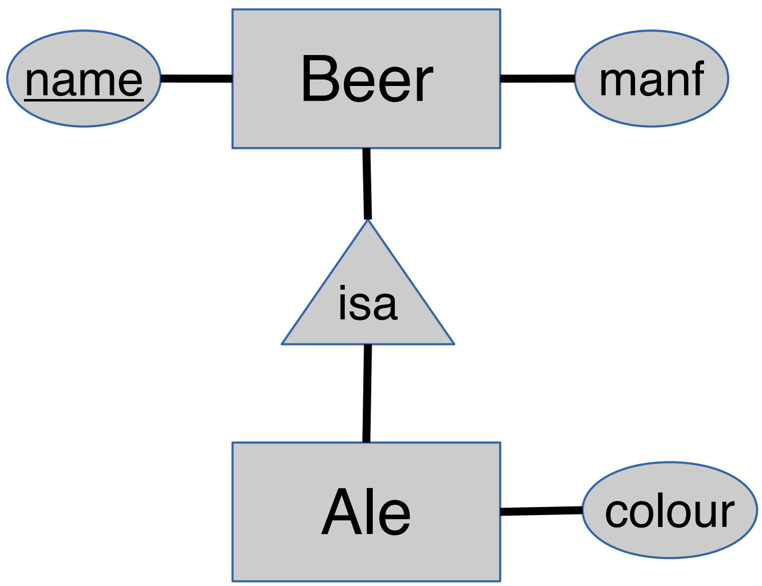
Note this is the same we did for a compulsory *many-one* before.

The addition here is the additional attr's in the key of the relation resulting from the weak entity on the “many“ side of the connectiong rel-ship.

Subclasses: Three approaches

1. **object-oriented**: a relation per *subset* of subclasses, with all relevant attributes.
2. **E/R style**: a relation for each subclass, with
   - key attribute(s)
   - attributes of that subclass
3. **using nulls**: one relation; “entities” have a *null* “value” in attributes that do not belong to them.

Subclass ⇒ relations



Object-oriented

Beer

name	manf
Bud	Anheuser Busch

Ale

name	manf	colour
Summerbrew	Pete's	dark

Good for queries like, “Find the color of ales made by Pete's.”

E/R style

Beer

name	manf
Bud	Anheuser Busch
Summerbrew	Pete's

Ale

name	manf	colour
Summerbrew	Pete's	dark

Good for queries like, “Find all beers (including ales) made by Pete's.”

Using nulls

Beer

name	manf	type	colour
Bud	Anheuser Busch	beer	*null*
Summerbrew	Pete's	ale	dark

Saves space, unless there are *lots* of attr. values that will be *null*.

And only a *single* relation, which is simpler.

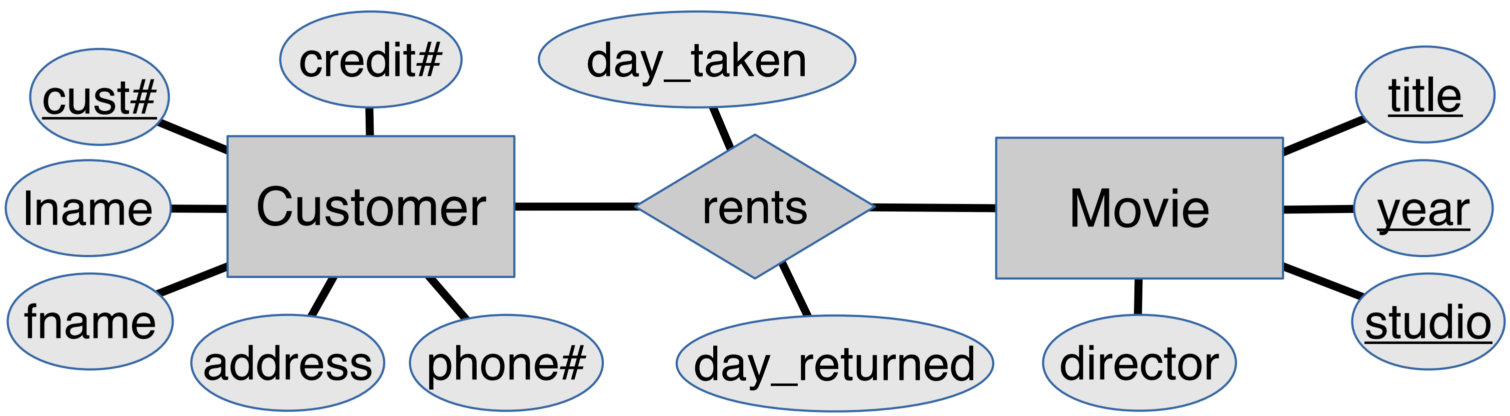
An Example Domain

“Previously”: Parke's Retro Video Shop

A movie rental domain

Scenario: We are running a movie rental store.

- We have *movies*.
- We have *customers*. (We hope!)
- Customers *rent* movies from us.



A good design?

- Any logical problems? Anything missing?
- Can the design accommodate the *questions* and *activities*?
- Can the design violate any real-world constraints?

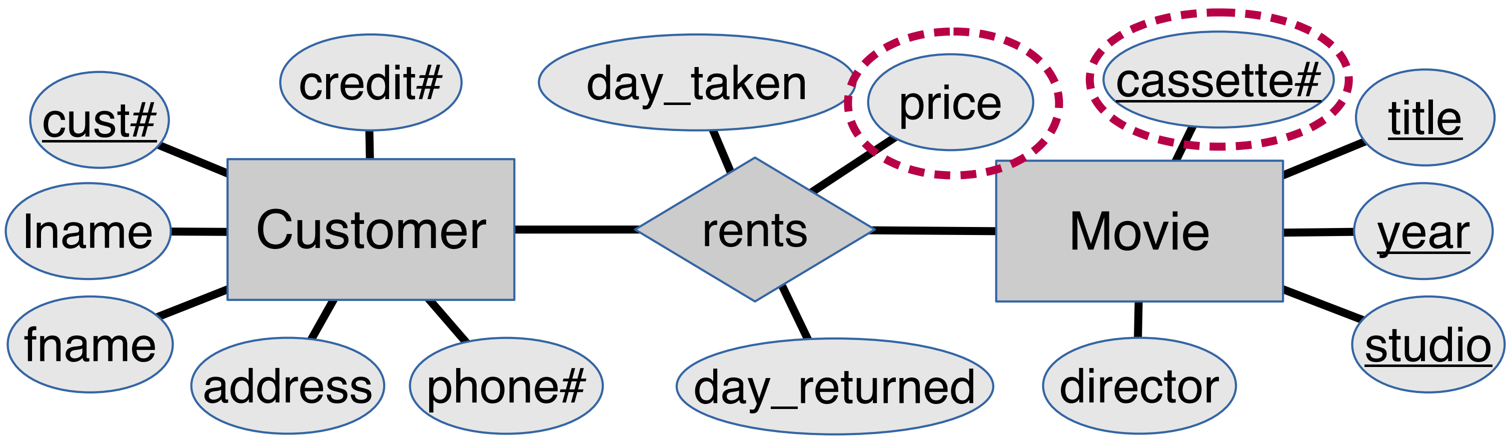
Anything missing?

Can add attr's, rel-ships, or entities.

Problem: multiple copies

We may have several copies of a given movie.

- More than one *copy* might be rented at the same time.
- We need to *know* which customer has which copy.



Attribute vs Entity

When to promote an attr. to an entity?

1. It needs to participate in rel-ships itself.
    - Then it must be an entity.
2. The values of the attr. in its entity would be repeated *often*, or there are other attr's associated with this one. (**Brevity**)

    E.g., **Professor** has *office#*.

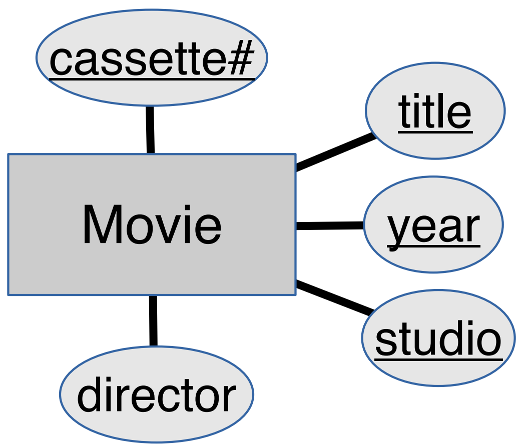
    - It probably should be an entity.
3. The values that the attr. may take are restricted. (*strong typing*)
    - Can accomplish this by making it an entity.

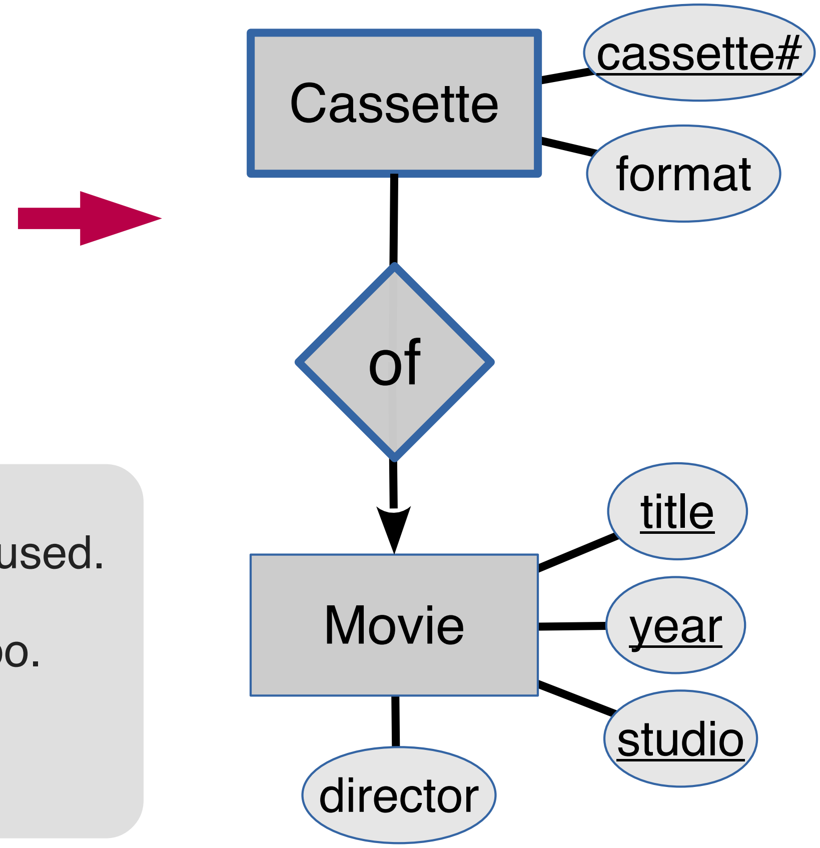
Movies and *copies*

- The information about a movie is *repeated* for as many copies of that movie we have. This seems awkward…
  - Soon, we will formalize *why* this *redundancy* is actually problematic, and not just inelegant.
  - What if we want to *catalog* a movie, but we have no copies yet?
- To fix this, we separate the notion of a *copy of a movie* — call this a *cassette* — and the *movie itself* (the “listing” of the movie).

  So, we add an entity **Cassette**.

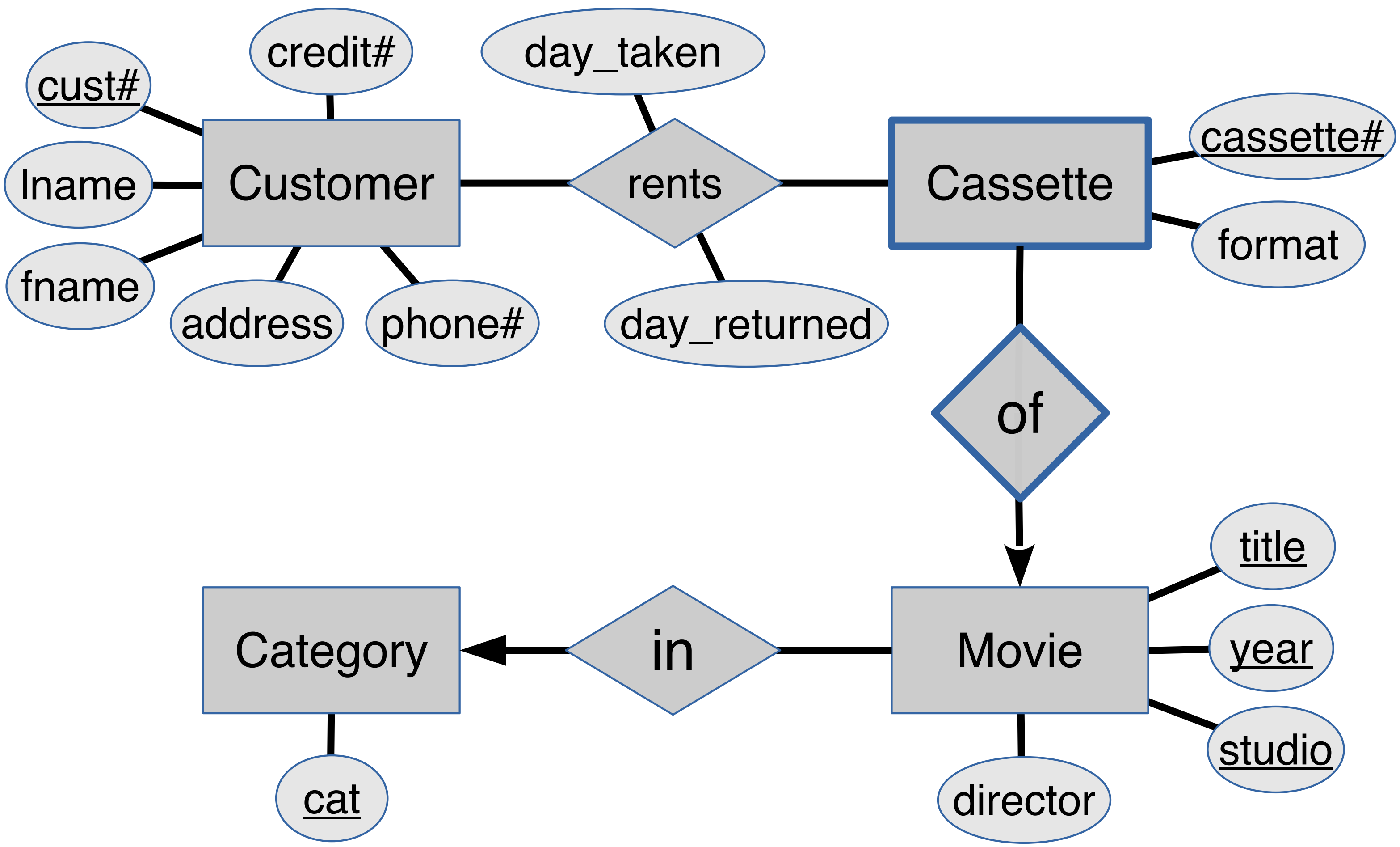
Cassettes





- A *weak entity* for **Cassette** can be used.
- But we could use a regular entity too.
- What are the differences?

A better design?



A better design? (2)

- I also added an entity **Category** for movies (e.g., *comedy*, *drama*, *scifi*).

  This strongly types the *category* a movie can be in.

Are we done yet? Can we stop?

Not even close!

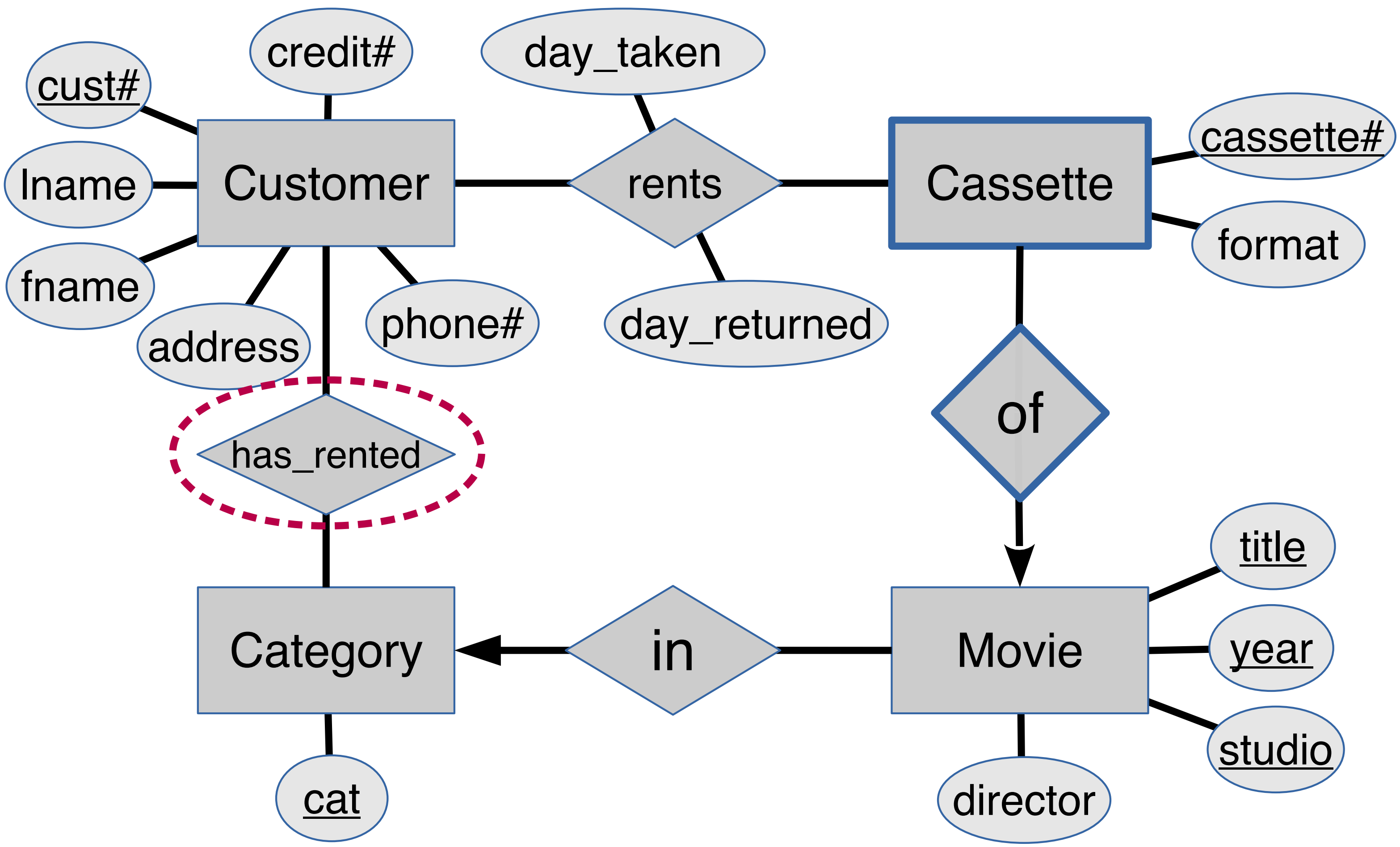
New Relationships

Careful for redundancy!

A requirement for our movie rental database is that we can determine *which* categories of movies a customer has rented (for promotions and such).

Q. Have we overlooked this?

Adding “has rented”…



That's redundant!

No! We had not overlooked this. Which categories a customer has rented is *derivable* from which movies he or she has rented.

Thus, the rel-ship **has_rented** would be *redundant*. It represents a new rel-ship that could be populated with completely *unrelated* data.

So, **has_rented** should *not* be added.

New rel-ships

But if not redundant…

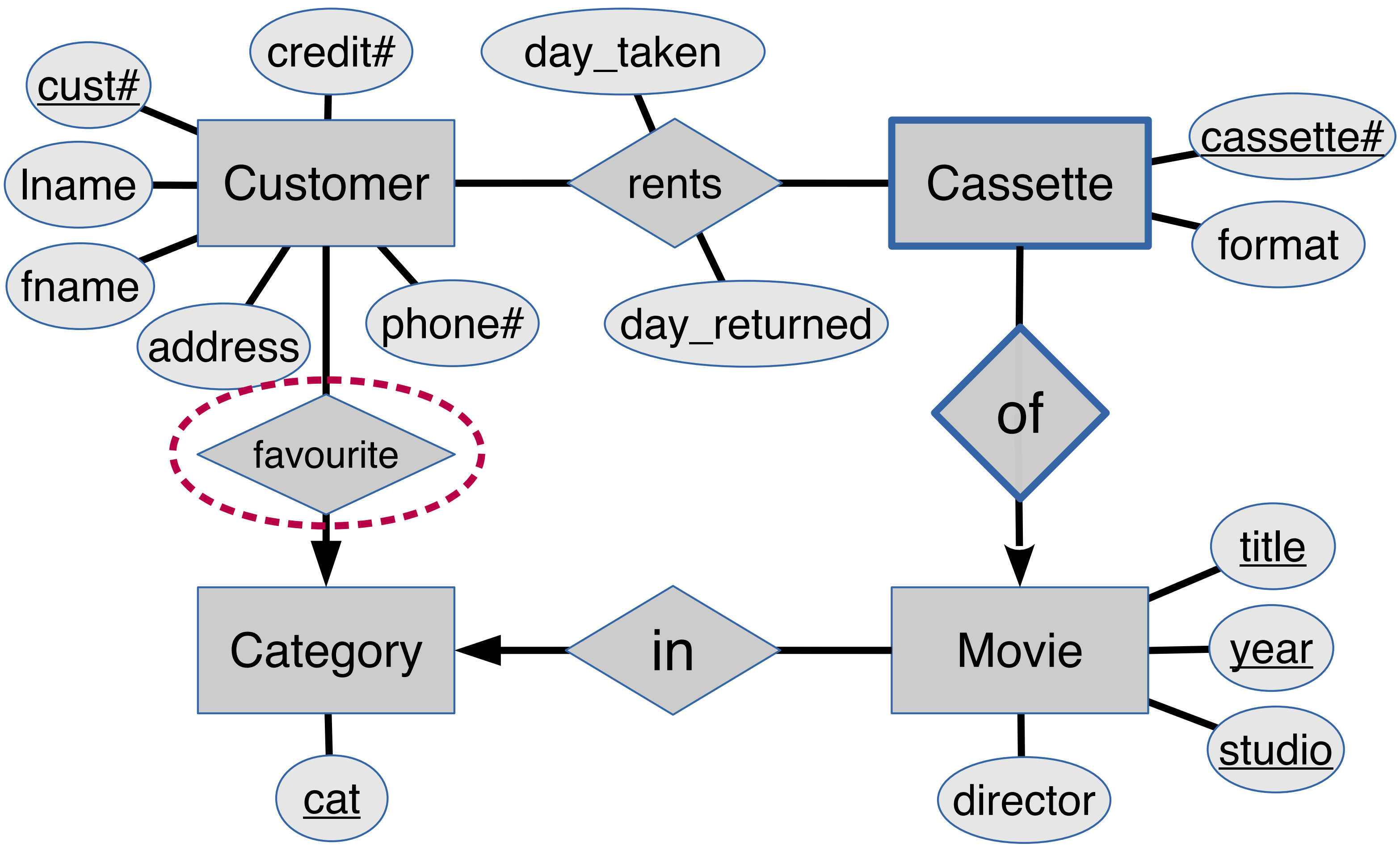
However, if a new rel-ship represents *new* information which is

- *not derivable* from what we already have, and
- which is needed for our intended *queries* and *applications*

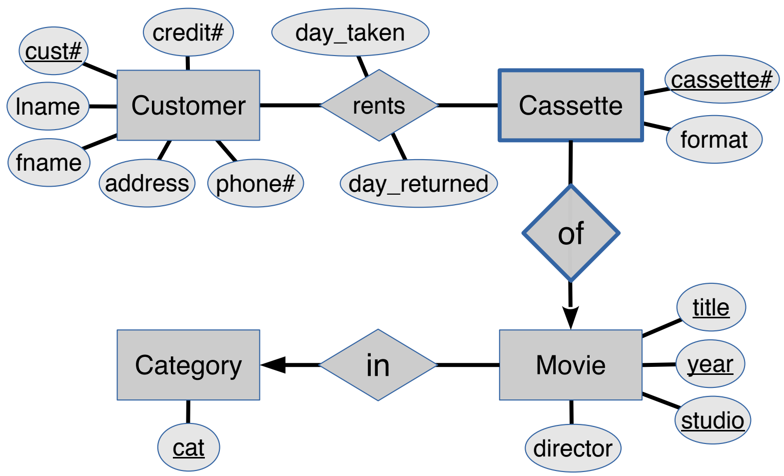
then we should added it.

For example, say we want to know a customer's favourite category (e.g., *romance*) for promotional purposes.

Favourite category



Ambiguity / missing “logic”

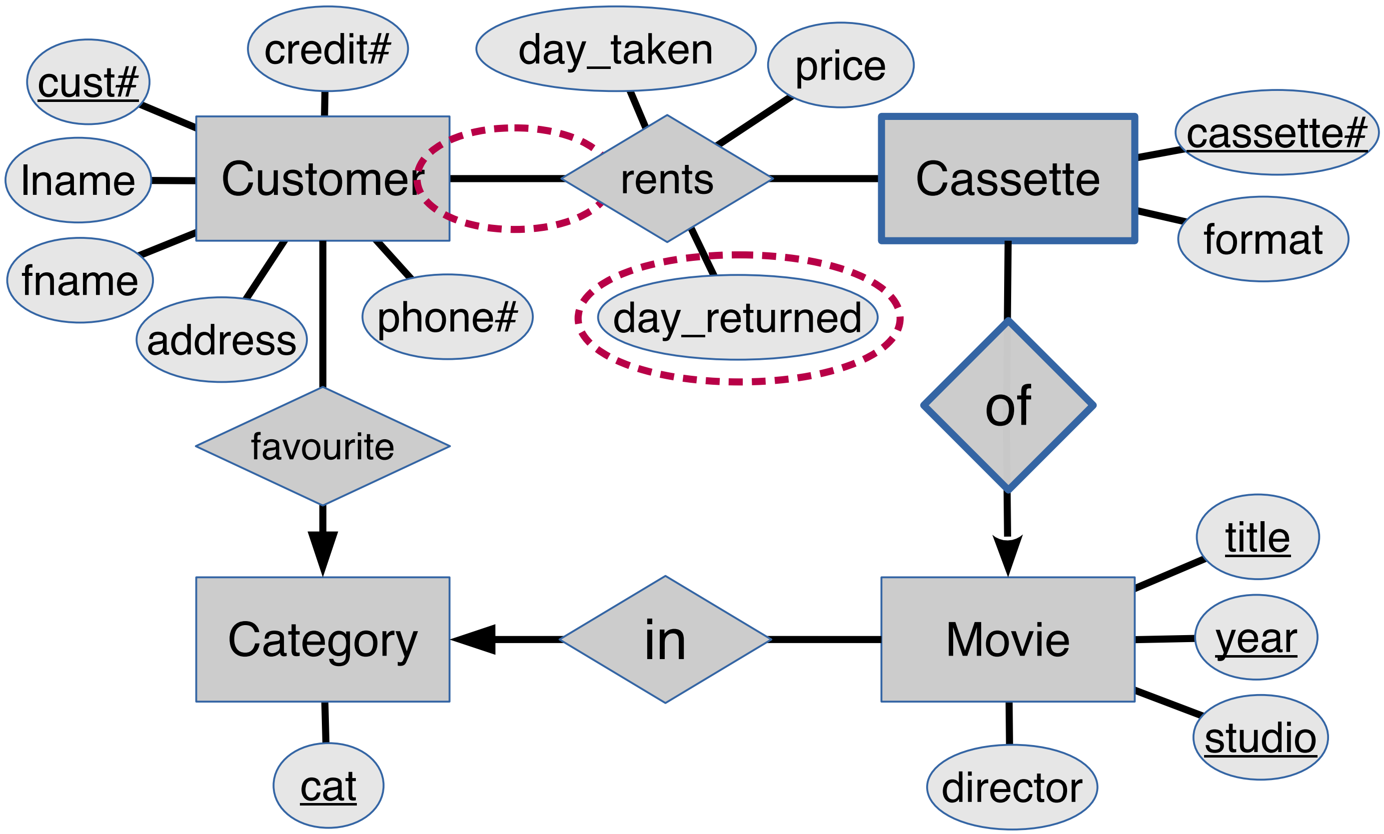


Missing

- A given copy cannot be rented at the same time by two or more customers.
- A given customer may rent the same copy of a movie more than one time.

Which do we miss? I don't know, but we do miss at least one of these!

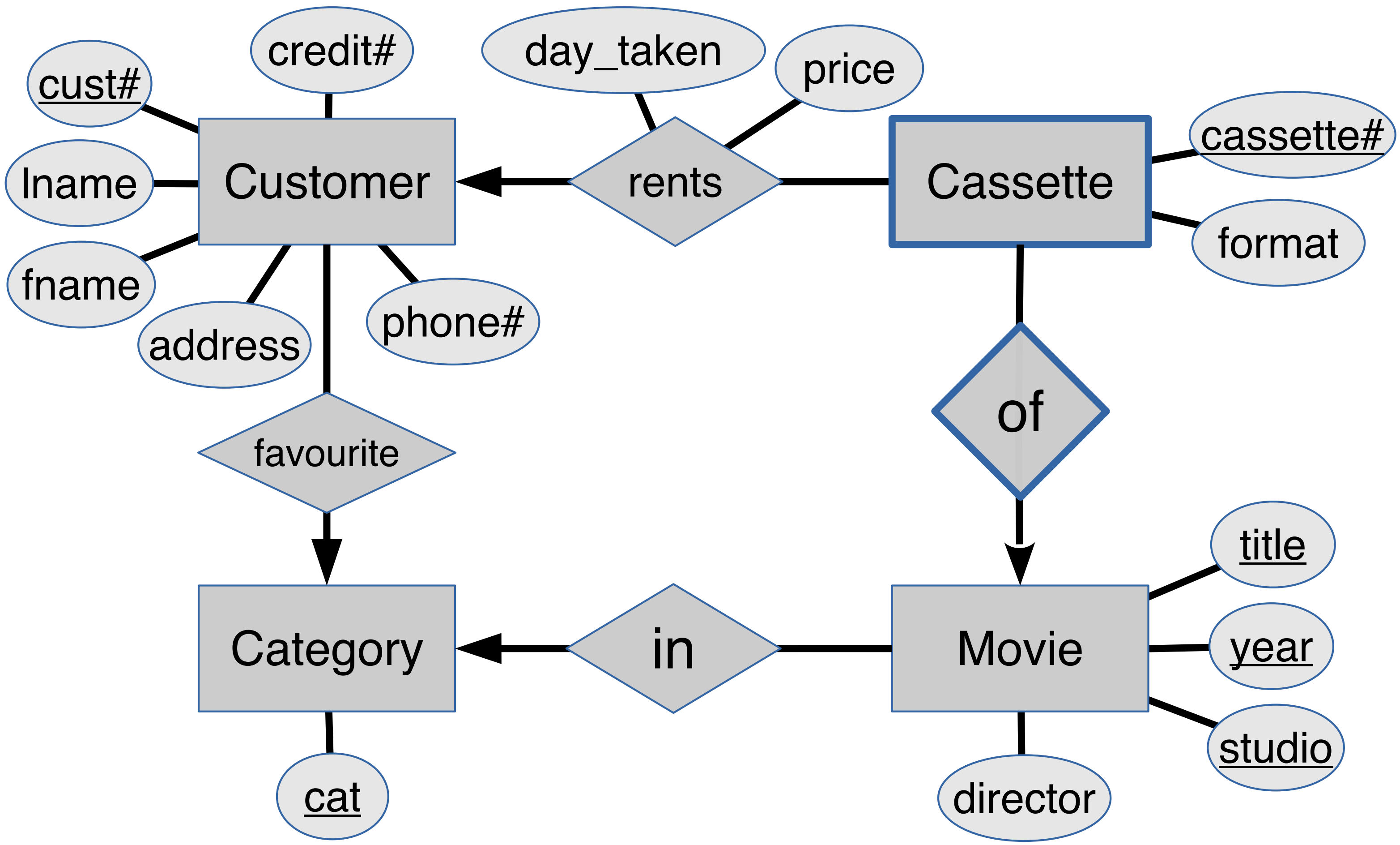
One needs changing!



Which to change?

Interpretation #1

Only track movies that are out



Interpretation #2

Record all rentals forever

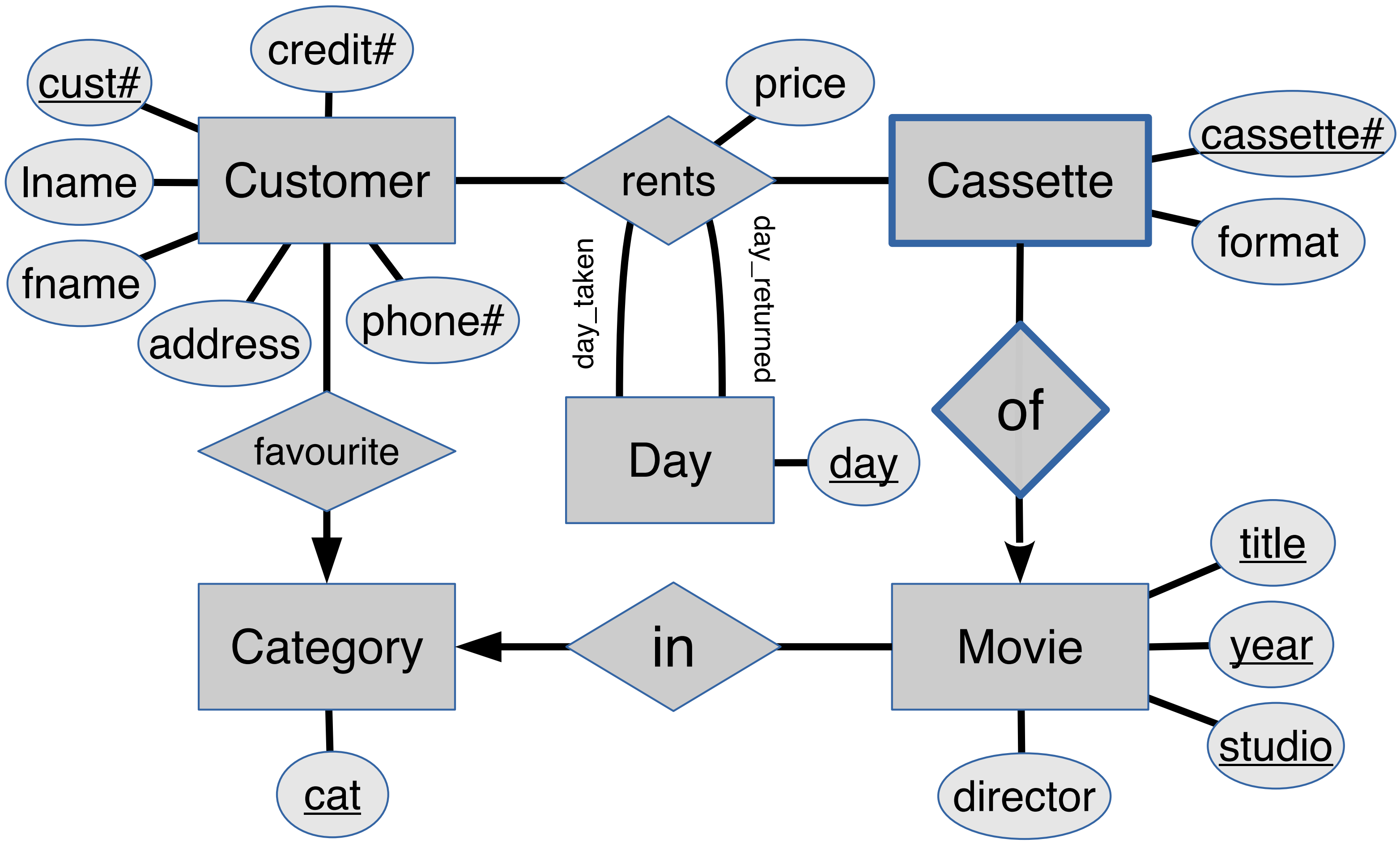
Once a rental is recorded in the database, it stays.

First Problem. A customer may rent (the same copy of) a movie more than one time. The rel-ship **rents** does not allow for this!

Solutions?

1. Have more entities participate in **rents** to fix this. (*a ternary rel-ship*)
2. Promote **rents** to be an entity. (**Rental**)

Ternary rel-ship attempt #1



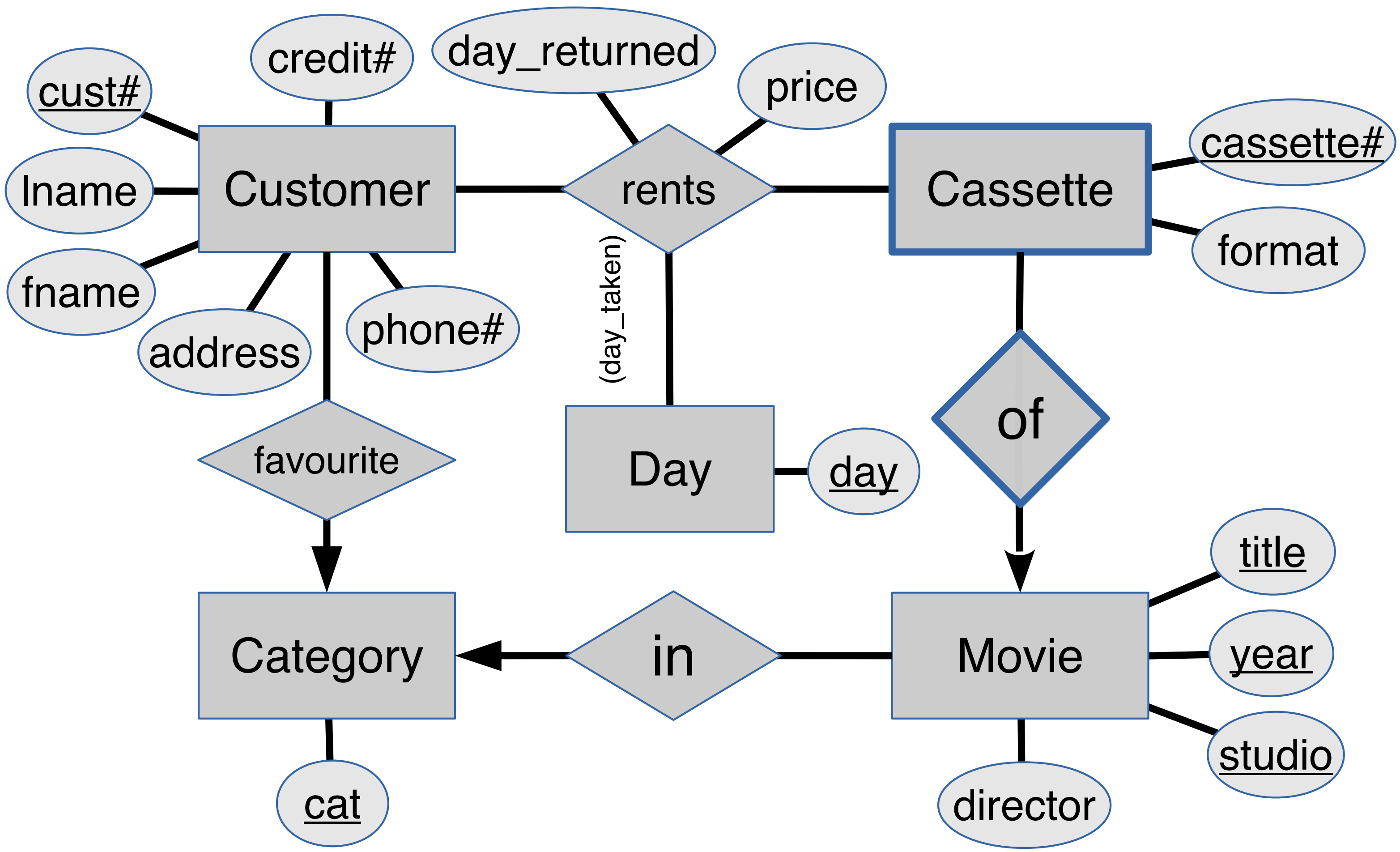
Ternary rel-ship attempt #1 (2)

Problems

1. Should **Day** be an entity? Do we really want to "store" days explicitly? (We wouldn't *actually* have to. But, still seems awkward!)
2. Rel-ship **rents** is **Customer** × **Cassette** × **Day** (*day_taken*) × **Day** (*day_returned*). But is *day_returned always* filled in? **No!**

This does *not* work because of #2.

Ternary rel-ship attempt #2



Ternary rel-ship attempt #2 (2)

This fixes the problematic — **fatal**! — aspect that the *returned day* is not known for any copies that are currently rented out by making it an attribute again.

But two customers still can rent the same copy of a movie on the same day!

- Why did the previous design (Ternary #1) not handle this?
- Why does this design (Ternary #2) not handle this?
- Can you fix it?

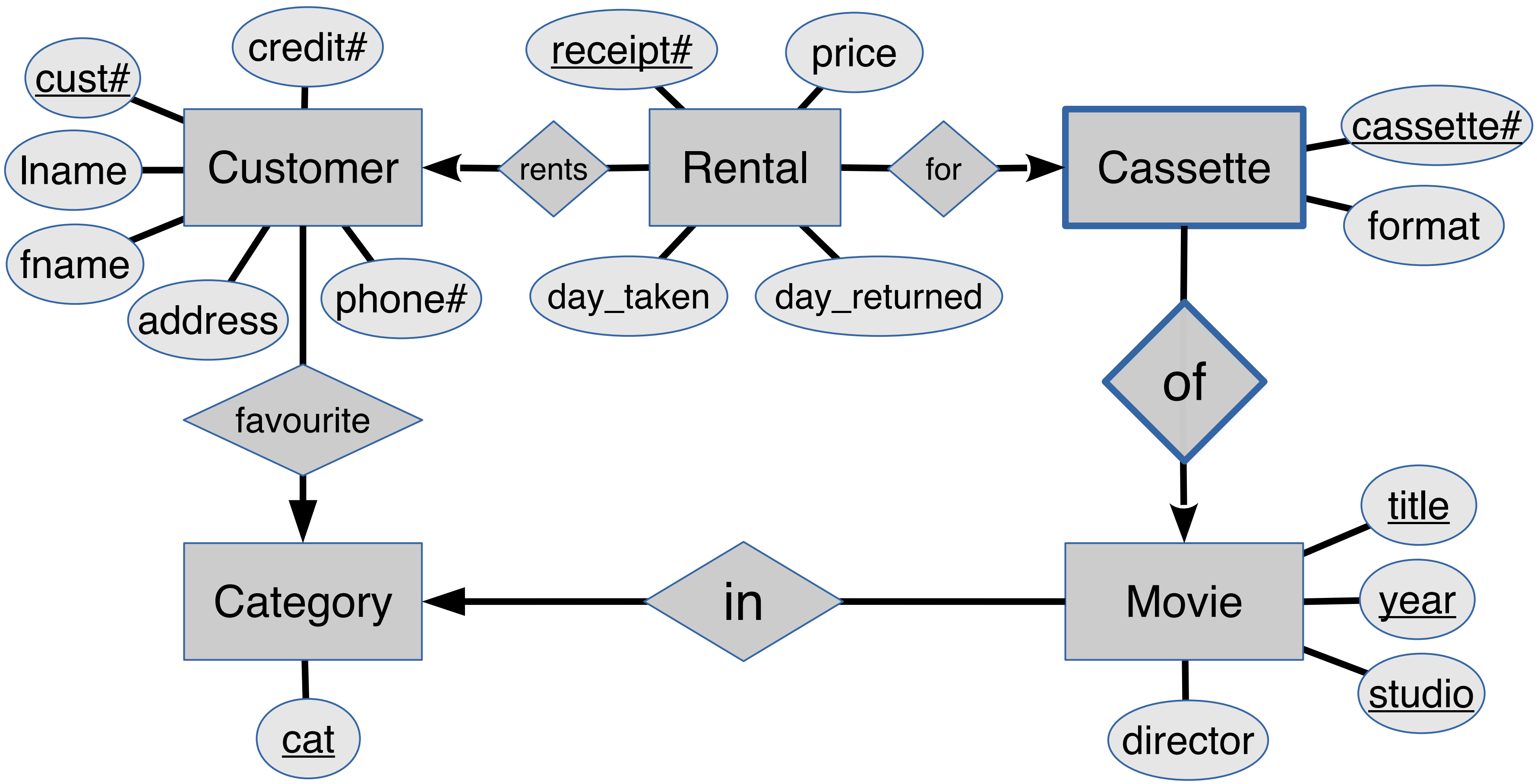
And still, overall, an awkward design anyway.

Rel-ship vs Entity

When to promote a rel-ship to an entity?

1. When the “key” of the rel-ship is too restrictive
    - Can promote the rel-ship to be an entity instead. Then it has a key.
    - *Or* the rel-ship should involve more entities so that its “key” is adequate. (Like in the *ternary* examples above.)
2. When it seems we need this rel-ship to be involved itself in other rel-ships.
    - Promote the rel-ship to be a *connecting* entity. Then it can participate in rel-ships with other entities.

Rental: Entity solution #1



Rental: Entity solution #1 (2)

One problem down, one remains

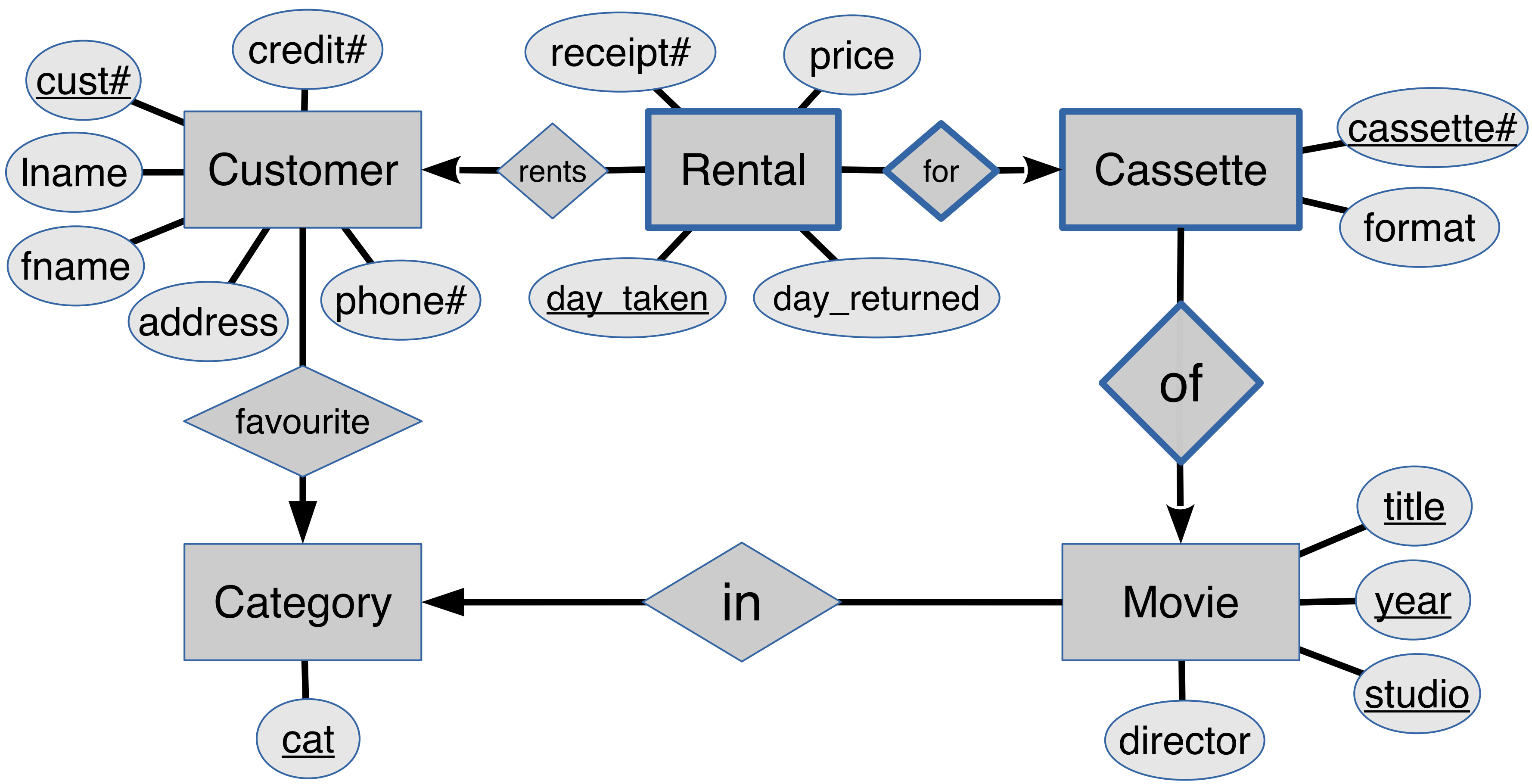
Handles *Problem #1*: A customer can now rent the same cassette a second time.

Still does not handle *Problem #2*: Two customers may rent the same cassette at the same time!

How to fix this second problem?

Rental: Entity solution #2

Better keys



Rental: Entity solution #2 (2)

Handles both problems now! (*Caveat*: Two customers cannot *take out* the same cassette on the same day.)

Can an entity have more than one key?

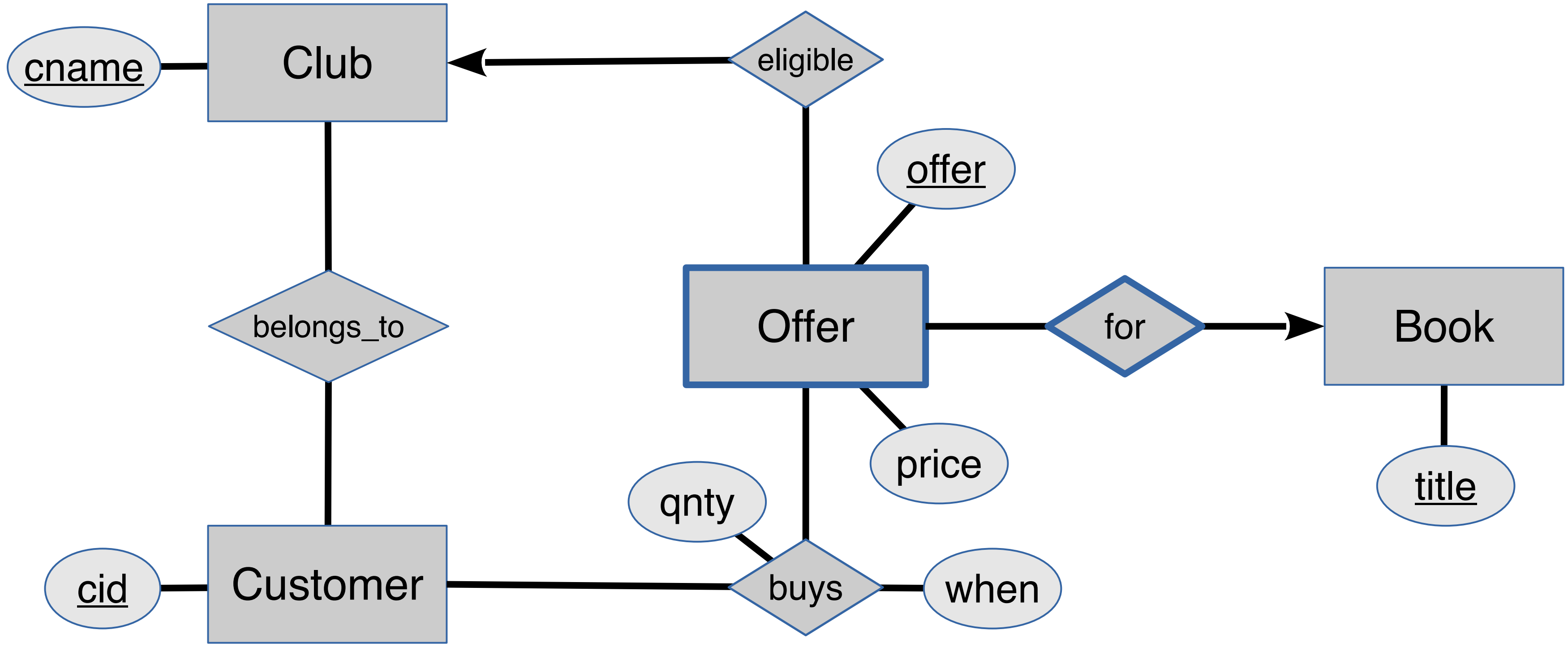
Of course!

We certainly could make **receipt#** a *second* key of the **Rental** table (relation) in our relational schema.

An Example Domain #2

StLoB: Saint Lawrence Online Bookshop

A online book purchase domain

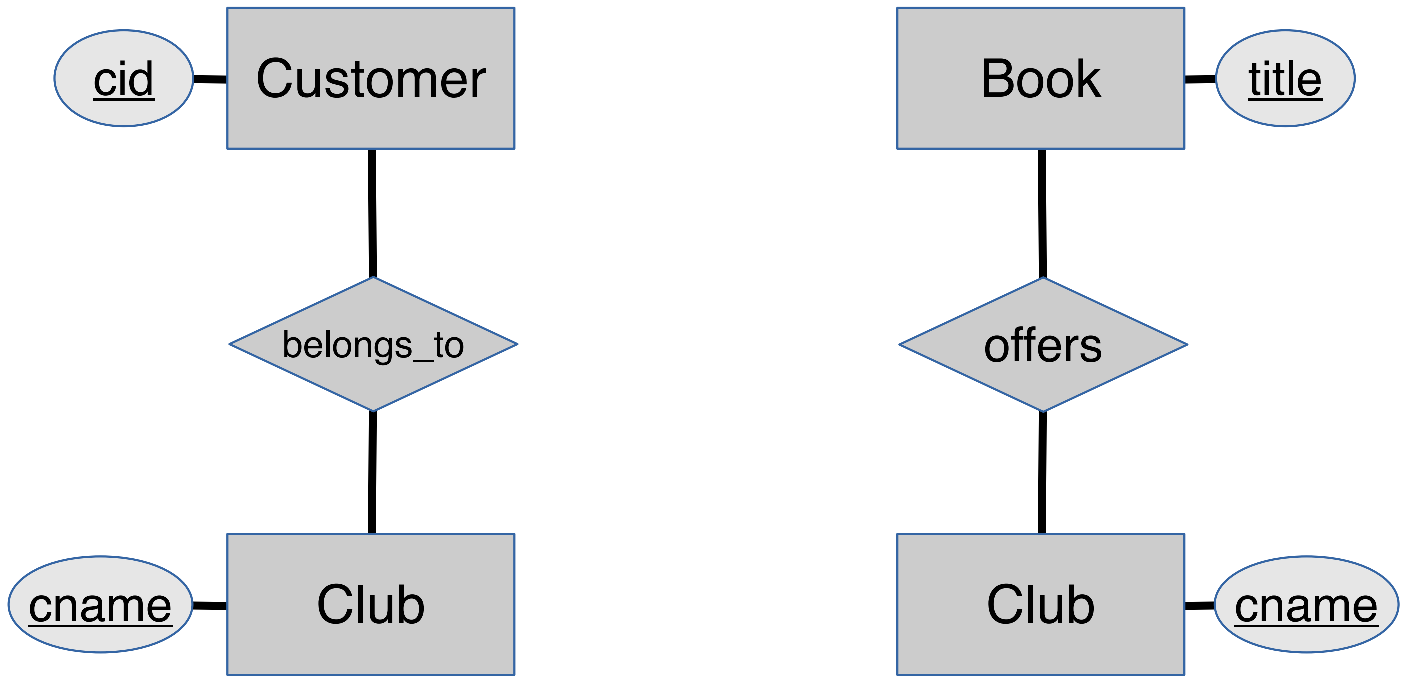


- **Gimmick**:
  - Customers belong to clubs.
  - Via clubs, there are offers on books.

Problems with our initial design?

1. Can a customer buy a book via an offer with a club a *second* time?
    - **No**.
2. Is *every* book *offered* by *every* club?
    - **No**. This does not have to be the case.
    - But say we do not want this.
3. Are we ensured that, when a customer *buys* a book via an offer, that he or she belongs to the club that the offer is through?
    - **No**.
    - This is my business model! I want to enforce this.

Back to the fundamentals



- **Customers** *belong to* **Clubs**
- **Books** *are offered by* **Clubs**

“buys”

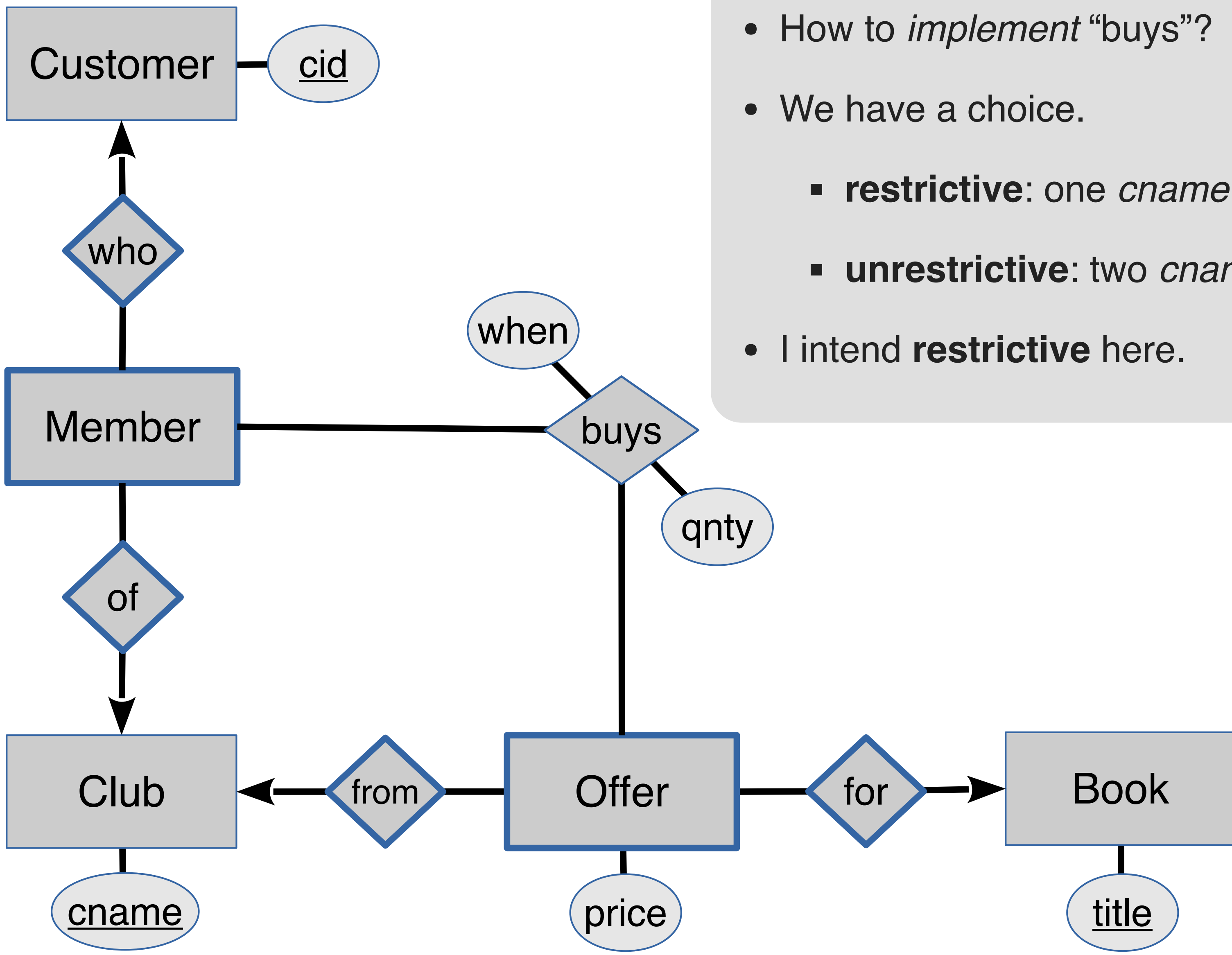
We need to *relate* — a rel-ship — *belongs_to* and *offers*.

But that is not legal in E/R!

Okay… Let's promote them to entities, then!

- *belongs_to* ⇒ **Member**
- *offers* ⇒ **Offer**

And put it all together…



- How to *implement* “buys”?
- We have a choice.
  - **restrictive**: one *cname*
  - **unrestrictive**: two *cnames*
- I intend **restrictive** here.

To relational

- **Customer(cid)**
- **Club(cname)**
- **Member(cid, cname)**
- **Book(title)**
- **Offer(cname, title, price)**
- **Buys(cid, title, cname, when, qnty)**

Foreign keys: Add §7.1.1 & §7.1.2 (pp.313–315) to your reading!

To relational w/ foreign keys

- Customer(<u>cid</u>)
- Club(<u>cname</u>)
- Member(<u>cid</u>, <u>cname</u>)
  - FK (cid) refs Customer
  - FK (cname) refs Club
- Book(<u>title</u>)
- Offer(<u>cname</u>, <u>title</u>, price)
  - FK (cname) refs Club
  - FK (title) refs Book
- Buys(<u>cid</u>, <u>title</u>, <u>cname</u>, when, qnty)
  - FK (cid, cname) refs Member
  - FK (cname, title) refs Offer