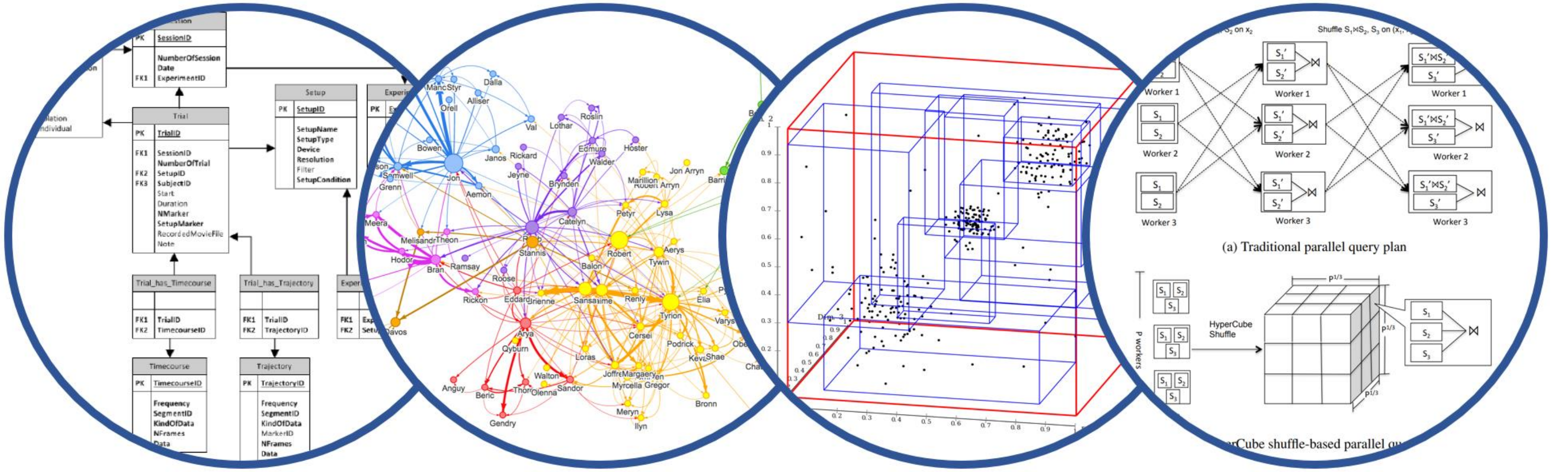


Introduction to Data Management

Database Design

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Recap: Relational Algebra

- SQL: declarative language; we say **what**
- RA: an algebra for saying **how**
- Optimizer converts SQL to RA

Recap: Relational Algebra

1. Selection $\sigma_{\text{condition}}(S)$
2. Projection $\Pi_{\text{attrs}}(S)$
3. Join $\mathrm{R} \bowtie_{\theta} S = \sigma_{\theta}(\mathrm{R} \times S)$
4. Union $\cup$
5. Set difference $-$

Rename ρ

Recap: Relational Algebra

1. Selection $\sigma_{\text{condition}}(S)$
2. Projection $\Pi_{\text{attrs}}(S)$
3. Join $\mathrm{R} \bowtie_{\theta} \mathrm{S} = \sigma_{\theta}(\mathrm{R} \times \mathrm{S})$
4. Union $\cup$

Monotone

5. Set difference $-$

Non-monotone

Rename ρ

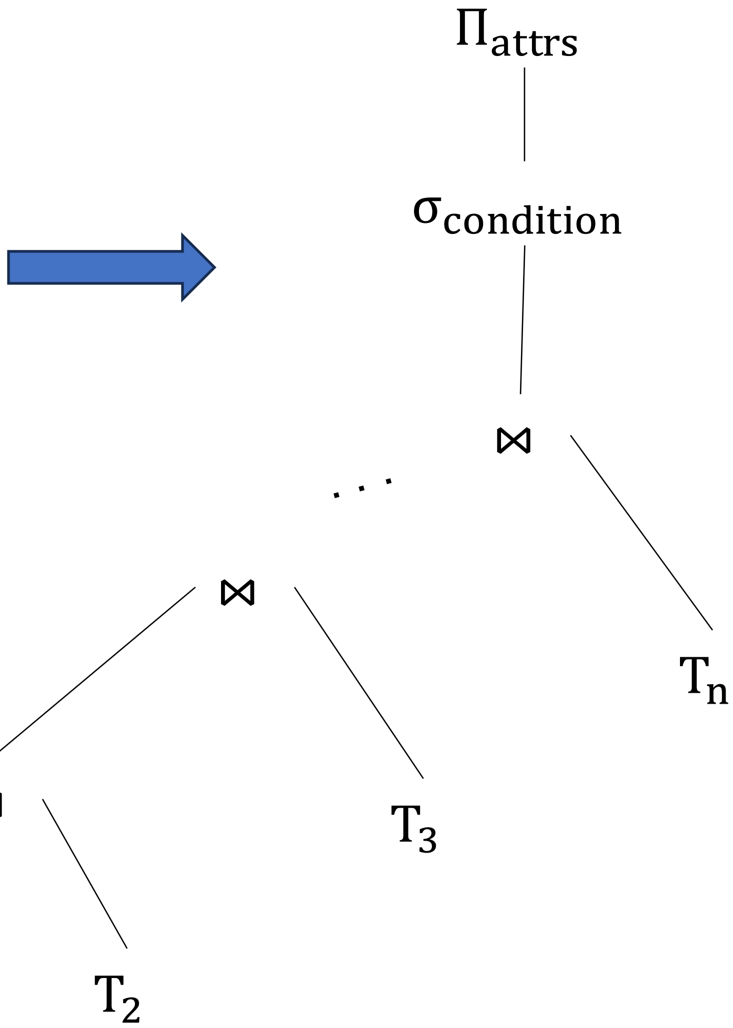
Monotone, but doesn’t do anything

Simple SQL to RA

SQL to RA

Single SELECT-FROM-WHERE query:

SELECT attrs
FROM $T_1, T_2, \ldots, T_n$
WHERE condition;



Π_{attrs}

$\sigma_{\text{condition}}$

$\bowtie$

$\ldots$

$\bowtie$

T_n

$\bowtie$

T_3

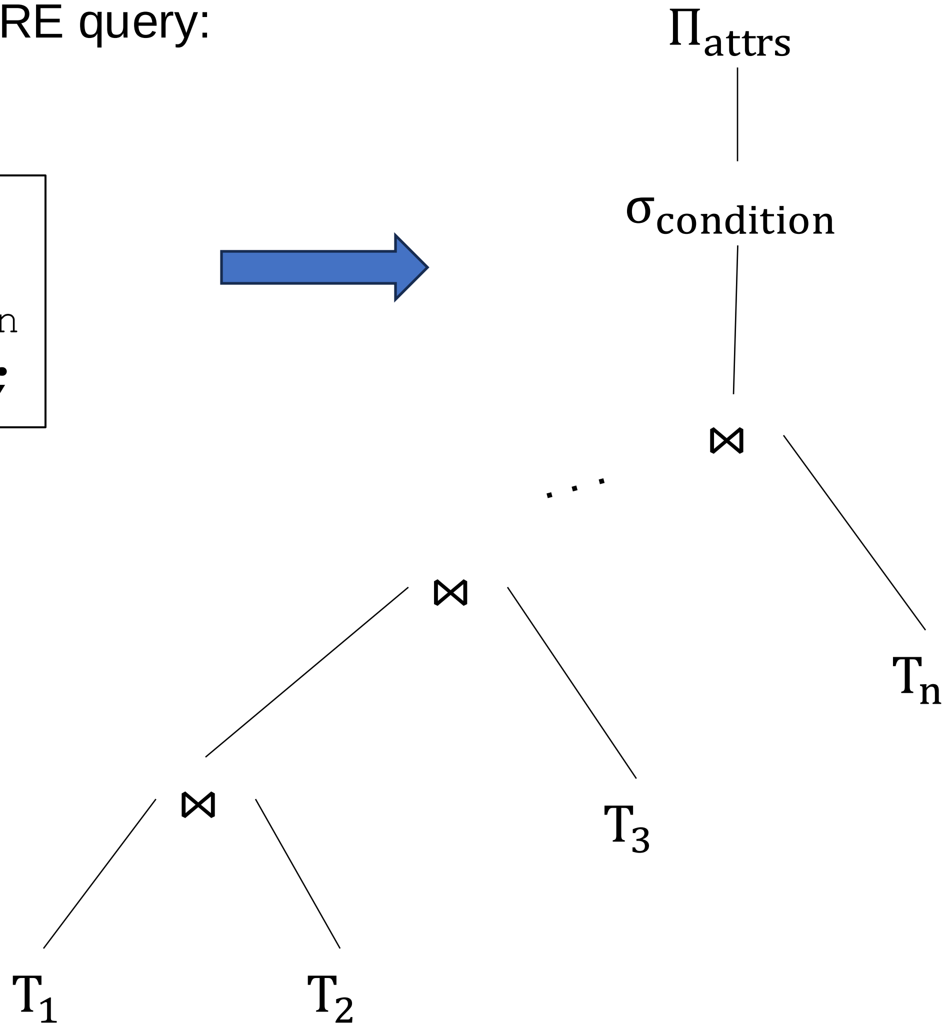
T_1

T_2

SQL to RA

Single SELECT-FROM-WHERE query:

```
SELECT attrs
FROM T1,T2,...,Tn
WHERE condition;
```



Next: to convert group-by we need to extend RA

Extended Relational Algebra

- Duplicate elimination δ
- Group-by aggregate $\gamma_{\mathrm{attr1,attr2,\ldots,agg1,\ldots}}$

Duplicate Elimination

$$\delta(\mathrm{T})$$

Eliminates duplicates from the bag T

```
SELECT DISTINCT *
FROM T;
```

Duplicate Elimination

$$\delta(\mathrm{T})$$

Eliminates duplicates from the bag T

```
SELECT DISTINCT *
FROM T;
```

$$\delta(\mathrm{R}) =$$

R

A	B
1	10
2	10
2	10
2	20
1	10

Duplicate Elimination

$$\delta(\mathrm{T})$$

Eliminates duplicates from the bag T

```
SELECT DISTINCT *
FROM T;
```

$$\delta(\mathrm{R}) =$$

A	B
1	10
2	10
2	20

R

A	B
1	10
2	10
2	10
2	20
1	10

GroupBy-Aggregate

$$\gamma_{attr1,attr2,\ldots,agg1,\ldots}(\mathrm{T})$$

Group-by, then aggregate

```
SELECT attr1,...,agg1,...
FROM T
GROUP BY attr1,...;
```

GroupBy-Aggregate

$$\gamma_{attr1,attr2,\ldots,agg1,\ldots}(\mathrm{T})$$

Group-by, then aggregate

$$\gamma_{\mathrm{Job},\mathrm{avg(Salary)}\rightarrow \mathrm{S}}(\mathrm{Payroll}) =$$

```
SELECT attr1,...,agg1,...
FROM T
GROUP BY attr1,...;
```

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

GroupBy-Aggregate

$$\gamma_{attr1,attr2,\ldots,agg1,\ldots}(\mathrm{T})$$

Group-by, then aggregate

```
SELECT attr1,...,agg1,...
FROM T
GROUP BY attr1,...;
```

$$\gamma_{\mathrm{Job,avg(Salary)}\to \mathrm{S}}(\mathrm{Payroll}) =$$

Job	S
TA	55000
Prof	95000

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

GroupBy-Aggregate

No need for a HAVING operator!

Find all jobs where the average salary of employees earning over 55000 is < 70000

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

GroupBy-Aggregate

No need for a HAVING operator!

Find all jobs where the average salary of employees earning over 55000 is < 70000

```
SELECT Job
FROM Payroll
WHERE Salary > 55000
GROUP BY Job
HAVING avg(Salary)<70000;
```

Payroll

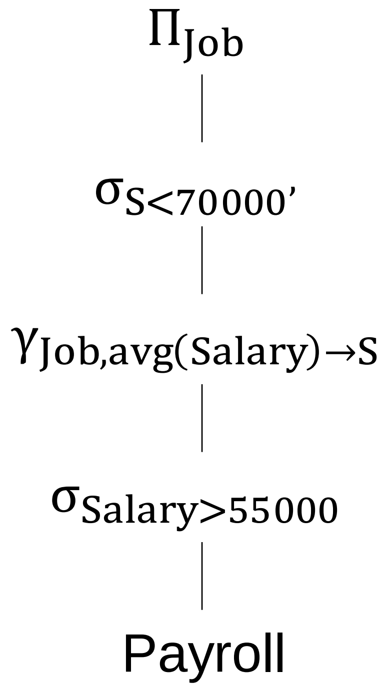
UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

GroupBy-Aggregate

No need for a HAVING operator!

Find all jobs where the average salary of employees earning over 55000 is < 70000

```
SELECT Job
FROM Payroll
WHERE Salary > 55000
GROUP BY Job
HAVING avg(Salary)<70000;
```



Π_{Job}

$\sigma_{\text{S<70000'}}$

$\gamma_{\text{Job,avg(Salary)}\rightarrow \text{S}}$

$\sigma_{\text{Salary>55000}}$

Payroll

Payroll

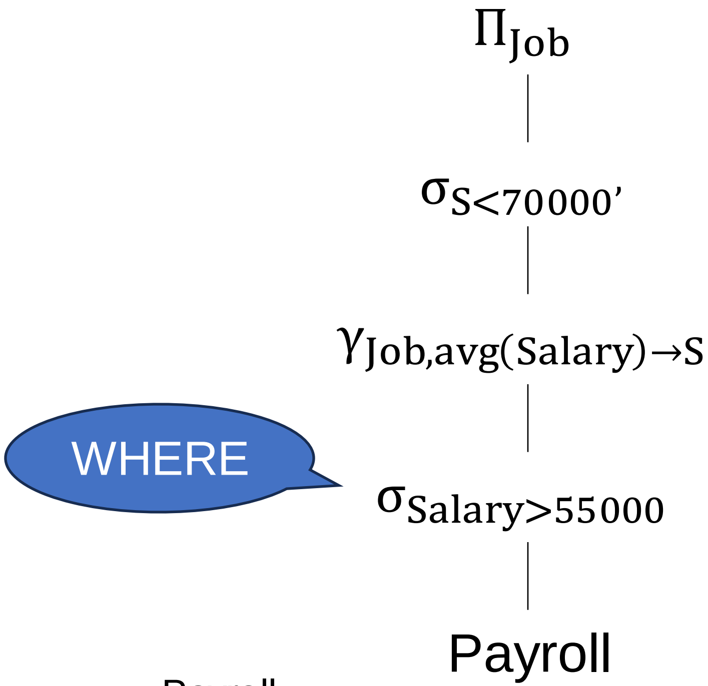
UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

GroupBy-Aggregate

No need for a HAVING operator!

Find all jobs where the average salary of employees earning over 55000 is < 70000

```
SELECT Job
FROM Payroll
WHERE Salary > 55000
GROUP BY Job
HAVING avg(Salary)<70000;
```



Π_{Job}

$\sigma_{S<70000'}$

$\gamma_{Job, avg(Salary) \to S}$

WHERE

$\sigma_{Salary>55000}$

Payroll

Payroll

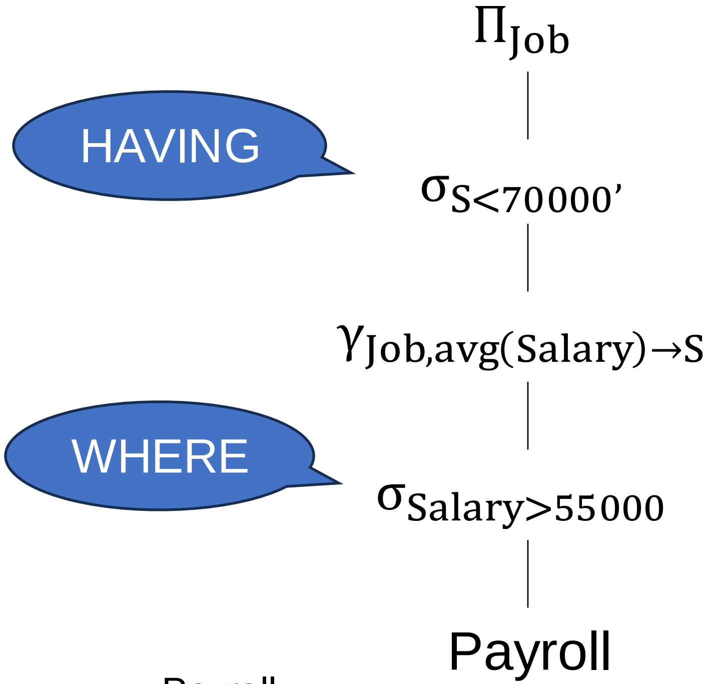
UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

GroupBy-Aggregate

No need for a HAVING operator!

Find all jobs where the average salary of employees earning over 55000 is < 70000

```
SELECT Job
FROM Payroll
WHERE Salary > 55000
GROUP BY Job
HAVING avg(Salary)<70000;
```



Π_{Job}

HAVING

$\sigma_{S<70000'}$

$\gamma_{\text{Job,avg(Salary)}\rightarrow S}$

WHERE

$\sigma_{\text{Salary}>55000}$

Payroll

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Discussion

The Greek alphabet soup:

- $\sigma, \Pi, \delta, \gamma$
- They are standard RA symbols, get used to them

Next: converting nested SQL queries to RA

Nested SQL to RA

Nested Queries to RA

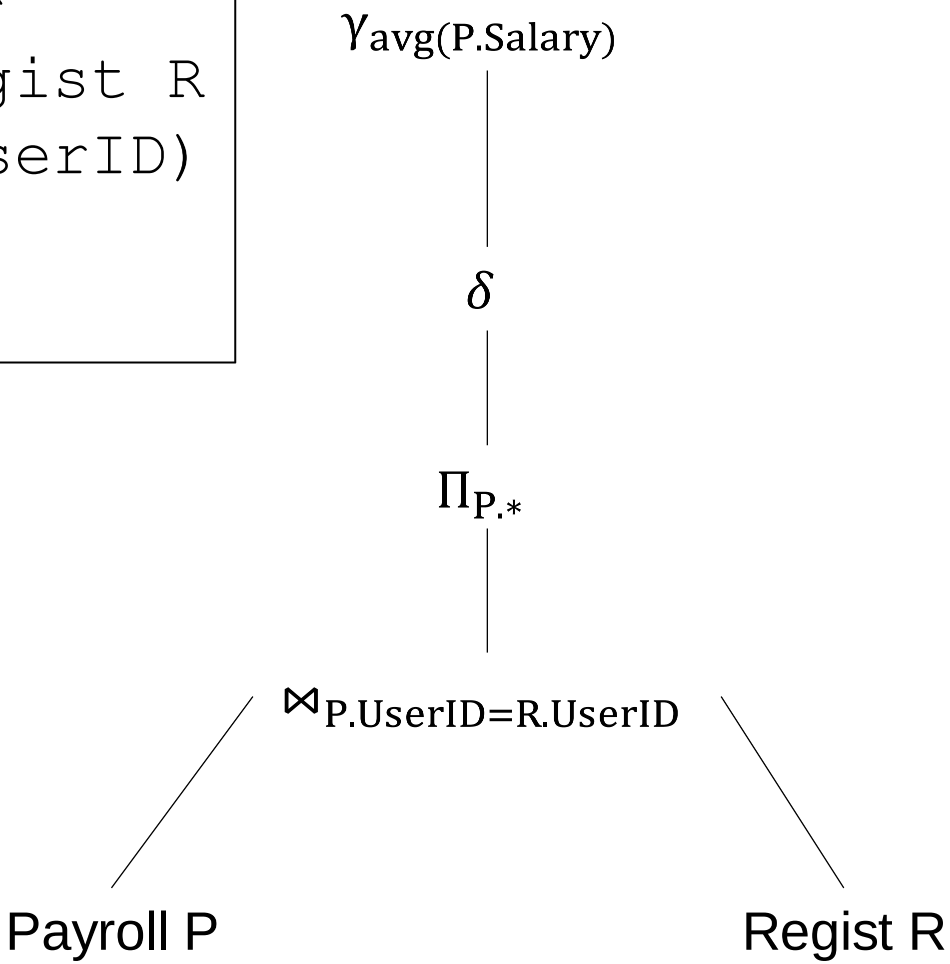
- RA is an algebra: has no nested expressions
- We cannot write EXISTS or NOT EXISTS in σ
- First unnest SQL query, then convert to RA

A Simple Case: the WITH Clause

```
WITH Cardrivers AS
   (SELECT DISTINCT P.*
    FROM Payroll P, Regist R
    WHERE P.UserId=R.UserID)
SELECT avg(Salary)
FROM Cardrivers;
```

A Simple Case: the WITH Clause

```
WITH Cardrivers AS
   (SELECT DISTINCT P.*
    FROM Payroll P, Regist R
    WHERE P.UserId=R.UserID)
SELECT avg(Salary)
FROM Cardrivers;
```



$\gamma_{\text{avg(P.Salary)}}$

δ

$\Pi_{\text{P.*}}$

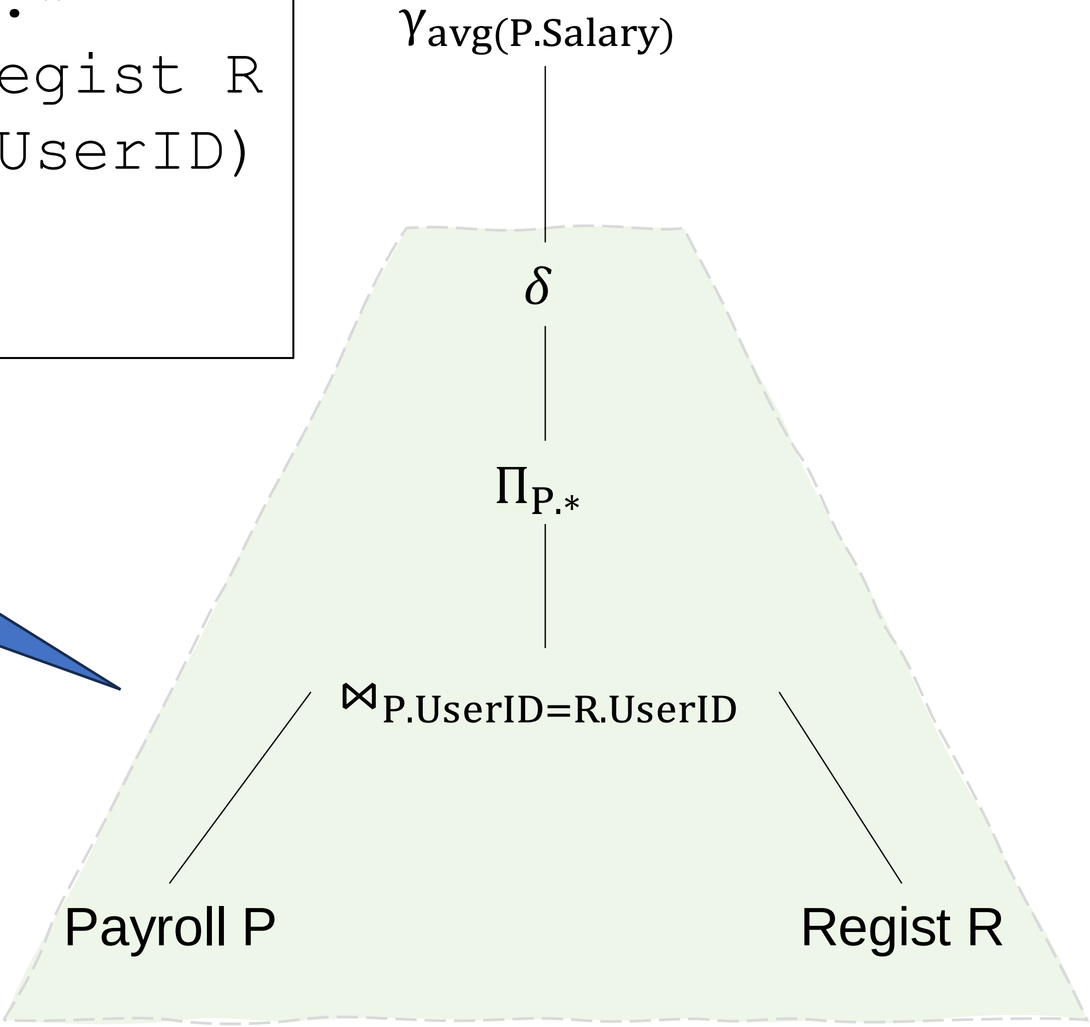
$\bowtie_{\text{P.UserID=R.UserID}}$

Payroll P

Regist R

A Simple Case: the WITH Clause

```
WITH Cardrivers AS
   (SELECT DISTINCT P.*
    FROM Payroll P, Regist R
    WHERE P.UserId=R.UserID)
SELECT avg(Salary)
FROM Cardrivers;
```



$\gamma_{avg(P.Salary)}$

δ

$\Pi_{P.*}$

$\bowtie_{P.UserID=R.UserID}$

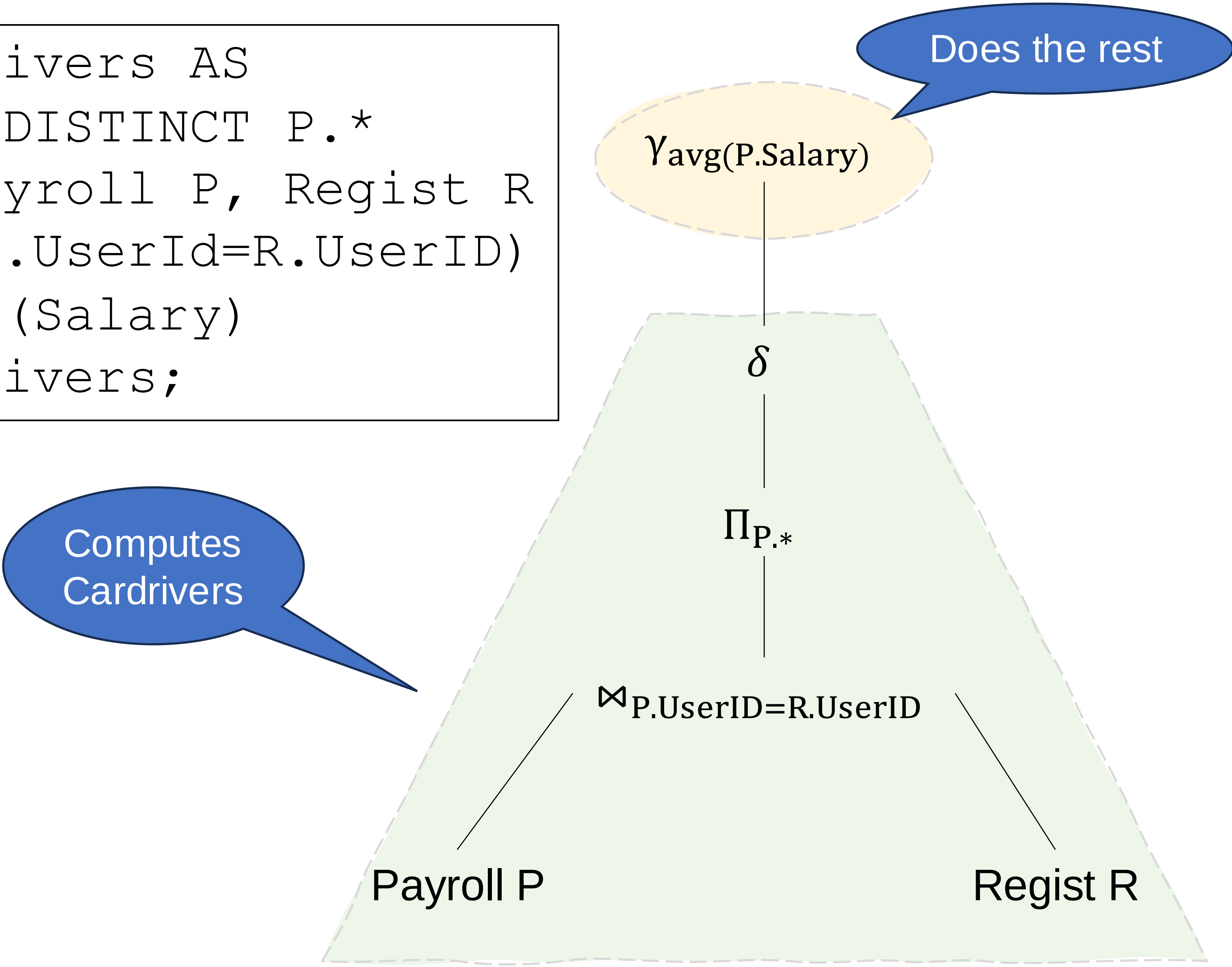
Payroll P

Regist R

Computes Cardrivers

A Simple Case: the WITH Clause

```
WITH Cardrivers AS
   (SELECT DISTINCT P.*
    FROM Payroll P, Regist R
    WHERE P.UserId=R.UserID)
SELECT avg(Salary)
FROM Cardrivers;
```



Does the rest

$\gamma_{avg(P.Salary)}$

δ

$\Pi_{P.*}$

$\bowtie_{P.UserID=R.UserID}$

Payroll P

Regist R

Computes Cardrivers

A Simple Case: a Monotone Query

```
SELECT P.UserID, P.Name
FROM Payroll P
WHERE exists
       (SELECT *
        FROM Regist R
        WHERE P.UserID = R.UserID);
```

A Simple Case: a Monotone Query

```
SELECT P.UserID, P.Name
FROM Payroll P
WHERE exists
      (SELECT *
       FROM Regist R
       WHERE P.UserID = R.UserID);
```

First
unnest

```
SELECT DISTINCT P.UserID, P.Name
FROM Payroll P, Regist R
WHERE P.UserID = R.UserID;
```

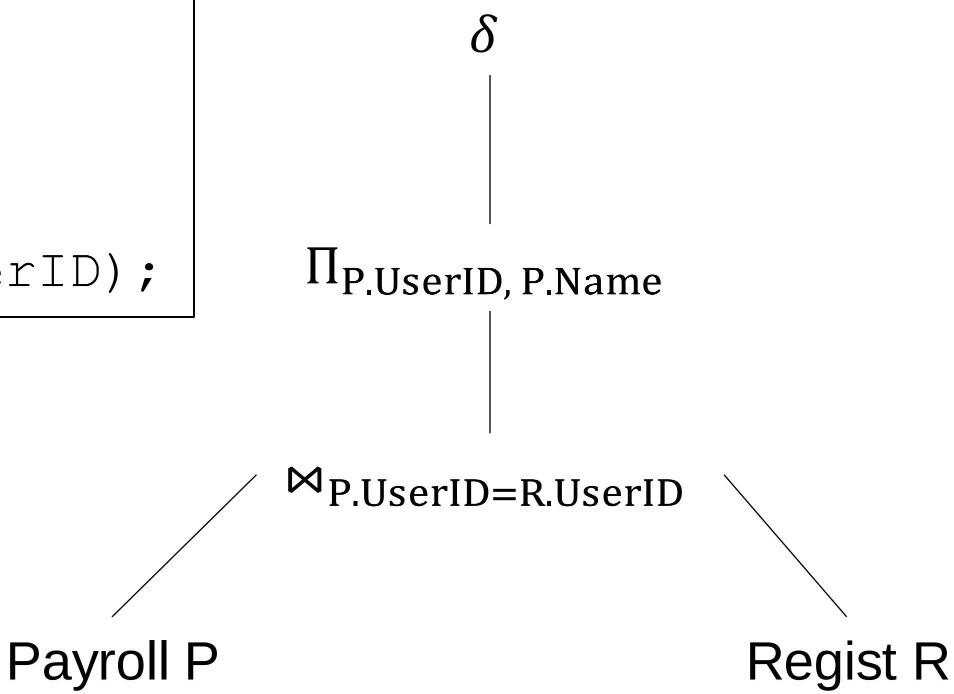
A Simple Case: a Monotone Query

```
SELECT P.UserID, P.Name
FROM Payroll P
WHERE exists
       (SELECT *
        FROM Regist R
        WHERE P.UserID = R.UserID);
```

First unnest

```
SELECT DISTINCT P.UserID, P.Name
FROM Payroll P, Regist R
WHERE P.UserID = R.UserID;
```

The convert to RA



δ

$\Pi_{P.UserID,\ P.Name}$

$\bowtie_{P.UserID=R.UserID}$

Payroll P

Regist R

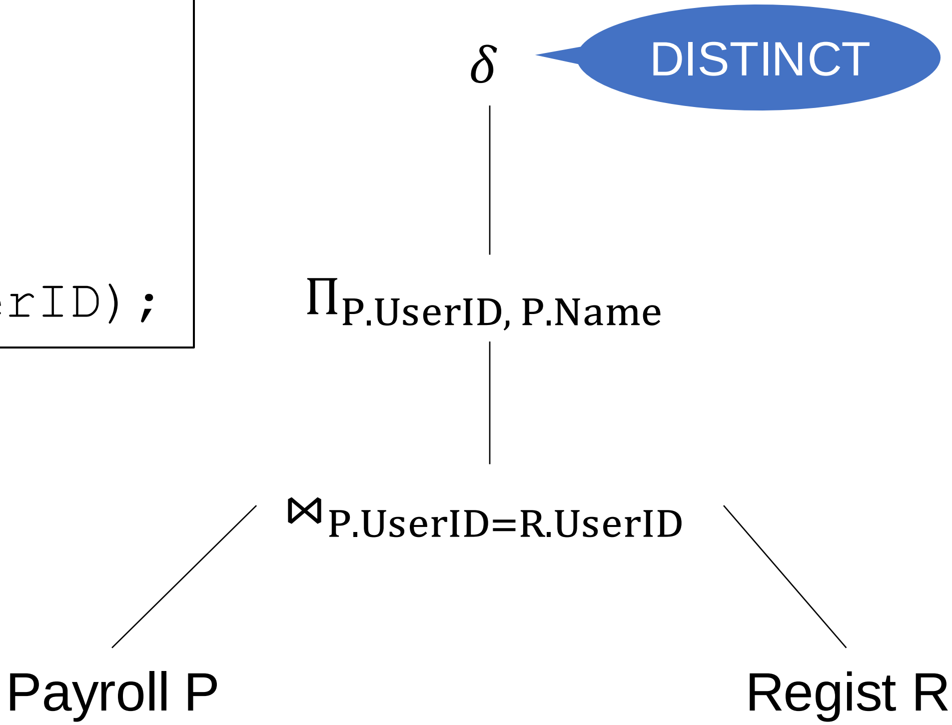
A Simple Case: a Monotone Query

```
SELECT P.UserID, P.Name
FROM Payroll P
WHERE exists
        (SELECT *
         FROM Regist R
         WHERE P.UserID = R.UserID);
```

First unnest

```
SELECT DISTINCT P.UserID, P.Name
FROM Payroll P, Regist R
WHERE P.UserID = R.UserID;
```

The convert to RA



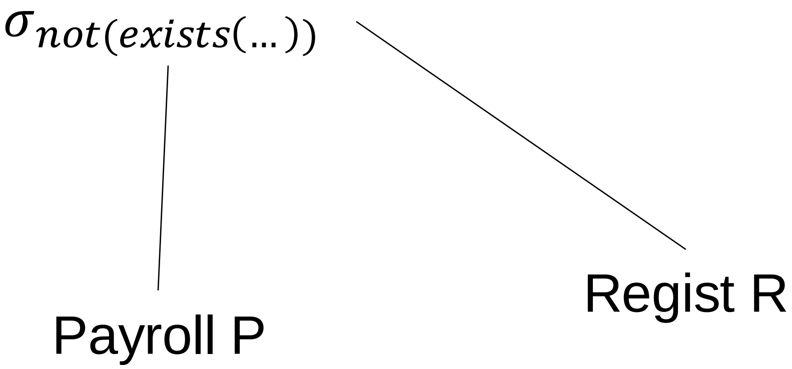
A Difficult Case: a Non-Monotone Query

```
SELECT P.UserID
FROM Payroll P
WHERE not exists
      (SELECT *
       FROM Regist R
       WHERE P.UserID = R.UserID);
```

A Difficult Case: a Non-Monotone Query

```
SELECT P.UserID
FROM Payroll P
WHERE not exists
      (SELECT *
       FROM Regist R
       WHERE P.UserID = R.UserID);
```

Totally, totally wrong!



$\sigma_{not(exists(\ldots))}$

Payroll P

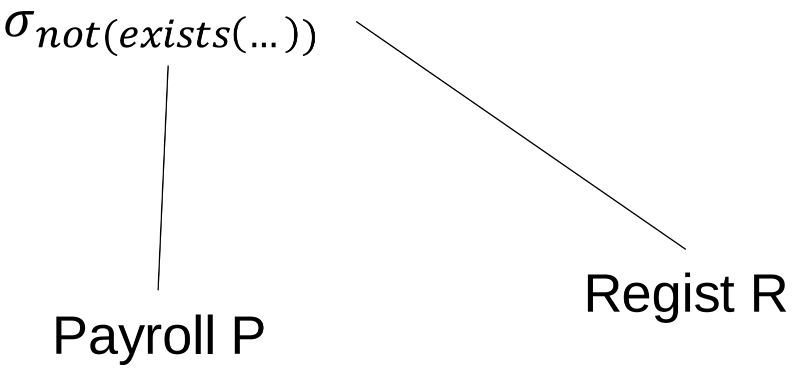
Regist R

A Difficult Case: a Non-Monotone Query

```
SELECT P.UserID
FROM Payroll P
WHERE not exists
      (SELECT *
       FROM Regist R
       WHERE P.UserID = R.UserID);
```

Totally, totally wrong!

There are no subqueries in RA.



$\sigma_{not(exists(\dots))}$

Payroll P

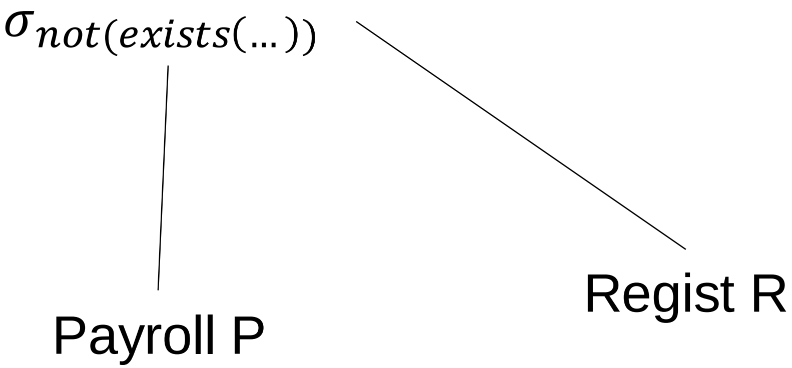
Regist R

A Difficult Case: a Non-Monotone Query

```
SELECT P.UserID
FROM Payroll P
WHERE not exists
        (SELECT *
         FROM Regist R
         WHERE P.UserID = R.UserID);
```

Totally, totally wrong!

There are no subqueries in RA.



$\sigma_{not(exists(\ldots))}$

Payroll P

Regist R

Need to unnest,
but first need to de-correlate.

A Difficult Case: a Non-Monotone Query

```
SELECT P.UserID
FROM Payroll P
WHERE not exists
       (SELECT *
        FROM Regist R
        WHERE P.UserID = R.UserID);
```

First
de-correlate

```
SELECT P.UserID
FROM Payroll P
WHERE P.UserID not in
       (SELECT R.UserID
        FROM Regist R);
```

A Difficult Case: a Non-Monotone Query

```
SELECT P.UserID
FROM Payroll P
WHERE not exists
       (SELECT *
        FROM Regist R
        WHERE P.UserID = R.UserID);
```

First
de-correlate

```
SELECT P.UserID
FROM Payroll P
WHERE P.UserID not in
        (SELECT R.UserID
         FROM Regist R);
```

Then unnest
using set difference

```
SELECT P.UserID
FROM Payroll P
   EXCEPT
SELECT R.UserID
FROM Regist R;
```

A Difficult Case: a Non-Monotone Query

```
SELECT P.UserID
FROM Payroll P
WHERE not exists
      (SELECT *
       FROM Regist R
       WHERE P.UserID = R.UserID);
```

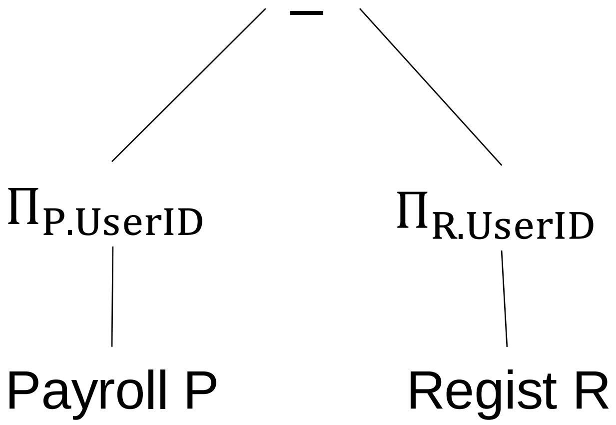
First
de-correlate

```
SELECT P.UserID
FROM Payroll P
WHERE P.UserID not in
       (SELECT R.UserID
        FROM Regist R);
```

Then unnest
using set difference

```
SELECT P.UserID
FROM Payroll P
   EXCEPT
SELECT R.UserID
FROM Regist R;
```

Finally,
rewrite to RA



$-$

$\Pi_{P.UserID}$ $\Pi_{R.UserID}$

Payroll P Regist R

Discussion

- SQL = declarative language; what we want
  RA = an algebra; how to get it

- We write in SQL, optimizers generates RA

- Some language resemble RA more than SQL, e.g. Spark

Next topic: how to design a database from scratch

Database Design

Database Design

- New application needs persistent database.

- The database will persist for a long period of time. We need a good design from day 1.

- Incorporate feedback from many stakeholders
  - Programmers, business teams, analysts, data scientists, product managers, …

The Database Design Process

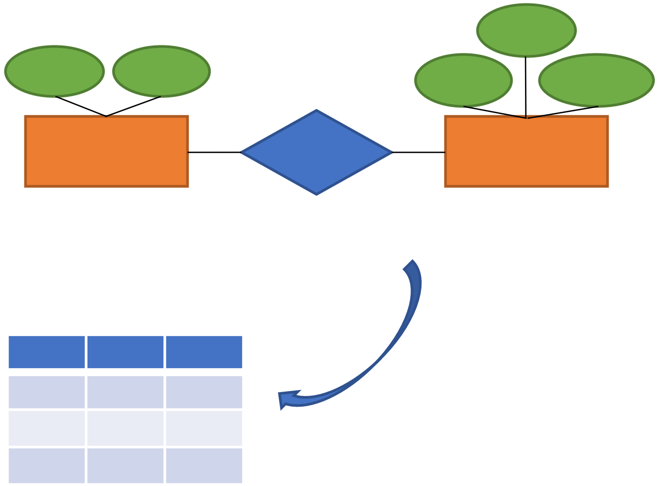
Today

Conceptual Model

Relational Model

+ Schema

+ Constraints



The Database Design Process

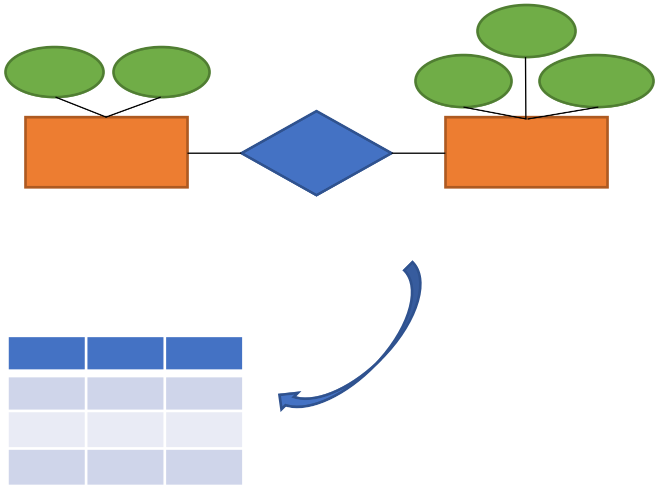
Today

Conceptual Model

Relational Model

+ Schema

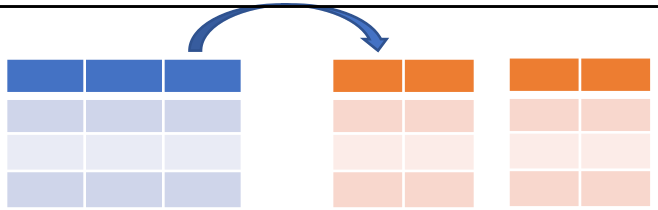
+ Constraints



Next Lectures

Conceptual Schema

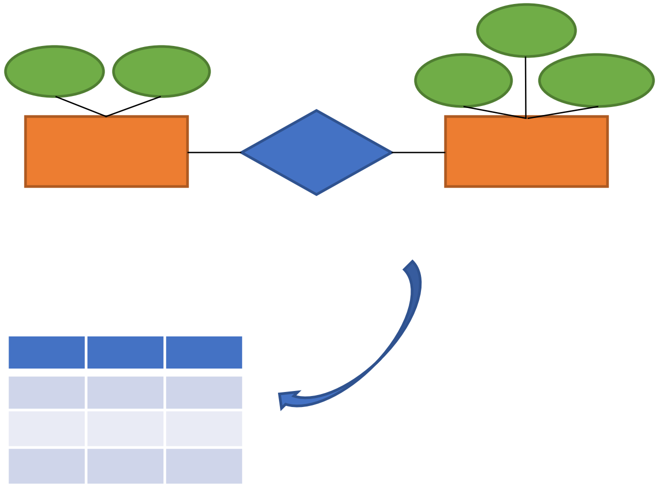
+ Normalization



The Database Design Process

Today

Conceptual Model



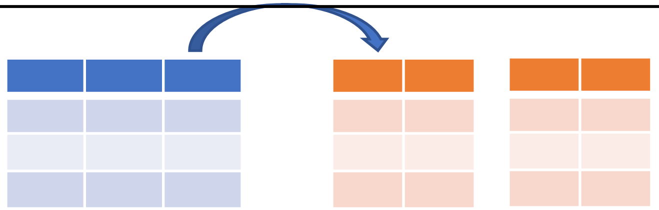
Relational Model

+ Schema
+ Constraints

Next Lectures

Conceptual Schema

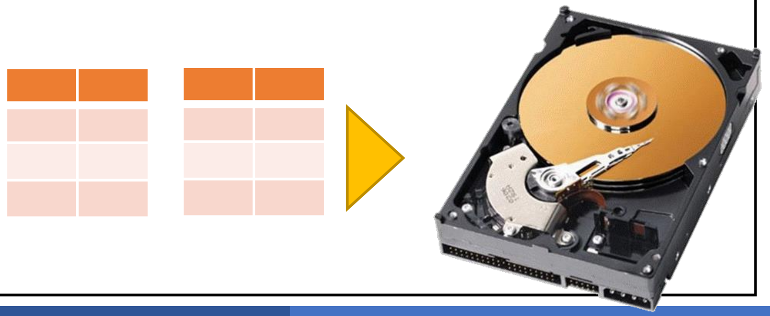
+ Normalization



Later…

Physical Schema

+ Partitioning
+ Indexing



ER Diagrams

Entity-Relationship (ER) Diagrams

- A visual way to describe the schema of a database
- Language independent: may implement in SQL, or some other data model

Example

Application to track the lifetime of products

- Keep information about Products: name, price, …
- Who manufactures them? Company name, address, their workers, …
- Who buys them? Customers with their names, …

Example: designing the Entity Sets

Product

Example: designing the Entity Sets

Product

Company

Worker

Example: designing the Entity Sets

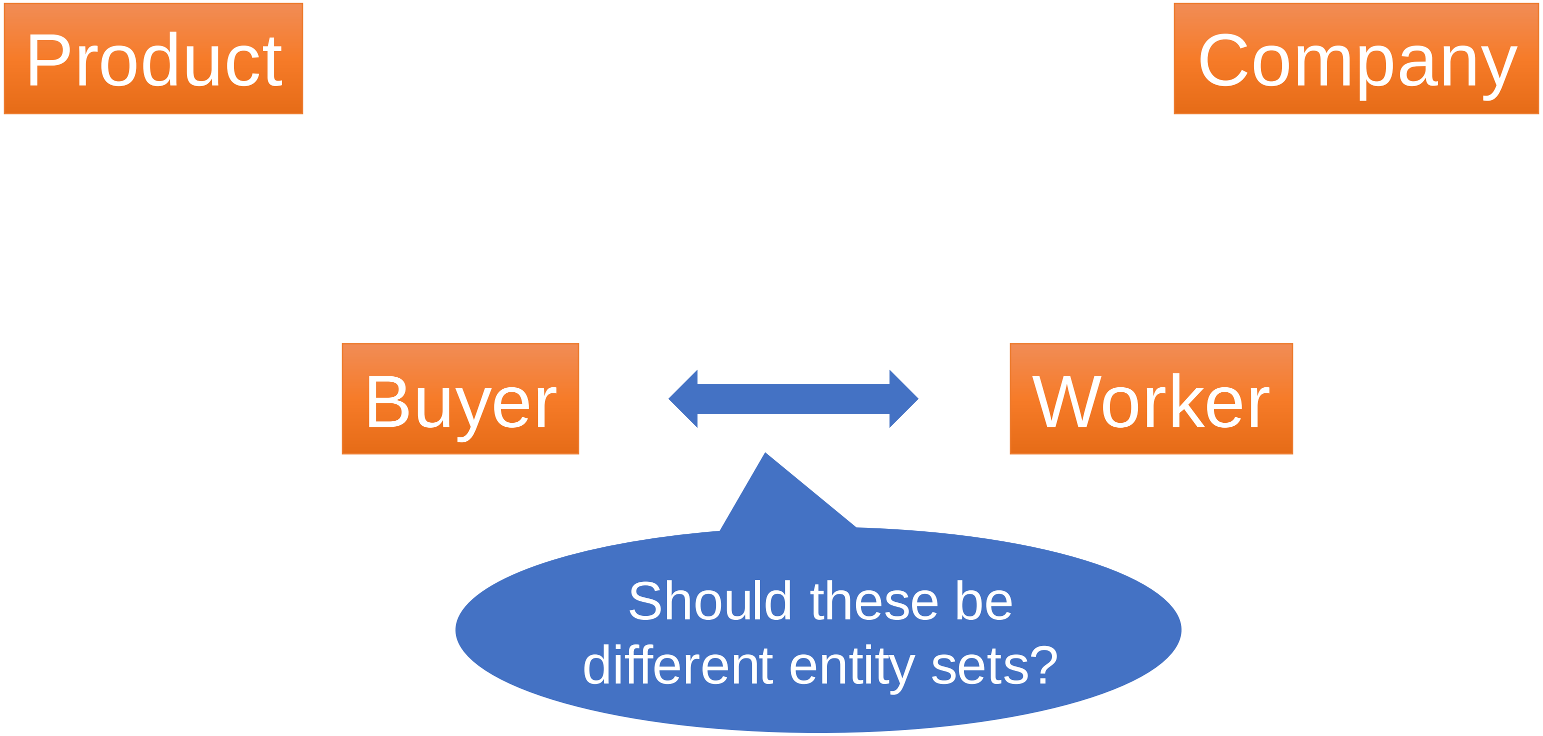
Product

Company

Buyer

Worker

Example: designing the Entity Sets



Product

Company

Buyer

Worker

Should these be different entity sets?

Example: designing the Entity Sets

Product

Company

Person

Let’s keep things simple for now

Example: adding Attributes

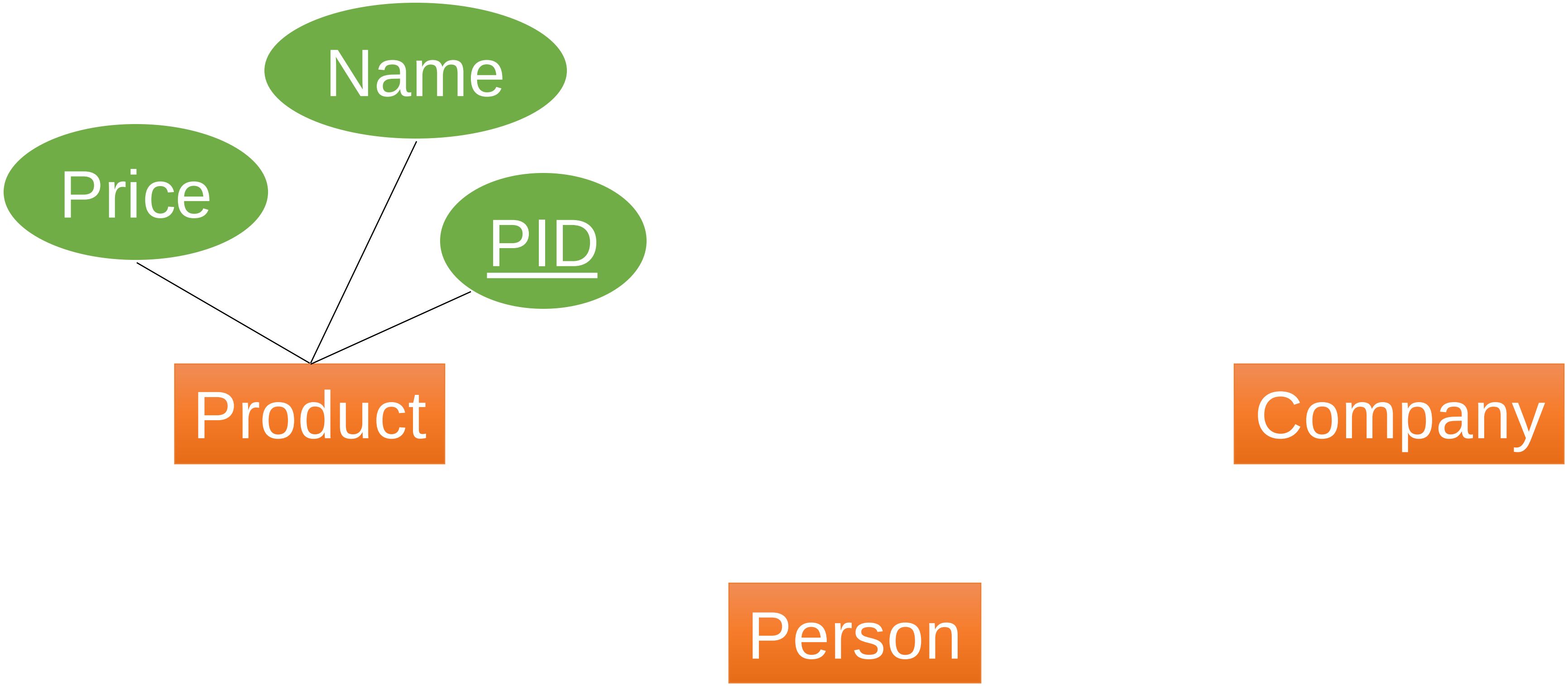
Next, let’s design their attributes

Product

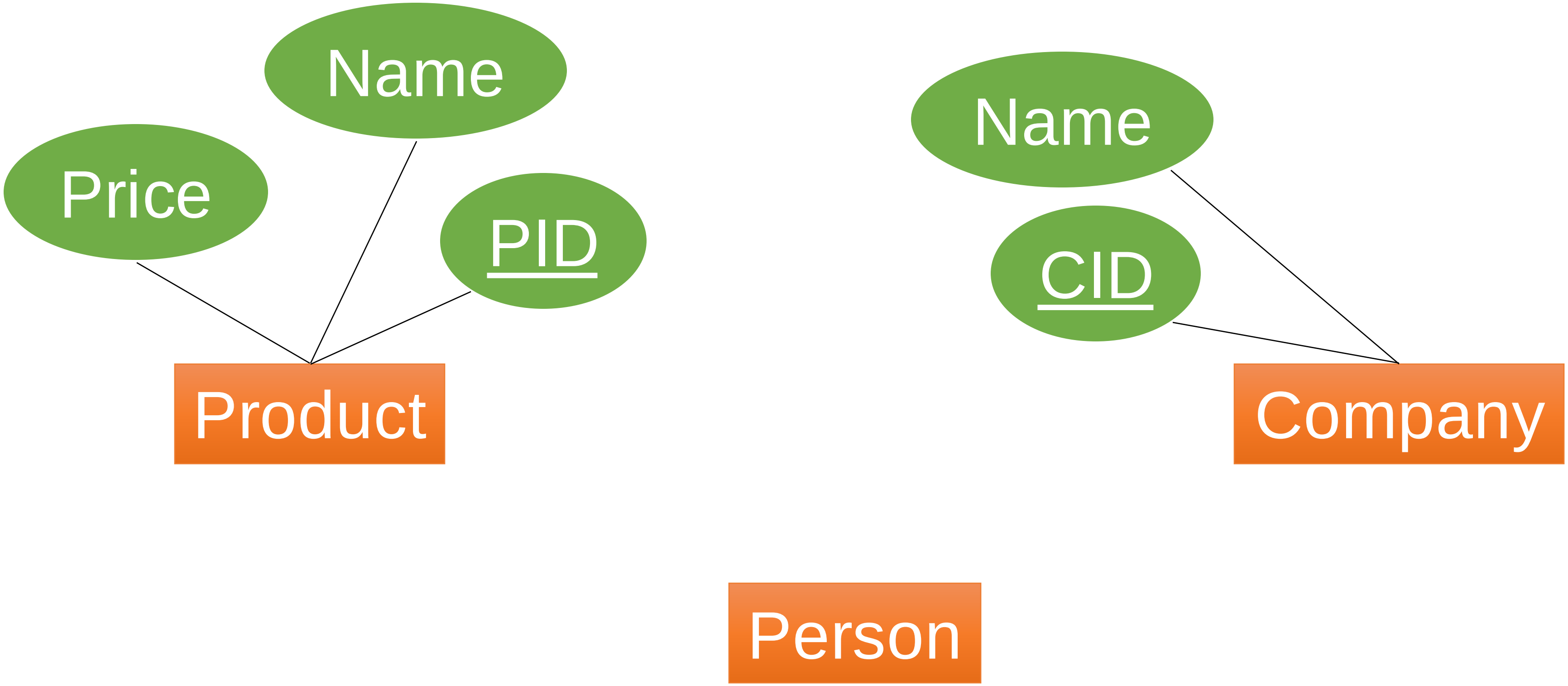
Company

Person

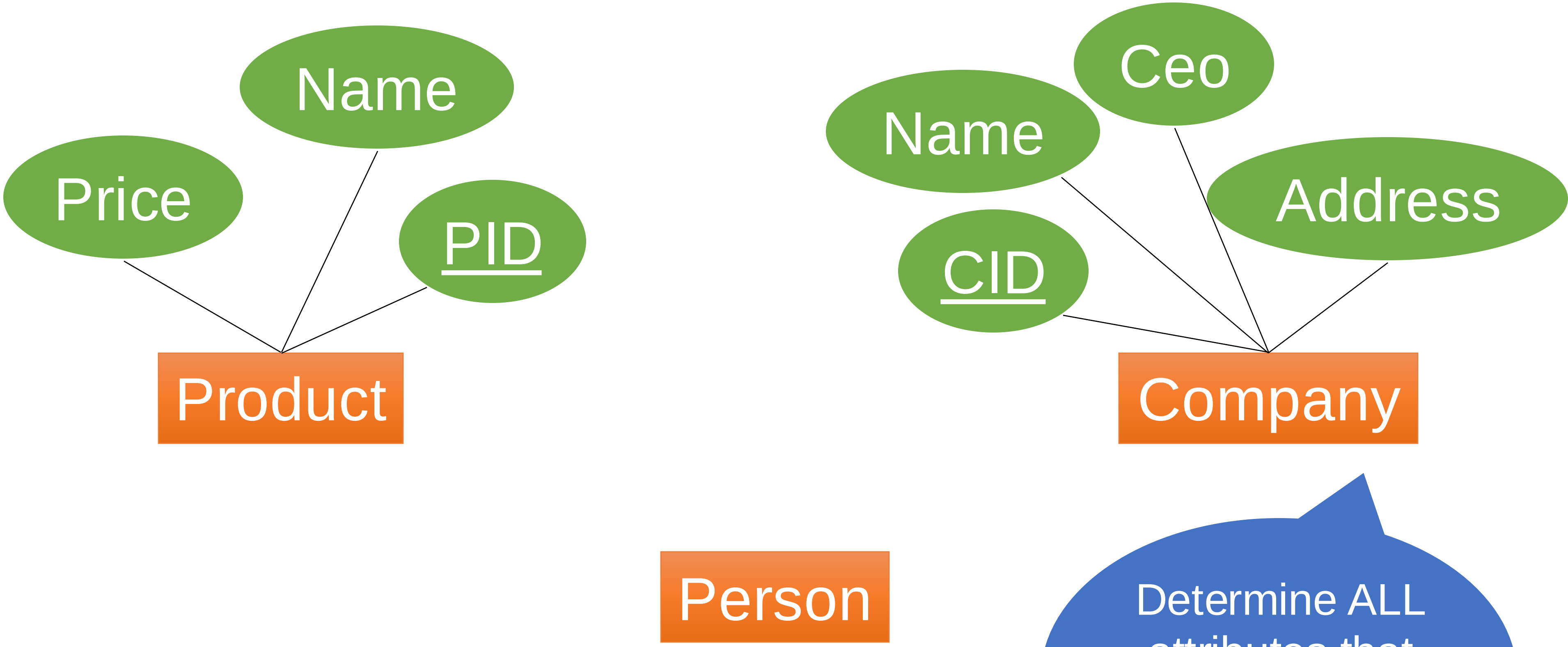
Example: adding Attributes



Example: adding Attributes



Example: adding Attributes



Price

Name

PID

Product

Name

Ceo

Address

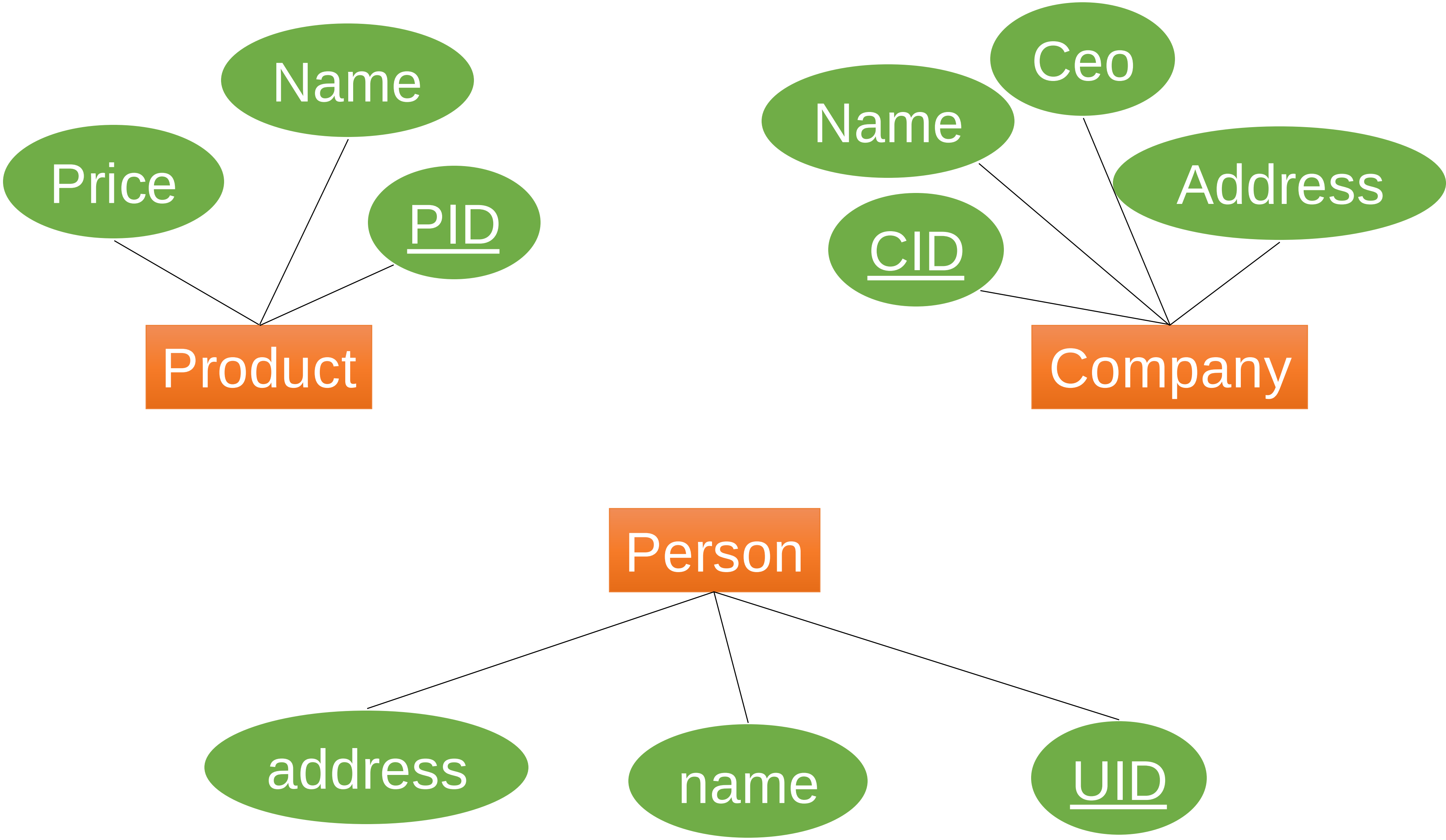
CID

Company

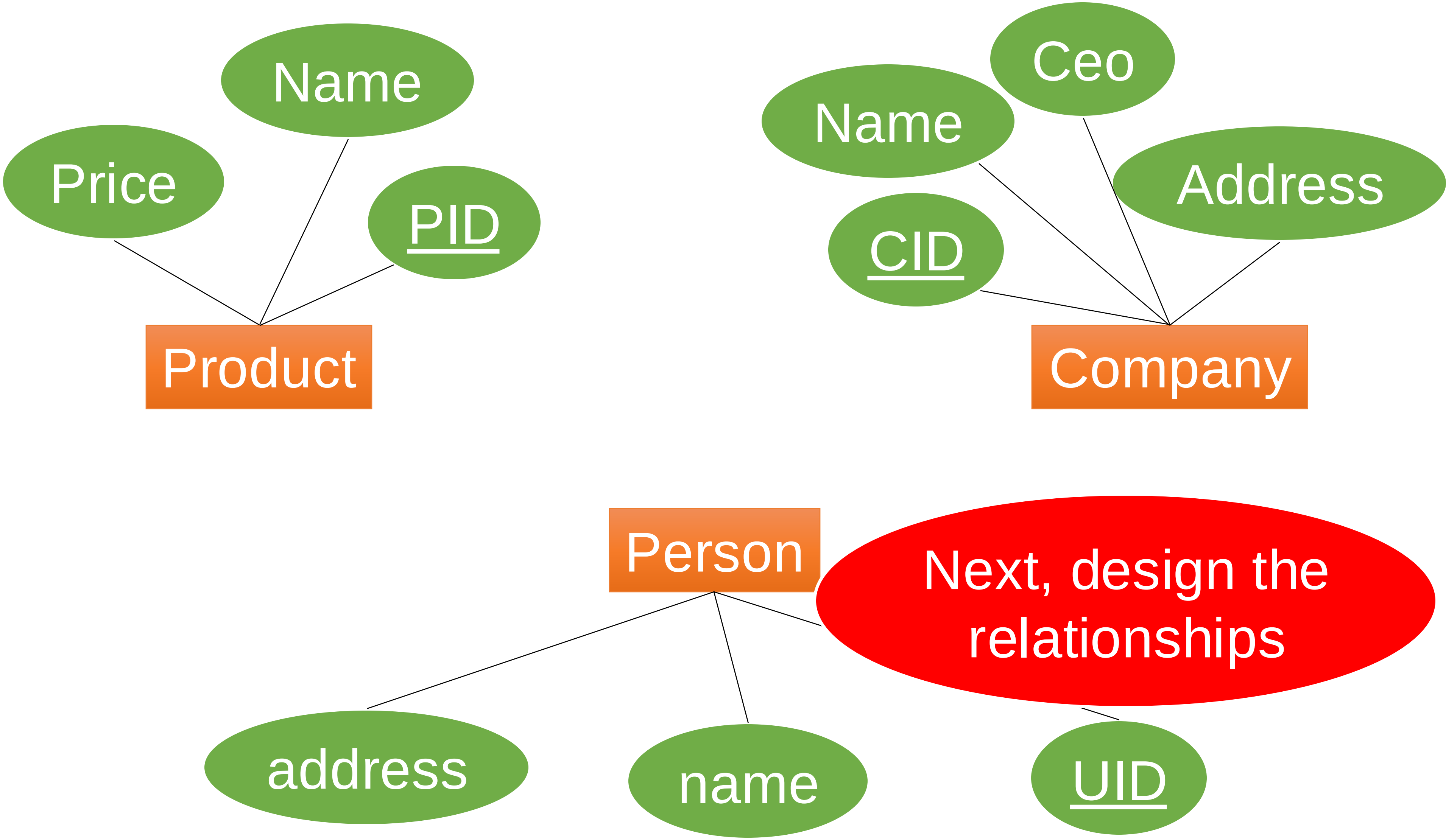
Person

Determine ALL attributes that your application needs

Example: adding Attributes

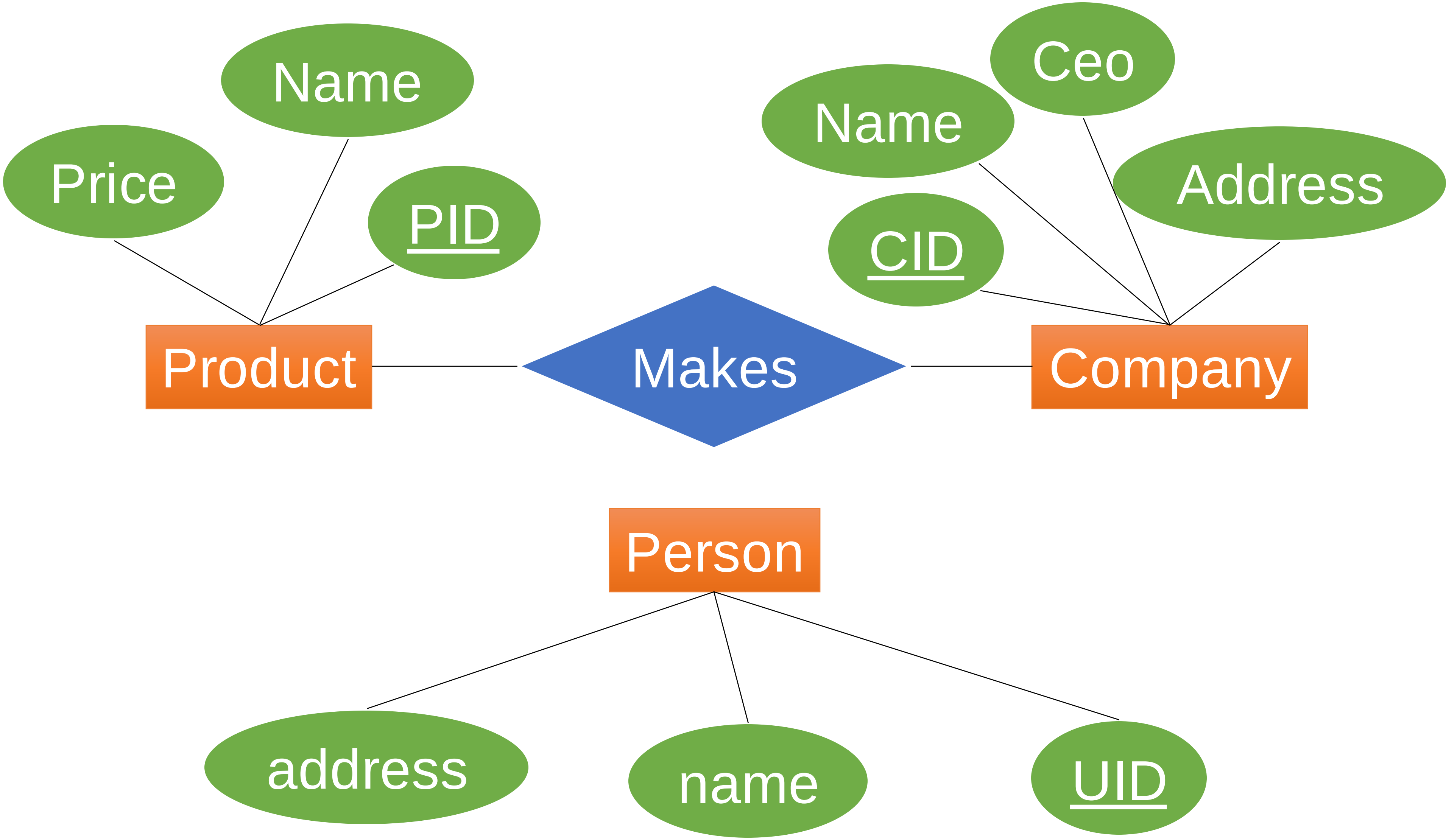


Example: adding Relationships

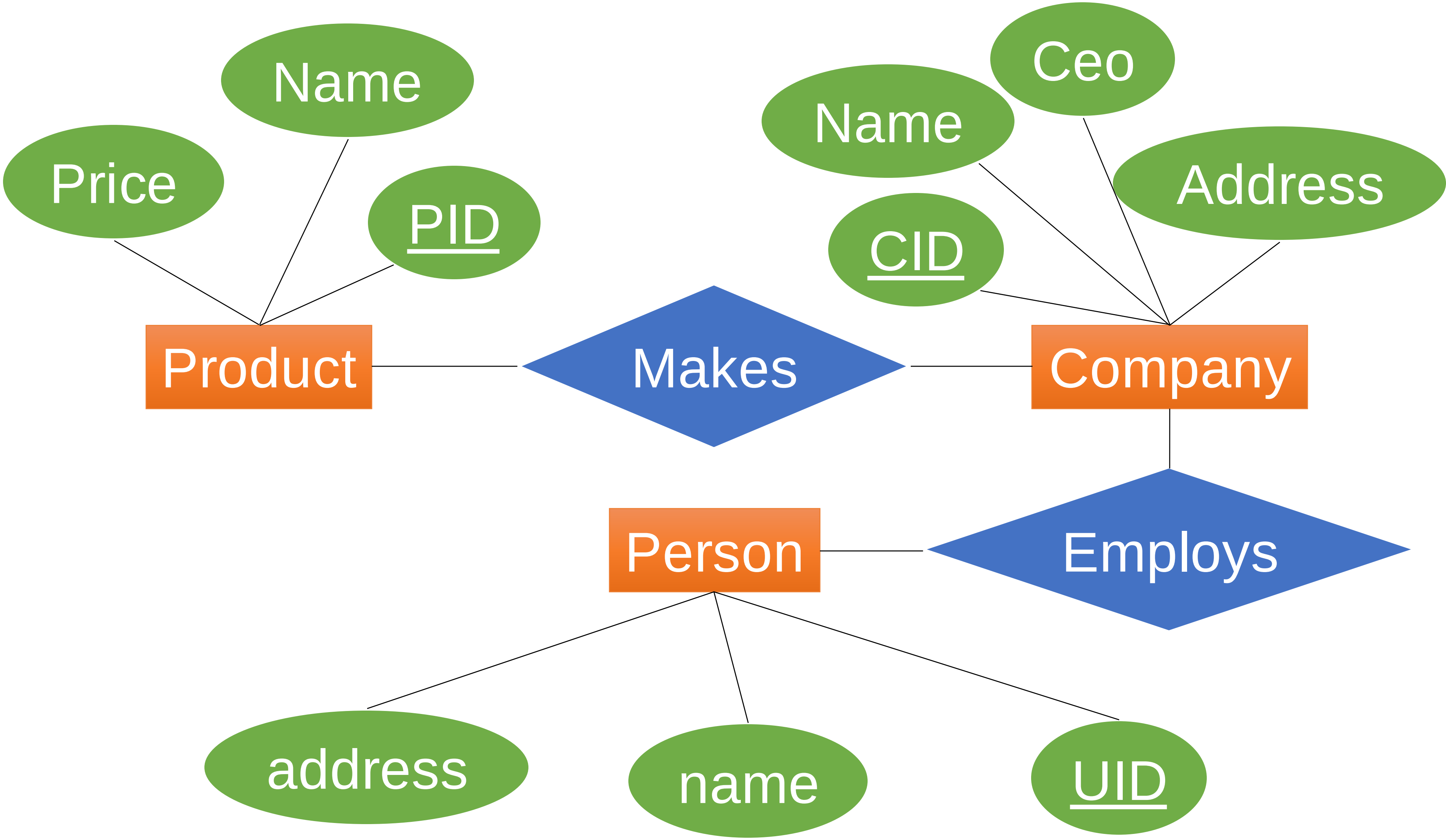


Next, design the relationships

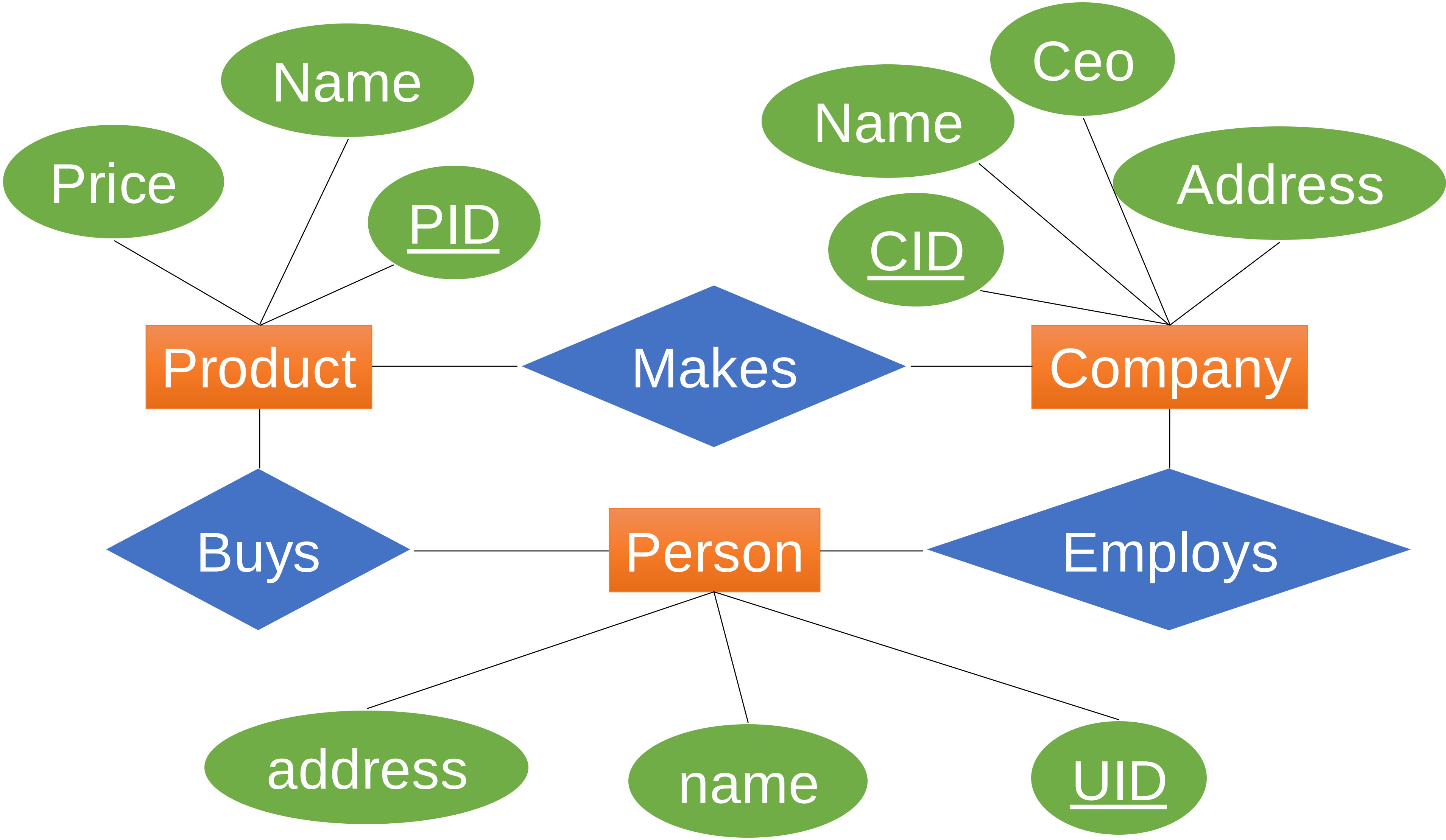
Example: adding Relationships



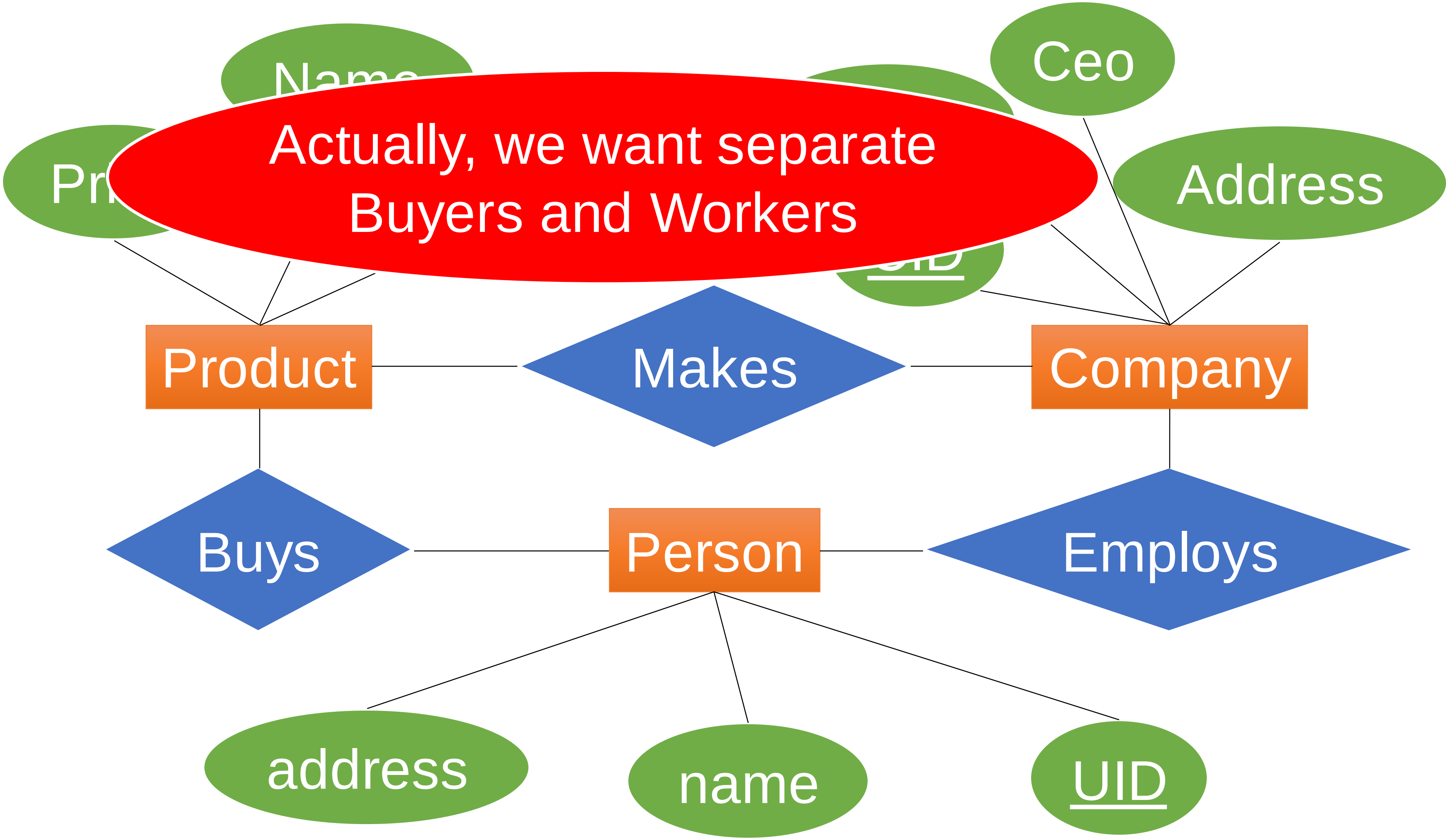
Example: adding Relationships



Example: adding Relationships

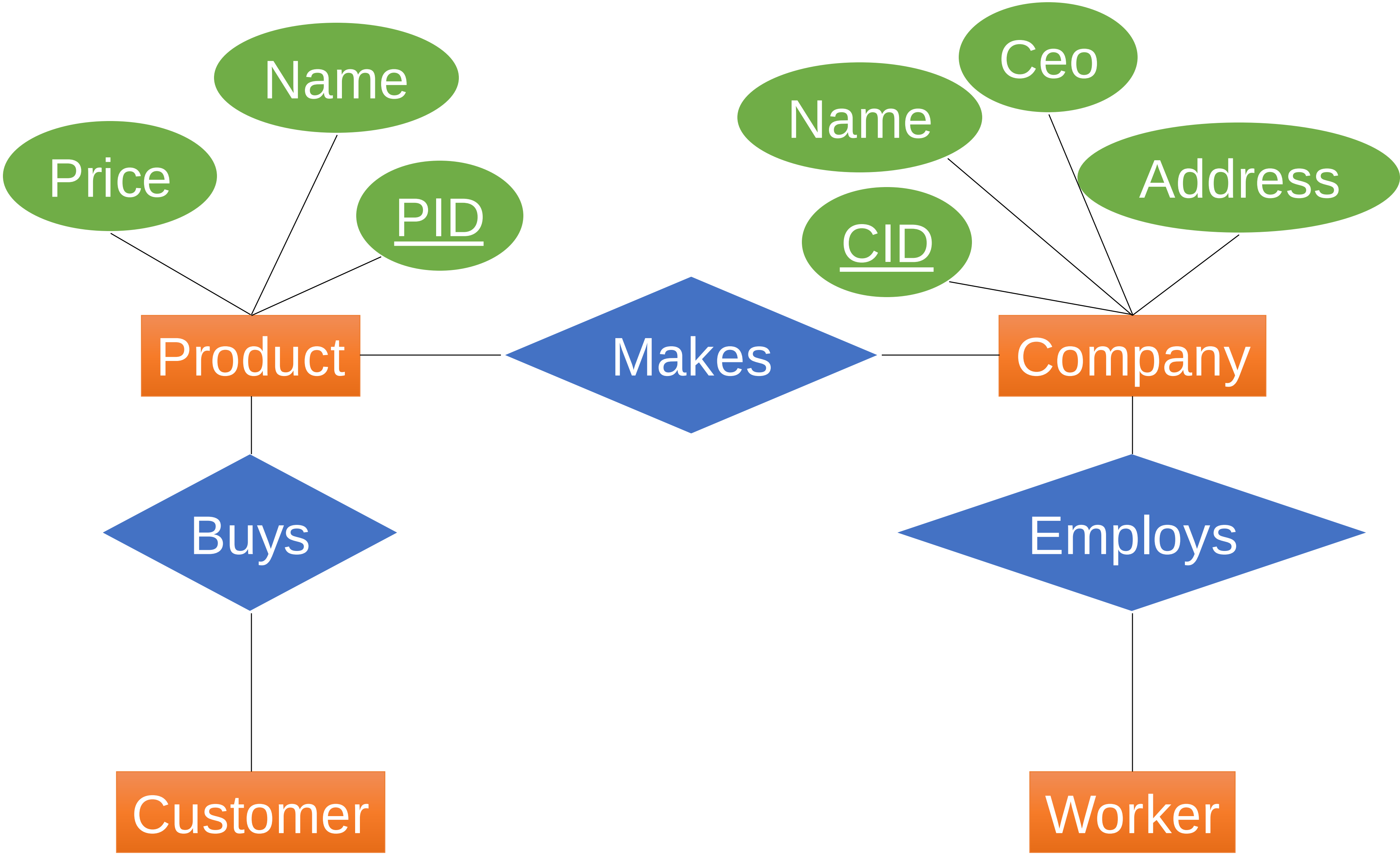


Example: Refining the Schema

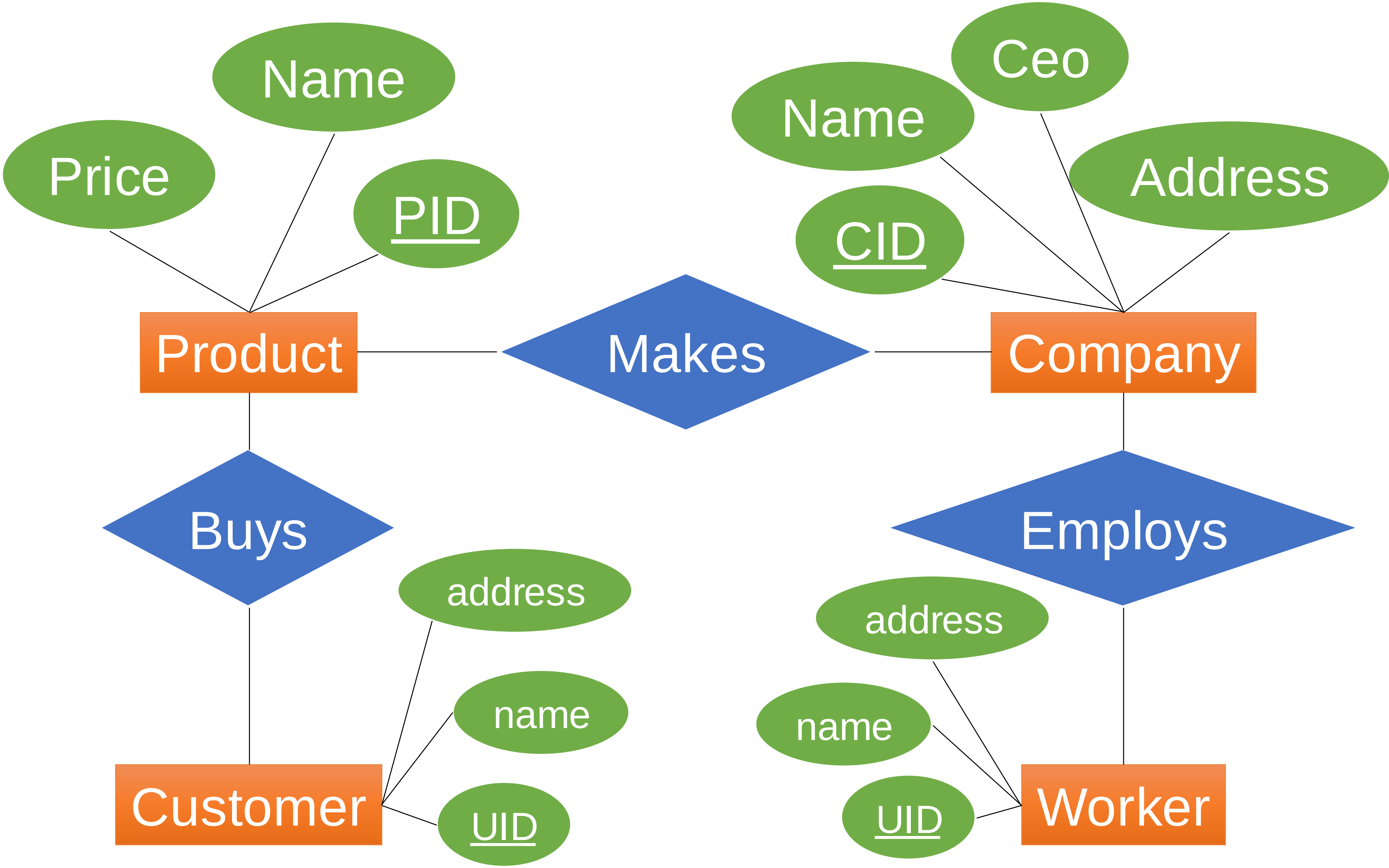


Actually, we want separate Buyers and Workers

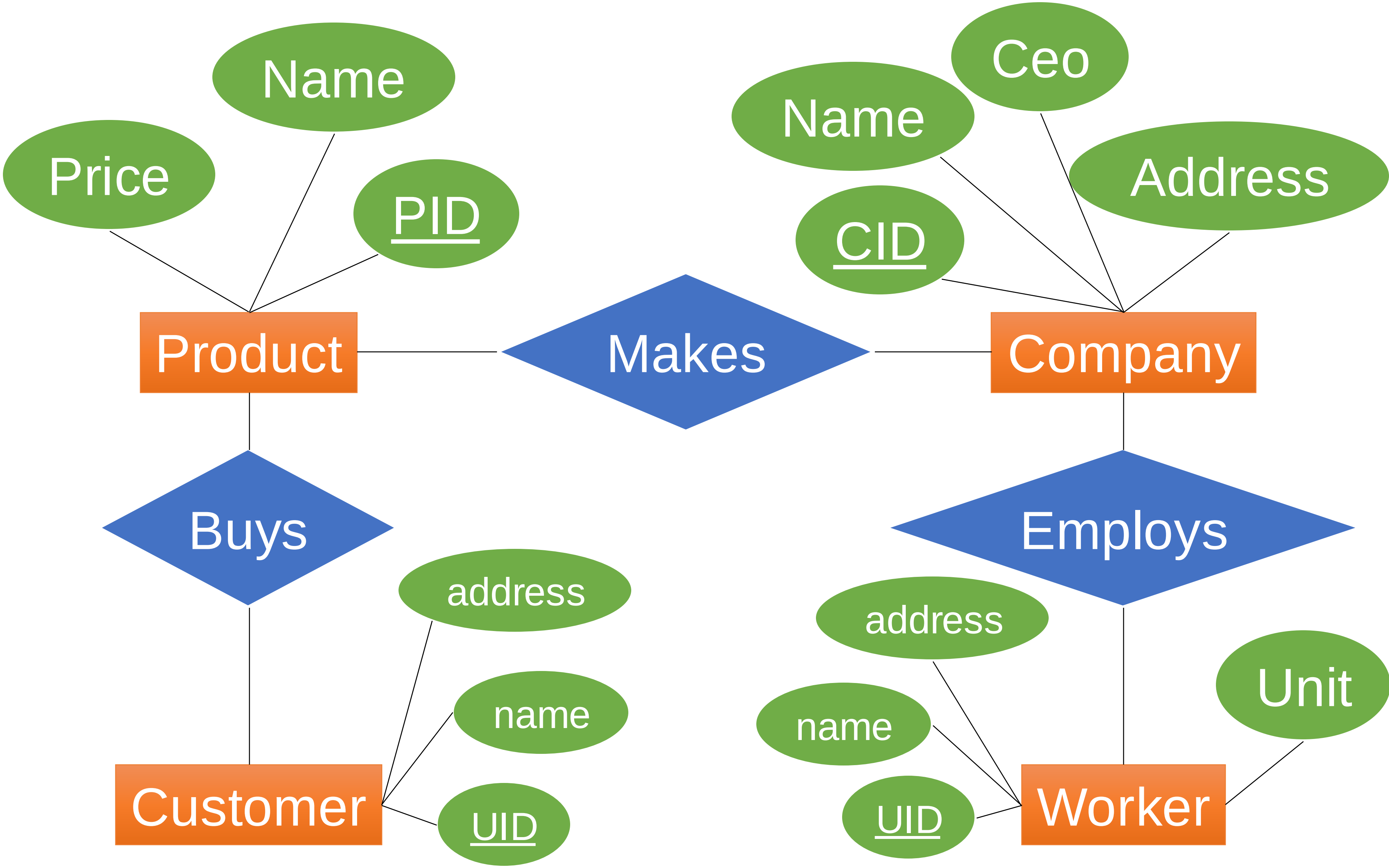
Example: Refining the Schema



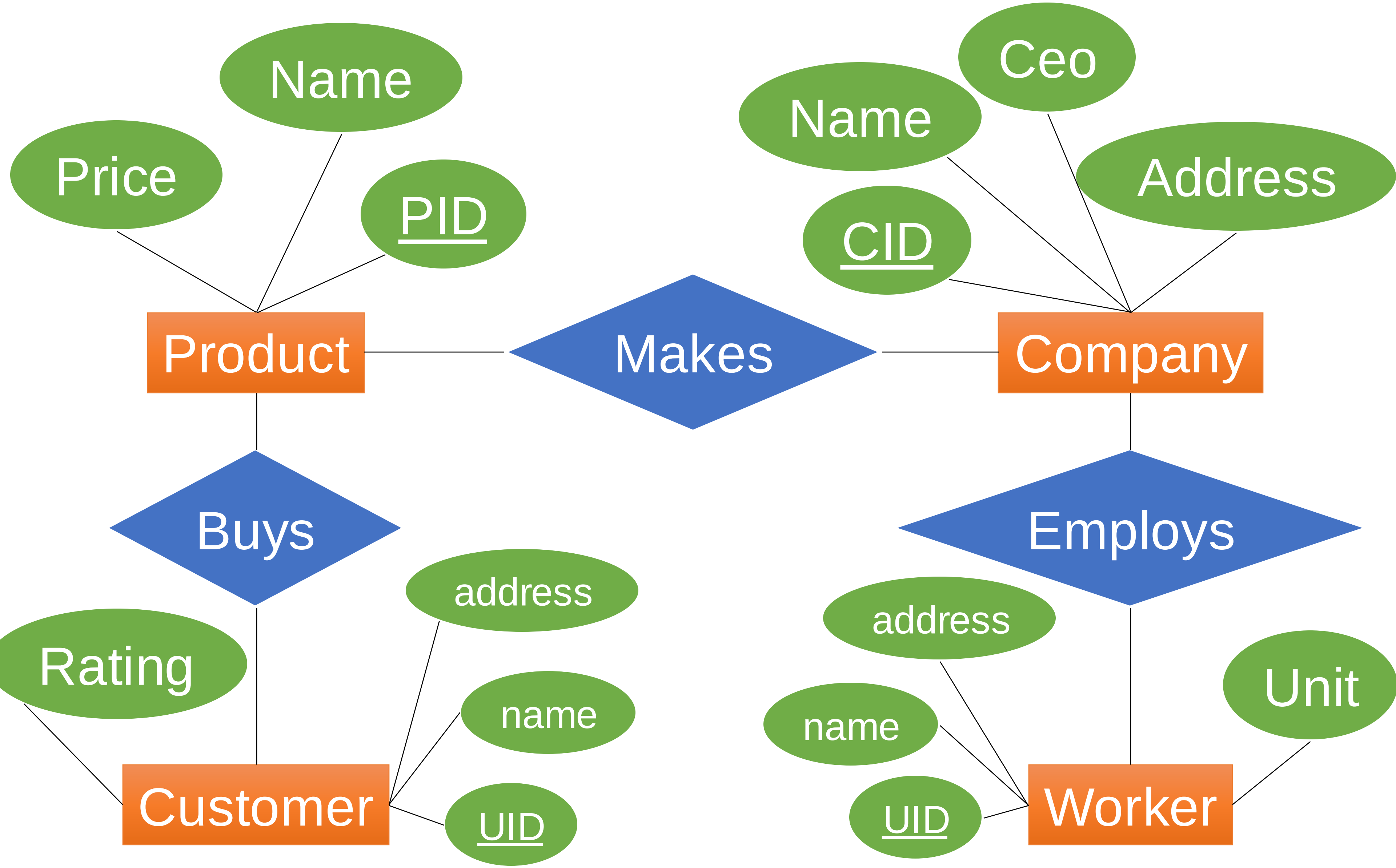
Example: Refining the Schema



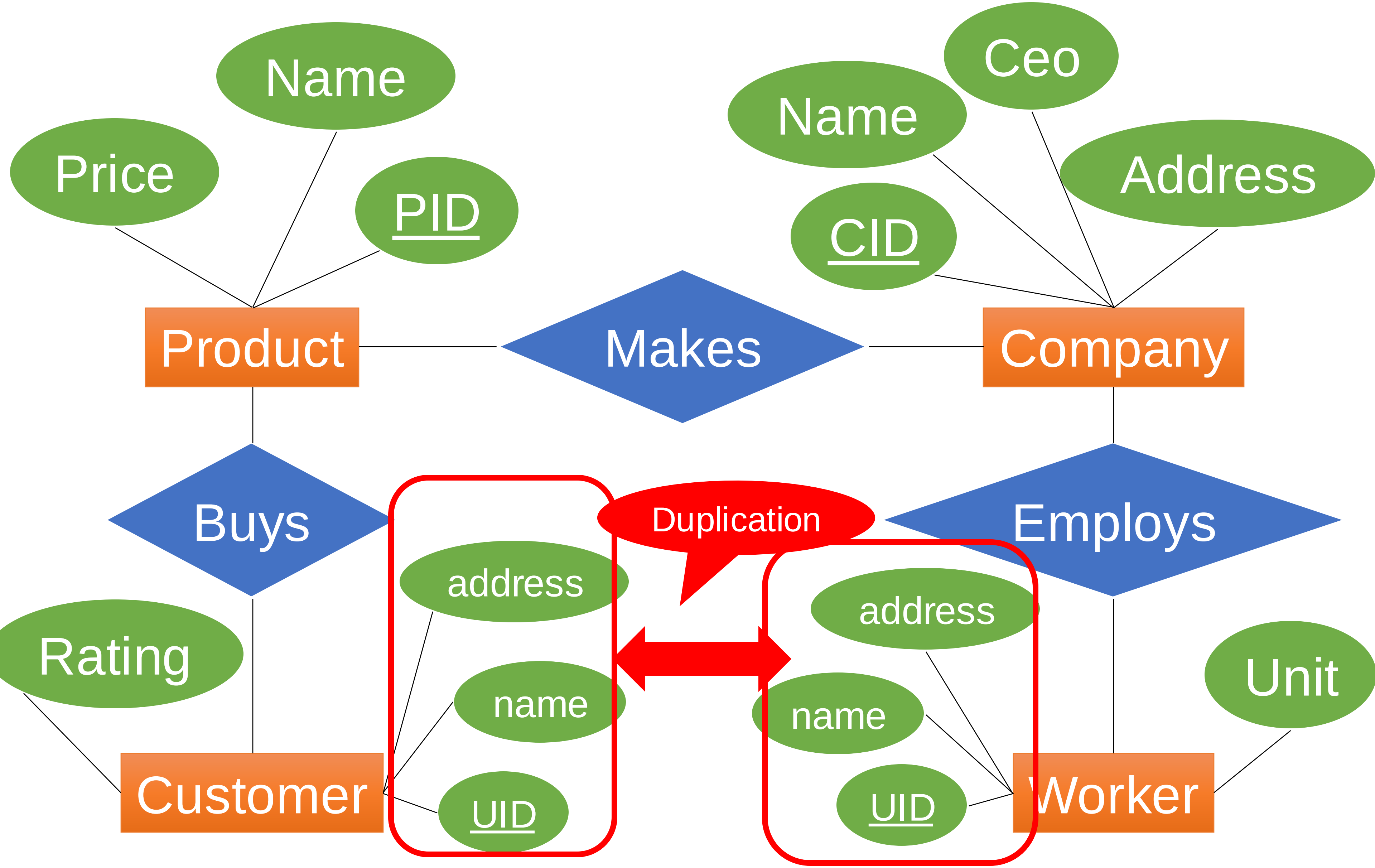
Example: Refining the Schema



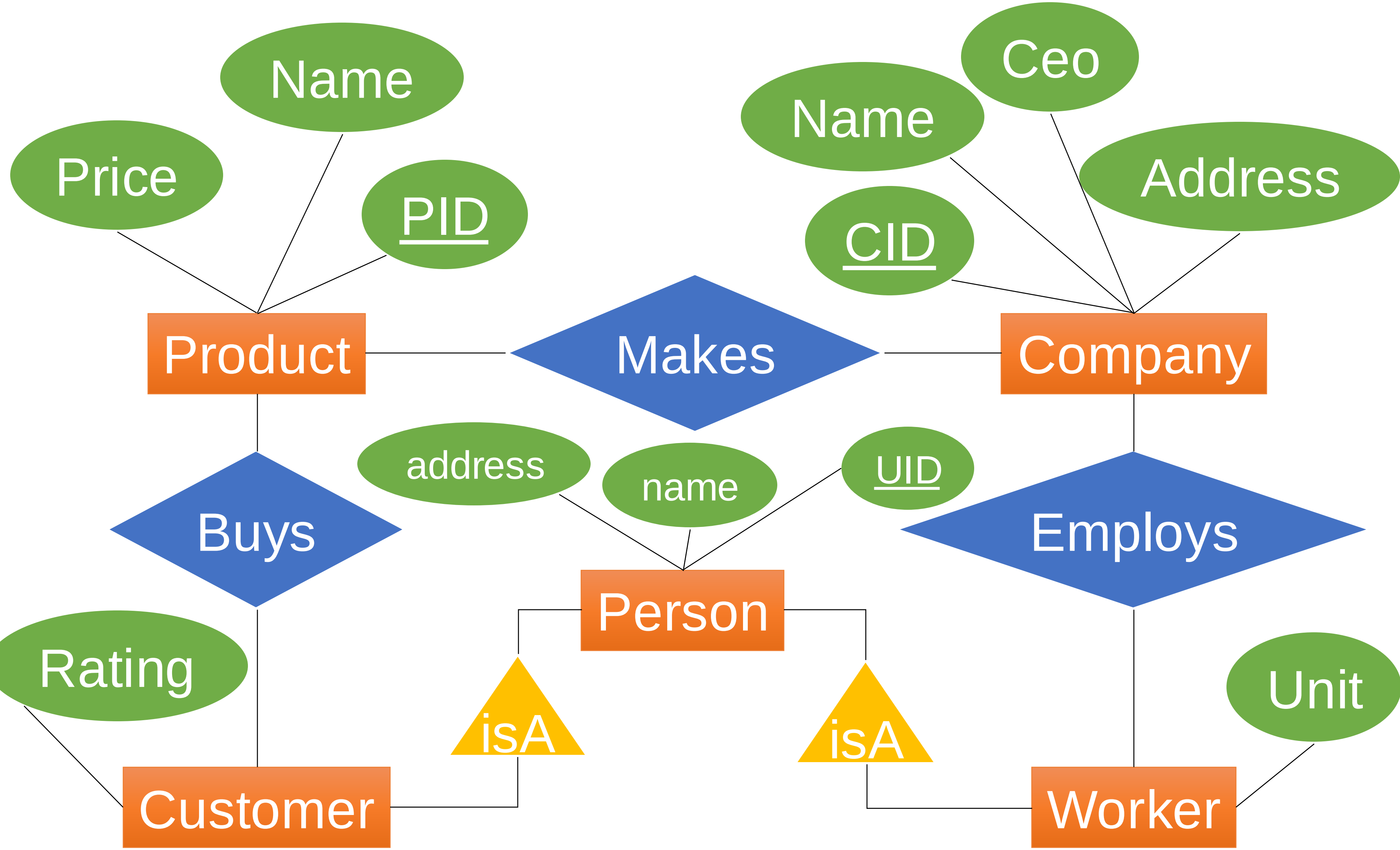
Example: Refining the Schema



Example: Refining the Schema



Example: Refining the Schema



Discussion

- ER diagram are easy to design,
  yet rigorous enough to convert to SQL

- Lots of ER diagram "dialects"
  - Textbook use rectangles/diamonds/ovals
  - Industry uses other standards

- In class we use the textbook version

Next: E/R diagrams in detail

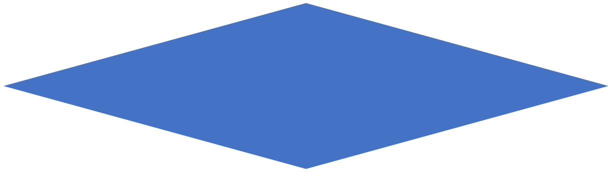
ER Diagrams: Building Blocks

- These are all the components we will learn about

Entity set

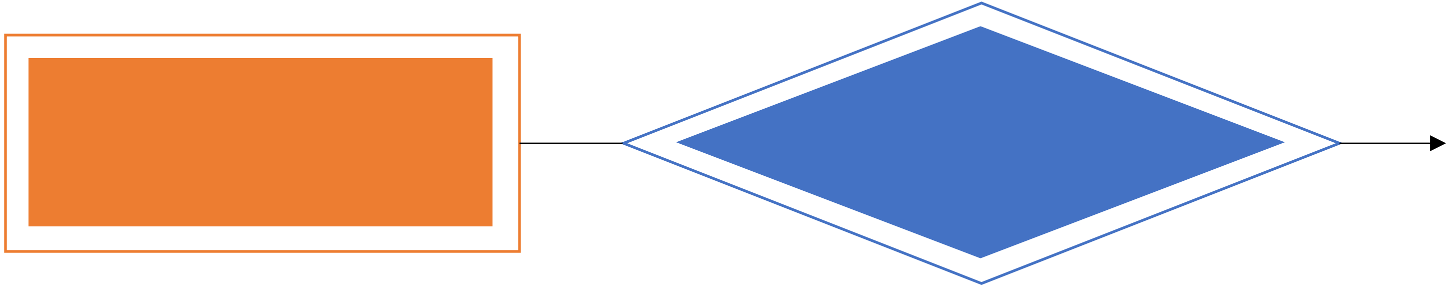
Attribute

Relationship



Subclass

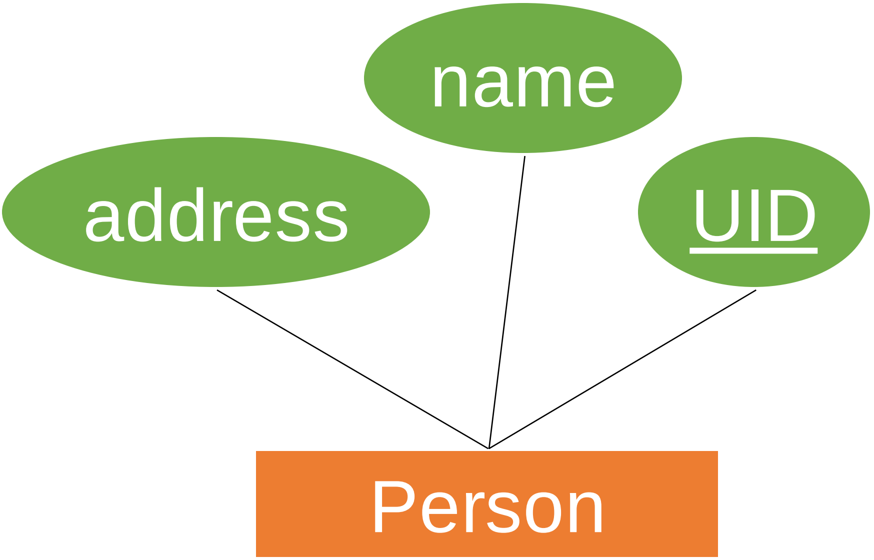
Weak Entity



Entity Sets

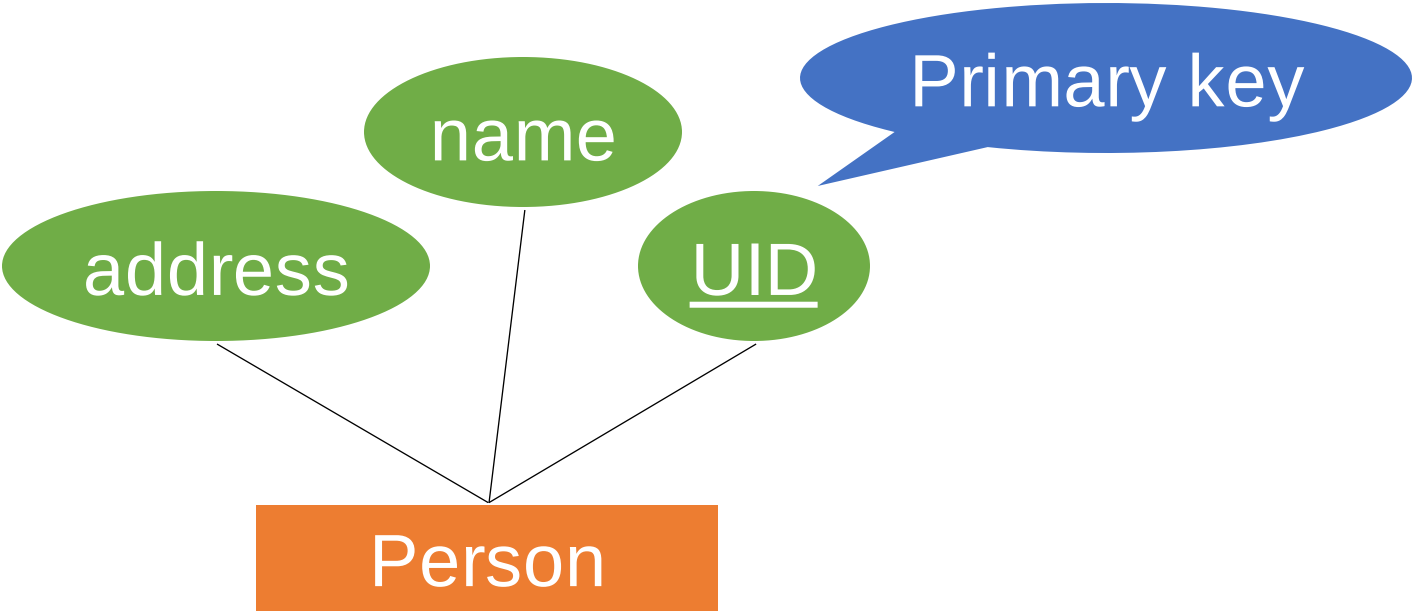
Entity Set

- **Entity set** is the same as a **class**
- An **entity** is the same as an **object**
- An **attribute** is the same as a **field** of a class



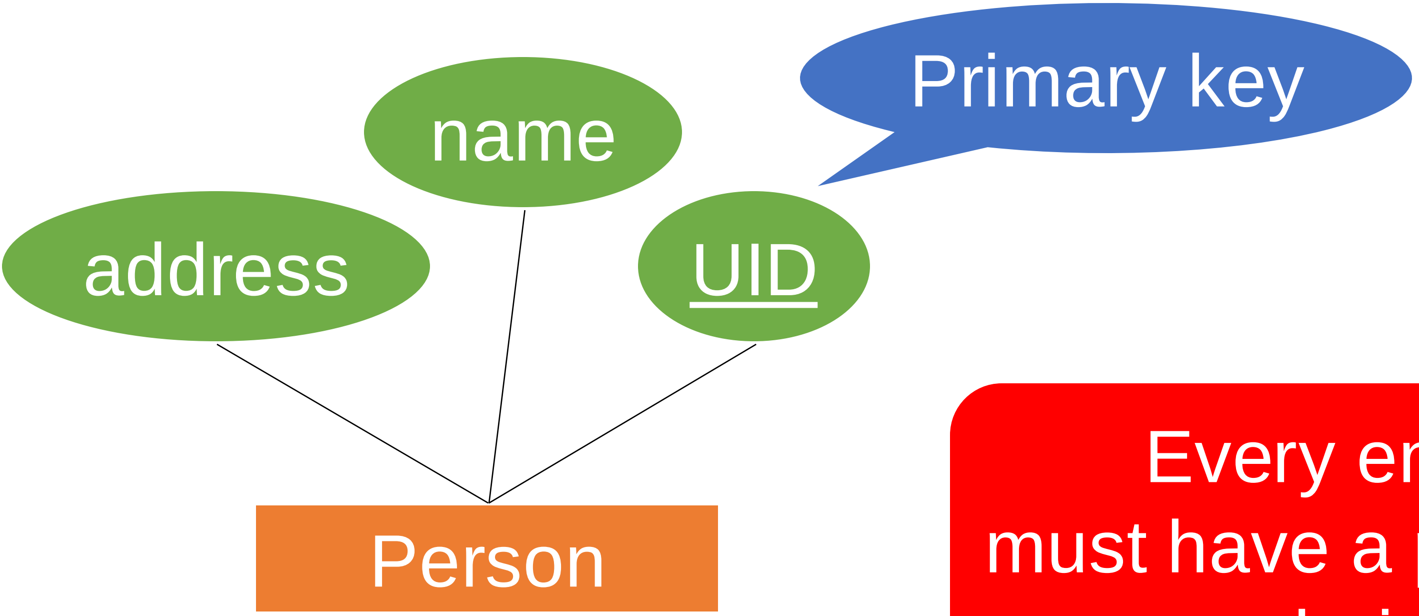
Entity Set

- **Entity set** is the same as a **class**
- An **entity** is the same as an **object**
- An **attribute** is the same as a **field** of a class



Entity Set

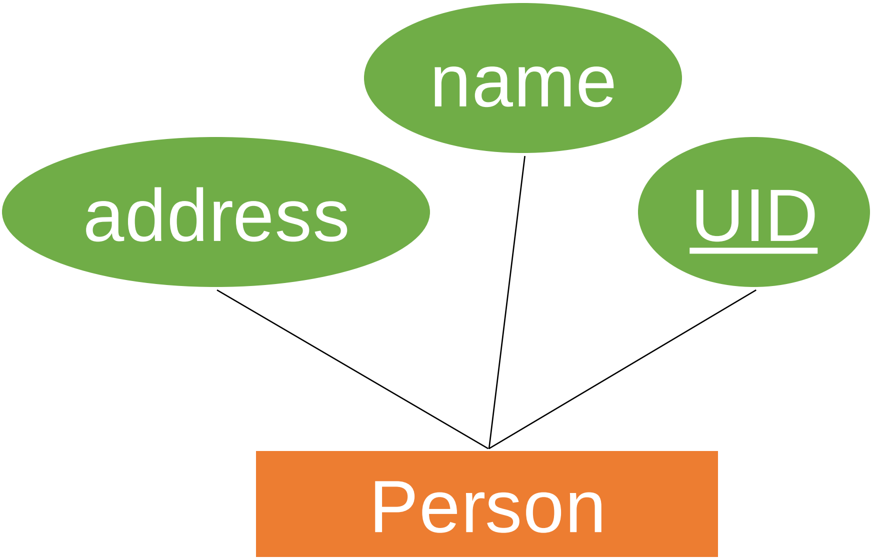
- **Entity set** is the same as a **class**
- An **entity** is the same as an **object**
- An **attribute** is the same as a **field** of a class



Every entity set
must have a primary key,
or a derived one.

Entity Set to SQL

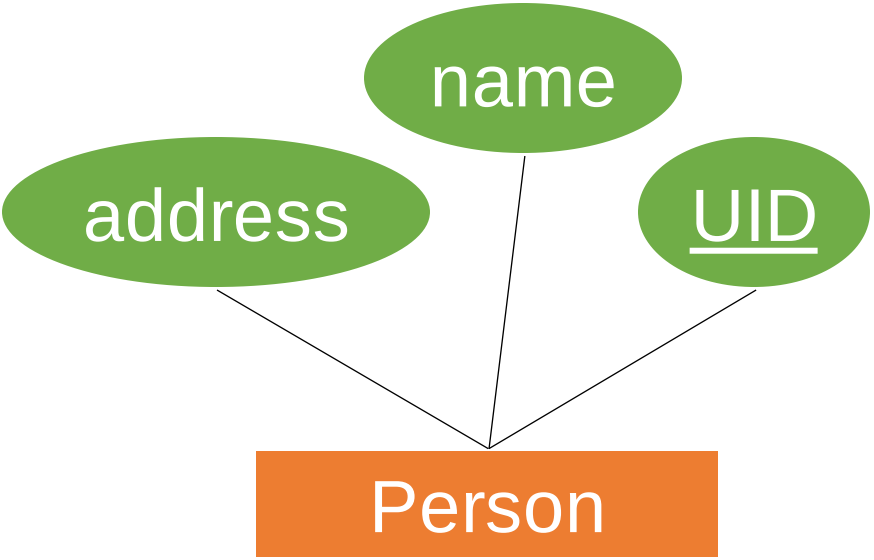
- **Entity set** is the same as a **class**
- An **entity** is the same as an **object**
- An **attribute** is the same as a **field** of a class



How do
we represent
in SQL?

Entity Set to SQL

- **Entity set** is the same as a **class**
- An **entity** is the same as an **object**
- An **attribute** is the same as a **field** of a class



How do we represent in SQL?

```
CREATE TABLE
   Person (
      UID INT PRIMARY KEY,
      name TEXT,
      address TEXT);
```

Relationships

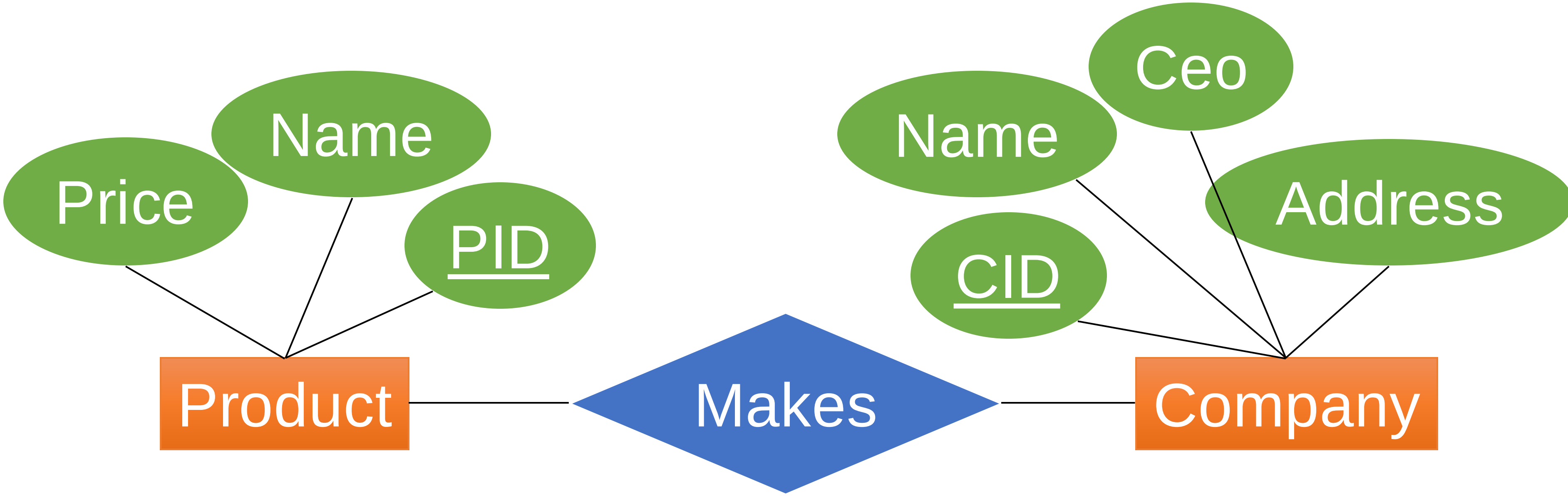
Relationships

- A **relationship** relates entities from two entity sets

A subset of the cross product: $R \subseteq A \times B$

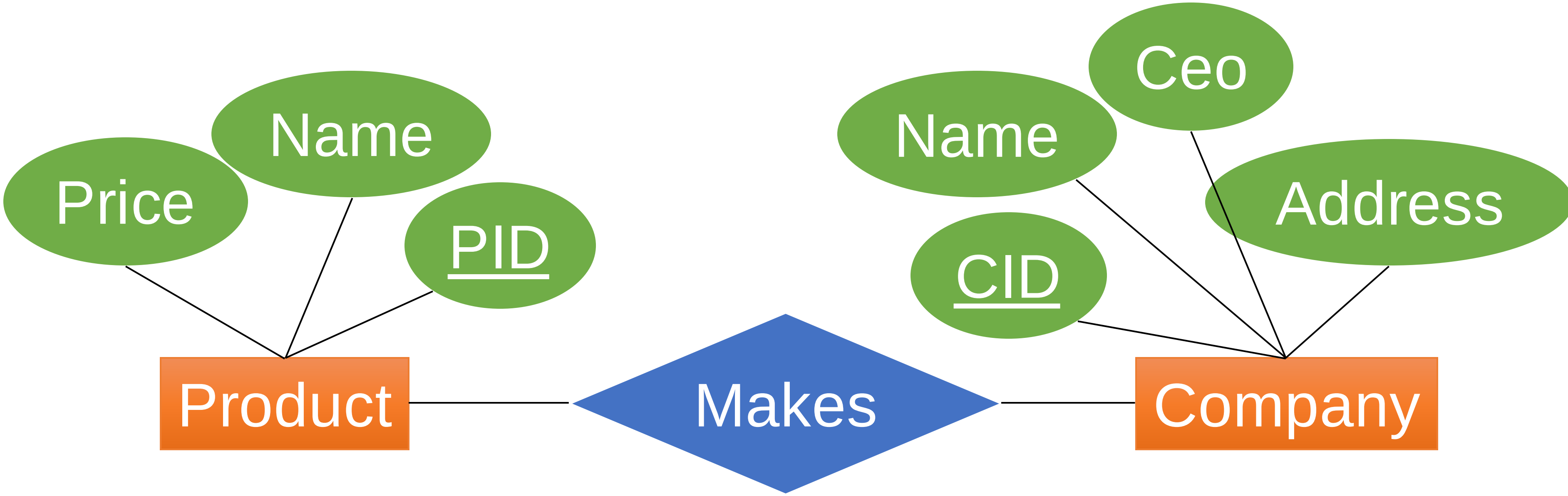
Relationships

- A **relationship** relates entities from two entity sets



Relationships

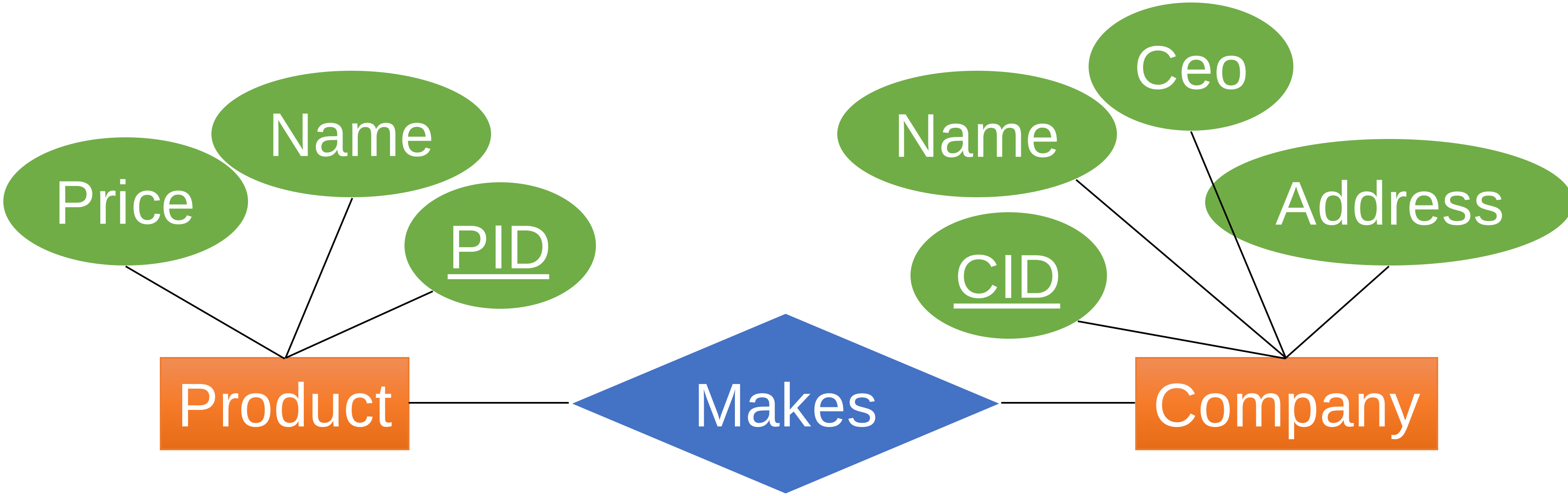
- A **relationship** relates entities from two entity sets



How do
we represent
in SQL?

Relationships

- A **relationship** relates entities from two entity sets

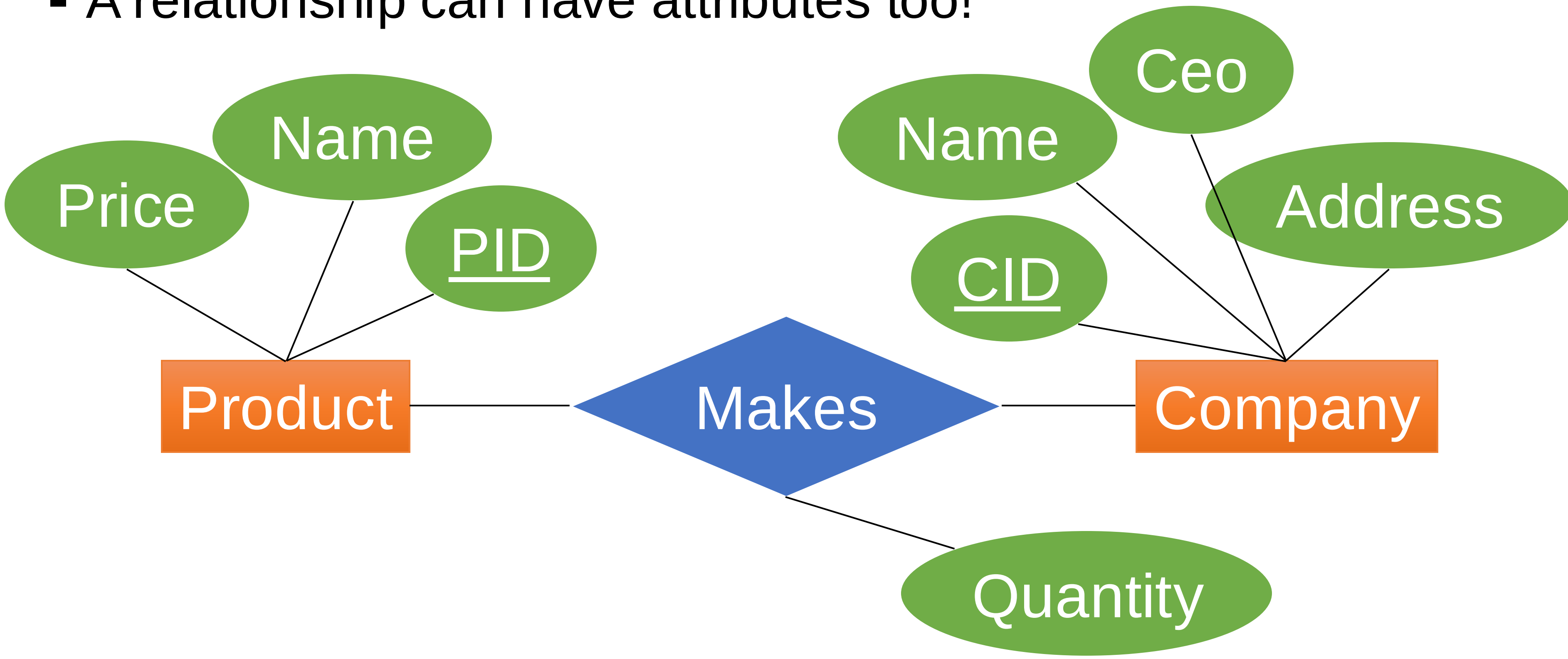


How do
we represent
in SQL?

```
CREATE TABLE
  Makes (
    PID INT References Product,
    CID INT References Company);
```

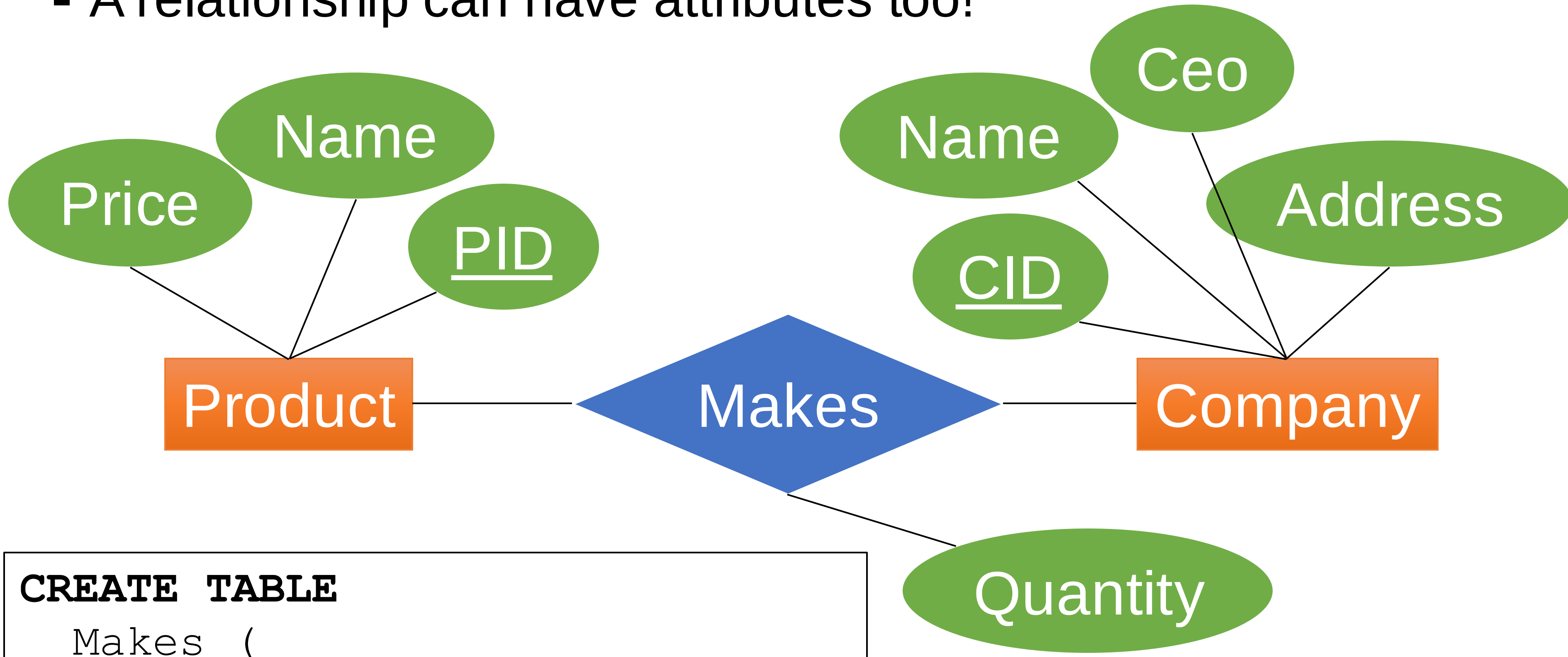
Relationships

- A **relationship** relates entities from two entity sets
- A relationship can have attributes too!



Relationships

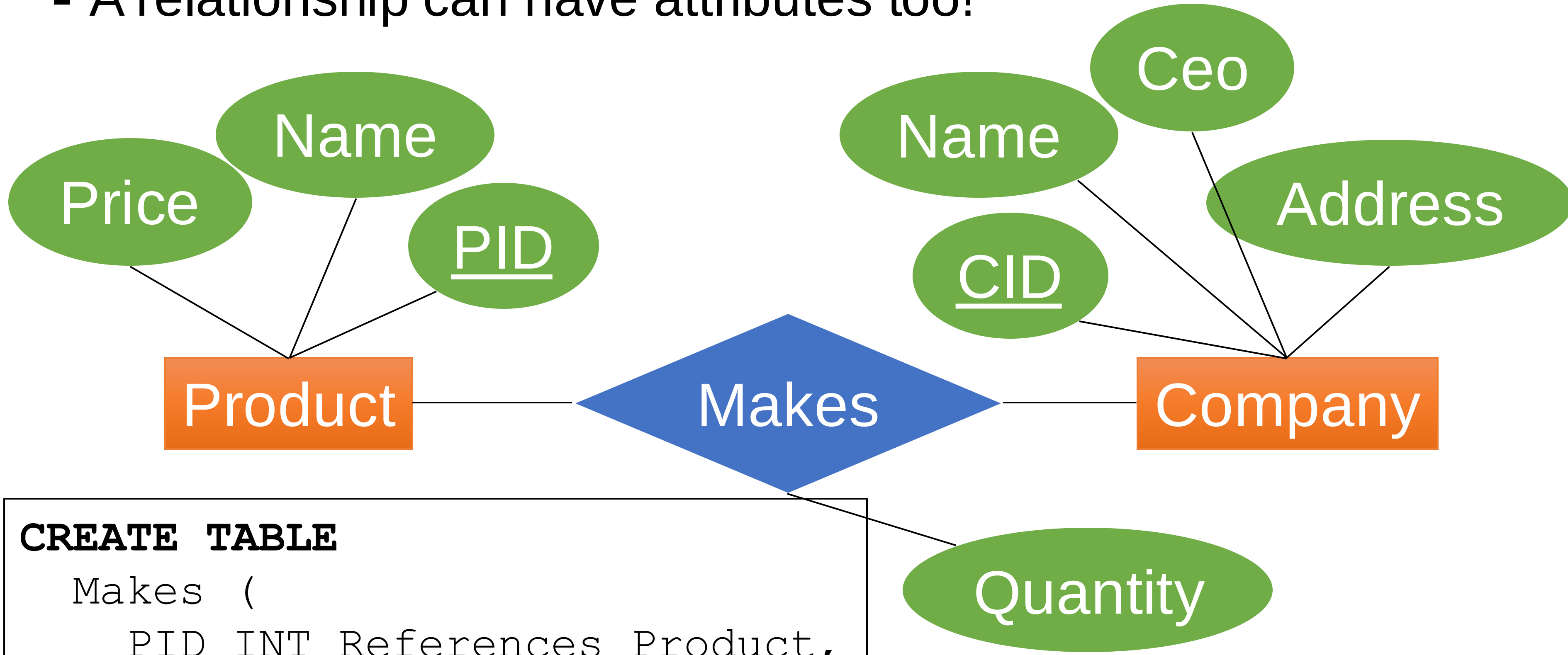
- A **relationship** relates entities from two entity sets
- A relationship can have attributes too!



```
CREATE TABLE
  Makes (
    PID INT References Product,
    CID INT References Company,
    Quantity Int);
```

Relationships

- A **relationship** relates entities from two entity sets
- A relationship can have attributes too!

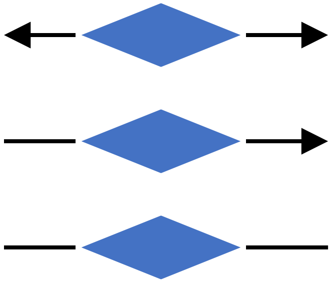


```
CREATE TABLE
  Makes (
    PID INT References Product,
    CID INT References Company,
    Quantity Int,
    Primary Key (UID, CID));
```

Key in a relationships consists of entities only

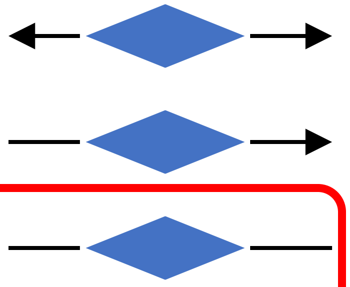
Relationship Multiplicity

- One-to-one
- Many-to-one
- Many-to-many



Relationship Multiplicity

- One-to-one
- Many-to-one
- **Many-to-many**



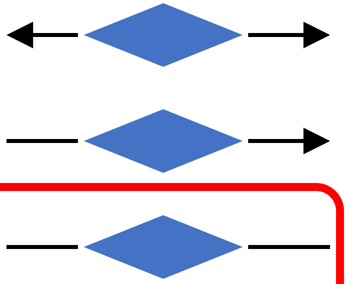
Product

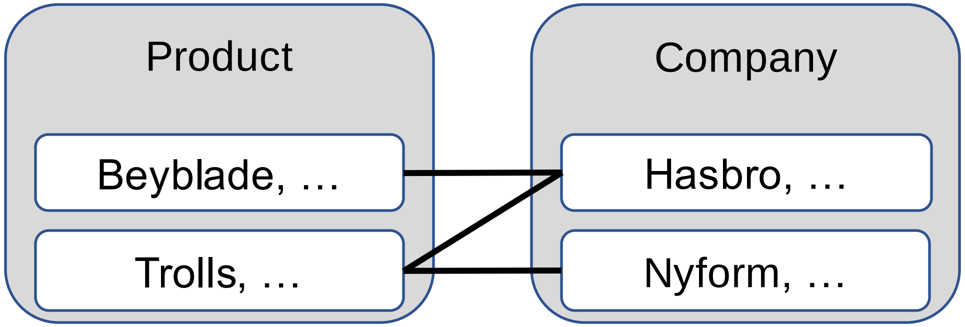
Makes

Company

Relationship Multiplicity

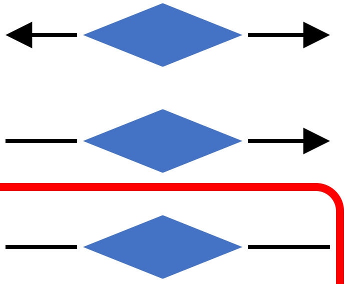
- One-to-one
- Many-to-one
- **Many-to-many**

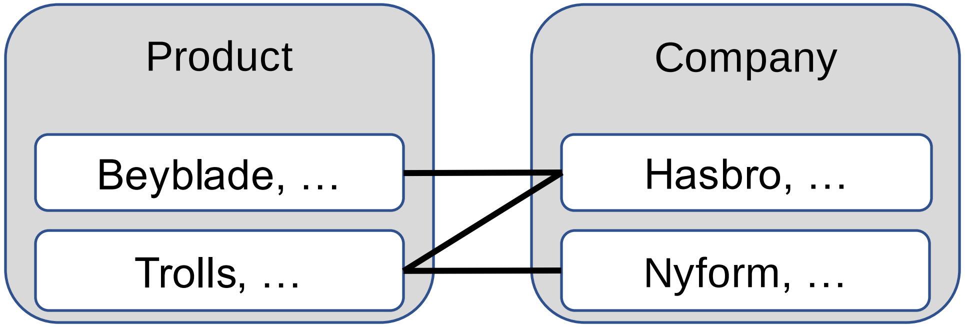




Relationship Multiplicity

- One-to-one
- Many-to-one
- **Many-to-many**

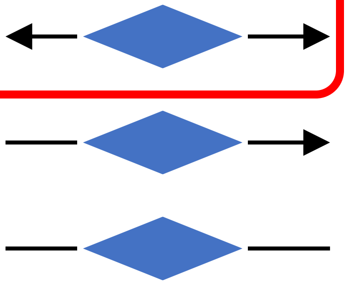




```
CREATE TABLE Product (
   PID int PRIMARY KEY, ...);
CREATE TABLE Company (
   CID int PRIMARY KEY, ...);
CREATE TABLE Makes (
   PID int REFERENCES Product,
   CID int REFERENCES Company);
```

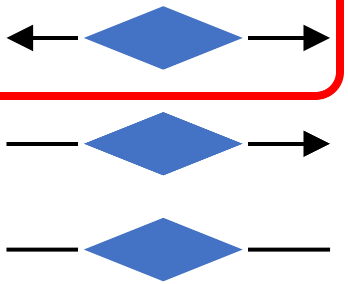
Relationship Multiplicity

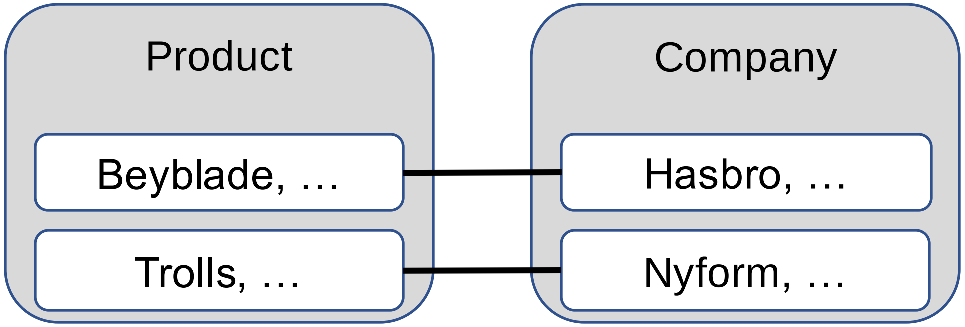
- **One-to-one**
- Many-to-one
- Many-to-many



Relationship Multiplicity

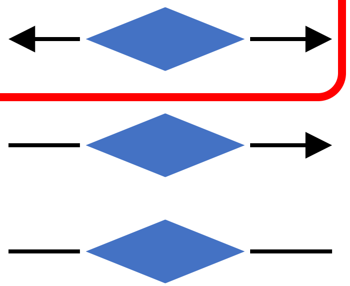
- **One-to-one**
- Many-to-one
- Many-to-many

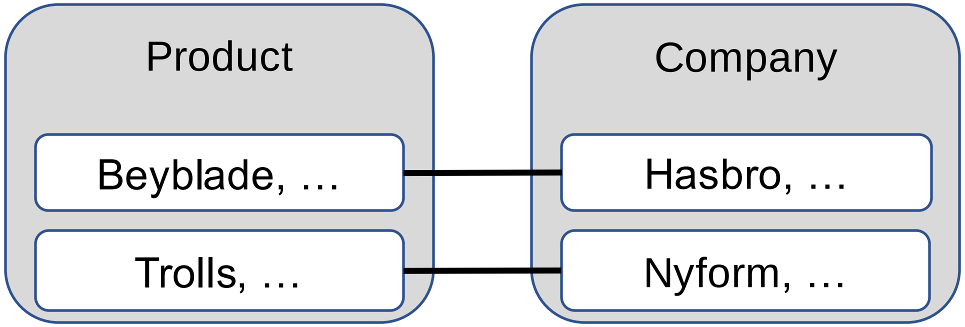




Relationship Multiplicity

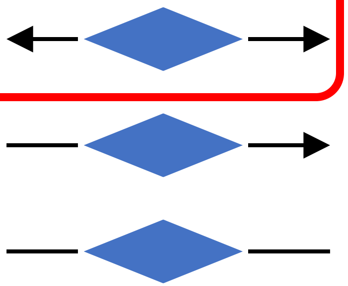
- **One-to-one**
- Many-to-one
- Many-to-many





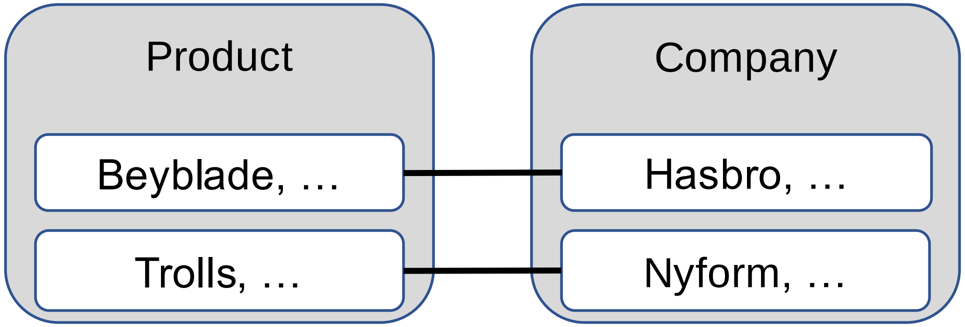
```
CREATE TABLE Product (
  PID int PRIMARY KEY, ...);
CREATE TABLE Company (
  CID int PRIMARY KEY, ...);
CREATE TABLE Makes (
  PID int UNIQUE
          REFERENCES Product,
  CID int UNIQUE
          REFERENCES Company);
```

Relationship Multiplicity



- **One-to-one**
- Many-to-one
- Many-to-many

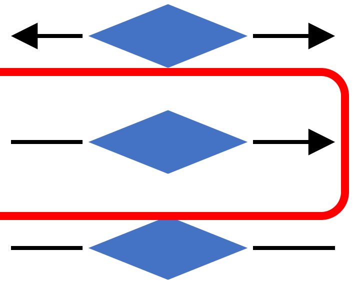
We will revise this shortly



```
CREATE TABLE Product (
  PID int PRIMARY KEY, ...);
CREATE TABLE Company (
  CID int PRIMARY KEY, ...);
CREATE TABLE Makes (
  PID int UNIQUE
          REFERENCES Product,
  CID int UNIQUE
          REFERENCES Company);
```

Relationship Multiplicity

- One-to-one
- **Many-to-one**
- Many-to-many



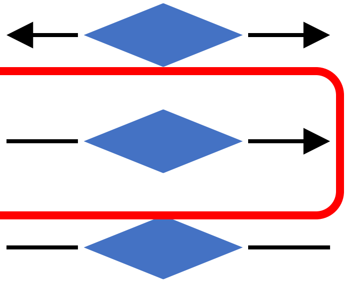
Product

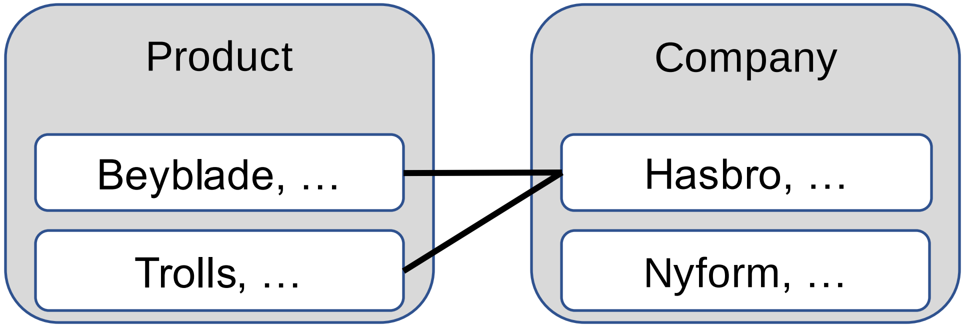
Makes

Company

Relationship Multiplicity

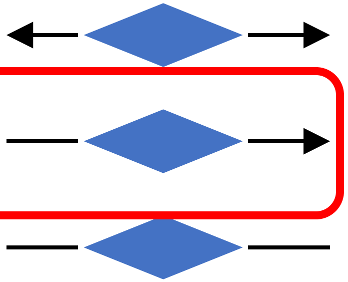
- One-to-one
- **Many-to-one**
- Many-to-many

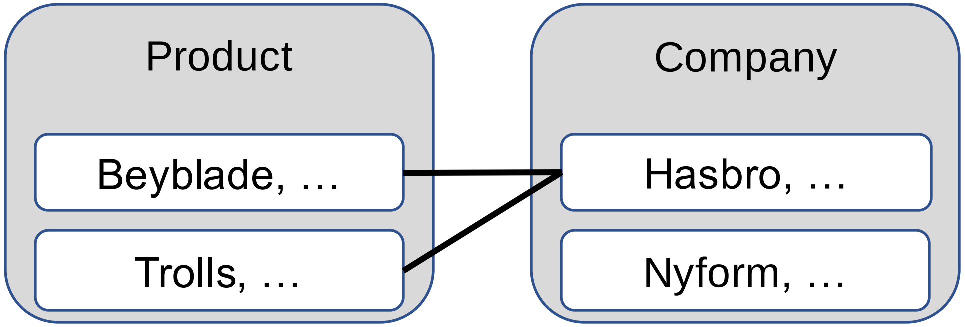




Relationship Multiplicity

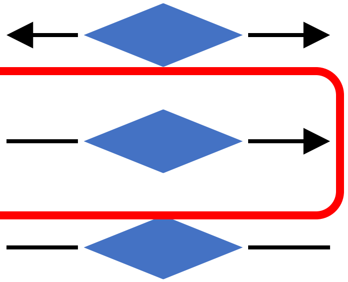
- One-to-one
- **Many-to-one**
- Many-to-many



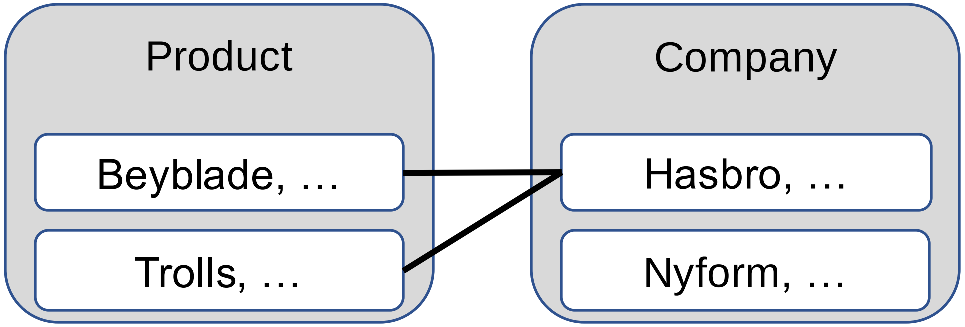


```
CREATE TABLE Product (
  PID int PRIMARY KEY, ...);
CREATE TABLE Company (
  CID int PRIMARY KEY, ...);
CREATE TABLE Makes (
  PID int UNIQUE
          REFERENCES Product,
  CID int REFERENCES Company);
```

Relationship Multiplicity



- One-to-one
- **Many-to-one**
- Many-to-many

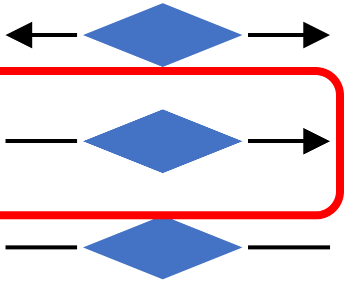


```
CREATE TABLE Product (
  PID int PRIMARY KEY, ...);
CREATE TABLE Company (
  CID int PRIMARY KEY, ...);
CREATE TABLE Makes (
  PID int PRIMARY KEY
          REFERENCES Product,
  CID int REFERENCES Company);
```

Better

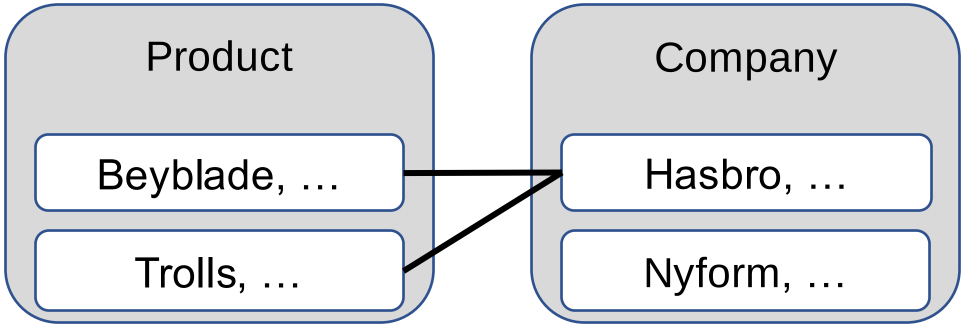
Relationship Multiplicity

- One-to-one
- **Many-to-one**
- Many-to-many



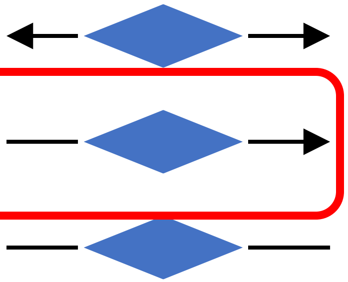
Do we need the Makes table?

```
CREATE TABLE Product (
  PID int PRIMARY KEY, ...);
CREATE TABLE Company (
  CID int PRIMARY KEY, ...);
CREATE TABLE Makes (
  PID int PRIMARY KEY
          REFERENCES Product,
  CID int REFERENCES Company);
```



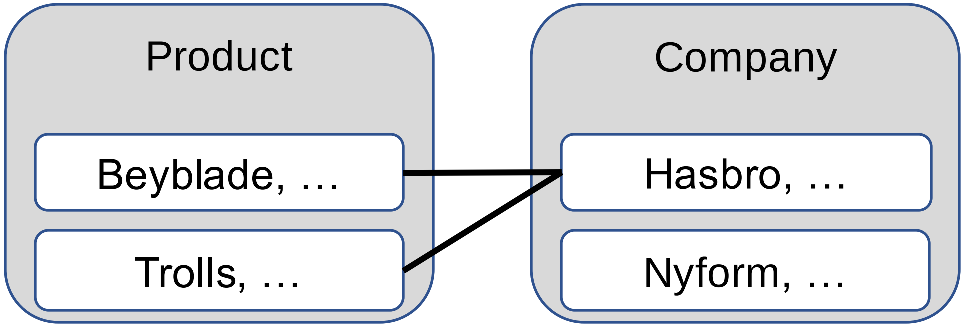
Relationship Multiplicity

- One-to-one
- **Many-to-one**
- Many-to-many



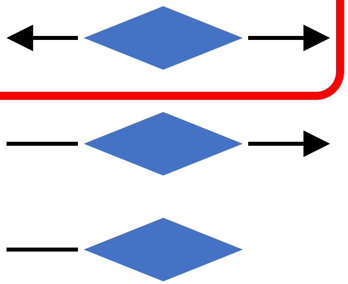
We don’t need separate table!

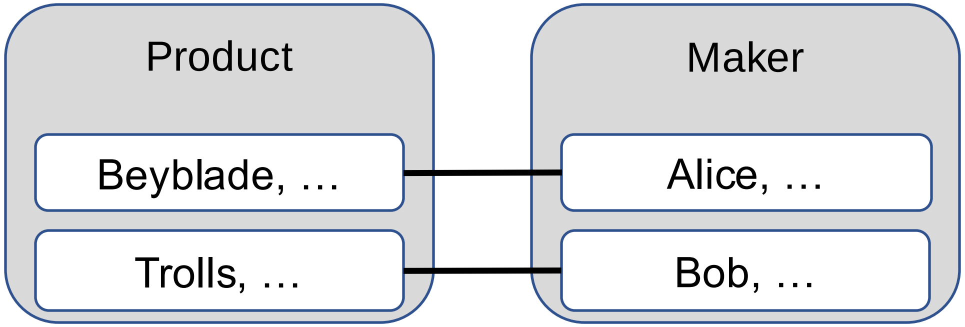
```
CREATE TABLE Product (
   PID int PRIMARY KEY,
   CID int REFERENCES Company,
   ...);
CREATE TABLE Company (
   CID int PRIMARY KEY, ...);
```



Relationship Multiplicity

- **One-to-one**
- Many-to-one
- Many-to-many





Product

Beyblade, …

Trolls, …

Maker

Alice, …

Bob, …

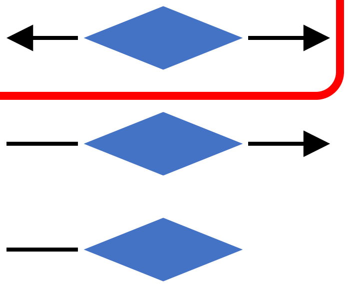
Product

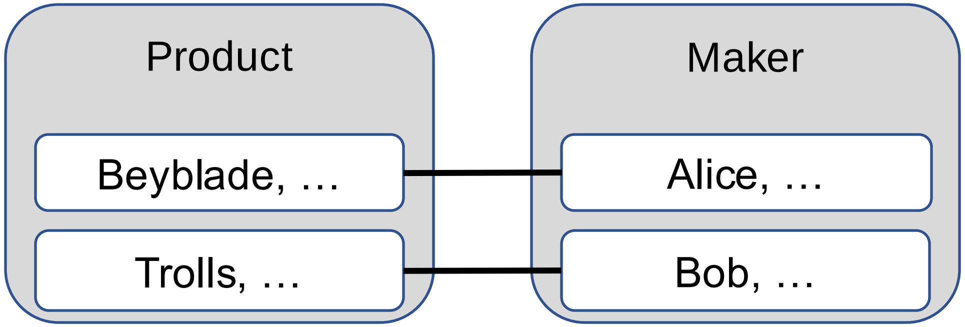
Makes

Maker

Relationship Multiplicity

- **One-to-one**
- Many-to-one
- Many-to-many

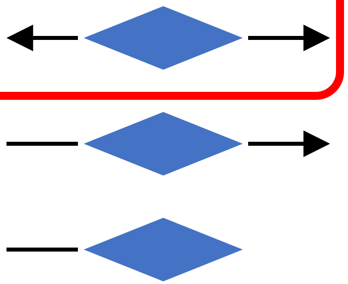




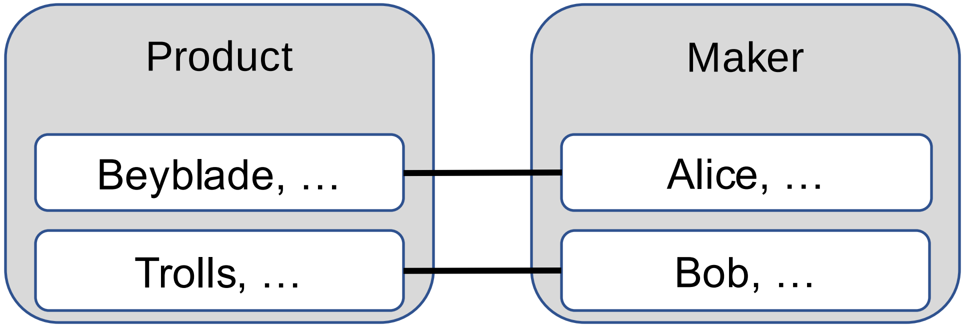
```
CREATE TABLE Product (
  PID int PRIMARY KEY,
  MID Int Reference Maker,
  ...);
CREATE TABLE Maker (
  MID int PRIMARY KEY,
  PID int Reference Product,
  ...);
```

Relationship Multiplicity

- **One-to-one**
- Many-to-one
- Many-to-many



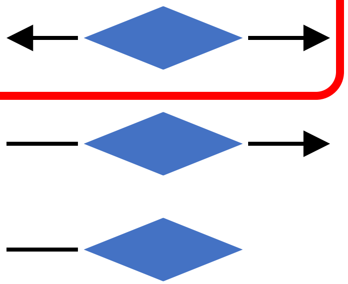
This is one option…

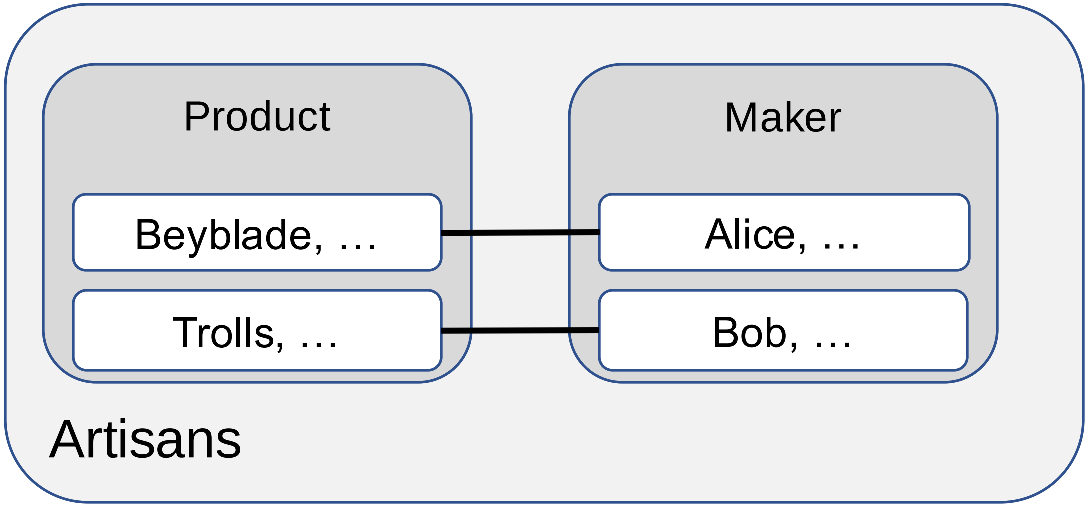


```
CREATE TABLE Product (
   PID int PRIMARY KEY,
   MID Int Reference Maker,
   ...);
CREATE TABLE Maker (
   MID int PRIMARY KEY,
   PID int Reference Product,
   ...);
```

Relationship Multiplicity

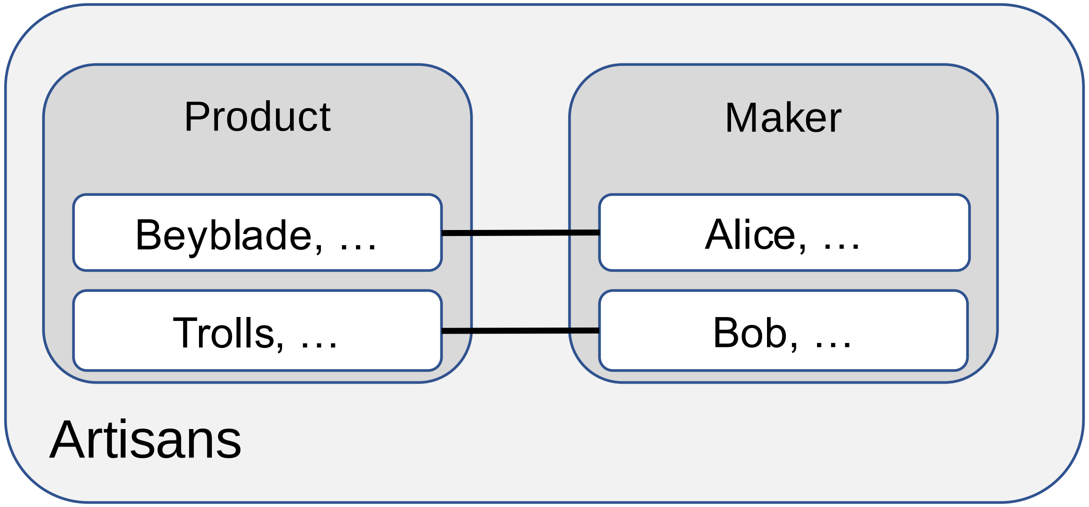
- **One-to-one**
- Many-to-one
- Many-to-many





Relationship Multiplicity

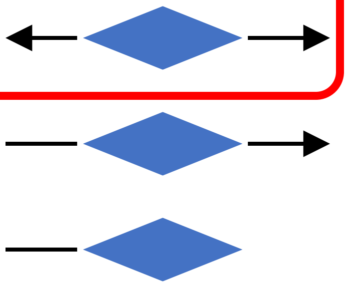
- **One-to-one** ←◆→
- Many-to-one —◆→
- Many-to-many —◆

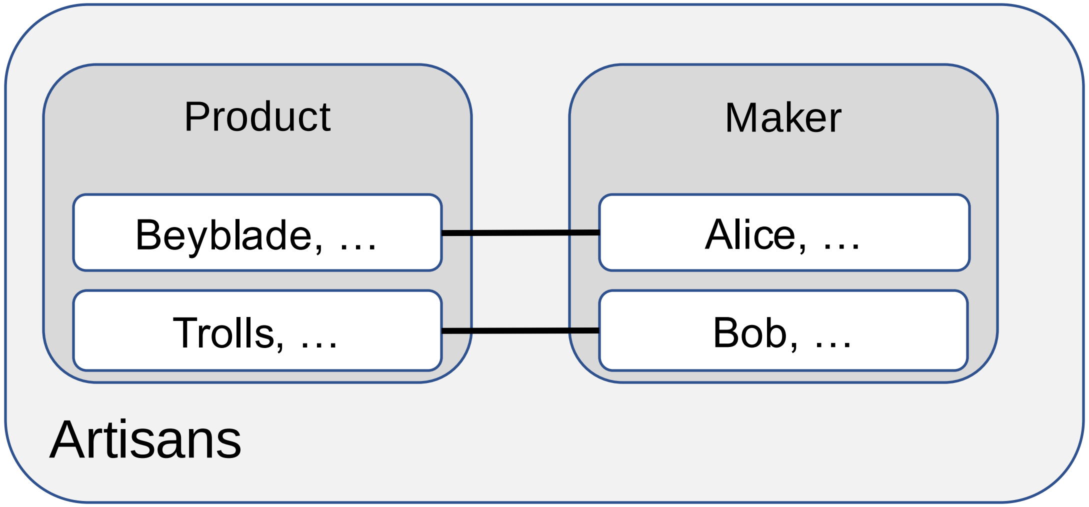


```
CREATE TABLE Artisans (
   AID int PRIMARY KEY,
   ProductName text,
   MakerName text,
   ...);
```

Relationship Multiplicity

- **One-to-one**
- Many-to-one
- Many-to-many

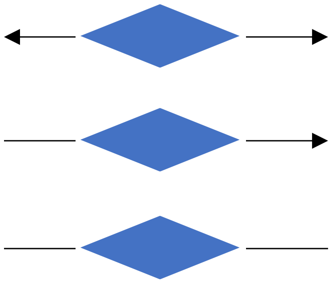




```
CREATE TABLE Artisans (
   AID int PRIMARY KEY,
   ProductName text,
   MakerName text,
   ...);
```

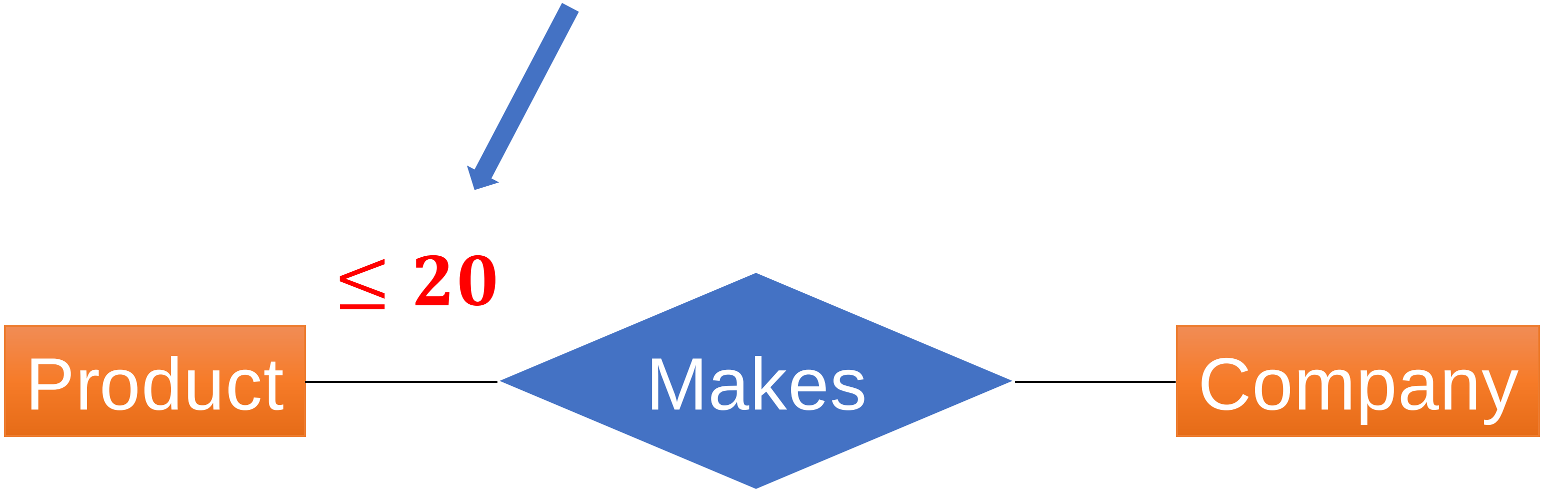
…and this is
the second option

Multiplicity Constraints



- One-to-one
- Many-to-one
- Many-to-many

- Each company manufactures at most 20 products
- OK in ER, but most SQL systems don't support



$\leq \mathbf{20}$

Product

Makes

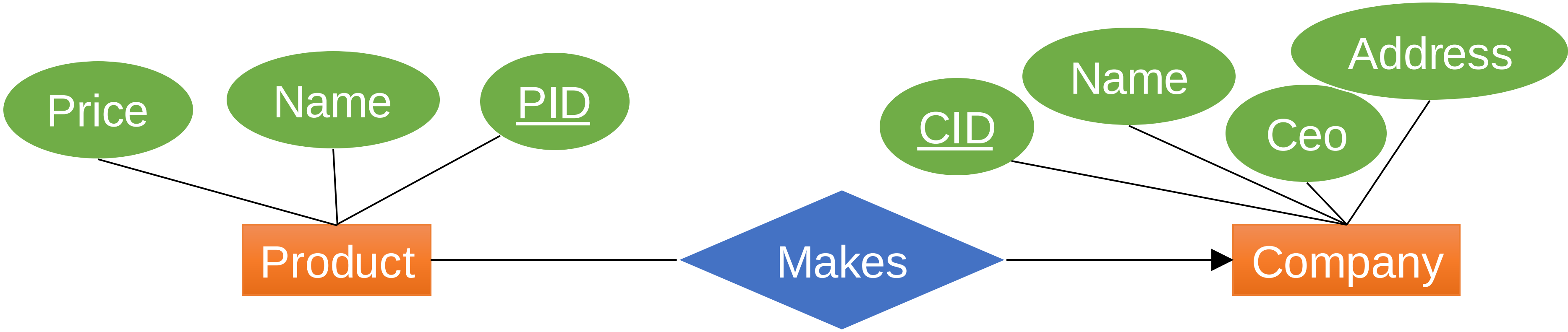
Company

Referential Integrity Constraints

(a complicated name for something very simple)

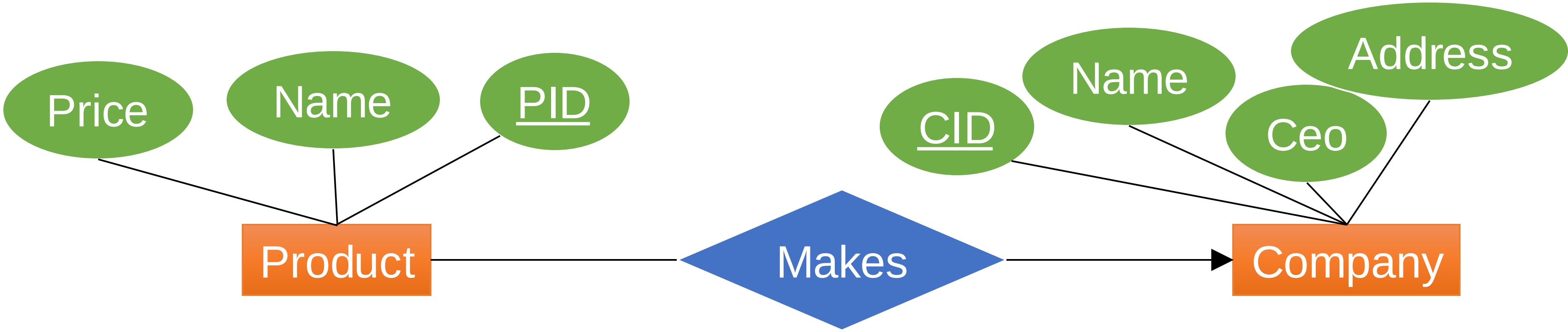
Referential Integrity Constraint

- Regular arrow: at most one
- Rounded arrow: exactly one



Referential Integrity Constraint

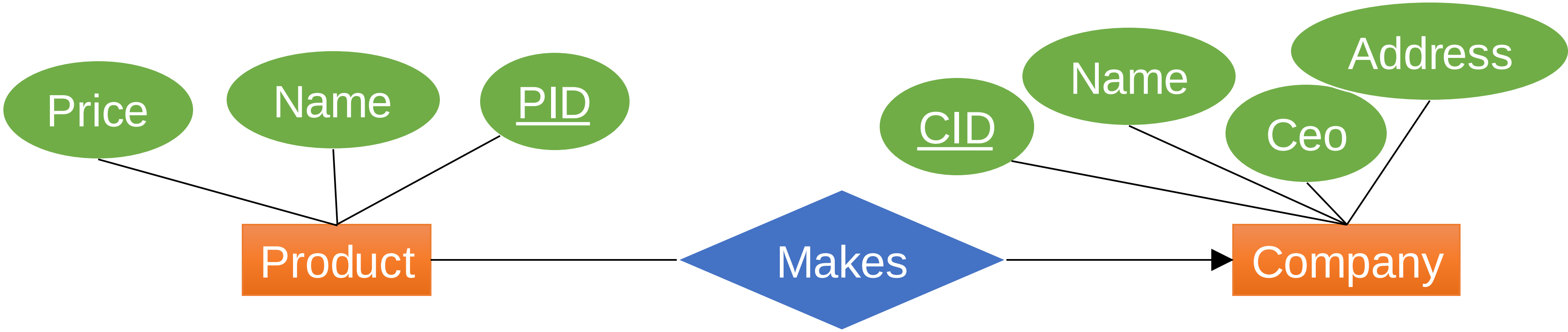
- **Regular arrow**: at most one
- Rounded arrow: exactly one



Referential Integrity Constraint

- **Regular arrow**: at most one
- Rounded arrow: exactly one

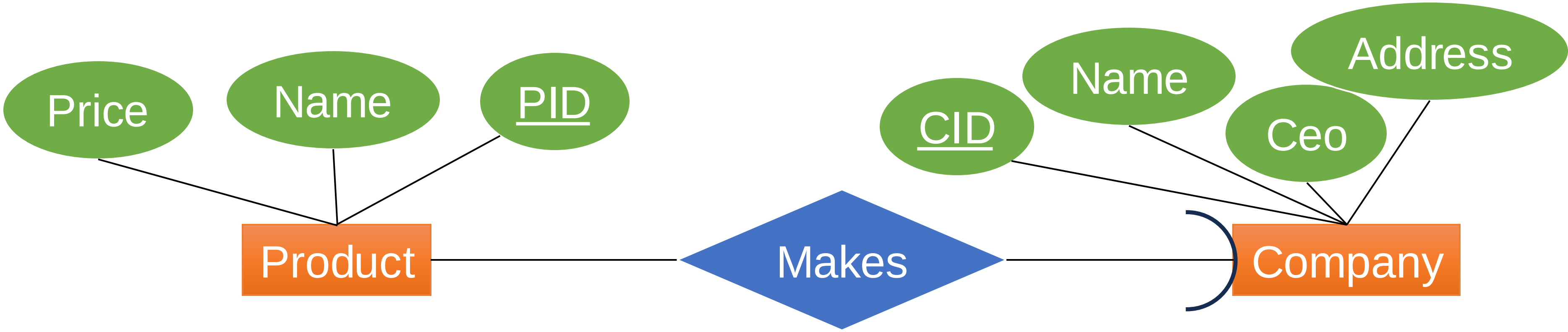
```
CREATE TABLE Product (
   PID int PRIMARY KEY,
   CID int REFERENCES Company,
   ...);
CREATE TABLE Company (
   CID int PRIMARY KEY, ...);
```



Referential Integrity Constraint

- Regular arrow: at most one
- **Rounded arrow**: exactly one

```
CREATE TABLE Product (
   PID int PRIMARY KEY,
   CID int REFERENCES Company
        NOT NULL,
   ...);
CREATE TABLE Company (
   CID int PRIMARY KEY, ...);
```

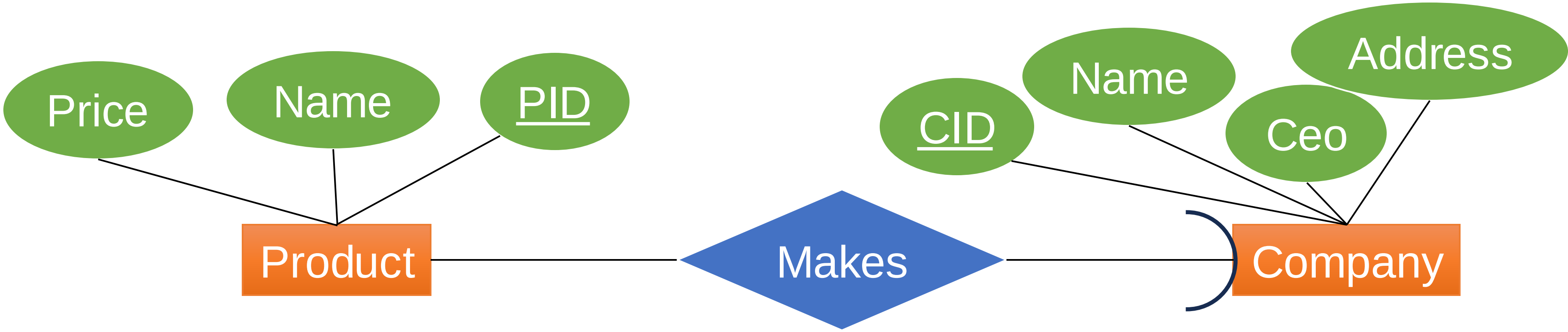


Referential Integrity Constraint

- Regular arrow: at most one
- **Rounded arrow**: exactly one

This is called a
“referential integrity constraint”

```
CREATE TABLE Product (
  PID int PRIMARY KEY,
  CID int REFERENCES Company
      NOT NULL,
  ...);
CREATE TABLE Company (
  CID int PRIMARY KEY, ...);
```



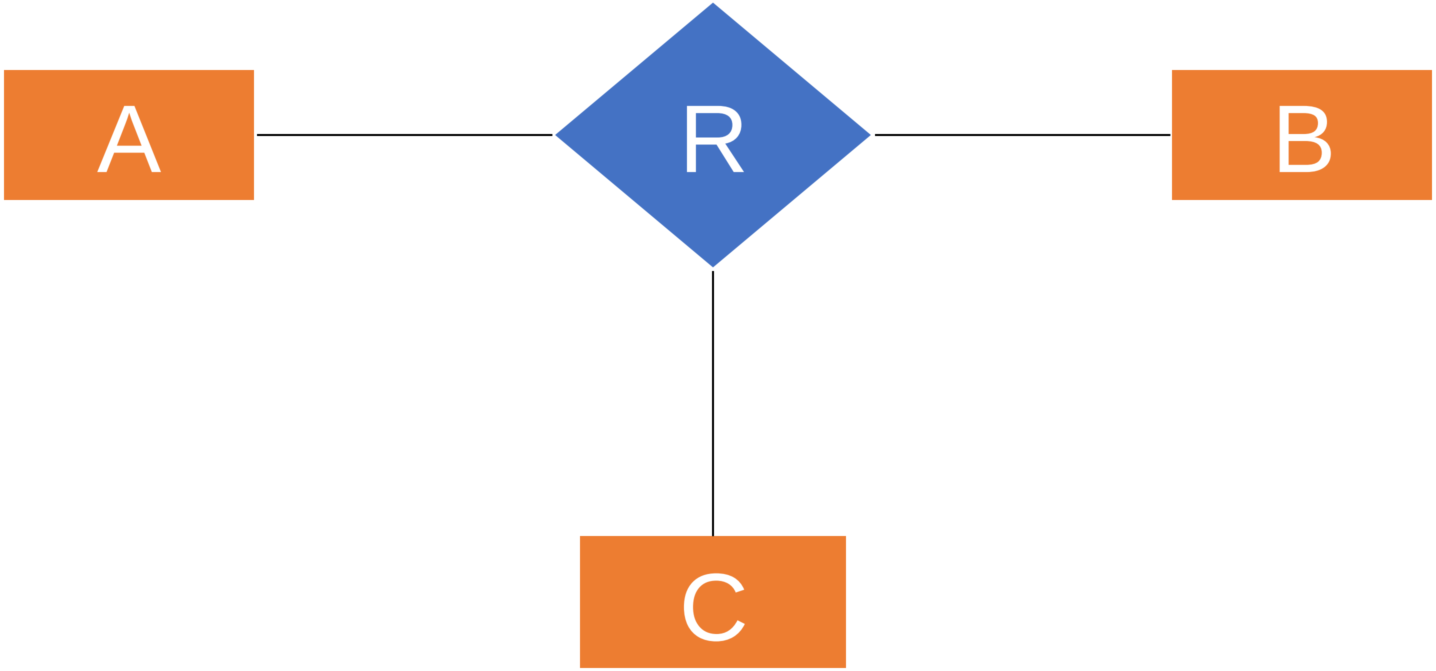
Multi-way Relationships

Multi-Way Relationships

- So far we saw **binary relationships**:
  they connect two entity sets

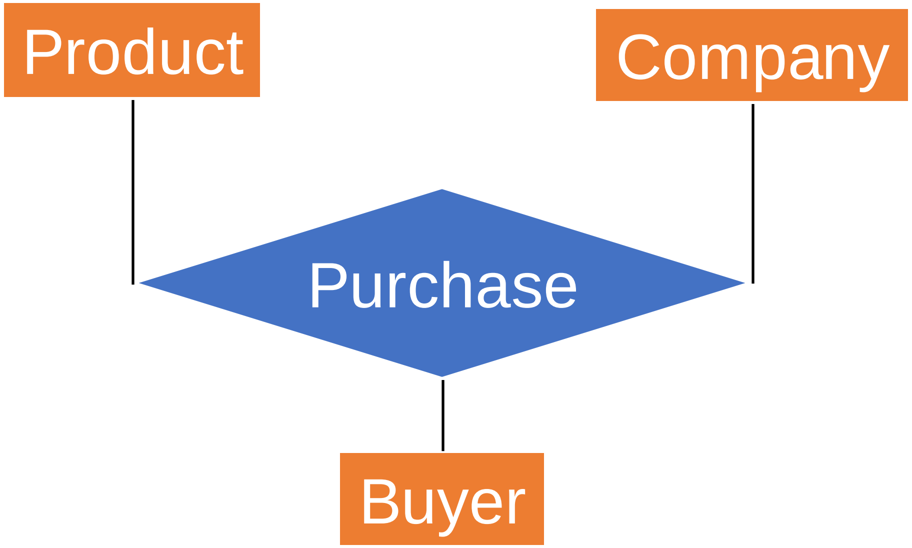
- Also possible: **multi-way relationships**:
  they connect three or more entity sets

Multi-Way Relationships

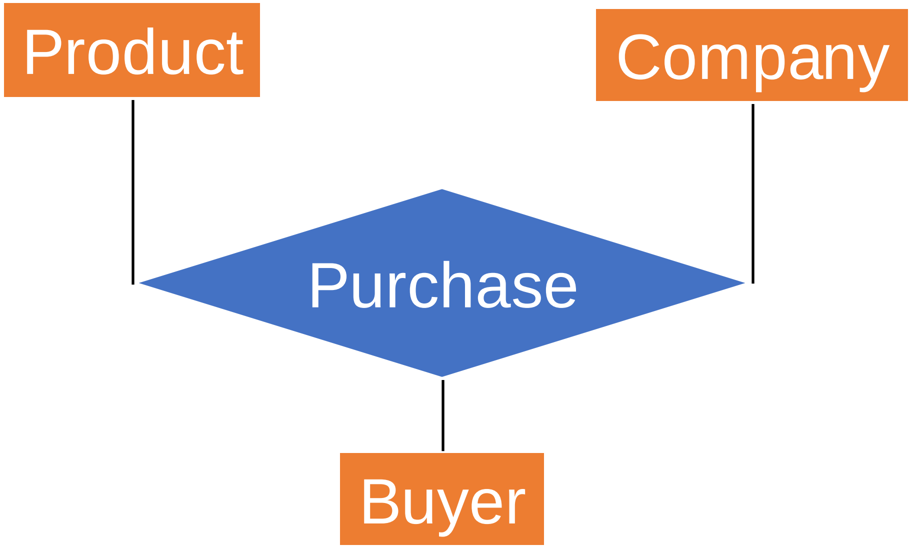


R is a subset of the cross product: $R \subseteq A \times B \times C$

Multi-Way Relationships

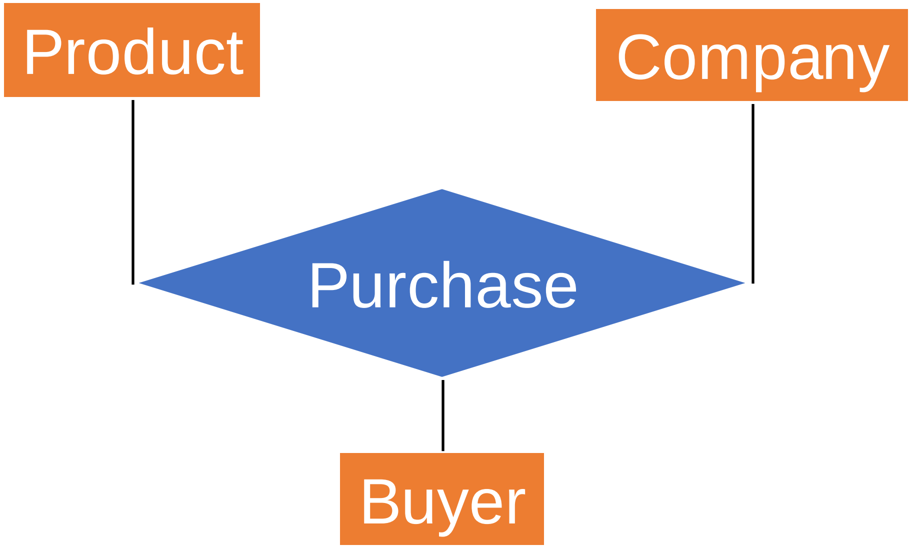


Multi-Way Relationships



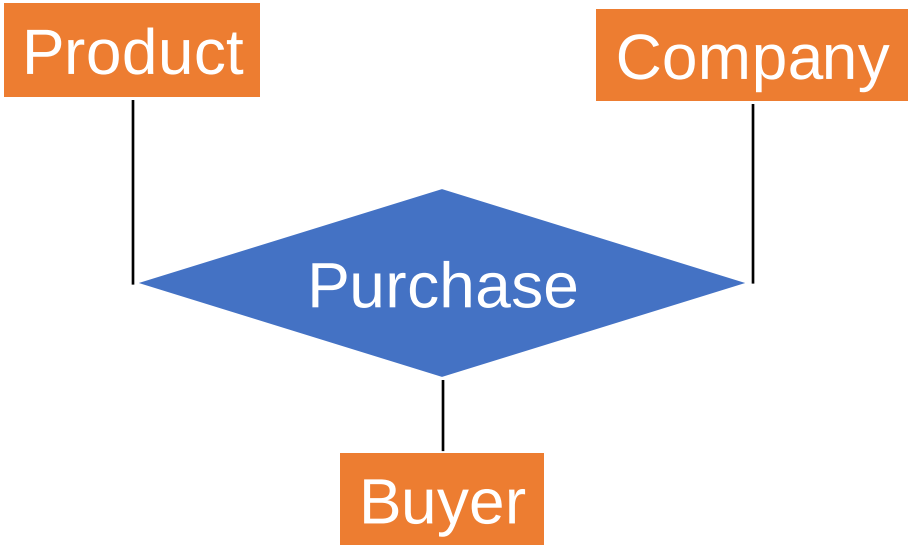
```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);
```

Multi-Way Relationships



```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    ...);
```

Multi-Way Relationships

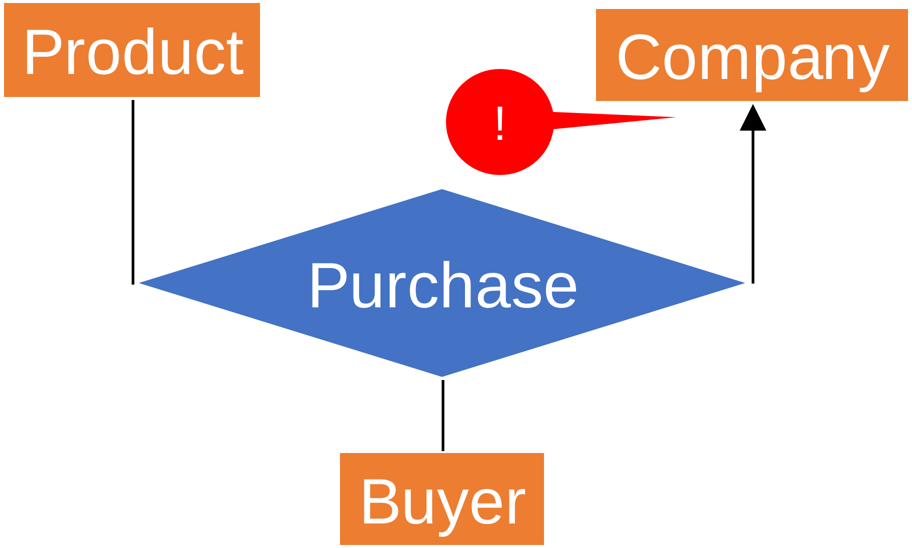


```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
…		

Multi-Way Relationships

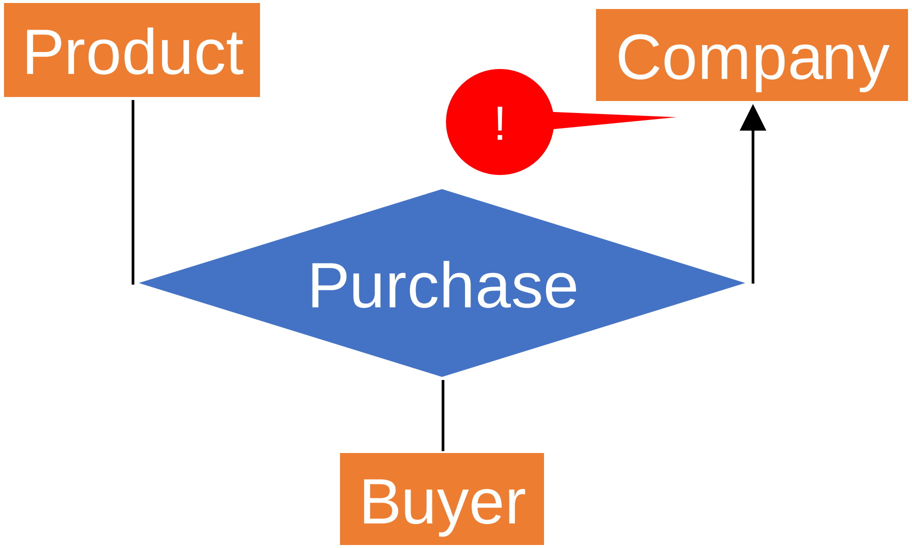


```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
…		

Multi-Way Relationships



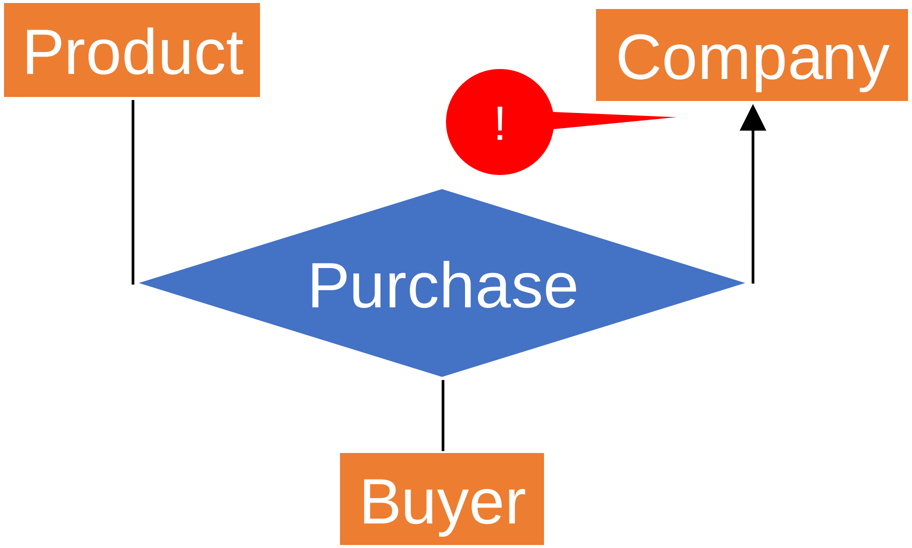
```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
…		

Arrow means:
a buyer always buys a product from the same company

Multi-Way Relationships



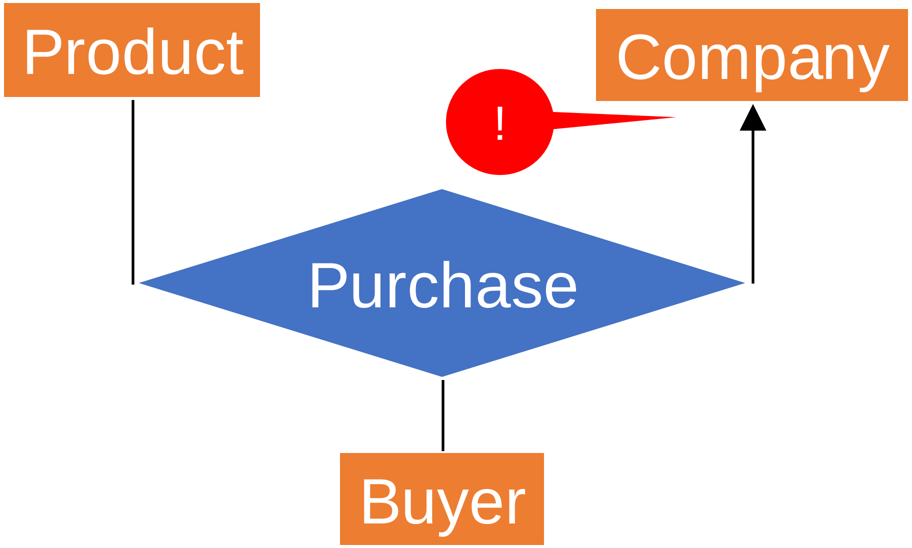
```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    PRIMARY KEY (BID, PID),
    ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
…		

Arrow means:
a buyer always buys a product from the same company

Multi-Way Relationships



```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    PRIMARY KEY (BID, PID),
    ...);
```

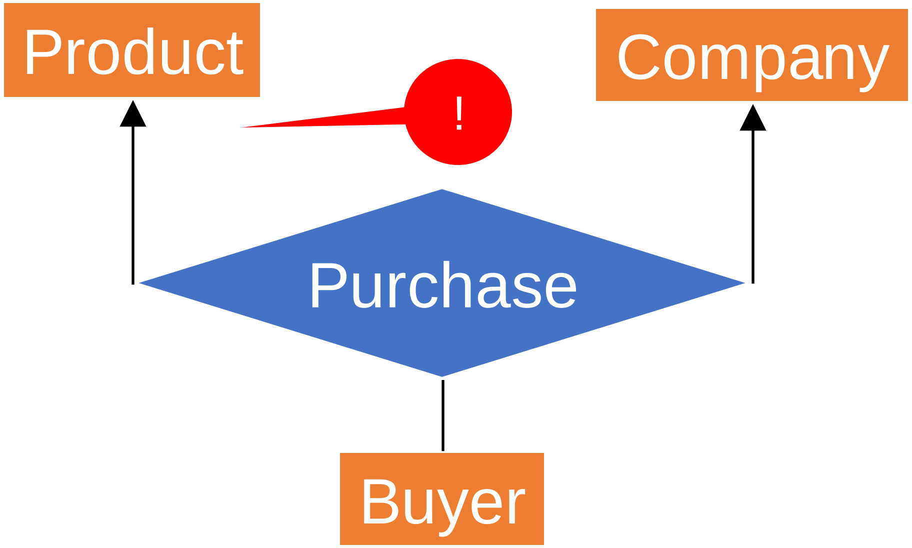
Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
0035 (soap)	**456 (Dove)**	**555 (Alice)**

Arrow means:
a buyer always buys a product from the same company

Not allowed

Multi-Way Relationships



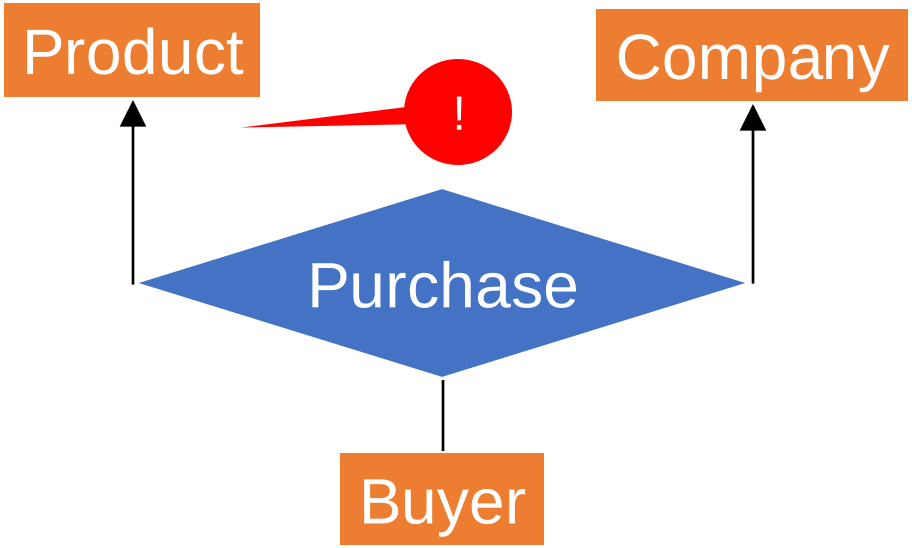
```
CREATE TABLE Product (
     PID INT PRIMARY KEY,...);
CREATE TABLE Company (
     CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
     BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
     PID INT REFERENCES Product,
     CID INT REFERENCES Company,
     BID INT REFERENCES Buyer,
     PRIMARY KEY (BID, PID),
     ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
…		

What do **two arrows** this mean?

Multi-Way Relationships



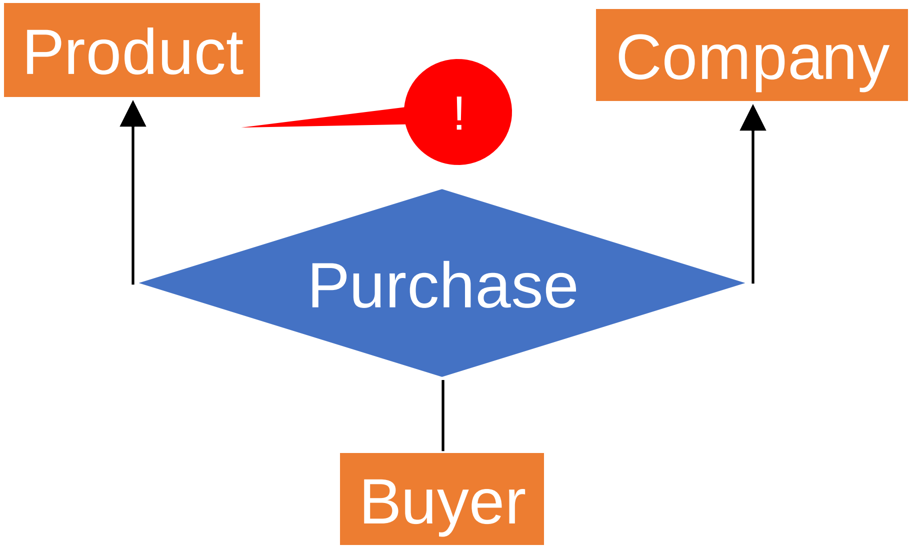
```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    PRIMARY KEY (BID, PID),
    ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
…		

What do **two arrows** this mean?
We read each arrow separately:

Multi-Way Relationships



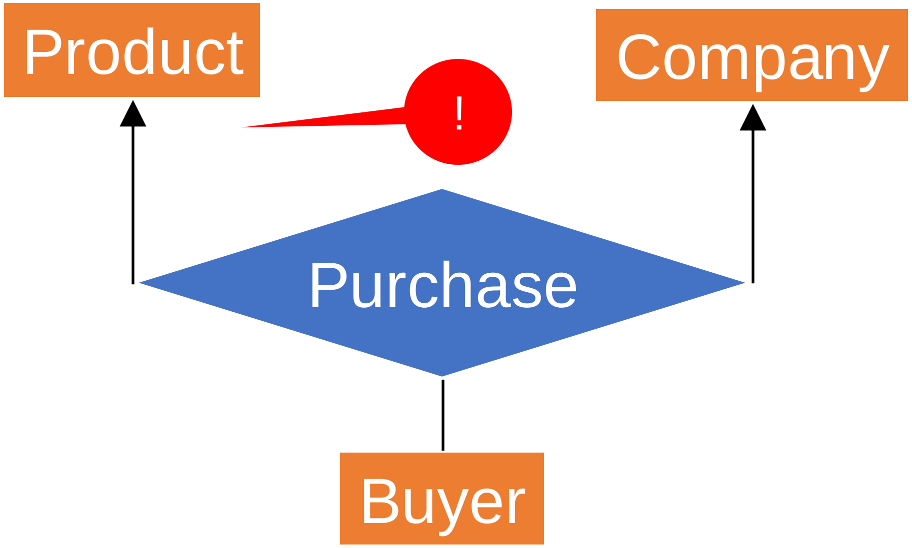
```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    UNIQUE (BID, PID),
    UNIQUE (BID, CID),
    ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
…		

What do **two arrows** this mean?
We read each arrow separately:

Multi-Way Relationships



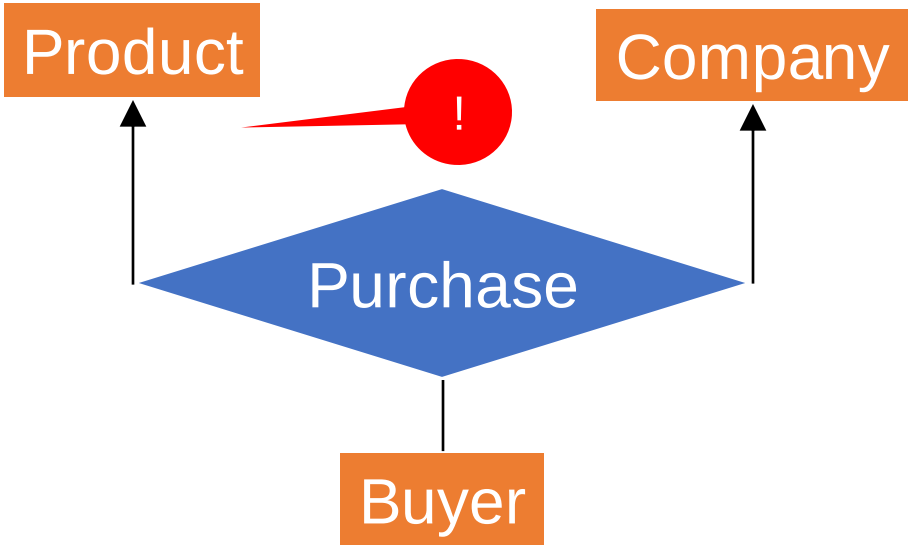
```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    UNIQUE (BID, PID),
    UNIQUE (BID, CID),
    ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
…		

What do **two arrows** this mean?
We read each arrow separately:
…
and every buyer buys at most one product from each company

Multi-Way Relationships



```
CREATE TABLE Product (
    PID INT PRIMARY KEY,...);
CREATE TABLE Company (
    CID INT PRIMARY KEY,...);
CREATE TABLE Buyer (
    BID INT PRIMARY KEY,...);

CREATE TABLE Purchase (
    PID INT REFERENCES Product,
    CID INT REFERENCES Company,
    BID INT REFERENCES Buyer,
    UNIQUE (BID, PID),
    UNIQUE (BID, CID),
    ...);
```

Purchase

PID	CID	BID
0035 (soap)	345 (Dial)	555 (Alice)
0035 (soap)	345 (Dial)	666 (Bob)
0041 (lotion)	123 (Nivea)	555 (Alice)
06 (soft soap)	**345 (Dial)**	**555 (Alice)**

What do **two arrows** this mean?
We read each arrow separately:
…
and every buyer buys at most one product from each company

Summary of Relationships

- Multiplicity constraints:
  - Many-many: separate table
  - Many-one: no separate table
  - Multiplicity constraints: only in ER

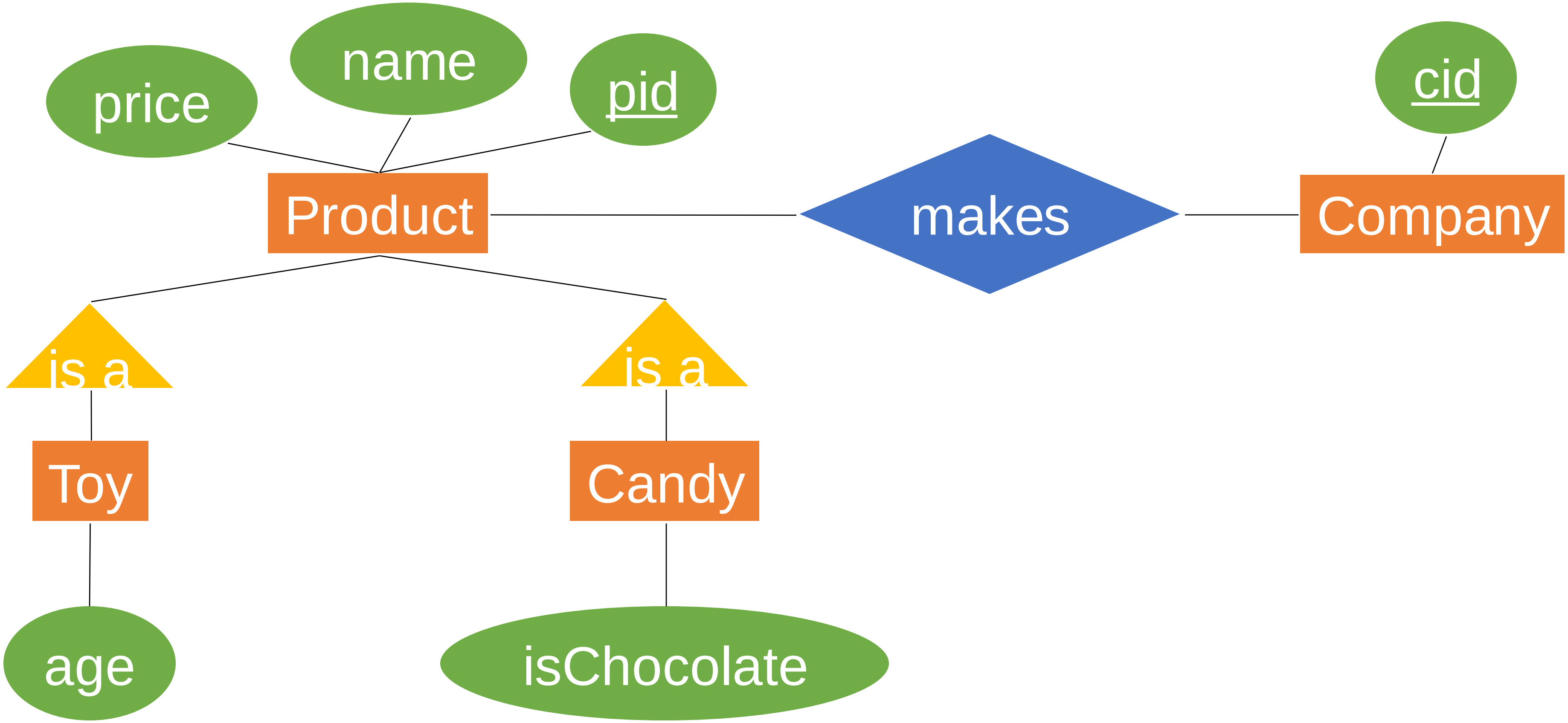
- Referential integrity: foreign key NOT NULL

- Multi-way relationships: foreign key to each

Subclasses

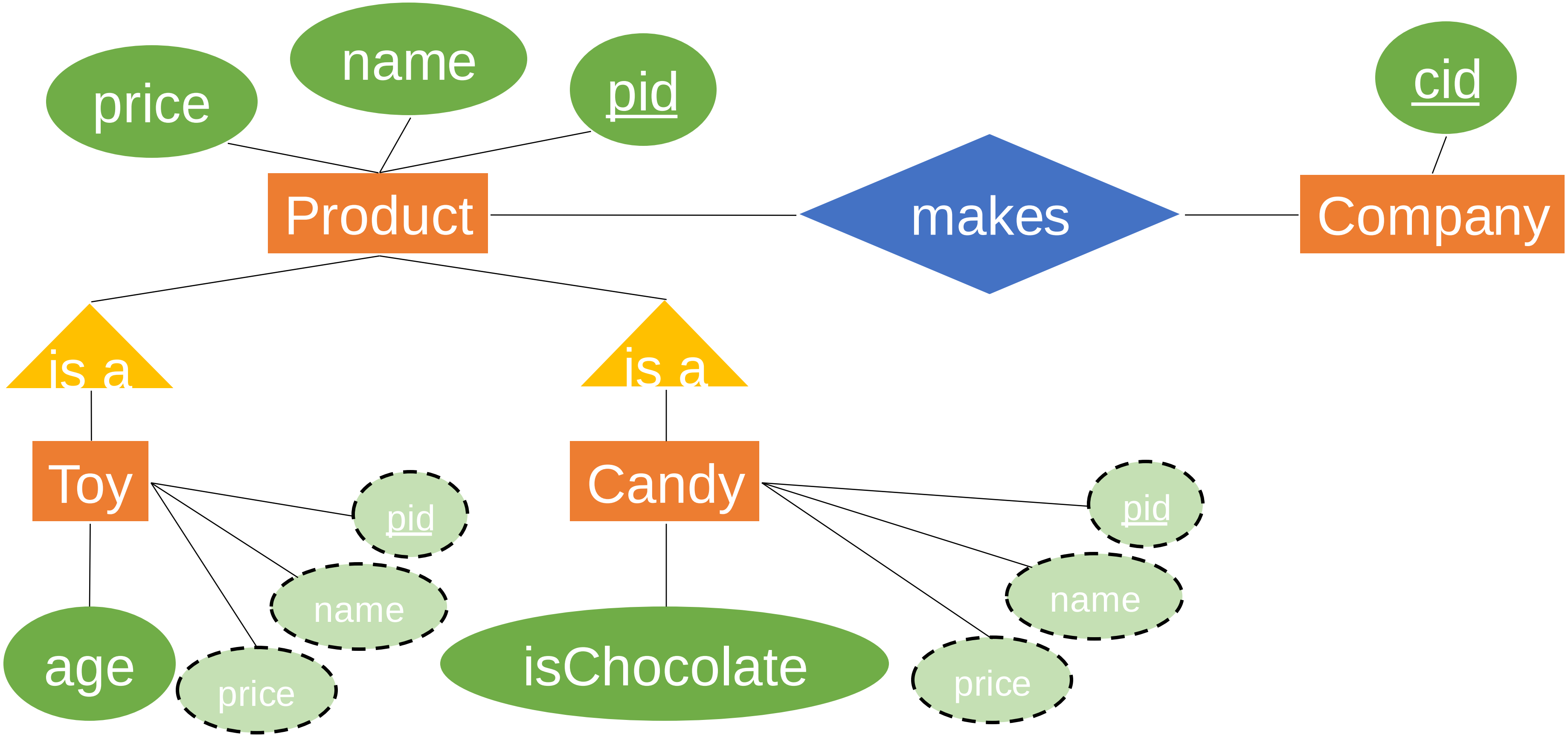
Subclassing

- Entity set may be a **subclass** of another entity set



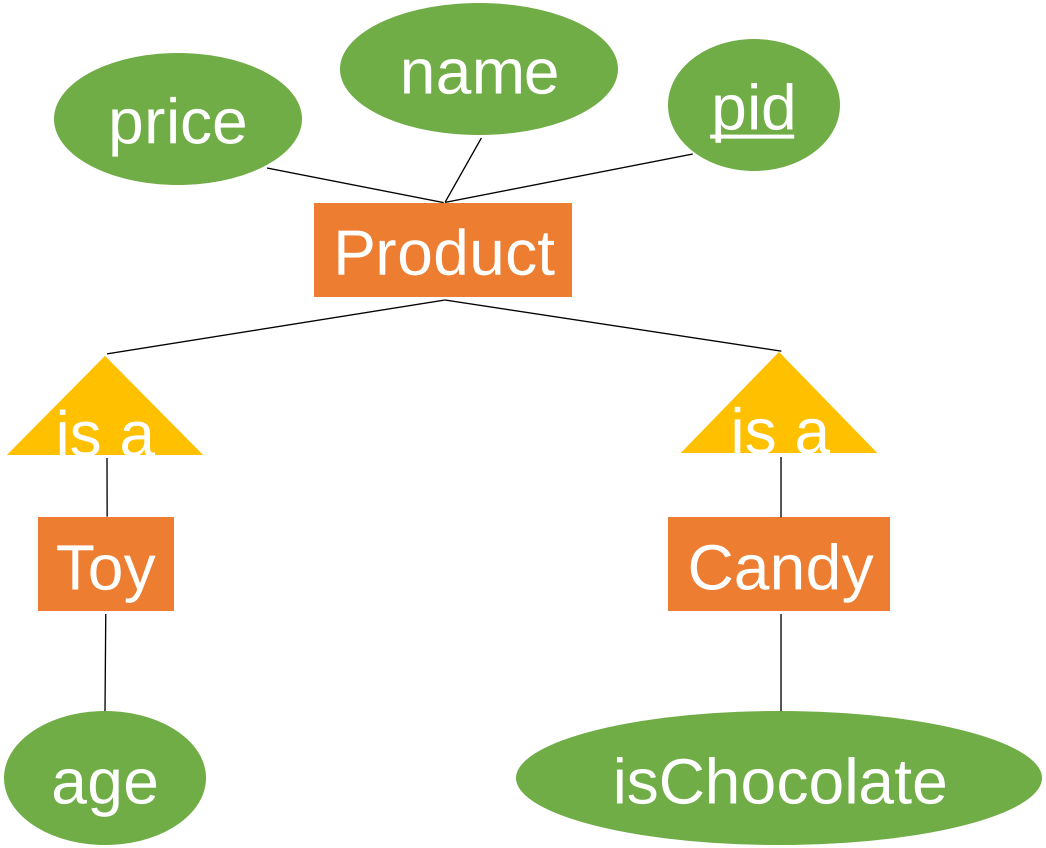
Subclassing

- Entity set may be a subclass of another entity set
- **Inherits** attributes of superclass



Representing Subclasses in SQL

- Each entity set becomes a relation

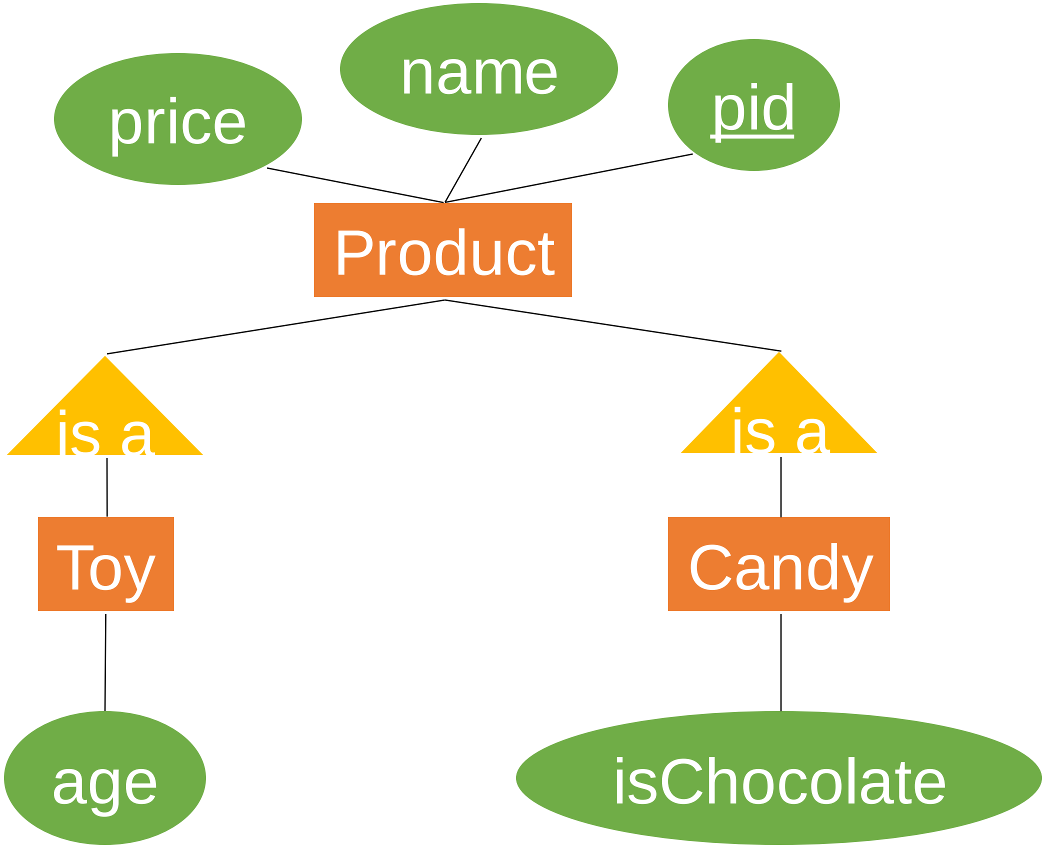


Representing Subclasses in SQL

- Each entity set becomes a relation

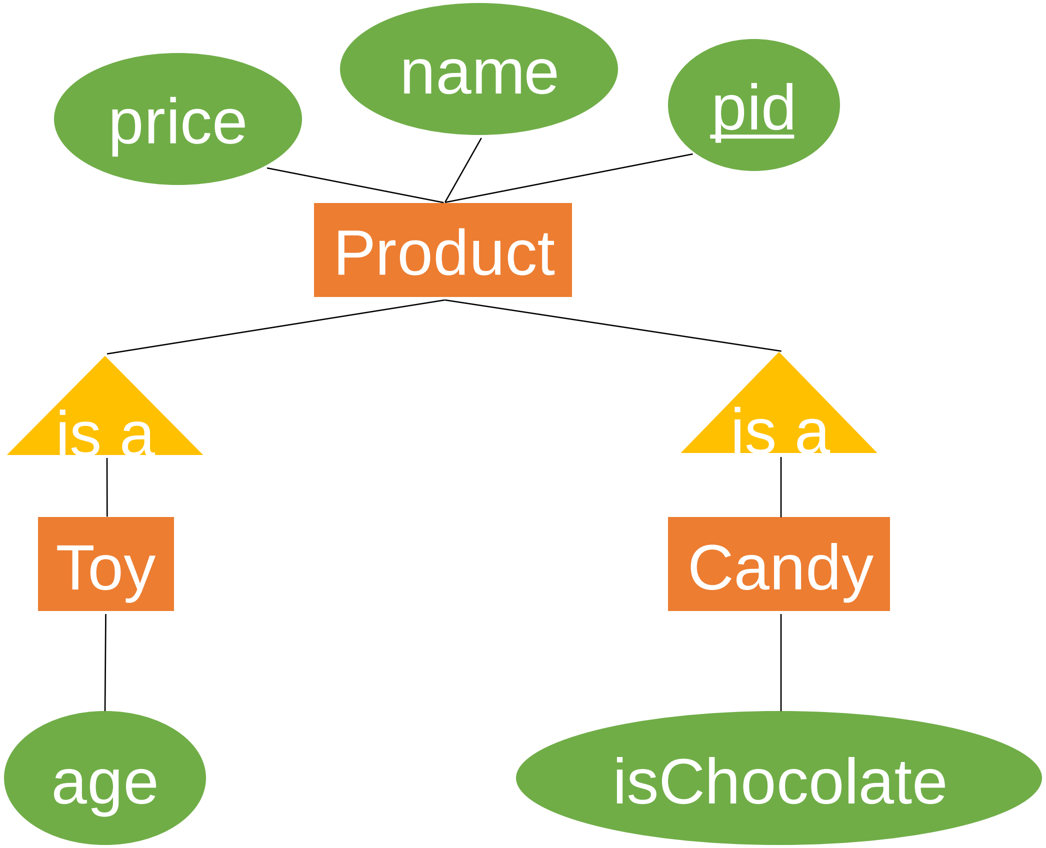
Product

<u>pid</u>	name	price
012	Lego	99
123	M&M	5
234	Computer	2999
345	Ball	15
456	Skittles	3
567	M&M toy	49



Representing Subclasses in SQL

- Each entity set becomes a relation



Product

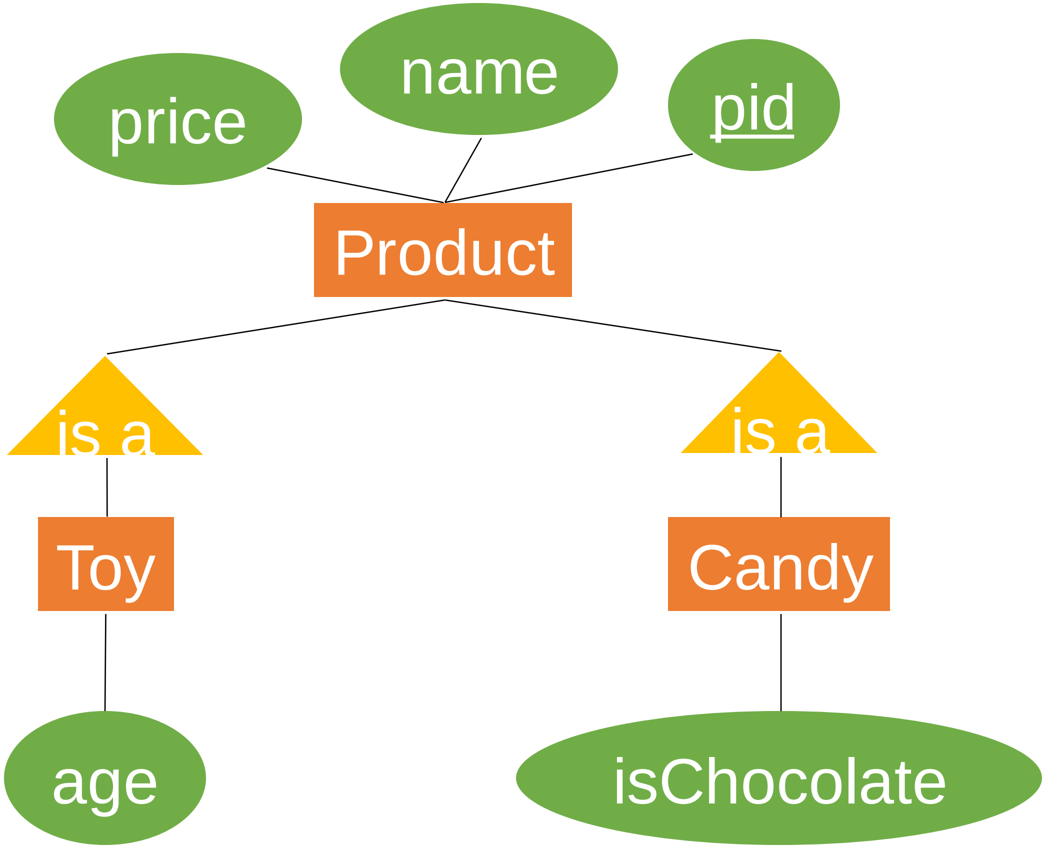
pid	name	price
012	Lego	99
123	M&M	5
234	Computer	2999
345	Ball	15
456	Skittles	3
567	M&M toy	49

Toy

pid	age
012	8
345	2
567	3

Representing Subclasses in SQL

- Each entity set becomes a relation



Product

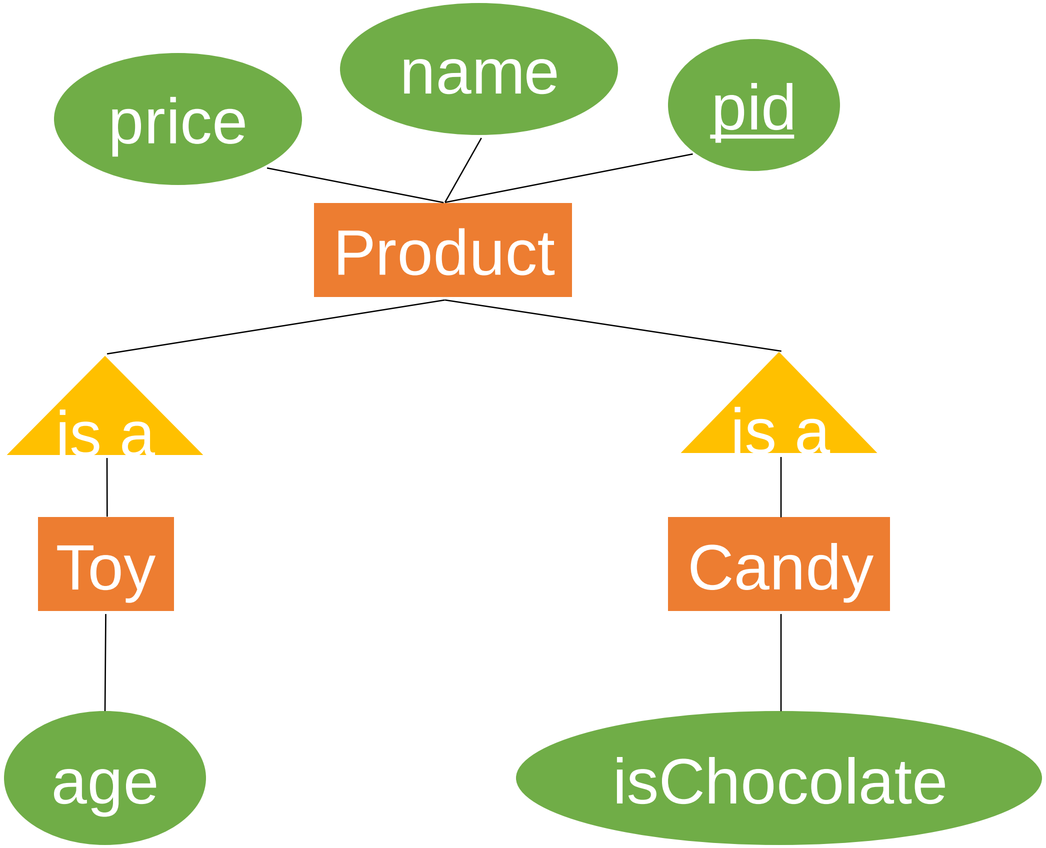
pid	name	price
012	Lego	99
123	M&M	5
234	Computer	2999
345	Ball	15
456	Skittles	3
567	M&M toy	49

Toy

pid	age
012	8
345	2
567	3

Representing Subclasses in SQL

- Each entity set becomes a relation



Product

pid	name	price
012	Lego	99
123	M&M	5
234	Computer	2999
345	Ball	15
456	Skittles	3
567	M&M toy	49

Toy

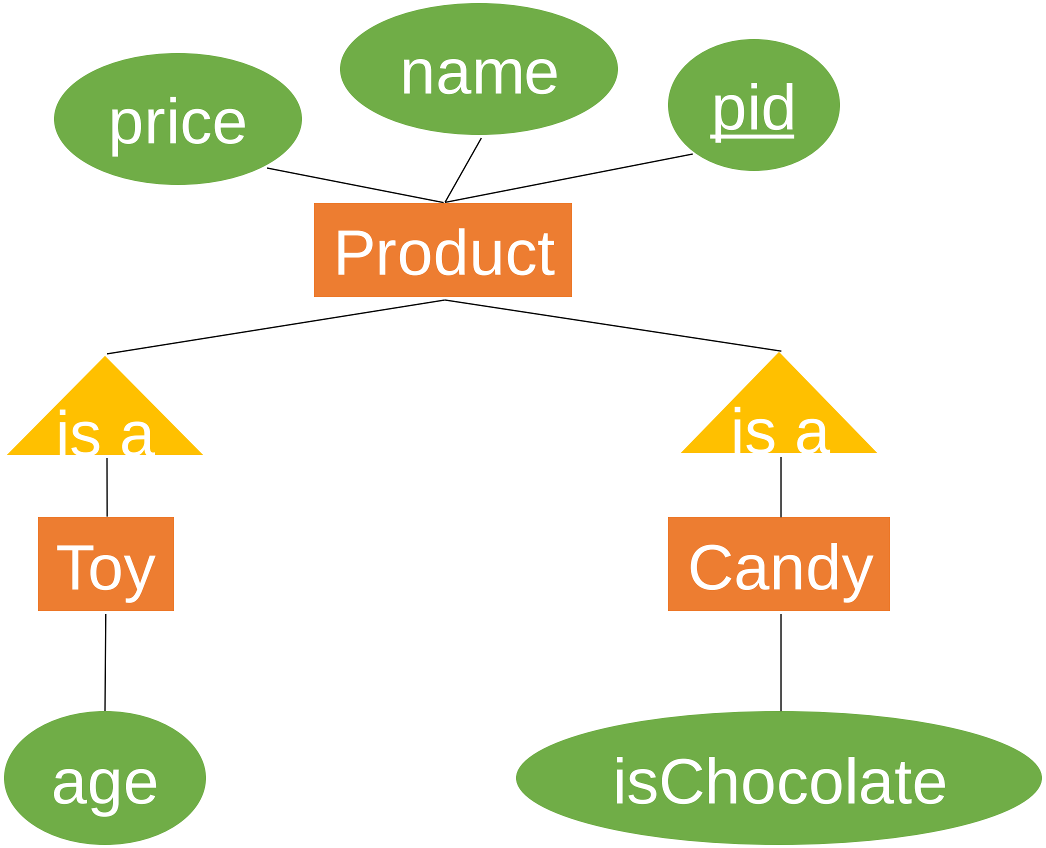
pid	age
012	8
345	2
567	3

Candy

pid	isChoc
123	yes
456	no
567	no

Representing Subclasses in SQL

- Each entity set becomes a relation



Product

pid	name	price
012	Lego	99
123	M&M	5
234	Computer	2999
345	Ball	15
456	Skittles	3
567	M&M toy	49

Toy

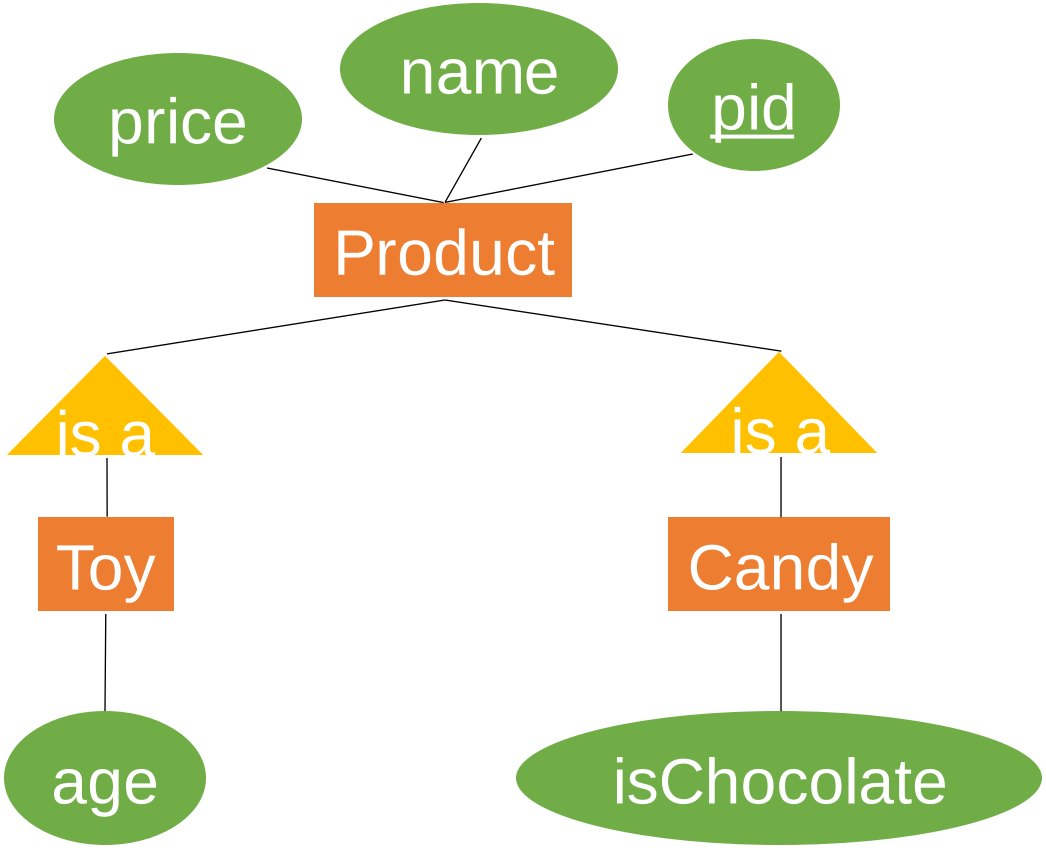
pid	age
012	8
345	2
567	3

Candy

pid	isChoc
123	yes
456	no
567	no

Representing Subclasses in SQL

- Each entity set becomes a relation



Product

pid	name	price
012	Lego	99
123	M&M	5
234	Computer	2999
345	Ball	15
456	Skittles	3
567	M&M toy	49

Toy

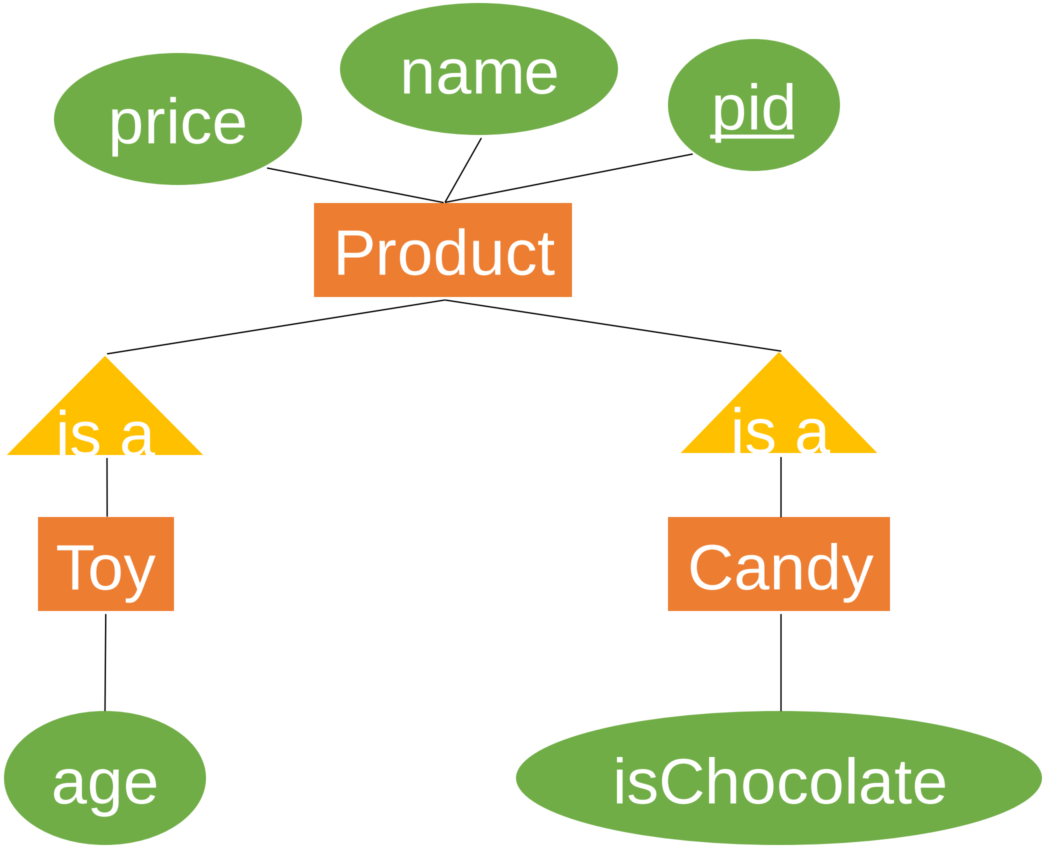
pid	age
012	8
345	2
567	3

Candy

pid	isChoc
123	yes
456	no
567	no

Representing Subclasses in SQL

- Each entity set becomes a relation



Product

pid	name	price
012	Lego	99
123	M&M	5
234	Computer	2999
345	Ball	15
456	Skittles	3
567	M&M toy	49

Toy

pid	age
012	8
345	2
567	3

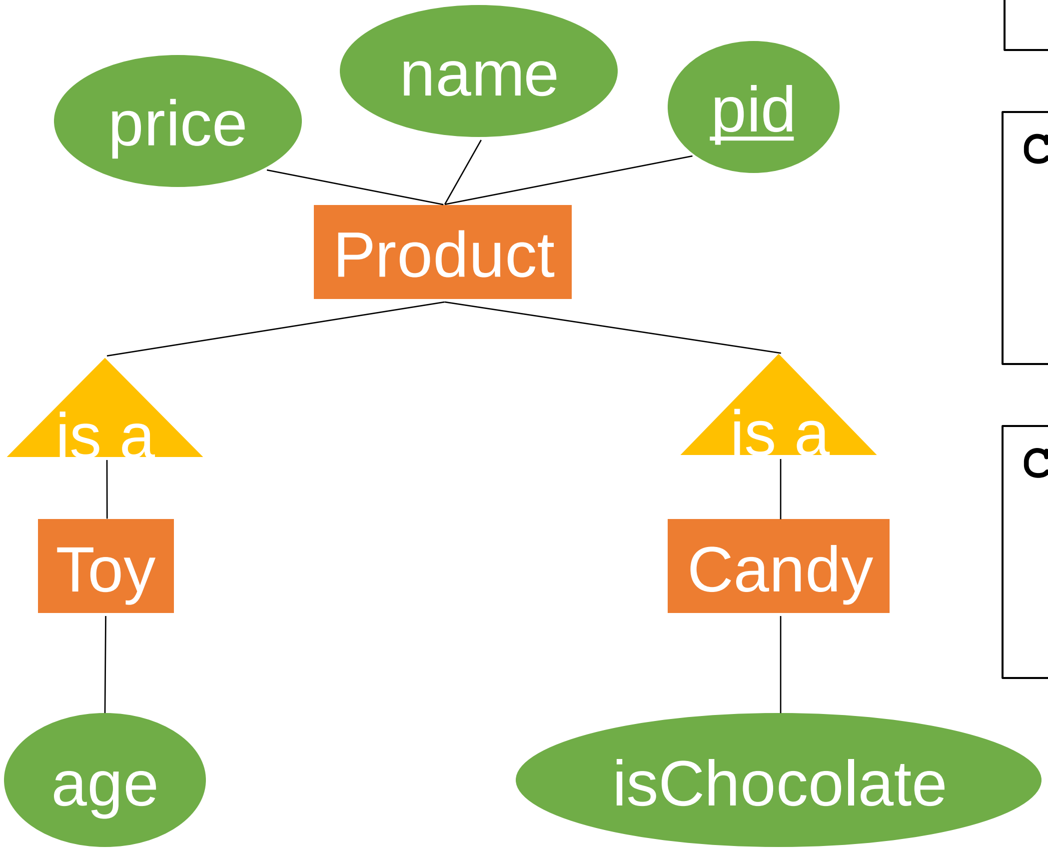
Candy

pid	isChoc
123	yes
456	no
567	no

567 MM& toy is both Toy and Candy!

Representing Subclasses in SQL

- Each entity set becomes a relation



```
CREATE TABLE Product (
  pid INT PRIMARY KEY,
  name TEXT,
  price FLOAT);
```

```
CREATE TABLE Toy (
  pid INT PRIMARY KEY
      REFERENCES Product,
  age INT);
```

```
CREATE TABLE Candy (
  pid INT PRIMARY KEY
      REFERENCES Product,
  isChocolate INT);
```

Discussion: Subclassing

- Entity set may be a subclass of another entity set
    - Inherits all the attributes of the superclass

- Some DBMSs support inheritance
    - However, we will simply represent inheritance using foreign keys and joins with the subclass and superclass

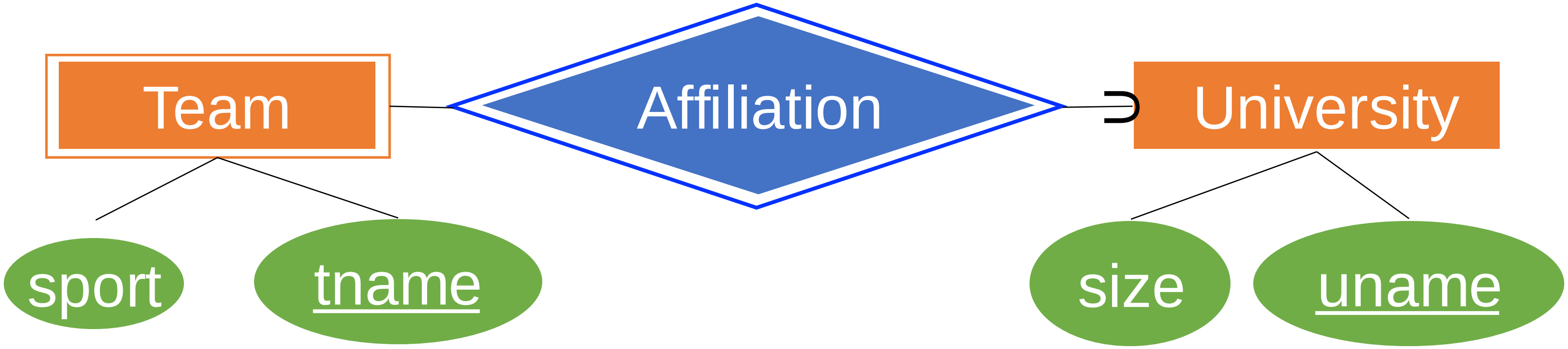
Weak Entity Sets

Weak Entity Set

- **Weak entity set:** key includes key from another entity set

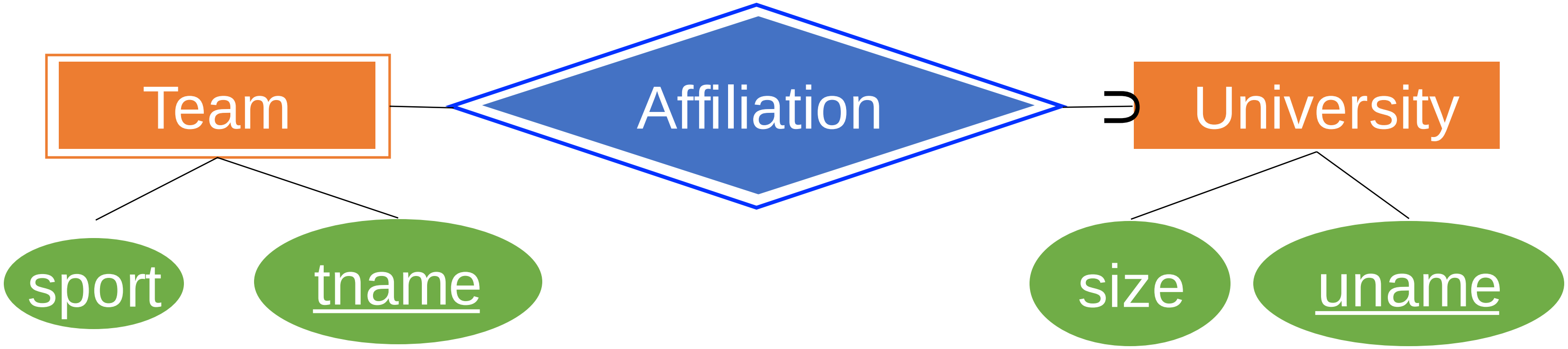
Weak Entity Set

- **Weak entity set:** key includes key from another entity set



Weak Entity Set

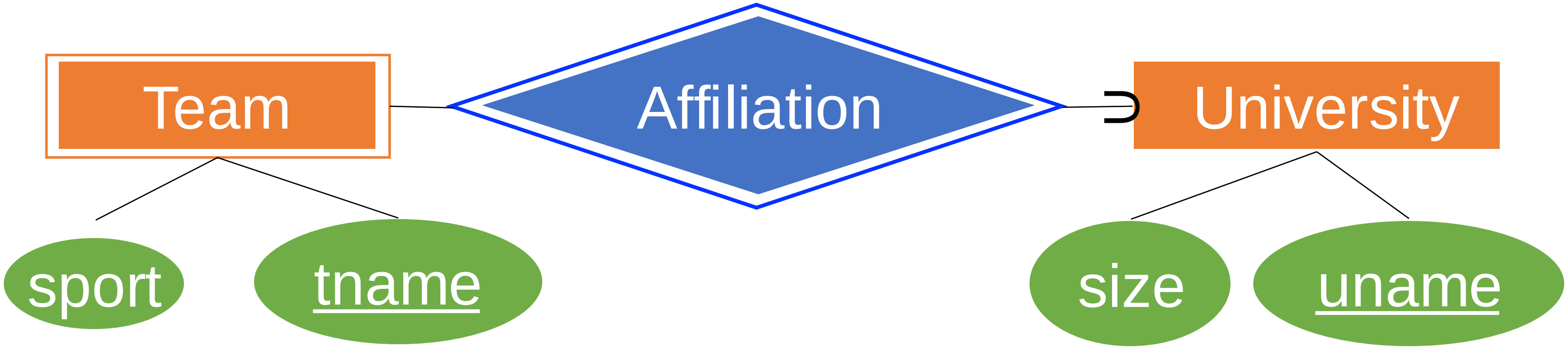
- **Weak entity set:** key includes key from another entity set



- The key of Team is (tname, uname) together
  - tname is not enough e.g. “Huskies” could be UCONN or UW

Weak Entity Set

- **Weak entity set:** key includes key from another entity set

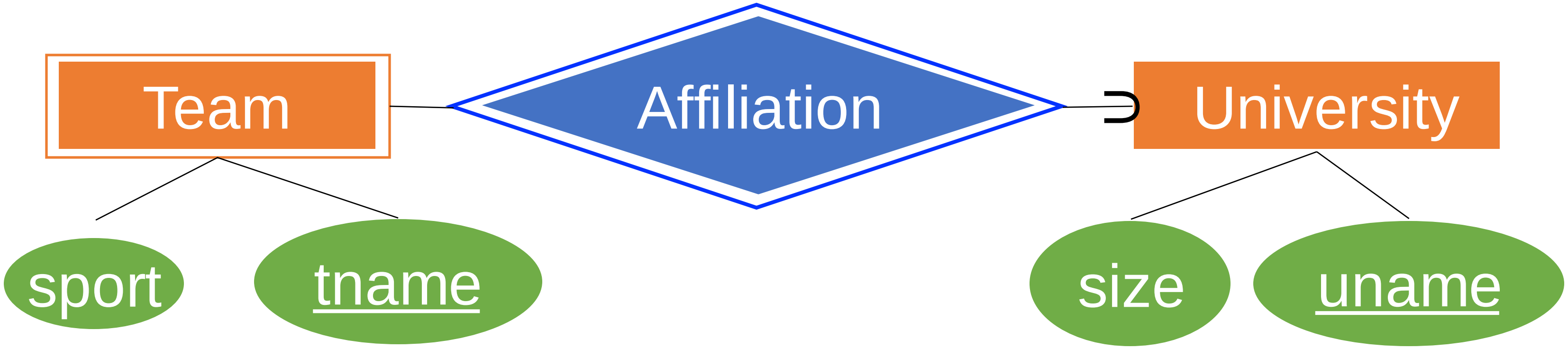


- The key of Team is (tname, uname) together
  - tname is not enough e.g. "Huskies" could be UCONN or UW

- The weak entity set and its relationship to the other (entity set's) key are both depicted with double-outlines

Weak Entity Set

- **Weak entity set:** key includes key from another entity set



```
CREATE TABLE University (
   uname TEXT PRIMARY KEY,
   size INT);
```

```
CREATE TABLE Team (
   uname TEXT REFERENCES University,
   tname TEXT,
   sport TEXT,
   PRIMARY KEY (uname, tname));
```

Discussion

What you should know:

- Design simple ER diagrams
- Understand:
  relationships, inheritance, weak entity sets
- Convert (correctly!) ER diagrams to SQL