ER-RM translation with examples

Taken form prof. Yannis Velegrakis classes

Conti Samuele Daniotti Filippo

October 1, 2019

Chapter 1

Translating ER into RM

The best way to learn this mechanism is to proceed by examples.
Let's start with this classic:

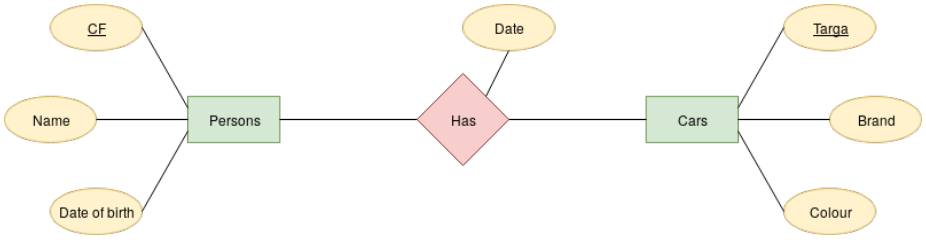

Figure 1.1: The classic example

When we want to translate an Entity Relationship Diagram (ERD) into a Relation Model (RM) we have to follow certain rules, let's check them out.

Rule #1 Every entity is a table and its attributes are the columns of the table.

Rule #2 Each relationship is a table and its attributes are part of the columns of the table; also the Keys of the entities involved in the relation become columns and they are flagged as Foreign Keys (FK). The Key of the table must be composed by the FKs.

Let's put this in practice:

PERSONS

CF	Name	Birth date
1	John	1978
5	Mary	1974

`PERSONS(CF, Name, Birth date)`

CARS

Targa	Brand	Color
xy	VW	Red
uv	Toyota	Blue

`CARS(Targa, Brand, Color)`

That was pretty easy, wasn't it?
Now it's time to draw the relation table:

HAS

CF	Targa	Date
1	xy	2002
5	uv	2004

`HAS(CF, Targa, Date)`

Well, that's clear. Now it's time to mix things up a little bit. What if a car can't be owned by different people? (see 1.2)

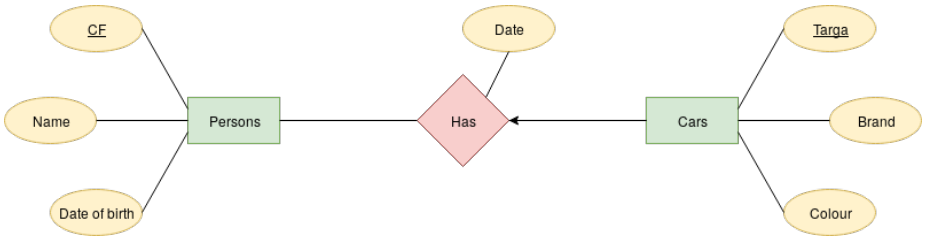

Figure 1.2: A car now can have at most one owner

We must use only the Targa as Key in the HAS relation. This leads us to the first exception of Rule #2 :

Each many to one relationship become a relation as in Rule #2 but the Key is *only* the Key of the "many" part.

Let's see this rule applied:

HAS

CF	Targa	Date
1	xy	2002
5	uv	2004

```
HAS(CF, Targa, Date)
    CF Foreign Key to PERSONS
    Targa Foreign Key to CARS
```

And what if we want the car to have also at least one owner? (see 1.3)

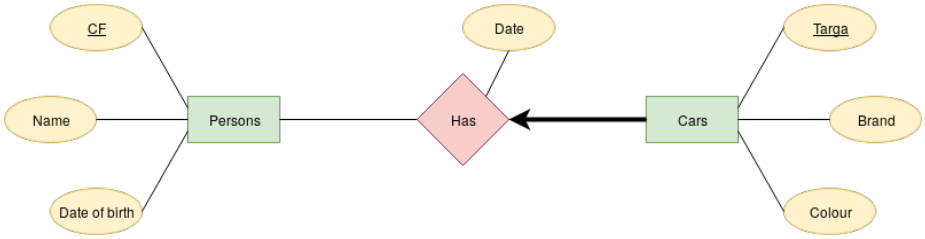

Figure 1.3: A car now can have at most one owner

This way a car must be owned by one (and only one) person, so it makes no sense to keep separately the HAS table and the CARS table, and this leads us to the second exception to Rule #2:

Each "many to one" and total partecipation relationship does not become a relation by itself, but its attributes and the Keys of the "one" entity are added in the attributes of the "many" entity. The Key of the "one" entity must be marked as Foreign Key and has to be requested to be `not null`.

Therefore in our case the solution is to delete the HAS table and add two columns to the CARS table, one for Date and one for CF, which will be a FK and will be requested to be `not null`. That's the representation in RM:

CARS

Targa	Brand	Color	CF	Date
xy	VW	Red	1	2002
uv	Toyota	Blue	5	2004

```
CARS(Targa, Brand, Color, CF, Date)
    CF not null
    CF Foreign Key to PERSONS
```

Going back to diagram 1.1, what if we want to keep in memory all the times

someone rent the same car? Now we cannot do that because, being the Key, Targa cannot appear twice in the table. The solution here is to use also Date as a Key. But that's not feasible because, as we stated in Rule #2, only Keys of the entities are used as Key in the relation table, and Date is not an entity; hence we have to redesign our diagram as this:

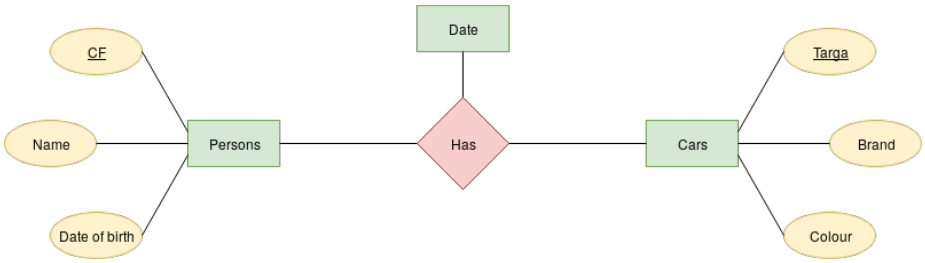

Figure 1.4: Now Date is an entity

Now we want every rent to have an insurance, so we have to create a new entity called INSURANCES and link it to the RENT relation as this:

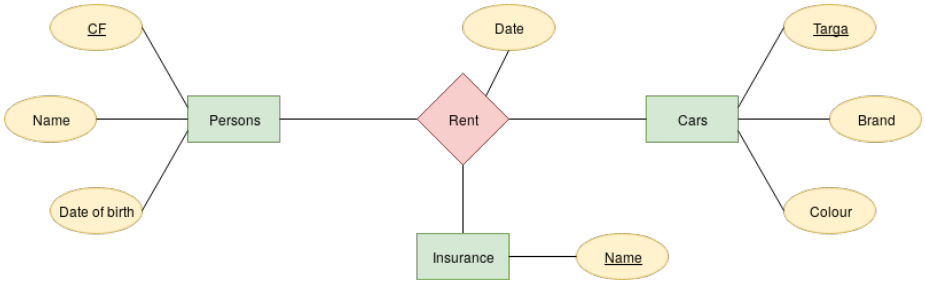

Figure 1.5: We added Insurance as an entity

```
RENT(Owner, Car, Name, date)
  Owner FK to PERSONS
  Car FK to CARS
  Name FK to INSURANCES
```

And now we decide that the insurance is no more necessary for a rent, and how do we set it as a field that can be `null`? We have to link it to RENT with a relationship. But we can't link two relationships together: we must aggregate PERSONS, RENT and CARS so that we can treat this set as an entity and then we can link to the aggregated entity the INSURANCES with another relationship (see 1.6).

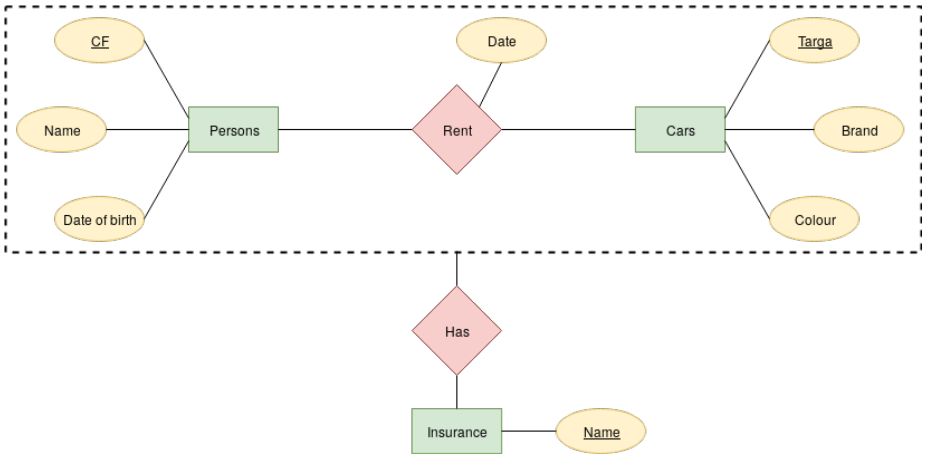

Figure 1.6: We added Insurance as an entity

In this case what's going to be the Key of RENT?

```
RENT(Owner, Car, date)
  Owner FK to PERSONS
  Car FK to CARS
```

And which should be the Key of HAS? Keeping in mind Rule #2 we have that the Keys of a relationship must be the Keys of the entities involved, marked as Foreign Key, therefore:

```
HAS(Owner, Car, Name)
  (Owner, Car) FK to RENT
  Name FK to INSURANCE
```

And now the last question about this example: what's the difference between 1.5 and 1.7?

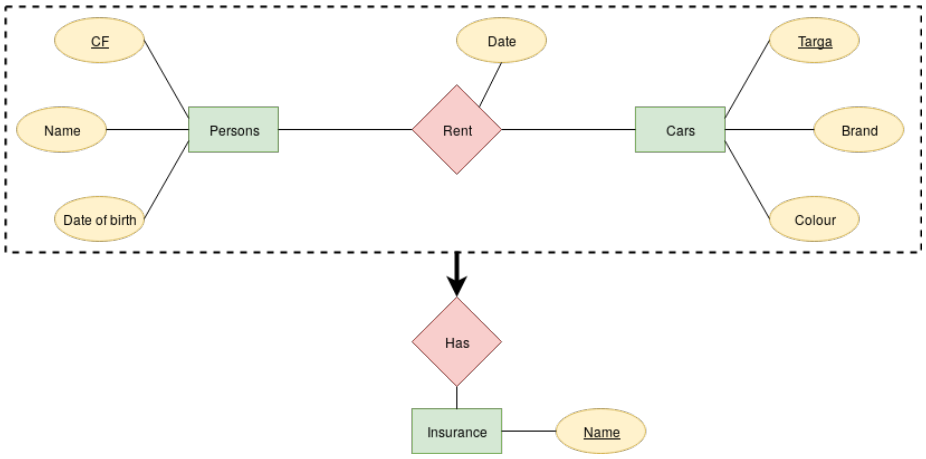

Figure 1.7: Note the particular form of the diagram

The difference is:

- In 1.5 you *must* use Name as Key in the RENT table, because INSURANCES is an entity involved in RENT relation (Rule #2); therefore a person can rent the same car multiple times given that he or she uses a different insurance.

- In 1.7 you don't have to use Name as Key in the RENT table, therefore it is not possible for a person to rent the same car with two different insurances (even if the insurance changes the Key remains the same).

1.1 Let's talk about inheritance

Now it's time to see a little bit of inheritance, as before we're starting with an example:

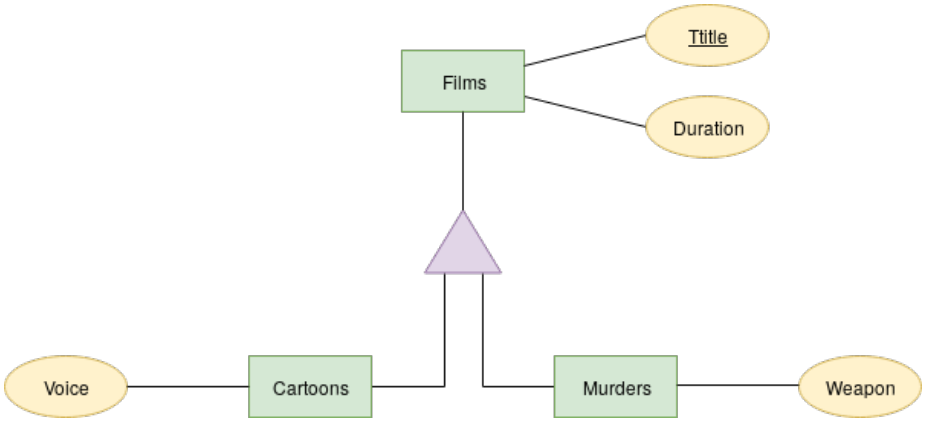

Figure 1.8: An example of inheritance

There are three ways to translate ER inheritance into RM:

1. The first approach is to have a different table for every entity, the tables of the children entities will contain also the attributes inherited by the parent entity.

   ```
   FILMS(Title, Duration)
   CARTOONS(Title, Duration, Voice)
        Title FK to FILMS
   MURDER(Title, Duration, Weapon)
        Title FK to FILMS
   ```

   The bad aspect of this technique is that we store the different entity-types in different places, for instance Shrek is stored into CARTOONS table, therefore if we want to query the database by Title and ask for Shrek we have to navigate through all the tables. Also if we want to query how many films the database contains we will have to search in all the tables.

2. The second approach is to have a different table for every entity, but the tables of the children entities contain only their specific attributes and the Key attribute of the parent entity, which will be also the child entity's Key (and Foreign Key).

   ```
   FILMS(Title, Duration)
   CARTOONS(Title, Voice)
        Title FK to FILMS
   MURDER(Title, Weapon)
        Title FK to FILMS
   ```

This strategy is better because all the entities are stored in the FILMS table and therefore if we're querying, for instance, Title or Duration, we have to look only into the FILMS table. The films that are cartoons - like Shrek - are going to be registered also in the CARTOONS table, and there we will find their Voice attribute. The side effect of this approach is that when we query, let's say, Duration and Voice we have to look into two different tables at the same time, and that's actually more complicated than we need.

3. The third approach is to store all the films in the same table with all the attributes of the parent entity and the children entities, these will be filled with a `null` value if the film stored isn't a cartoon or a murder.

```
FILMS(Title, Duration, Voice, Weapon)
   Duration is not-null
   Voice and Weapon can be null
```

Thinking in a querying way that's the best approach, for every research in the database you have to navigate only through a table. Anyway in the real life, where problems are way much complex than our examples, this technique can become tricky too.

Chapter 2

Reverse engineering: from RM to ER

Sometimes you may want to translate an RM back to its ER model, so here we provide some examples and basic rules in order to get it done.

For instance, let's consider the following statements:

```
A(x,y)
B(u, v)
   v Foreign Key to A
   u Foreign Key to C
C(m, n)
```

B has two Foreign Keys, one points to A and the other one points to C, so B is a relationship between A and C.

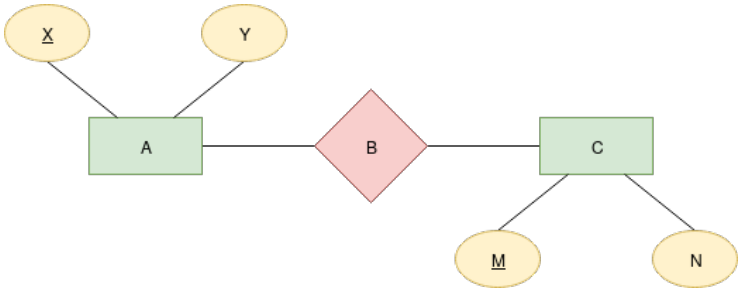

Figure 2.1:

But now we change it up just a little, and we want something like this:

```
A(x,y)
B(u, v)
   v Foreign Key to A
```

```
   u Foreign Key to C
C(m, n)
```

It looks similar, but it's not the same: `v` is not a Key of `B` anymore, so we can have multiple vs for just one `u`, so the relationship from `C` to `A` has become a many to one relationship.

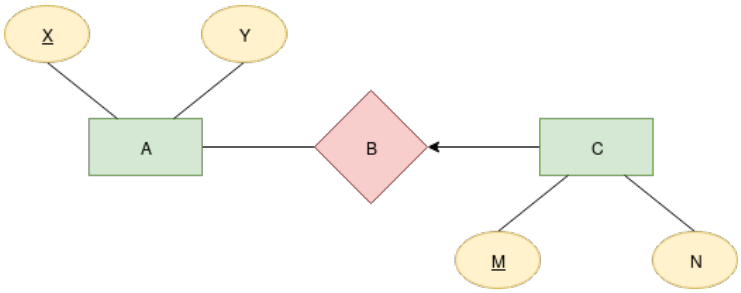

Figure 2.2:

Here it is another example:

```
C(m, n, v)
   m Foreign Key to A
A(x,y)
```

This is a IS-A and C is child to A, because the Foreign Key `m` is also the only Key of C.

2.1 Weak Entities and summing it up

Now let's take a quick look at the translation of Weak Entities, i.e. entities that cannot exist on their own.

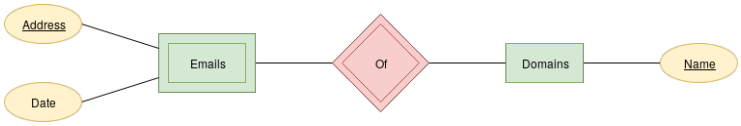

Figure 2.3: E-mail cannot exist if domain doesn't

The translation is simple: DOMAINS is a straight entity, so we list all its attributes and underline the Key, and then we do the same for E-mails, and its Composite Key is going to be its Key (Address) plus the Key of its straight entity (Name, in this case), which is also a Foreign Key.

```
DOMAINS(name)
EMAILS(address, date, name)
   name Foreign Key to DOMAINS
```

Now let's point out three basic rules if you are into troubles with reverse engineering and you want to have it done quickly:

Rule #1 If the Foreign Key is not a part of the Key, what we have is a *relationship*;

Rule #2 If the Foreign Key is the only Key of that table, what we have is an IS-A;

Rule #3 If the Foreign Key is a part of the Key but it is NOT the whole Key of that table, what we have is a Weak Entity relationship.