# Entities and relationships

## Introduction

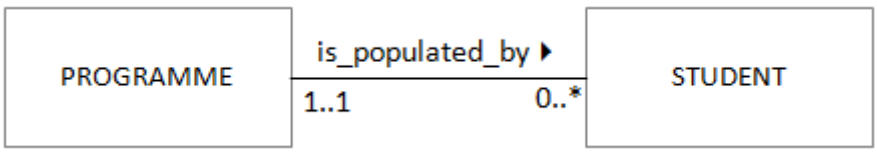
All data models must represent the relationships that exist between entities. The practical question for us is how to do that using a relational database. Using columns to represent the attributes of a particular type of entity is an easy step, but relationships exist *outside* an entity: how then can we used the basic structures available to represent relationships as well?

This set of notes covers probably the single most important topic in the module. Without a good understanding of the concepts discussed here, you will not be able to use or develop databases with confidence. Once you do understand how relationships are represented, you will be able to handle all of the other topics more easily.

## Foreign keys

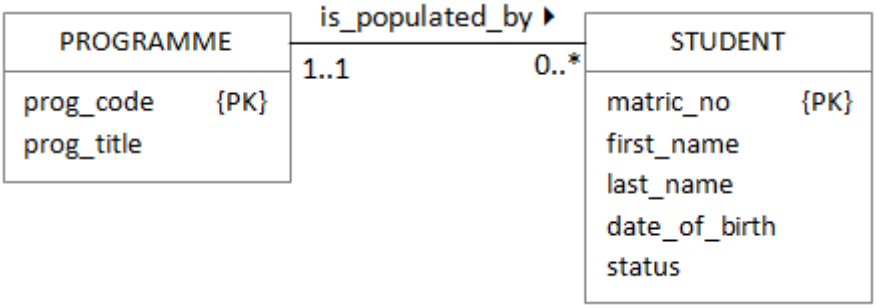
A *schema* is a set of database tables that together make up the model of the subject area. For example, a university database is likely to have tables to represent students, modules and programmes. Each table represents a set of entities of the same type, and can potentially contain thousands of rows each of which represents a single entity. In order to locate just the rows that we are interested in, each table needs a *primary key*. Recall that a primary key is an attribute or group of attributes which uniquely identifies an entity. A primary key which is made up of several attributes (columns) is known as a *composite* primary key.

So far, so good. Now let's use a familiar example to work out how to represent relationships. Take a look at the ER diagram below.



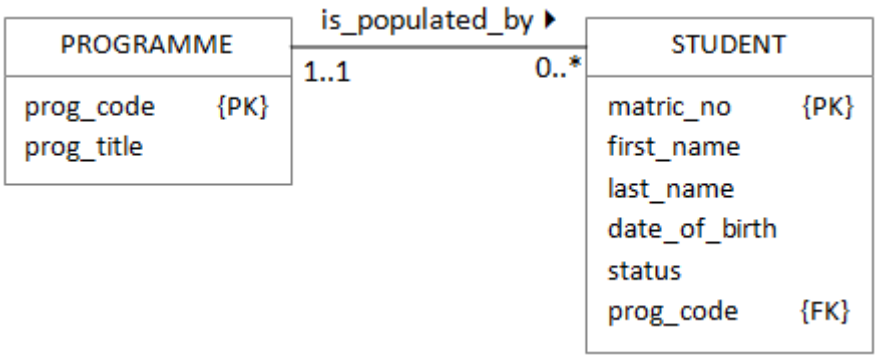

This diagram says that one PROGRAMME is populated by many students. The multiplicity labels tell us that a STUDENT must be associated with a PROGRAMME, but that a PROGRAMME can exist without any students (think of a new programme that has not yet recruited).

Put another way, a group of records in the STUDENT table are related to a unique record in the PROGRAMME table: this sounds like something we can use the primary key for. If we attach the primary key from the PROGRAMME record to the related STUDENT records, that will accurately represent the relationship. That is exactly what happens, but before looking at the way it is done, let's think quickly about the columns that these two tables might have.



This diagram extends the basic form of an entity in an ER diagram to include the entity's attributes. The PROGRAMME table contains the programme code and title, and the programme code is used as a primary key. Notice that it is labelled in the diagram. The STUDENT table contains the personal details of the students along with some administrative information to help the university operate. Each STUDENT has a matriculation number, for example, and a status code. Not surprisingly, the matriculation number is used as the STUDENT primary key.

To associate a group of records in the STUDENT table with a row in the PROGRAMME table, we need a place to put the primary key from the PROGRAMME table. The only way to store this kind of information in a table is as a column; therefore we need to add a new column to the STUDENT table as shown below.




Now for every STUDENT, we can store the programme code for the corresponding PROGRAMME record. The new column is imported from another table where it is the primary key. In the STUDENT table, it is called a *foreign key*, and its job is to identify a unique record in another table.

Looking at some example data can help make this explanation clearer. Let's assume that there are three students on the Flat-Pack Design programme, and three on the Fast Food Management programme. The data might look like this:

PROGRAMME

| prog_code | prog_title |
|---|---|
| MCD | Fast Food Management |
| IK | Flat-Pack Design |

STUDENT

| matric_no | first_name | last_name | date_of_birth | status | prog_code |
|---|---|---|---|---|---|
| 1234 | Anne | Archer | 12/4/90 | C | MCD |
| 1235 | Bob | Barber | 20/3/91 | C | IK |
| 1236 | Carole | Cook | 7/7/92 | C | IK |
| 1237 | David | Driver | 1/12/91 | M | MCD |
| 1238 | Erica | Eggler | 23/3/90 | C | MCD |
| 1239 | Fred | Fletcher | 17/5/91 | C | IK |

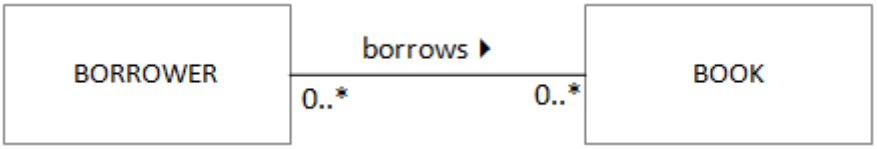
From the values in the foreign key column, it is clear which of the six students is on which programme.

Remember:

- A foreign key always corresponds to a primary key in another table
- The foreign key is always at the "many" end of a relationship

## Many to many relationships

A primary key - foreign key pair can be used to represent a 1:* relationship; however, they are not the only kind you will come across. Consider a library database that stores information about books and borrowers. The ER diagram might look like this:

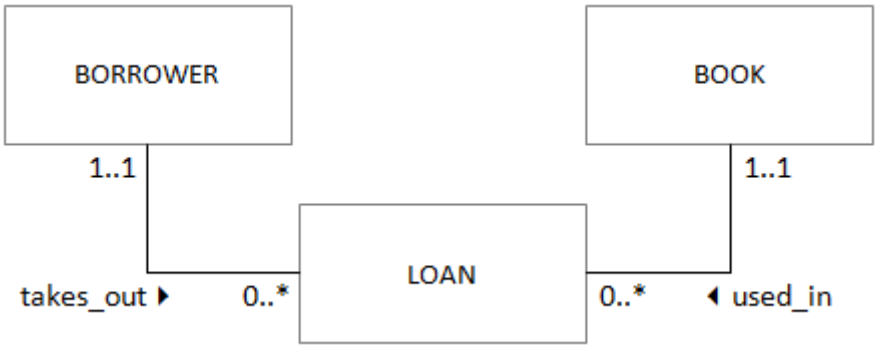



The diagram tells us that a BORROWER may borrow zero to many BOOKS, and that a BOOK may be borrowed by zero to many BORROWERS. The relationship is therefore *:*, so where does the foreign key go?

The answer is that you can't represent a *:* relationship directly using a relational database. You always have to resolve it into a set of 1:* relationships. This actually turns out to be very easy to do, and the solution is always the same:

1. Create a third table to represent the relationship
2. Add the primary keys from the original tables as foreign keys in the new one

The result is shown below. In effect, you have inserted the third table in between the original ones. Notice the multiplicity labels on the new relationships. Because the third table has foreign keys to both of the original tables, it is at the "many" end of both of the new relationships.




In library terminology, a "loan" is an instance of borrowing an item. It is therefore the natural name for our new table. At the moment, the LOAN table only contains two foreign keys. In many cases, that is all that is required, and the table known as a *bridge table* or a *link table*. Its only function is to resolve the *:* problem. In this case, though, there is some more information that needs to be attached to a LOAN, such as the date the item was borrowed, and the date it is due to be returned.

## One to one relationships

Most relationships between entities will be of the 1:* kind, and *:* relationships can be decomposed into two 1:* relationships. However, that still leaves 1:1 relationships. These are rarer, but here are some examples:

- One EMPLOYEE has one CONTRACT
- One PLAYER captains one TEAM
- One EMPLOYEE leases one CAR

An immediate question that should come to mind is whether we need to keep 1:1 relationships in the database. After all, one of the reasons for using a database it to eliminate redundancy, so if we can subsume one table into another, isn't that a good thing? The answer is yes, but this is not always desirable, and occasionally impossible.

The starting point for deciding how to handle 1:1 relationships is to look at the optionality of the relationship where there are three possibilities.

The relationship can be:

1. mandatory at both ends
2. mandatory at one end and optional at the other
3. optional at both ends

## Mandatory at both ends

If the relationship is mandatory at both ends it is often possible to subsume one entity into the other.

- The choice of which entity type subsumes the other depends on which is the most important entity type (more attributes, better key, semantic nature of them)
- The result of this amalgamation is that all the attributes of the 'swallowed up' entity become attributes of the more important entity
- The primary key of the subsumed entity type becomes a normal attribute
- If there are any attributes in common, the duplicates are removed
- The primary key of the new combined entity is usually the same as that of the original more important entity type

**When not to combine**

There are a few reasons why you might not combine a 1:1 mandatory relationship.

- the two entity types represent different entities in the real world
- the entities participate in very different relationships with other entities
- a combined entity would slow down some database operations

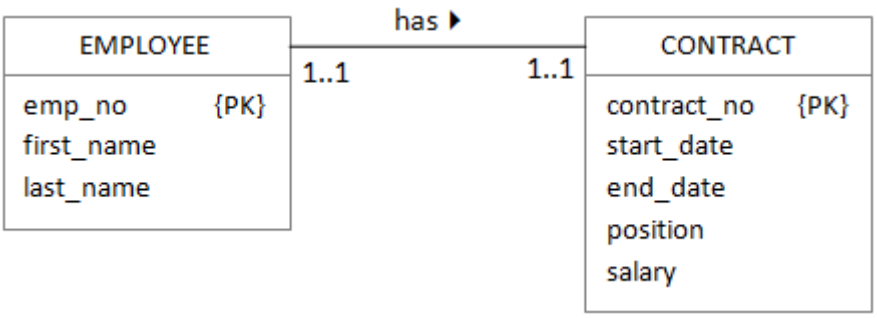
**If not combined...**

If the two entity types are kept separate then the association between them must be represented by a foreign key.

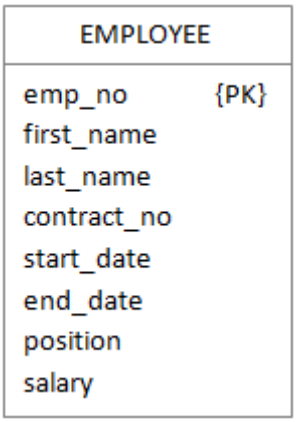
- The primary key of one entity type becomes the foreign key in the other.
- It does not matter which way around it is done but you should **not** have a foreign key in both entities

**Example**

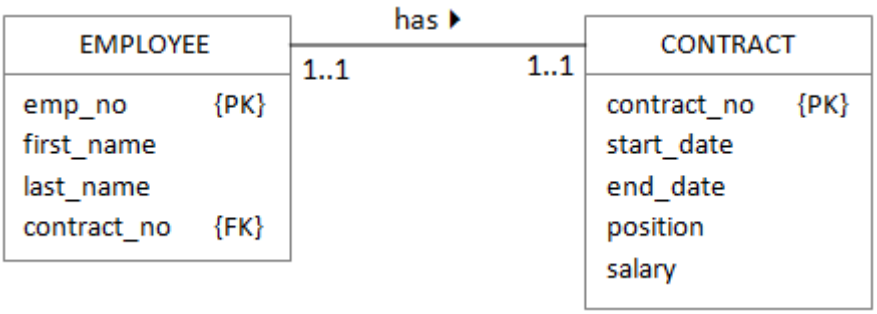
- Two entities are EMPLOYEE and CONTRACT
- Each EMPLOYEE must have one CONTRACT and each CONTRACT must have one EMPLOYEE associated with it
- It is therefore a mandatory relations at both ends

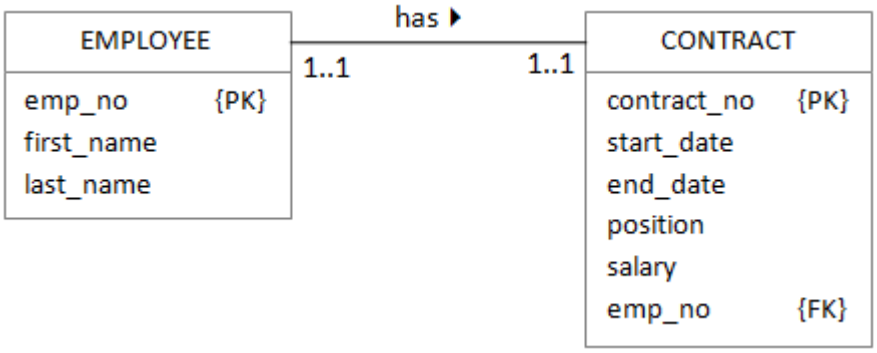



These two entity types could be combined into one:




Alternatively, they could be kept separate, in which case there are two possibilities:




Depending on which you consider to be the more important, either of these two solutions is acceptable. Which one do you find more satisfactory?

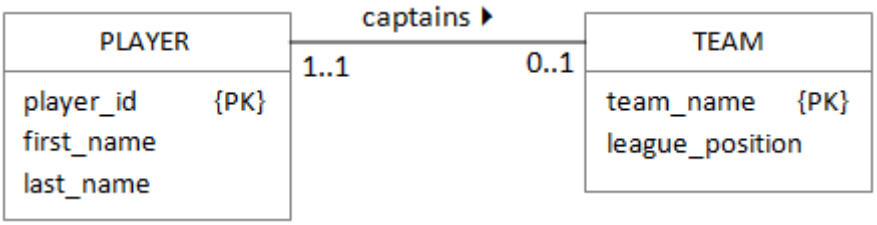
## Mandatory / Optional

The entity at the optional end may be subsumed into the one at the mandatory end, or both entities may be retained.

**Example**

- Two entities are PLAYER and TEAM
- One PLAYER may captain one TEAM, and each TEAM must have one CAPTAIN
- However, there are other PLAYERS who do not captain a TEAM; therefore the relationship is mandatory on one direction and optional in the other

Notice the use of *may* and *must* in the relationship description, and the multiplicity labels in the diagram



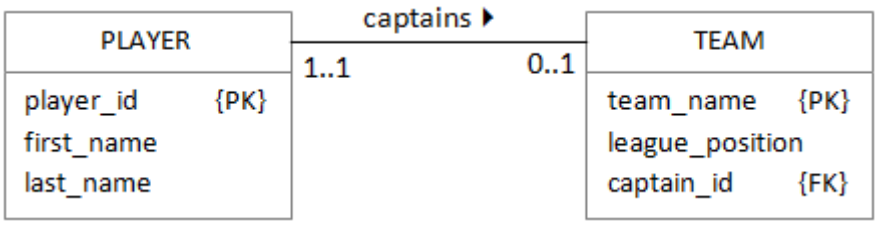

Combining the two entities would give the result below. Only those players who are team captains would have values in the *team_name* and *league_position*.


The reasons for not subsuming are the same as before with one additional reason:

- the two entity types represent different entities in the real world
- the entities participate in very different relationships with other entities
- a combined entity would slow down some database operations
- *very few of the entities from the mandatory end are involved in the relationship. This could cause a lot of wasted space with many blank or null entries.*

Thinking about the team captain example, there will be many PLAYERS who are not the captain of a TEAM. As noted above, the *team_name* and *league_position* fields would be empty for these people. That is a lot of wasted space, and in this case we would probably maintain two separate entities.

To summarise the approach for 1:1 mandatory/optional relationships, take the primary key from the 'mandatory end' and add it to the 'optional end' as a foreign key:




Notice in this example that the name of the foreign key column in TEAM is different from the name of the primary key column in PLAYER. This is perfectly legal, and in this case it makes the purpose of the column clearer. In most cases, however, the two columns will have the same name.

## Optional at both ends

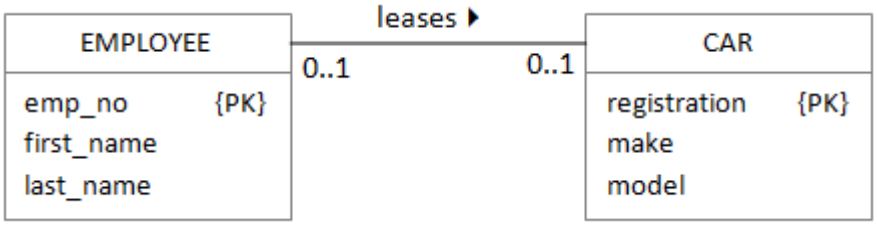
Such examples cannot be combined because you could not select a primary key. Instead, one foreign key is used as before.

**Example:**

Imagine a company that operates a company car scheme where some employees can have exclusive use of a car.

- Two entities are EMPLOYEE and CAR
- One EMPLOYEE may lease one CAR, and one CAR may be leased by one EMPLOYEE
- However, there are EMPLOYEES who do not lease CARS, and CARS that are not allocated to EMPLOYEES. The relationship is therefore optional in both directions

Notice the use of the word *may* in the description, and the multiplicity labels on the diagram.




Remember that every record in a table should have a primary key. If we tried to combined these two entities, we have three options for the primary key:

- Use *emp_no*
- Use *registration*
- Us a composite key made up up *emp_no* and *registration*

If we use *emp_no*, any unallocated cars would have no primary key

If we use *registration*, any employees without a car would have no key

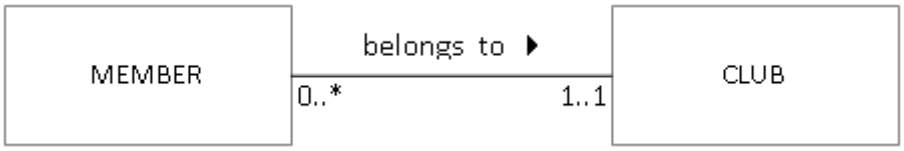
If we use the composite solution, sometimes one of the key attributes would have no value. This is not permitted: a primary key field must always have a value.

The only solution is to maintain two separate entities. There is still one remaining problem however: where does the foreign key go? Putting it in either table would work, but would also require null values.

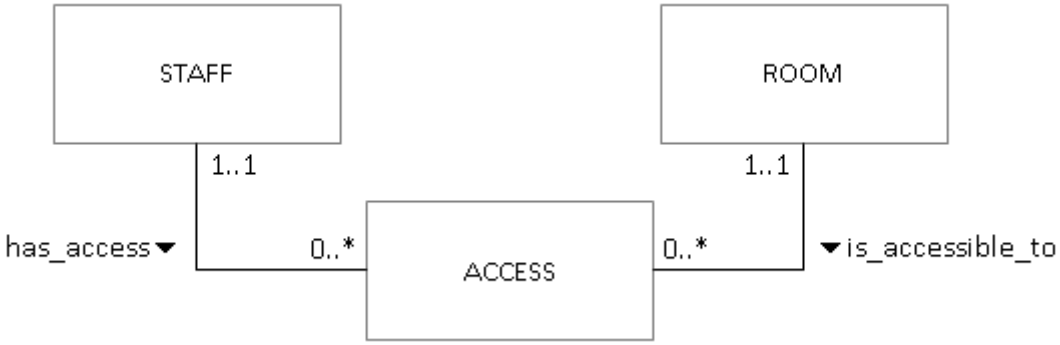
## Referential integrity

Apart from considerations of efficiency and performance, the job of the DBMS is to maintain various forms of *data integrity* which concerns the type of data stored in a column and any constraints that might apply such as a restricted range of values. *Entity integrity* is the requirement to be able to uniquely identify each individual entity which translates into the need for primary keys. *Referential integrity* is the requirement that the value of any foreign key field must identify a unique record in the related parent table. The set of primary key values in the parent table define the domain of the foreign key column in the child table.

If a foreign key value has no matching primary key in the parent table the data is inconsistent. The existence of a foreign key value suggests that a relationship exists, but if the related record cannot be found vital information may be missing about the child record. An example might be a club membership database as represented below. If the foreign key value on a **MEMBER** record did not correspond to a primary key in **CLUB**, then there is no way to know which club the member belongs to.




At the extreme, orphan foreign key values can introduce completely meaningless data into the database. Consider a bridge table which consists only of two foreign keys such as the one which links **STAFF** and **ROOM** in a system which controls door access by means of swipe cards. It is likely that both STAFF_ID and ROOM_ID will be synthetic keys. With no matching parent records, values in the bridge table are simply meaningless numbers.




## Sanity checks

Data integrity is maintained by carefully designing the definition of a column at the point the table is created. This can go beyond just specifying a datatype by including programmatic rules called *constraints* which validate the data before it can be stored. Constraints will be discussed in more detail later in the module. The main point to note here is that the DBMS will prevent the user from storing any data which does not satisfy the criteria set of a particular column.

In a well-designed database, every table should have an appropriate primary key defined. In certain cases, a database may still behave as expected if one or two tables do not adhere to this rule, but in general leaving out primary keys is bad practice and can cause data anomalies which will be discussed later in the module. One practical implication of defining primary keys is that the DBMS will not allow the user to store a record unless a) the primary key field has a value and b) that value does not appear on any other record.

The relationships between entities are represented in the database by *foreign key constraints*. Really, this just means that when a child table is created one (or more) columns are defined as the foreign key and in addition, the related parent table is also specified. In effect, this means that the value in the foreign key field of the child table is the same as the value in the primary key field in the parent table - ie the values match. Referential integrity is the requirement that the corresponding parent record MUST exist for a foreign key value. If this rule were not enforced, it would be possible to have meaningless data stored in the database, either in the form of child records with no parent or foreign key values with no referential target.

Again, as in the case of other types of integrity, the DBMS will prevent the user from doing certain operations which would cause a loss of referential integrity. In particular:

- A child record cannot be inserted with an unknown foreign key value - ie one that does not exist as a primary key in the parent table
- A parent record cannot be deleted while related child records still exist
- A primary key value cannot be updated while related child records exist (although, this situation should never actually arise)

These rules have practical implication for data maintenance. For example, when inserting a set of data into a pair of related tables, each parent record must be inserted before the child records. Otherwise you are trying to insert a foreign key value before it exists as a primary key value in the parent table. Likewise if you are deleting a set of data, child records must be deleted before related parent records.

## Data maintenance

There may be genuine reasons for deleting a parent record from the database. If that situation arises, there are essentially two options:

- Insist that the user deletes any related child records before attempting to delete the parent record
- Automatically delete any related child records when the parent record is deleted.

The first case is usually the default and in that situation an attempt to delete the parent record would result in an error message if related child records exist. However, an application designer can decide to manage the deletion of child records with application code which explicitly goes through any related records and applies appropriate conditions to determine whether or not to delete them automatically. This allows for complex situations and fine-grained record handling compared to embedding this behaviour in the database definition.

Automatic deletion of child records is know as *cascading* the delete operation. As part of the database definition, the cascade behaviour can be specified in the CREATE TABLE statement:

```
CREATE TABLE access (
    staff_id NUMBER REFERENCES staff(staff_id) ON DELETE CASCADE,
    room_id NUMBER REFERENCES room(room_id) ON DELETE CASCADE
);
```

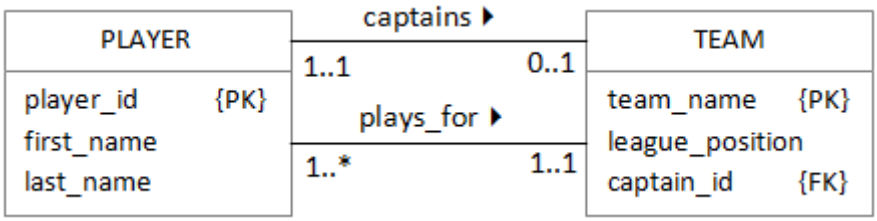
Notice that the cascade behaviour is part of the definition of the foreign key constraint, and therefore attached to the child table. The definition above means that if a parent **STAFF** or **ROOM** record is deleted, any related records in **ACCESS** are also deleted automatically.

Clearly, whether or not to include automatic deletion of child records in the database requires careful consideration. The deletion of child records will happen without any warning to the user, and so if there are situations in which that should not happen, the better option is to handle those cases explicitly with application code.

# Advanced ER modelling

## Parallel relationships

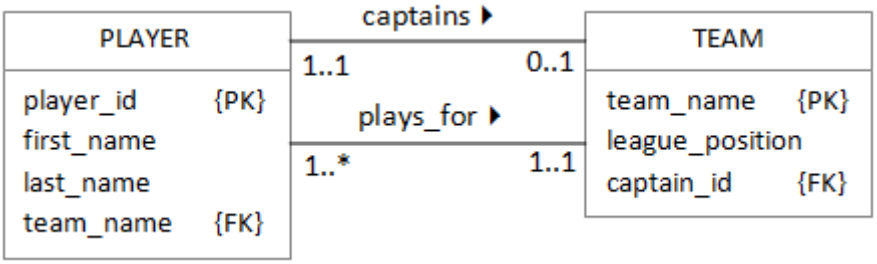
Parallel relationships occur when there are two or more relationships between two entity types. Thinking about the example of PLAYERS and their TEAMS, we have seen that one PLAYER can captain one TEAM. Another more obvious relationship between these two entities is that each PLAYER plays for one TEAM. We can show this on an ER diagram like this:




Each relationships is shown separately. Notice that the *plays_for* relationship is shown from the PLAYER perspective. Usually it is best to represent relationships from the perspective of the entity at the 'one' end of the relationship. In this case, the version shown is a more natural expression.

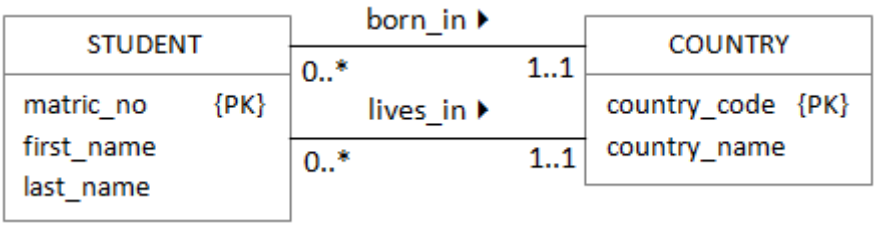
The multiplicity label at the PLAYER end of the *plays_for* relationship shows that a TEAM must have at least one PLAYER, and the relationship is mandatory in both directions.

The foreign key for the *captains* relationship is already shown in the TEAM table. The *plays_for* relationship must also be represented by a pair of keys. Because the TEAM is at the 'one' end of the relationship, we must add the TEAM primary key as a foreign key to the PLAYER table as shown below.

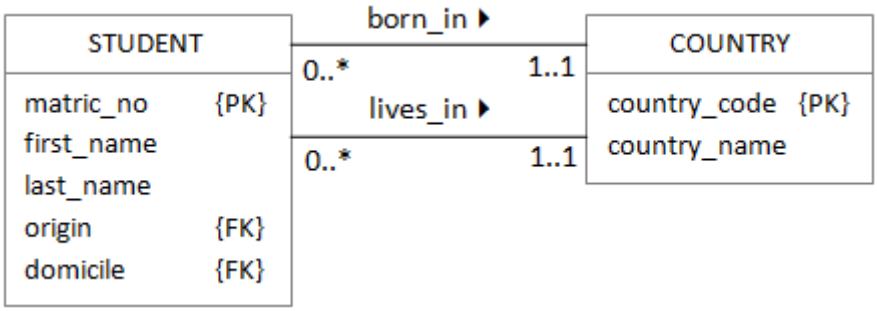



Now we have a complicated situation in which the primary key of each table is used a foreign key in the other. This *only* happens for parallel relationships - two foreign keys means two relationships. Notice that because we have used the name *captain_id* for the foreign key in the TEAM table, it is easier to work out which keys belong to which relationship.

Using another example of a university database that stores details of students, we can see that the university needs some information about the country the student is from in order to calculate fees correctly. However, a STUDENT may have been born in one COUNTRY, but live in another. These are two different relationships between STUDENTS and COUNTRIES. Where do the foreign keys go?



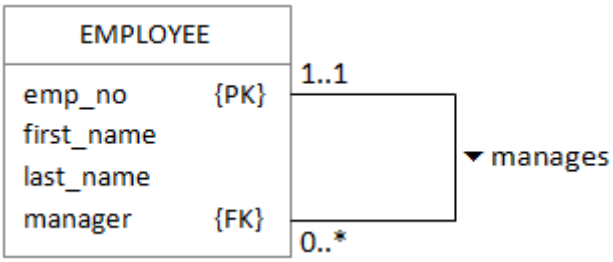

The multiplicity labels tell us that both relationships are *:1 and both are in the same direction. That means that for each one, we take the primary key for the COUNTRY table and include it as a foreign key in the STUDENT table. This is perfectly legal, and we already know that we can give the foreign key column a different name. The result would be something like this:




Remember: each foreign key represents a different relationship.

## Recursive relationships

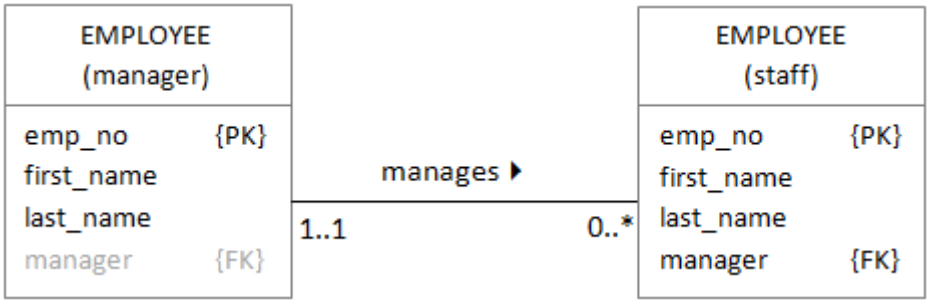
In some cases, an entity may be related to another entity of the same type. The classic example of this is in a company where some employees are managers and are therefore responsible for other employees. The ER diagram would look like this:




The *manager* column contains the *emp_no* for an EMPLOYEE's manager, so if Erik Eriksson is the manager of a team of three other employees, the data might look like this:

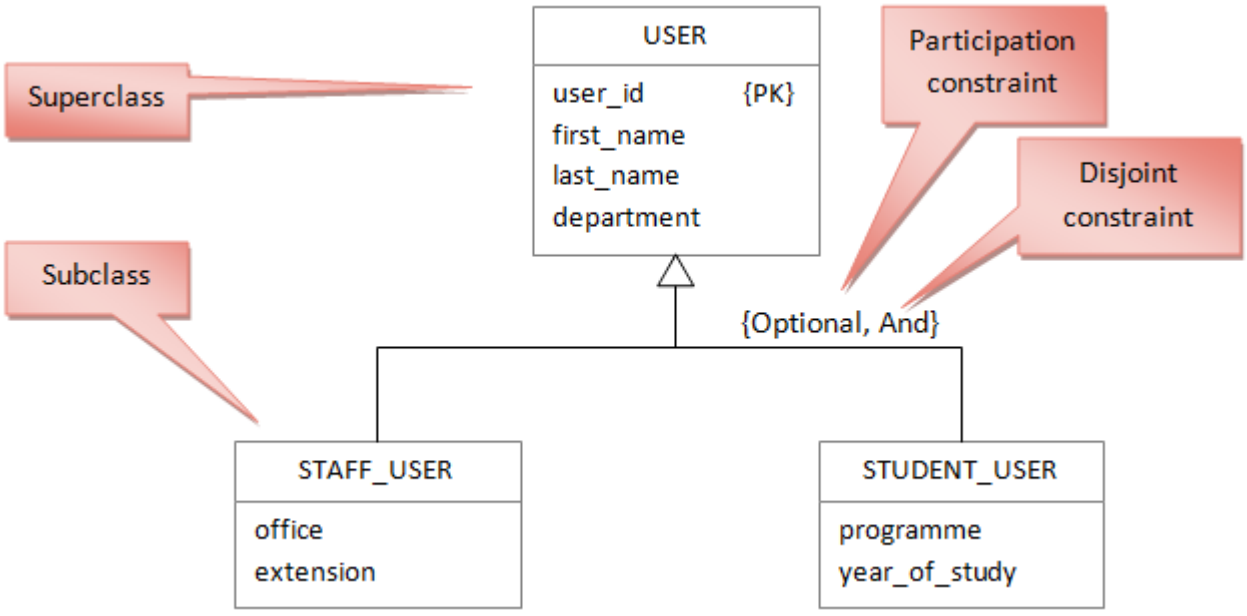
| emp_no | first_name | last_name | manager |
|---|---|---|---|
| 234 | Erik | Eriksson | |
| 432 | Donald | McDonald | 234 |
| 524 | Ivan | Ivanovich | 234 |
| 312 | Connor | o'Connor | 234 |

Recursive relationships can be difficult to understand. Sometimes it can help to imagine two copies of the table just while you are mapping the entities into database tables as shown below. It would be wrong to leave the extra copy in the ER diagram, though.




## Specialisation / generalisation

Often, a category of entities can be broken down into smaller categories. Each of the smaller categories shares some attributes, but also have differences in other attributes. If you are familiar with object orientation, this idea should be familiar. Imagine, for example, a database used by a university IT service department to store user details. All users will share some attributes such as first name, last name and department/school; however some of them will be staff and some will be students. Each of those sub-types has additional specific attributes. You can represent this in an enhanced ER diagram like this:



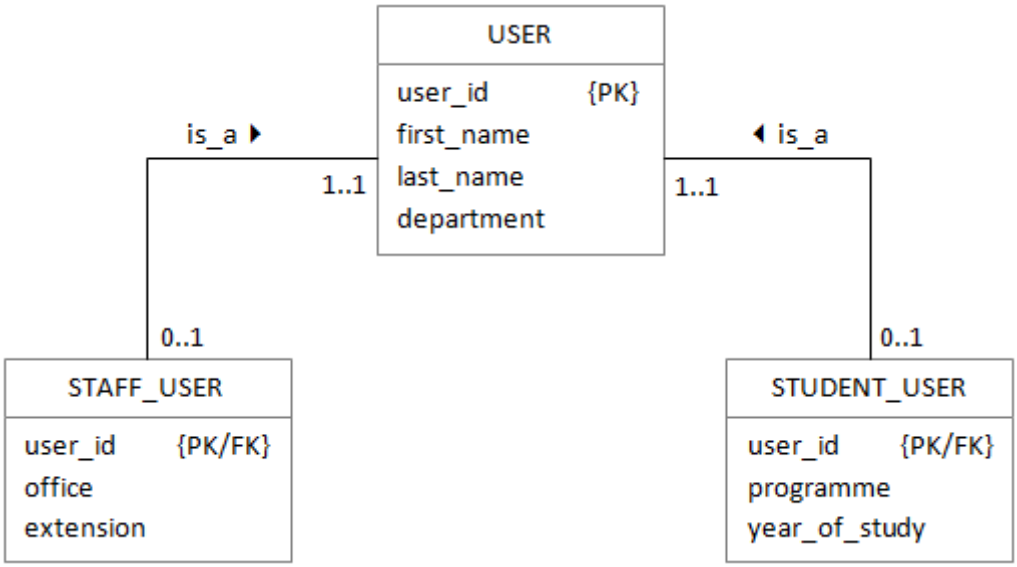

The diagram shows the USER superclass split into two subclasses, STAFF_USER and STUDENT_USER. Notice the large white arrow that indicates the specialisation/generalisation relationship and the two constraints.

The *participation constraint* takes one of two values. *Mandatory* would mean that every USER entity must be either a STAFF_USER or a STUDENT_USER. The value o*ptional* used here shows that there may be other types of USER (guests, for example) who are neither staff nor student.

The *disjoint constraint* also takes one of two values. *Or* would mean that an entity must be in only one subclass; the value *and* in the diagram shows that a USER can be both a STAFF_USER and a STUDENT_USER (if for example, a lecturer is taking a postgraduate course).

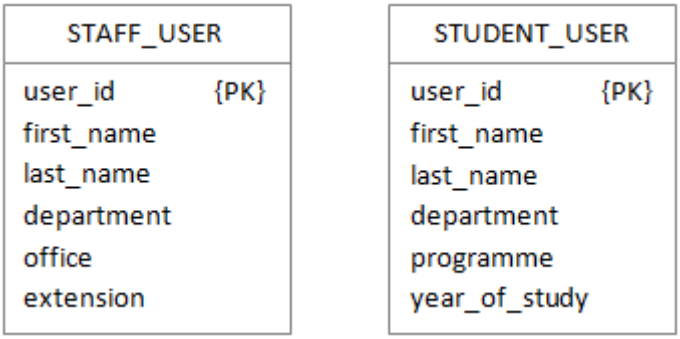
This level of detail is possible when modelling real entities and their relationships, but you still need to decide how to map them into database tables. You have three options, and your choice will depend on the application you are designing. Only the first method is a true reflection of the superclasses and subclasses. If either of the other methods is chosen as the most suitable then the specialisation/generalisation relationship should be removed from the model.

**1. One relation for the superclass and one relation for each subclass**



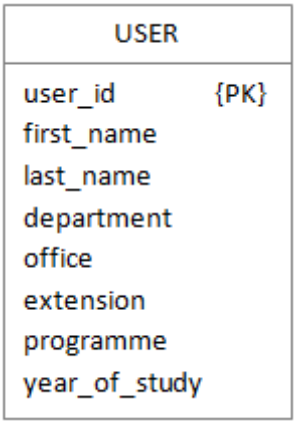

The primary key of the superclass is mapped into each subclass and becomes the subclasses primary key. This represents most closely the Enhanced ER model. However it can cause efficiency problems as there needs to be a lot of joins if the additional information is often needed for all staff.

**2. One relation for each subclass**




All attributes are mapped into each subclass. It is equivalent to having three separate entity types and no superclass.

**3. One relation for the superclass**



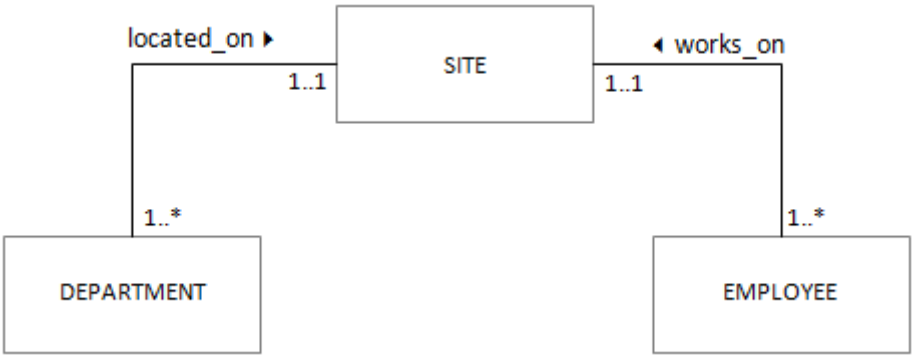

This represents a single entity type with no subclasses. This is no good if the subclasses are not disjoint or if there are relationships between the subclasses and the other entities. In addition, there will be many null fields if the subclasses do not overlap a lot. However, it avoids any joins to get additional information about each member of staff.

## Problems

Entity relationship modelling is simple enough in theory, but many situations will arise that need careful thought. There are also a couple of common error situations that you need to be aware of. They arise quite often because they are not immediately obvious.

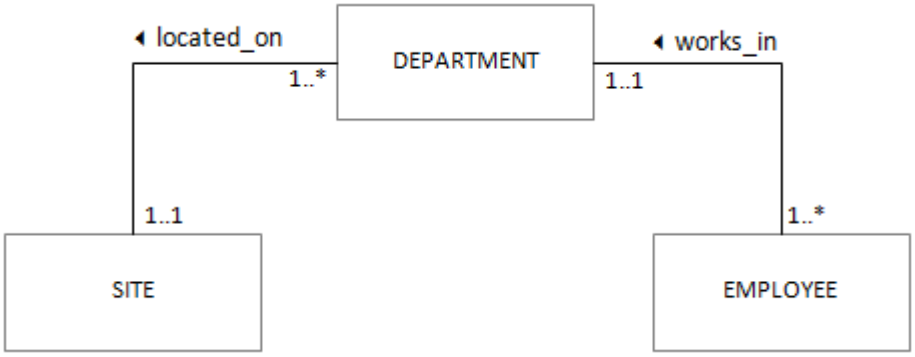
### Fan traps

A fan trap occurs when a model represents a relationship between entity types, but the pathway between certain entity occurrences is ambiguous. It occurs when 1:* relationships fan out from a single entity.




A single site contains many departments and employs many staff. However, which staff work in a particular department?

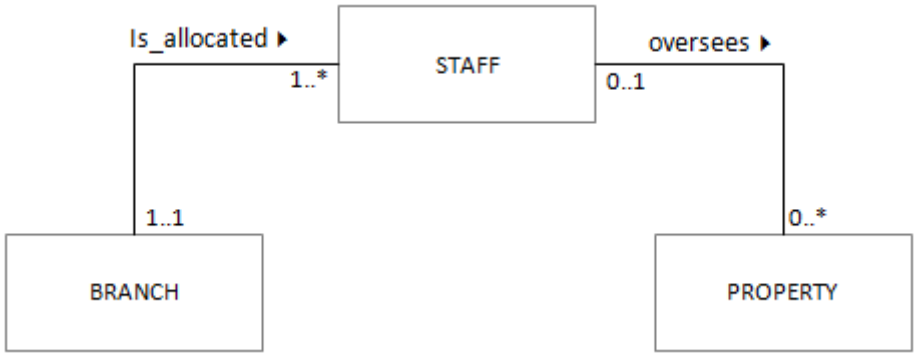
The fan trap is resolved by restructuring the original ER model to represent the correct association.




## Chasm traps

A chasm trap occurs when a model suggests the existence of a relationship between entity types, but the pathway does not exist between certain entity occurrences.

It occurs where there is a relationship with partial participation, which forms part of the pathway between entities that are related.




- A single branch is allocated many staff who oversee the management of properties for rent. Not all staff oversee property and not all property is managed by a member of staff.
- What properties are available at a branch?
- The partial participation of STAFF and PROPERTY in the oversees relation means that some properties cannot be associated with a branch office through a member of staff.
- We need to add the missing relationship which is called *has* between the BRANCH and the PROPERTY entities.
- NB. You need to be careful when you remove relationships which you consider to be redundant.