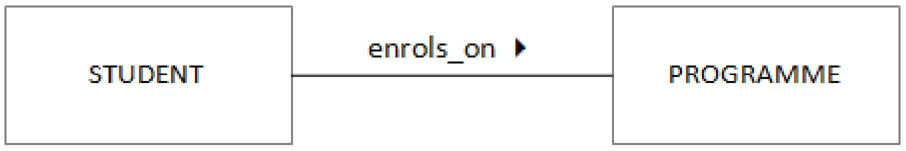
The relational model

Introduction

In last week's material we briefly mentioned the four essential elements of a data model. It is very important to know what they are and to understand their role, so here is a reminder:

Component	Description	Examples
Entities	The important things in the real world that need to be modelled Groups of the same type of objects are called *entity types* or *entity sets* Entity sets are represented in a database by tables, and each row in a table represents one entity of that type	People, places, objects, events, etc.
Attributes	Individual items of data associated with an entity Entities of the same type have the same attributes, but the values of those attributes may differ from one entity to another. That is the same as saying that all records in a table share a particular column, but the values in that column may be different for each row.	Name, national insurance number, weight, date of manufacture
Relationships	Ways in which entities are connected	A is part of B, A lives in B, A produces B, A takes place in B, etc.
Constraints	Rules which place limits on the data that is allowed	Every A must have a B, Only future dates are allowed, etc.

When describing a model of a real-world situation, it is useful to show it in a diagram. We can use rectangles to represent entities, and connecting lines to represent relationships between entities:



This type of diagram is called an entity-relationship diagram (ER diagram or ERD) for obvious reasons. This is a very simple example, and we will go on to explore how the essential information about the details of the database structure can be included.

Being able to read and draw accurate ER diagrams that correctly reflect the database structure is essential. This is one of the key skills in this module, and you will need to demonstrate your skills in this area in the coursework *and* the exam.

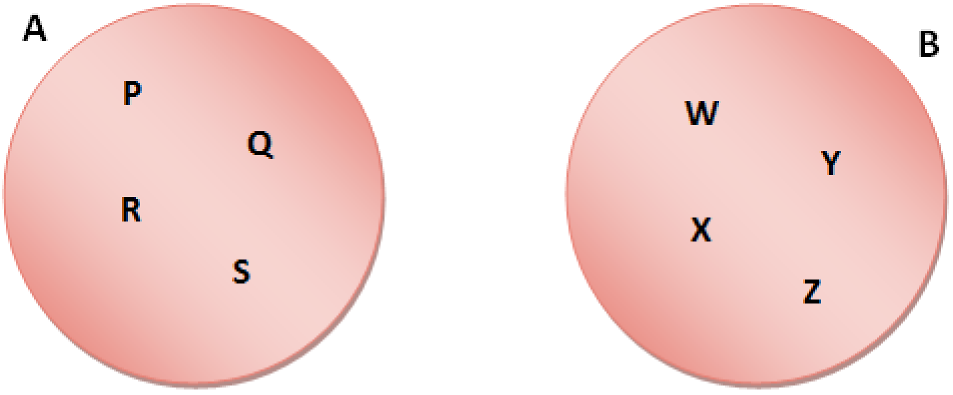
What is a relation?

Take a deep breath: here comes some maths.

The word *relationship* is a common one which you think you understand. You have probably assumed - and most people do - that the name *relational database* comes from the fact that relationships are defined between entities. In fact this is not the case. Any kind of data model is made up of four main components: entities, attributes, relationships and constraints regardless of its type. Thus you don't need a relational database to capture relationships.

The term *relation* comes from a branch of mathematics called set theory which deals with groups of objects and the relationships between them. A *relation* is therefore a precisely defined formal construct.

Imagine two sets, A and B, as shown below. Each one represents a group with four members

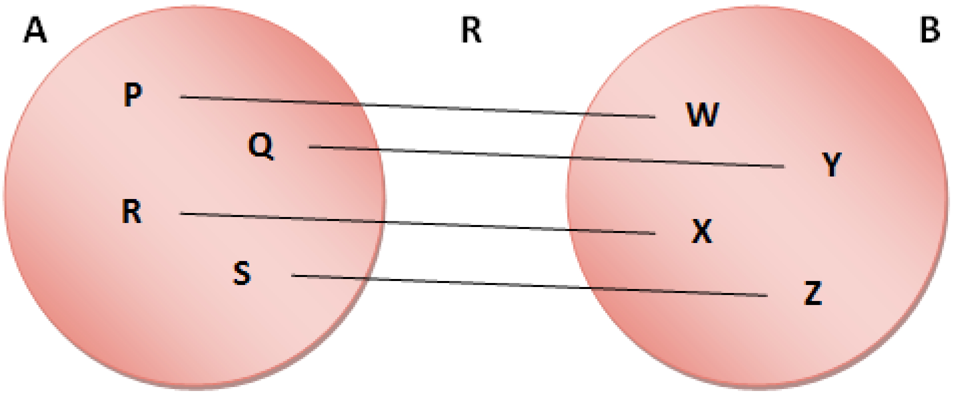


We could also represent these sets without using a diagram like this:

A (P, Q, R, S)

B (W, X, Y, Z)

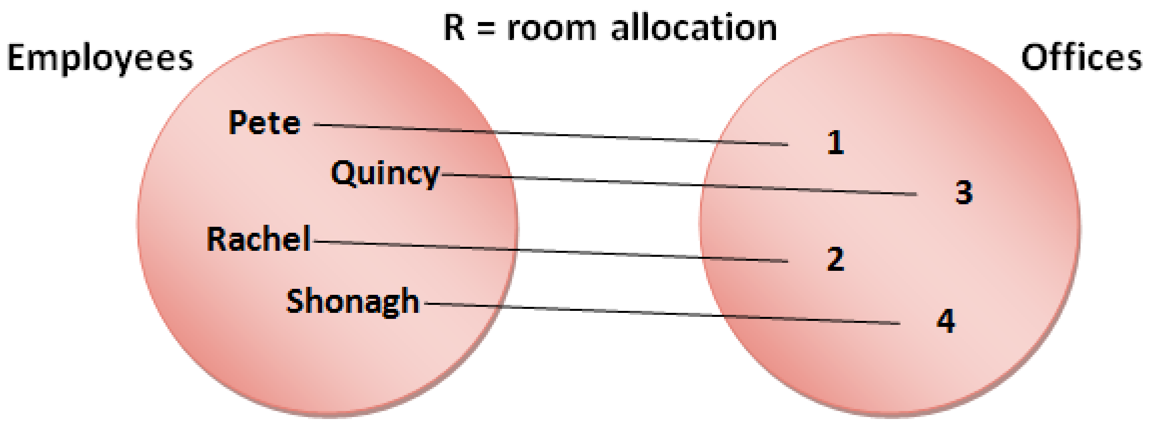
A mathematical relation is a mapping of members of one set to members of the other set as shown in the diagram below. We can give a relation a name - the one in the diagram is called R.



In the example, each member of set A is linked to one and only one member of set B, and the relation can be expressed as a set of pairs:

R = ((P, W), (Q, Y), (R, X), (S, Z))

All well and good, but why is this interesting? The relevance to database structure becomes more obvious if we take some real examples rather than the traditional mathematical symbols. Let's say that set A represents the employees of a small company, and set B represents the rooms in the company's premises. Relation R now represents the allocation of people to offices.



Now the relation makes some sense:

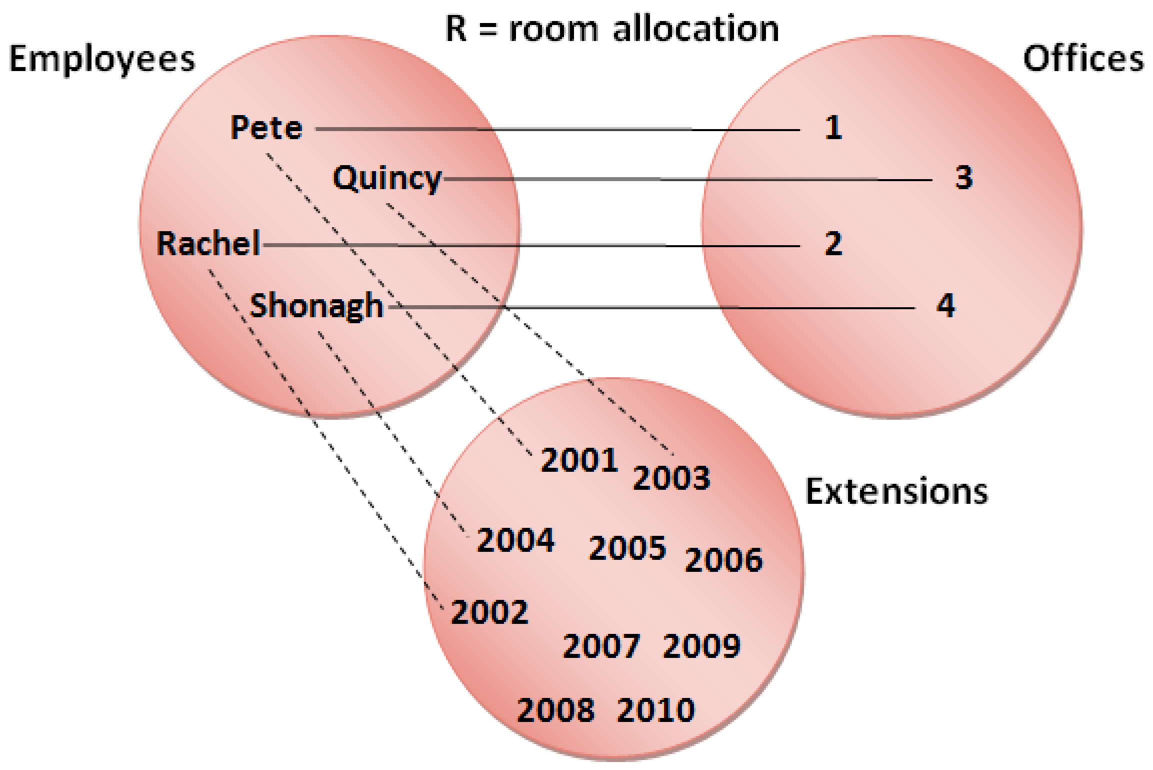
R = ((Pete, 1), (Quincy, 3), (Rachel, 2), (Shonagh, 4))

We can now go one more step and represent the relation as a table rather than a list of pairs:

employee	office
Pete	1
Quincy	3
Rachel	2
Shonagh	4

Because the values in the first column all come from the set of employees, we can say that this set is the *domain* for this column. Likewise the set of all offices is the domain for the second column. In this trivial example, the importance of domains is not obvious; however, controlling the values that are allowed in a particular column is an important part of maintaining data integrity. The database designer therefore needs to identify a column's domain, and implement the necessary constraints to ensure that only legitimate values may be inserted. This is known as*domain integrity*.

We could add another set to our example, such as the set of all telephone extensions in the company:



The relation now defines a set of triples rather than pairs - that is, each member of the relation is a group of three values, one from each set. The number of sets in a relation is known as its *degree*, and if we apply the same term to a database table, its degree is simply the number of columns it contains. Database tables can have many columns, and after pair, triple and quadruple, the terminology becomes awkward, so the generic term *tuple* is used to mean a group of related values. Sometimes, the degree is added and so you may see the expressions *4-tuple* or*5-tuple*, for example.

Notice in the the three-set example that not all of the values from the extensions set are used. This is an important feature of domains: they define *possible* values, not actual ones. the equivalent database table is shown below. Remember that a table represents an entity set, and the columns represent the attributes that each entity may have.

employee	office	extension
Pete	1	2001
Quincy	3	2003
Rachel	2	2002
Shonagh	4	2004

A feature of the values in a column is that they are all of the same type. This is quite a simple observation, but again there are practical consequences for relational databases.
The *Employee* column contains the first names of people, which in computing terms are text strings. This is the *datatype* of the attribute. The datatype of each column is part of the definition of a table in a database, and the most basic constraint on the data stored in the table is that it must be of the appropriate datatype. Trying to store a string value in a number column, for example, would cause an error.

In this example, we have used two of the three basic datatypes used by all relational databases, strings and numbers. The third is the datatype used to store dates and times which takes different names in different database platforms. Typically, computer systems treat hours, minutes and seconds as fractions of a day, and therefore dates and times are represented in the same way.

Thinking about the datatype of an attribute and the domain of that attribute, you can see that sometimes they are the same. For example, a table containing historical events might have a *description* column, and an *event_date* column. Although we could say that only dates in the past are permitted in *event_date*, it is difficult to imagine what constraints to place on description values, especially when modern databases can hold strings of indefinite length. In practice though, there is usually some upper limit on the length of text string that can be stored, and this demonstrates the difference between datatype and domain: Although the datatype defines a potentially infinite set of values (think of the set of all integers), the domain of an attribute is usually a limited subset.

Keys

One of the main purposes of a database is to be able to find the information you need as quickly as possible. This requirement leads to the seemingly trivial observation that it must be possible to uniquely identify a specific row in a table. However, this is not always as easy as it sounds. For example, given a group of four people like the employees in our imaginary company, how might we uniquely identify them?

employee	office	extension
Pete	1	2001
Quincy	3	2003
Rachel	2	2002
Shonagh	4	2004

Using their names might be a good method. However, if our group was 4,000,000 instead of just four, would the name method still work? The chance of several people having the same name in a group that size is quite high, and therefore we need some other method for finding specific individuals.

In the relational model, a *key* is an attribute or group of attributes whose values can be used to uniquely identify a row. The concept of a key is quite easy to grasp since there are many everyday examples. Telephone numbers act as keys by identifying one and only one fixed line or mobile SIM card. You use these keys without realising every time you make a telephone call. A key is simply a piece of data that can be used to look up other data reliably. When someone calls you, the network users the telephone number to find any other information it needs about you to connect the cal.

When we have a key such as a telephone number which can be used to look up a set of further attributes from the same row in a database table, we say that the key attribute *functionally determines* the other attributes. In other words, if you can look up, say, the network using the number, then the number determines the network. Notice that this relationship is not reversible: knowing the network does not enable you to look up the number - there will be many numbers on the same network. Instead, we say that the network is *functionally dependent* on the number. You will need these concepts in week 6 when we look at normalisation.

Say that our company needed to employ a new member of staff, and that the new person has to share a room. Coincidentally, the new employee's name is Pete:

employee	office	extension
Pete	1	2001
Quincy	3	2003
Rachel	2	2002
Shonagh	4	2004
Pete	4	2004

Looking at the table, there is clearly no single attribute that can be used to identify each row uniquely. However, we could use the combined values of the *employee* and *office* attributes. This is perfectly acceptable in the relational model, and a key that includes more than one attribute is known as a *composite key*.

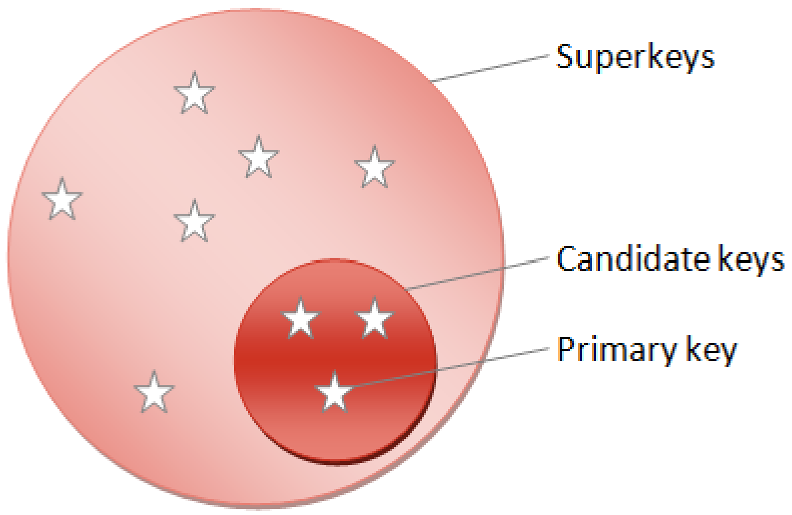
In fact, it is likely that the EMPLOYEE table would also include other data such as last name and national insurance number as shown below:

first_name	last_name	date_of_birth	ni_number	office	extension
Pete	Brown	16/4/1964	12435 B6934C	1	2001
Quincy	Green	12/11/1974	23412 D6274C	3	2003
Rachel	White	15/2/1985	34253 F6351B	2	2002
Shonagh	Brown	20/7/1969	23415 K3652A	4	2004
Terry	Black	12/11/1974	31243 W2784D	4	2004

Now there are actually many keys that we could use to uniquely identify a row. The national insurance number is guaranteed to be unique for each individual, so it is a natural key. There are also many possible composite keys - the list below shows the options:

1. ni_number
2. ni_number, last_name
3. date_of_birth, office, first_name
4. first_name, last_name, date_of_birth
5. first_name, last_name, date_of_birth, extension

Most of the options contain redundant attributes - they specify more data than is necessary to identify one row. For example, *last_name* is redundant in option 2 because *ni_number* is already enough to uniquely identify a row. Any attribute or combination of attributes that can be used to uniquely identify a row is known as a *superkey*. A *superkey* which contains no redundant attributes is known as a *candidate key*. It is then up to the database designer to select one of the candidate keys as the one that will be recognised in the database as the proper way to identify rows in the table. The final selection is known as the *primary key* for the table, and is one of the most important elements of relational database design. The existence of a primary key guarantees a property called *entity integrity* which means that every individual entity in a set can be uniquely identified. The diagram below summarises the different types of key:



NB. Sometimes it may not be possible to find a single attribute or group of attributes that uniquely identify an entity. The only solution in that case is to introduce a unique id number. This is known as a *synthetic key*, and all reputable relational database platforms have ways to let you generate unique numbers. It may be tempting to add a synthetic key to every table, but that may lead to errors in normalisation. In general, you should only use synthetic keys if

- There is no viable group of attributes that can be used as a primary key
- There is a potential candidate key, but its value may change over time
- There is currently a good candidate key, but future records may have the same value
- The only viable candidate key is composite and includes a date value - dates are difficult to use and can lead to query errors
- The only viable candidate key is a composite key with several components - a synthetic key may be used for convenience to simplify joins

Relationships

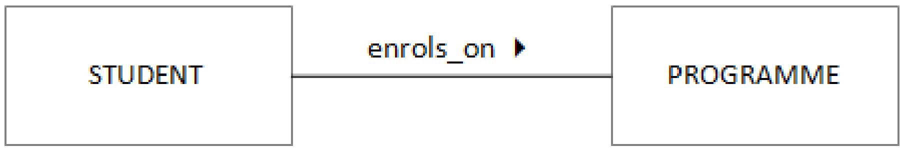
Summarising what you know so far:

- *Entities* are important things that we need to store information about
- Similar entities can be grouped in *entity types*
- An entity type is represented in a database by a *table*
- Each *row* in a table represents one entity of that type
- The individual pieces of information about an entity are its *attributes*
- Attributes correspond to the *columns* in a database table
- Entities can have many different *relationships* with each other
- Relationships between entities are shown using an *entity relationship diagram* (ER diagram or ERD)

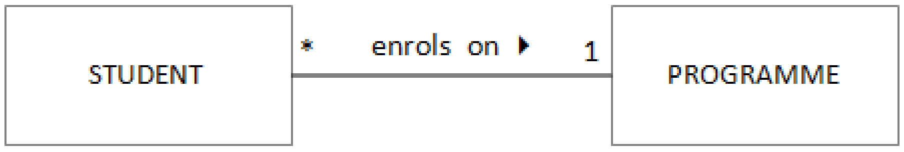
Real world situations are typically vague and poorly structured. Part of the job of a database designer is to use the simple structures available within a relational database to represent these quite complex situations. To bridge the gap between the capabilities of the relational model and the real world, we need to think about the different types of relationship that can exist between entities. We also need to know how to represent the kind of uncertainty that is found in the real world. We can do this by using some simple examples using the conventions of ER diagrams. As we go through, we will add more information to the diagram so that it represents the real situation more closely.

Multiplicity

First, take the example that was in this week's introduction:

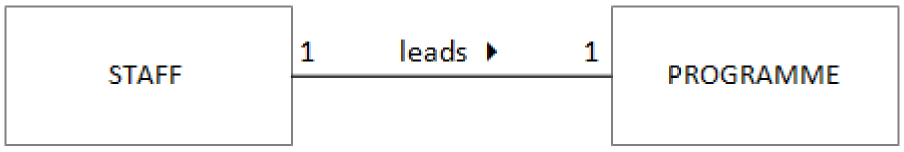


This diagram adequately captures the idea that a STUDENT enrols on a PROGRAMME; however, that is clearly not sufficient. Many students will usually usually enrol on the same programme. We therefore need some indication on the diagram of the number of entities of each type that are involved in the relationship, and this is shown in the version below.

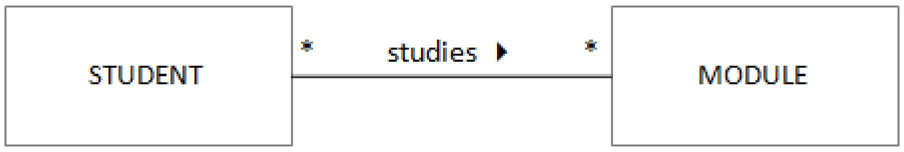


In ER diagrams, an asterisk (star) is interpreted as "many". *Enrols_on* can therefore be described as a "many-to-one" relationship, usually written **:1.* Notice that the relationship has direction: *enrols_on* is equivalent to a relationship called *is_populated_by* in the opposite direction. We would call *is_populated_by* a 1:* relationship. A good rule of thumb is to name the relationship from the perspective of the entity at the "one" end. Most relationships will be 1:* or *:1, but other types also exist:

1:1 (one-to-one): one member of staff is the leader of one programme



: (many-to-many): one student studies many modules

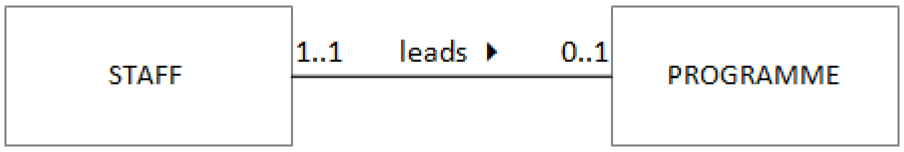


The specification of the number of entities involved in the relationship is known as its *multiplicity*; however, this is not yet the end of the story.

Notice that you must be careful when defining the cardinality of a relationship. The 1:1 relationship above is only correct if there is some rule that says a member of staff can only be leader of one programme at a time. If such a rule does not exist, one person could be the leader of many programmes, and the relationship is actually 1:*. This is a very common error, and to identify it, you need to think about the relationship in *both* directions.

Cardinality and optionality

The last diagram above seems correct... However, what about the new student who has not yet registered for any modules? Similarly, what about the new module which has not yet been run? What about the majority of staff who are not programme leaders? Clearly, we need to able to show whether a relationship is *mandatory* (it must always be true) or *optional* (it can sometimes be true). We can do this quite easily on our diagram by adding a little more information to the multiplicity labels:



The multiplicity labels now show a range of values of the form *minimum .. maximum*. The diagram above can therefore be read from left to right as

A member of staff leads zero or one programmes

This captures the idea that a member of staff does not have to be a programme leader, because that person can be linked to zero programmes via this relationship. Therefore, the relationship is optional. Notice that this is true if there is a rule that prevents a member of staff from leading more than one programme. The relationship can also be read in the other direction as

A programme is led by one and only one member of staff

This leaves no question of there being a programme with no programme leader, and the relationship is therefore mandatory in this direction. This is just one of the ways in which direction is important in defining relationships.

To summarise:

- A full definition of a relationship includes the number of participants - this is the cardinality of the relationship, and the options are 1:1, 1:* or *:*
- A relationship can be mandatory or optional, and each direction has to be considered independently
- Optionality can be expressed on an ER diagram by extending the cardinality labels
- Cardinality and optionality together make up the multiplicity of the relationship.

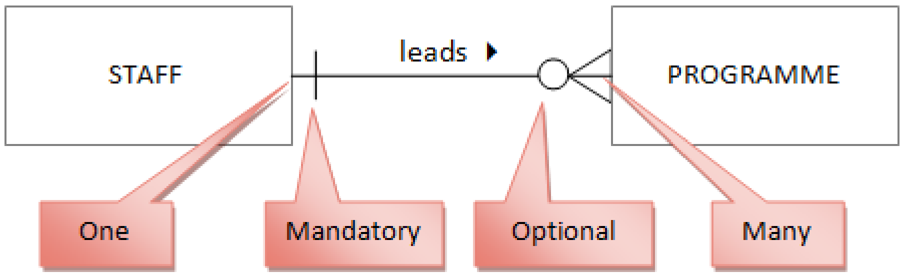
NB. Do not confuse the two types of shorthand:

- The labels *1:**, **:1*, *1:1* and **:** refer to *cardinality* only, and are applied to relationships.
- A multiplicity label of the form *n..m* describes *one end* of a relationship and includes information about cardinality *and* optionality.

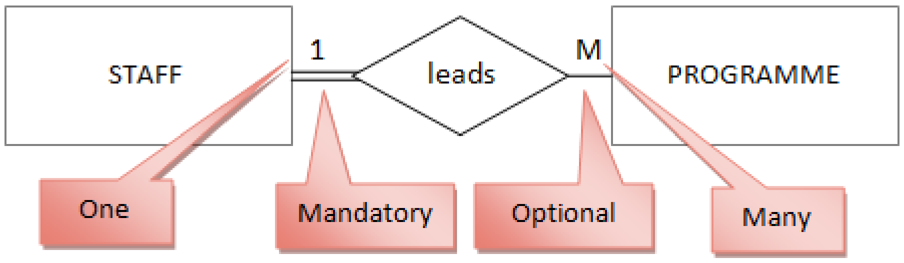
ER diagrams

The diagramming conventions that we are using are in fact a simple version of the class diagram defined in the Universal Modelling Language (UML) standard toolset. Although there are some differences between object-oriented modelling and relational modelling, the diagram conventions can be applied in both situations. This is convenient because you are likely to use other diagrams from UML in other modules. However, you should be aware that there are other conventions for setting out ER diagrams which use different symbols.

Crow's feet notation uses single or multiple ends on a relationship line to represent cardinality:



One of the first people to formulate a set of conventions for ER diagrams was P. P. Chen in 1976. His diagram placed the relationship name in a diamond, and used different types of line:

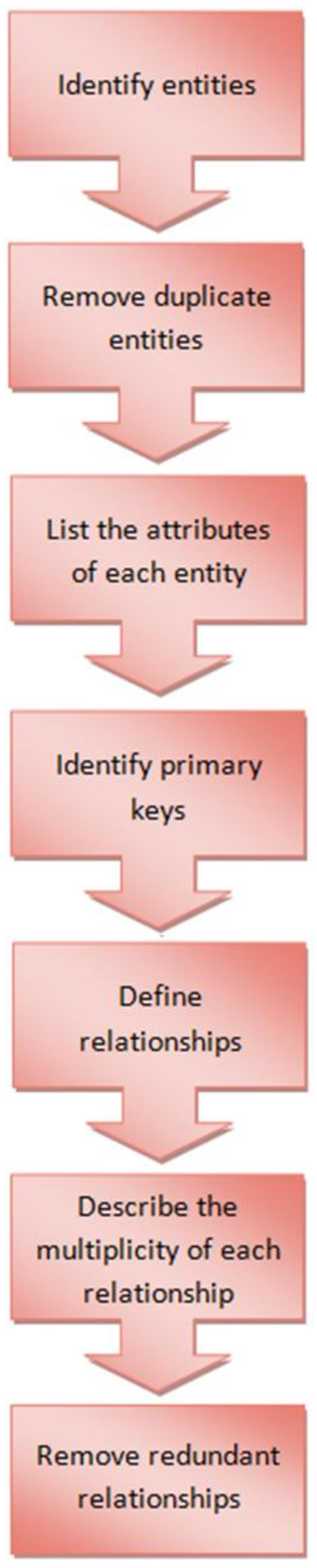


Notice that the different diagram styles convey the same essential information. Another benefit of using the UML class diagram is that it allows you to be very specific about the multiplicity of a relationship. For more information about alternative diagrams, see Connolly and Begg, or simply spend some time on the Web.

Developing an ER diagram

The starting point for any database development is to extract information about the real world situation, usually through direct observation and speaking to the people involved (the eventual users of your system). If you are lucky, an analyst may already have started the process, and there may be a structured requirements specification document available. In this case, it is best to approach the spec with some scepticism: writing something down in words is a lot easier than constructing a formal consistent model, and it is highly likely that there are details missing. In a real project, you would document any assumptions that you are making in the rest of your analysis where certain information has not been provided.

As you got through the available information, try to follow the process below. Like all analysis and design processes, it is unlikely that your first pass through the steps will give an accurate result, and you will need to *iterate*. The process is complete when you find no more changes are required.


Identify entities
Remove duplicate entities
List the attributes of each entity
Identify primary keys
Define relationships
Describe the multiplicity of each relationship
Remove redundant relationships

These are the object of interest in the system.
It is better to put too many entities in at the beginning and them discard them later if necessary.

Ensure that they really are separate entity types or just two names for the same thing
Also do not include the system as an entity type
e.g. if modelling a library, the entity types are books, borrowers, etc.
The library is the system, thus should not be an entity type.

Check that some of your entities are not simply attributes of another entity type.
Check that attributes are attached to the correct entity

Which attributes uniquely identify instances of that entity type?
This may not be possible for some entities.

Include the relationships that are important in the context
Like entities, add more relationships early and remove them later if necessary

Think about the relationship in both directions
Use real examples to help you decide what is correct

Go through your relationships to see whether they are all required
Does one relationship duplicate another?

This process is generally known as *entity-relationship modelling*. Notice that it is a *top-down* process: you start from an overall idea of what you are trying to achieve, and work down through the various levels of detail.

Entity relationship modelling: example
Simple e-commerce site

The purpose of an e-commerce site is to allow customers to buy products. A customer can place one or more orders, and each order may include one or more products. During the checkout process, the customer is presented with an invoice for the order. Each product is supplied to the e-commerce company by a vendor. One employee of the e-commerce company processes each order, and one employee (who may or may not be the same person) prepares a shipment. A shipment contains only items for a single customer, but because there may be ordering delays, it may contain incomplete orders. Because a customer may make several orders, one shipment may also contain items from several orders.

On the basis of a scenario like this, the job of the database designer is to follow the process outlined above, starting with the identification of significant entities.

Entities

Customer	The e-commerce company needs detail of each customer to be able to send orders to the right place, and for billing
Order	For obvious legal and commercial purposes, orders need to be recorded
Order detail lines	Because an order may consist of several items, each of which has its own price and quantity, each line must be treated as a separate entity
Invoice	An invoice is an important financial document, and records the payment due for an order
Product	Product details need to be shown on the site, and customers must be able to identify the products they want
Vendor	The e-commerce company needs to be able to contact vendors for more stock when a product starts to run low
Employee	As well as setting staffing rotas and making salary payments, it is important to know who processed an order and who prepared a shipment for quality assurance reasons
Shipment	The details of a shipment need to be recorded so that the customer can track deliveries.

Remove duplicate entities

If customers were allowed to make many orders and to pay for them all in one go, it would be necessary to keep orders and invoices as separate entities because there would be a one-to-many relationship between them. In this case, however, there is a one-to-one relationship between them, and so we should consider combining them. In fact the only additional attribute required would be the date that the order was paid for.

List attributes

customer
first_name
last_name
address
phone_no
email

order
reference_no
order_date
discount
subtotal
delivery
tax
total
paid_date

detail_line
line_number
product_id
quantity
unit_price

product
product_id
name
description
price
quantity_on_hand
minimum_stock

vendor
name
address
phone_no
contact_first_name
contact_last_name
vat_no

employee
first_name
last_name
address
phone_no
email
ni_no

shipment
shipment_date
delivery_date

The QUANTITY_ON_HAND attribute in the **PRODUCT** entity is for recording the current stock level. A value of 100, for example, would mean that there were 100 units of the product currently available. The MINIMUM_STOCK_LEVEL value is another stock control feature. Once theQUANTITY_ON_HAND drops below the MINIMUM_STOCK_LEVEL, more product needs to be ordered from the vendor.

If the meaning and use of any of the other attributes are not clear, please be sure to ask for an explanation from the tutor.

Identify primary keys

Every entity needs a primary key in order to guarantee entity integrity (each individual entity or row in a table must be uniquely identifiable). Primary keys can be single attributes or composite keys, made up of several attributes. If no viable key is available, an easy solution is to introduce an id number. This new attribute carries no information, and only serves as a unique key for each entity. It is known as a *synthetic key* and will be discussed further later.

In the tables below, primary keys are shown underlined, and you can see that most of them seem to be synthetic keys. A primary key must satisfy three criteria: it must always have a value, that value must always be unique and the value must not change. Can you see which of these apply here? Notice too that **ORDER** has a composite primary key.

One tricky case is the NI_NO attribute in **EMPLOYEE** - surely all National Insurance numbers are unique? One reason for using a synthetic key for employee is purely pragmatic in that where the employee record is first set up, that person's NI number may not be known. The new employee might supply it some time after being taken on. In that case, we don't want to create a bureaucratic delay just because we don't have a particular piece of data. Instead, we want to create the new record and add the missing data later.

customer
customer_id
first_name
last_name
address
phone_no
email

order
reference_no
order_date
discount
subtotal
delivery
tax
total
paid_date

detail_line
reference_no
line_number
product_id
quantity
unit_price

product
product_id
name
description
price
quantity
minimum_stock

vendor
vendor_id
name
address
phone_no
contact_first_name
contact_last_name
vat_no

employee
employee_id
first_name
last_name
address
phone_no
email
ni_no

shipment
shipment_id
shipment_date
delivery_date

Define relationships

The relationships between entities can be extracted from the description of the business domain:

- One customer may make many orders
- One order must consist of one or more detail lines
- One product may appear in one or more detail lines
- One employee may process one or more orders
- One shipment must contain one or more detail line items
- One shipment is prepared by one employee
- One vendor may supply one or more products

Describe multiplicities

Remember that multiplicity is composed of cardinality and optionality. Cardinality is the easier of the two, but it still needs to be worked out carefully to avoid mistakes. Taking cardinality first, most of the relationships are obvious from the relationships that have been identified. What about the relationship between **SHIPMENT** and **EMPLOYEE** though? From the wording above it looks like a one-to-one relationship; however, if you check it in both directions you will see that it is actually one-to-many:

- One shipment is prepared by one employee
- One employee may prepare one or more shipments

To be sure that your cardinalities are correct, you need to check each relationship like this.

Now thinking about optionality, the key question for each relationship is whether one of the related entities can exist without the other. If so, the relationship is optional in one direction. If each entity may exist without the other, it is optional in both directions. Optionality can be represented in words by careful use of *may* to indicate an optional relationship and *must* to indicate a mandatory one. As with cardinality, you need to check every relationship in both directions to avoid mistakes. The final version of the relationships in words looks like this:

- One customer may make many orders
- One order must be made by one customer
- One order must consist of one or more detail lines
- One detail line must be related to one order
- One product may appear in one or more detail lines
- One detail line must refer to one product
- One employee may process one or more orders
- One order must be processed by one employee
- One shipment must contain one or more detail line items
- One detail line item may be part of one shipment
- One shipment must be prepared by one employee
- One employee may prepare one or more shipments
- One vendor may supply one or more products
- One product must be supplied by one vendor

Each relationship must be represented by a pair of keys. We have already identified the primary keys, but some of the foreign keys are missing and we need to add them now. In the tables below, foreign keys are shown in *italic*. Remember that the foreign key goes at the *many* end of the relationship.

customer
customer_id
first_name
last_name
address
phone_no
email

order
reference_no
order_date
customer_id
discount
subtotal
delivery
tax
total
paid_date

detail_line
reference_no
line_number
product_id
quantity
unit_price
shipment_date

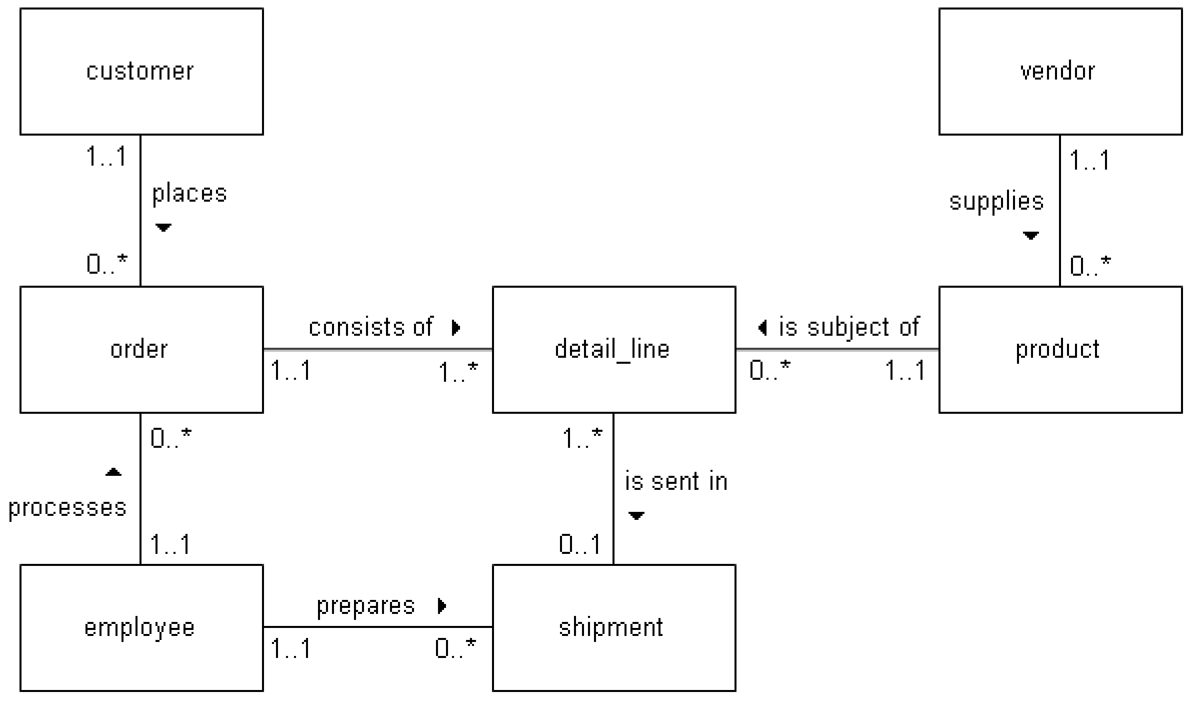
product
product_id
name
description
price
quantity
minimum_stock
vendor_id

vendor
vendor_id
name
address
phone_no
contact_first_name
contact_last_name
vat_no

employee
employee_id
first_name
last_name
address
phone_no
email
manager_id
ni_no

shipment
shipment_id
shipment_date
customer_id
delivery_date

Draw E-R Diagram



Final question: can you explain why the relationship from **DETAIL_LINE** to **SHIPMENT** is optional?