Ian Kenny

October 14, 2016

Databases

Lecture 4

Announcements

- Course schedule - tests
- First test next Friday at 1200, no lecture.
- Notes from previous years.

Introduction

In this lecture

- Overall database design process.
- Database planning.
- Requirements gathering.
- Database design phases.
- Application design.
- Entity-Relationship modelling.

Database planning

As discussed in the first lecture, many systems require a database of some kind. An individual organisation may have many systems and many databases. Databases are a critical component of most (at least moderately-sized) businesses.

When an enterprise requires an *Information System* it will generally require database functionality. Thus, the database system will require planning alongside the more general organisation information system (or systems).

We will concentrate on the database component of the information system. It is likely to be a crucial component. The design of the database system will, however, run in parallel with the design of other aspects of the system and will affect those and be affected itself.

Database planning

Every organisation will have a *mission* (or should have) and the mission will drive the choices made in planning and designing a database system.

The database system needs to help fulfil the organisation's mission hence enable the organisation to achieve its goals. An organisation will have many goals and the database system will contribute to their achievement or, importantly, non-achievement.

So in planning a database system, an organisation's high-level goals must be considered, but considered in the context of other factors such as any existing systems and legacy systems that may exist. Does the new system need to supplement or replace those systems. Does it need to be integrated with them.

Database planning

The boundaries between the new system and existing systems needs to be very clearly defined otherwise much waste and chaos could ensue.

The *user views* will also need to be established early on, i.e. what will be required of the database (in general) from the perspective of all of the different groups of users who will need to use it. This could have a big impact on the overall design and must be considered early. The data and operations needed for each group of users must be considered.

The planning of a database system also enables the organisation to make the most of new opportunities, perhaps giving it a competitive edge.

So in the planning phase, an organisation will seek to commence the creation of a database system that supports its goals, but with constraints and opportunities in mind.

Requirements gathering

Informed by the planning stage, this stage will gather a very detailed record of the data and operations needed from the perspective of each group of users. This is a crucial phase and is difficult to get right (hence the long history of failed and inadequate IT systems). Many techniques exist for requirements gathering such as structured interviews, questionnaires, examining existing documentation, observations, user stories, etc. We are not going to discuss them.

The purpose is to gain a detailed and clear specification of the required operation of the system, and the data to be stored and produced.

If this phase is too lengthy and involved, however, it could lead to over-analysis and a lack of progress, or to overly complex requirements (hence an overly complex system). If it is too short and not involved enough, the requirements may be inadequate.

Of course, agile methodologies *should* help to avoid such problems...

Requirements gathering: views

A consideration at this point is, how the requirements are managed with respect to the different views for the system.

With the *centralised* approach there is basically one list of requirements. Splitting them off for the separate views is done later. The advantage this approach is that there is a managed, central list, and overlap of requirements can be identified more easily.

With the *view integration* approach, the requirements are divided up amongst the views and are integrated at a later point. This may lead to more focussed and specific requirements per view but may also lead to overlap and redundancy.

Database design: general

For the actual design phase of the database system, there are two basic overall approaches.

The *bottom up* approach starts at the lowest level - with the entities, attributes and relationships - and builds the system up from there. The advantage of this is that it starts with the actual 'nuts and bolts' of the system. The disadvantage is that this method does not scale well to large, complex systems which may have hundreds or thousands of entities.

The *top down* approach starts at the highest level and considers only a few high-level entities and relationships and then gradually breaks those down. The disadvantage of this approach is that it may take a while to get to a level of detail that allows some 'bigger' decisions to be made.

In practice, of course, the design phase will feature a mixture of these methods and other methods that are based upon them.

Data modelling

One of the most important activities in the design phase is *data modelling*. We will be considering this in more detail on this course.

However, in overview, the purpose of data modelling is to attempt to understand the structure and meaning of the data, and, importantly, to facilitate communication about this to all interested groups, including users. This means that data modelling must include information in a format that general users can understand and discuss.

The designers and the users need to establish a shared language about the design and data modelling approaches aim to do that.

Data modelling

The data modelling approach selected should have certain characteristics.

Clearly, it should enable the data and processes of the enterprise in question to be fully represented. It should not be the case that some data and process cannot be represented. This includes all constraints on the data, for example.

It should be easy to understand by all involved parties.

It should not be tied to a particular technology. It should be possible to implement the model on different technologies. The consideration of the particular DBMS technology to be used should come later on (as we will discuss).

Database design phases

It is common to conceive of the database design process as having three phases

- The Conceptual Design Phase.
- The Logical Design Phase.
- The Physical Design Phase.

Conceptual design

In the conceptual design phase, a model of the data used in an enterprise is constructed without regard for logical and physical considerations, i.e. the particular database model is ignored as are any considerations of physical storage, indexes, etc.

In this phase, associated application programs, programming languages, hardware, etc. are ignored.

We will shortly consider conceptual design in more detail.

Logical design

In the logical design phase, the actual database model is considered. For example, if a relational model is to be used then the relations and attributes, etc, are determined.

This is done without considering the particular DBMS that might be used, apart from the fact, for example, that it will be a relational database.

We will be considering logical design in a future week.

Physical design

In the physical design phase, the actual DBMS is selected and targeted for the data model (of course, it may have actually been selected long before).

With the relational model, for example, this phase will involve the actual creation of the relational tables, the implementation of the constraints, the selection of storage parameters to enable optimal searching, implementing security measures, etc.

We will be considering physical design in a future week.

Application design

We won't be focusing on application design but it is worth briefly considering in this context.

The database system will, of course, generally be supporting applications in an enterprise's overall IS system. The design of the application(s), however, will be in parallel with the design of the database system. This is because there are clearly dependencies between them. The database system must support the needs of the applications and the applications must provide the correct access to the database system.

A key component of this will be the design of the user interface. This will need to provide the correct access to the application and the database system (without omissions and complexity, of course).

The user interface design may need to take into account the different views that were determined earlier on in the database design phase.

Transaction design

Another key element of this relationship between the database and application designs is the transaction design.

Each required transaction will need to be specified and implemented by the application, in conjunction with the database system. Each transaction will map to a series of operations on the database. This needs to be carefully designed so that transactions are not missed and that they contain what they need to contain (and not more).

Transactions will be *retrieval transactions* where data is sought from the database, *update transactions* where the database is modified somehow. and transactions that mix both processes.

Conceptual design

We will now look at conceptual design using Entity-Relationship Modelling as the tool to determine the design of the database.

In the conceptual design phase we identify:

- Entity types.
- Relationship types.
- Attributes and attribute domains.
- Keys including primary keys.
- Integrity constraints.

Entity-Relationship modelling

We will now look at a widely-used technique in database design: *Entity-Relationship (ER) modelling*. We will (later) be looking at the extended version of ER modelling.

ER modelling is simply a method for describing the entities, attributes and relations in a database design. It is used with diagrammatic techniques so that the design may be understood and discussed with the various interested parties.

There are a number of different formats for the diagrams. In this module we will look at standard ER diagrams (with extensions). Other techniques include the Unified Modelling Language (UML) which has its origins in object-oriented analysis and design but has been extended to cover database design. For our purposes, there is little difference between the two methods.

Terminology

ER modelling is based upon *entities* and the *relationships* between them (unsurprisingly).

When considering the design of a system, an *entity* is an object (physical or abstract) that exists in the system. Example entities might be Employee or Invoice.

Relationships are relationships between entities. For example, IssuedBy might be a relationship between Employee and Invoice.

It is not always clear whether something is an entity or a relationship. When it comes to the actual database design, it isn't that important since we must *still* translate the entities and relationships to tables anyway (if using the relational model, that is).

Entities

Entities are the 'things' in the system. They may correspond to actual physical entities (e.g. Customer) or more abstract entities (e.g. Department). They are the 'things' that the database will store and process information about.

Entities are often identified as *nouns* in a textual description of a system, for example

Employees work in a **department** and manage a maximum of 100 **customers**.

Entities

The term Entity in a database system can be used to refer to the *set* of entities, i.e. all actual entities of a particular type. This type can also be called the *Entity Type*. It can be thought of like a type in object-oriented programming. In the relational model, this sense of the term entity corresponds to a table.

Sometimes, however, the term Entity is used to refer to an Entity Type *instance*, i.e. some actual entity in the system. In the relational model, this sense of the term Entity corresponds to a *tuple* of a table.

The context should make clear which sense of the term is being used.

Attributes

Attributes are the *properties* of the entities. They provide information about each specific entity in the database system. For example, the Employee entity might have attributes such as payrollNumber, address, salary, manager. The Invoice entity may have attributes such as invoiceNumber, raisedBy, invoiceAmount.

Attributes are usually *simple*, which means they are atomic and cannot be broken down further. *Composite* attributes are complex attributes that contain multiple components. An example of this would be an address. An address could be a simple attribute hence the entire address text would be a single attribute, or it could be composite where the parts of the address are split across multiple attributes (for example, houseNumber, street, postcode, etc.). The decision about whether such an attribute is simple or composite would depend on the application.

Attributes

Each attribute has a *domain*, which is the set of the values from which its value may be drawn. Example domains are 'integers', 'strings of length 10', etc.

At the conceptual stage in a design, we don't concern ourselves with how the values are restricted to values in the domain. These additional constraints can be applied later.

Derived attributes

Some of the attributes of an entity can be derived from other attributes in combination with other information. For example, an *age* attribute can be derived from a *date of birth* attribute and the current date (a system value).

Should derived attributes be stored or computed as needed (by an application, for example)? If they are to be stored, how often should they be updated? That depends on factors such as how frequently they will be updated, how 'crucial' they are, etc. To compute a derived attribute, the processing time will have to be expended at some point. The question is whether to compute it immediately every time one of its terms changes, to update it overnight or when the system is less busy. If it is essential that the derived attribute is correct at all times, then it will be updated immediately. If it can wait, then it could be updated overnight.

Keys

A *key* is a minimal set of attributes that uniquely identifies each instance of an entity, i.e. each row of a table, in the relational model. There may be multiple *candidate keys*.

Keys must not contain null values.

The *primary key* for an entity is the candidate key selected as the key for the entity. Which candidate key is selected as the primary key will depend on various considerations, such as simplicity, key length, future guarantee of uniqueness, etc.

Relationships

Relationships usually correspond to the verbs in a description of a system. In this way they relate the objects (entities) in the system together, as verbs relate the objects in a sentence.

Employees **work in** a department and **manage** a maximum of 100 customers.

A *relationship instance* is a particular instance of a relationship, i.e. the association between actual entity instances in the model.

Strong and weak entity types

A *strong entity type* is an entity type that does not rely on the existence of another entity for its existence. Strong entity instances can be identified by the primary key of that type.

A *weak entity type* is an entity type that relies on the existence of another entity for its existence, i.e. it is not independent. Weak entity instances cannot be uniquely identified only by the attributes of the entity type, i.e. they do not have a primary key.

Weak entity instances can only be identified through their relationship with another entity. An example is an entity representing the dependants of a member of staff (for the purpose of employment-related benefits). An instance of this entity will only exist because of the existence of another entity (e.g. StaffMember), and can only be identifed through the key of the other entity.

Entity-Relationship modelling

For any system being designed, which 'things' become entities, which become attributes and which become relationships may be clear at the beginning, or an understanding may emerge as work progresses on the design. A further step (when it comes to the logical design) is which of the entities and relationships become tables (in the relational model).

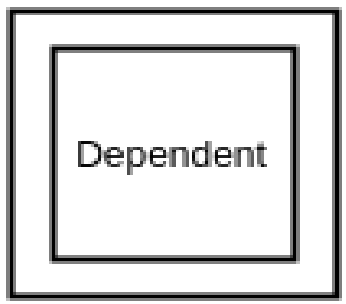
During the conceptual design phase, diagrammatic methods will be used to shape and discuss the system. We will now look at the use of ER diagrams to model a system. The basics of the ER diagram are very easy. The trick is to get them right.

ER diagrams: entities

In an ER diagram, an entity is usually represented by a rectangular box with sharp or rounded corners, for example

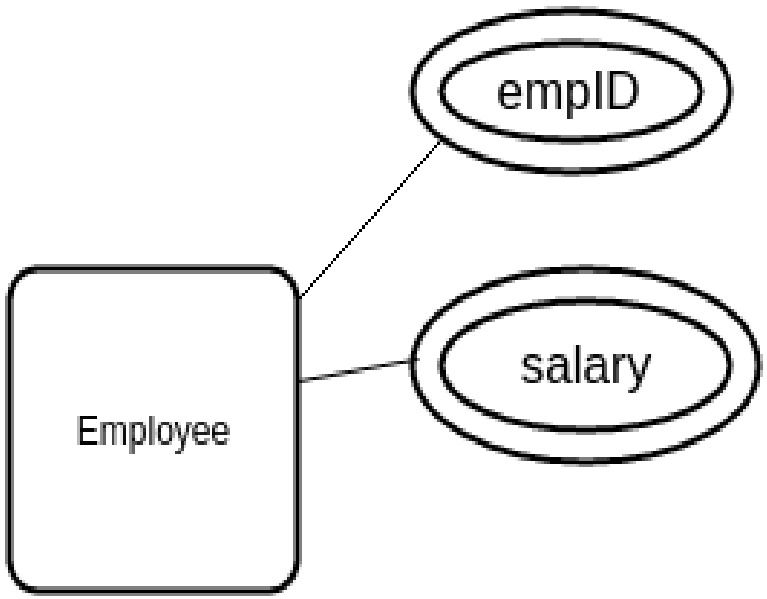
ER diagrams: weak entities

In an ER diagram, a weak entity is usually represented by a rectangular box with sharp or rounded corners with another box inside it, for example

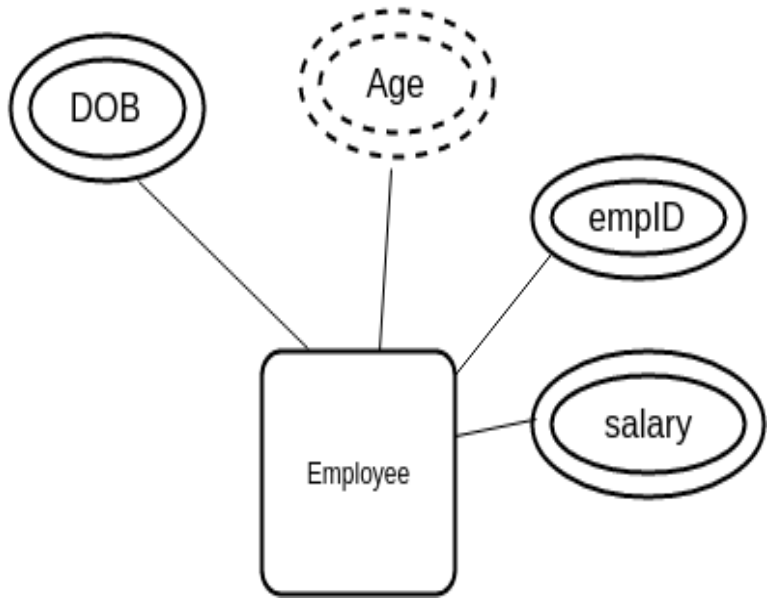


ER diagrams: attributes

Attributes are shown as ellipses with the name of the attribute inside, and they are connected to the entity they are an attribute of, for example

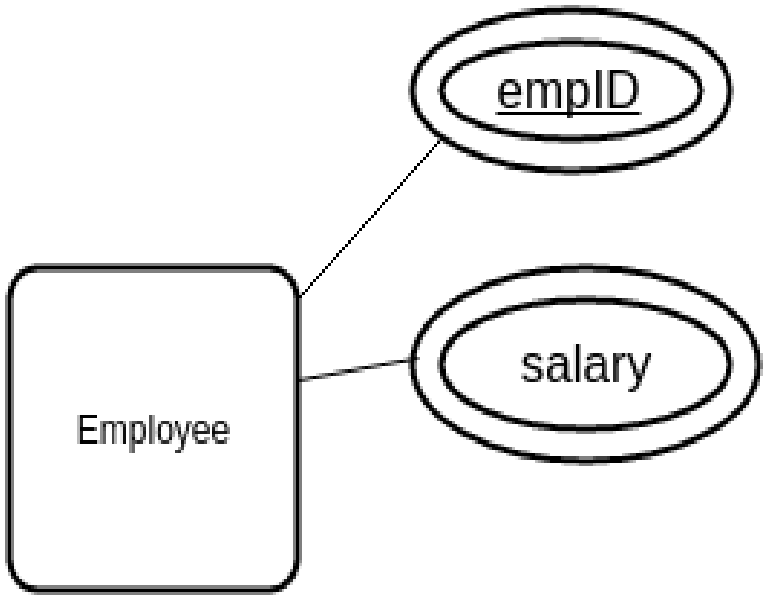


ER diagrams: derived attributes



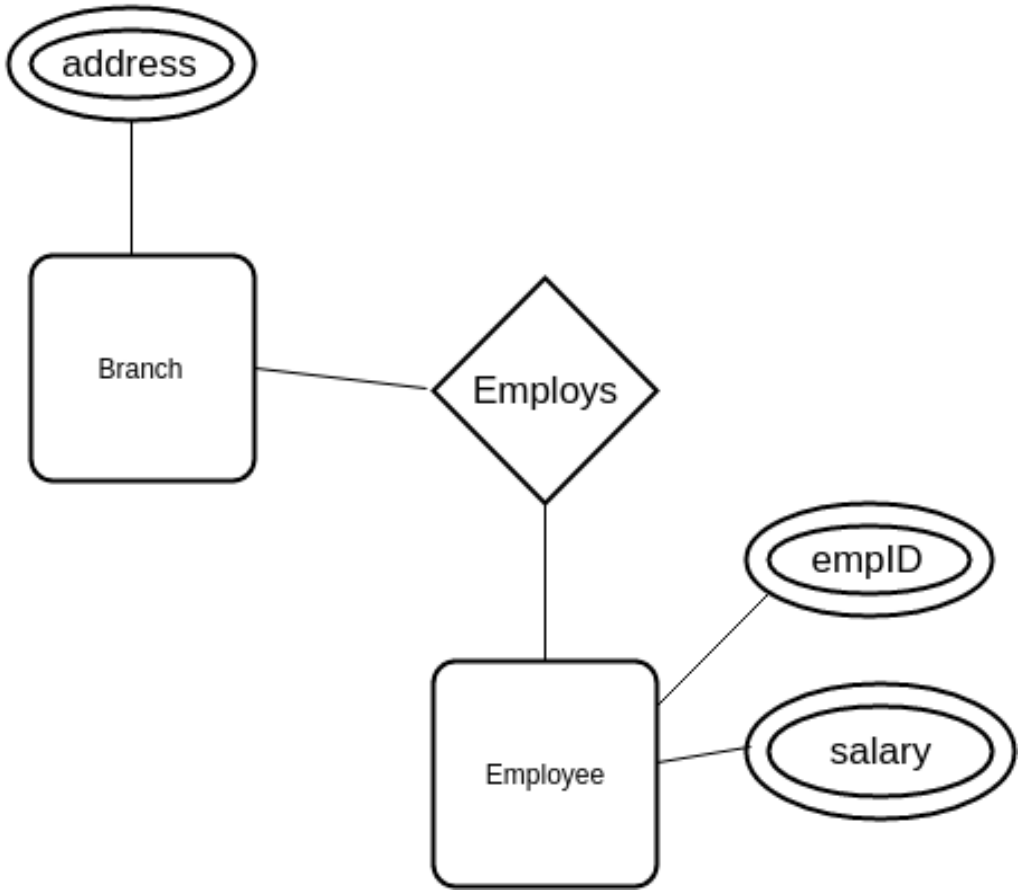
Primary keys

Primary keys can be shown simply by underlining the primary key attribute name.



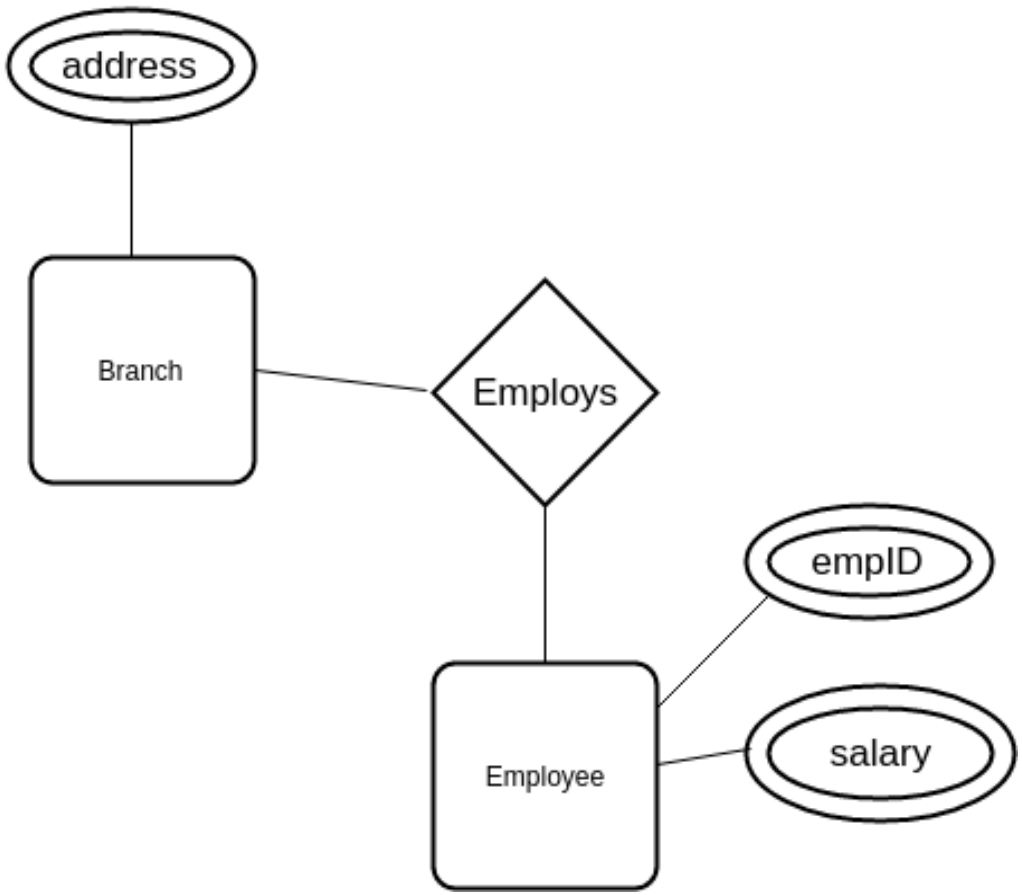
ER diagrams: relationships

Relationships are drawn as diamond shapes with the name of the relationship inside,for example



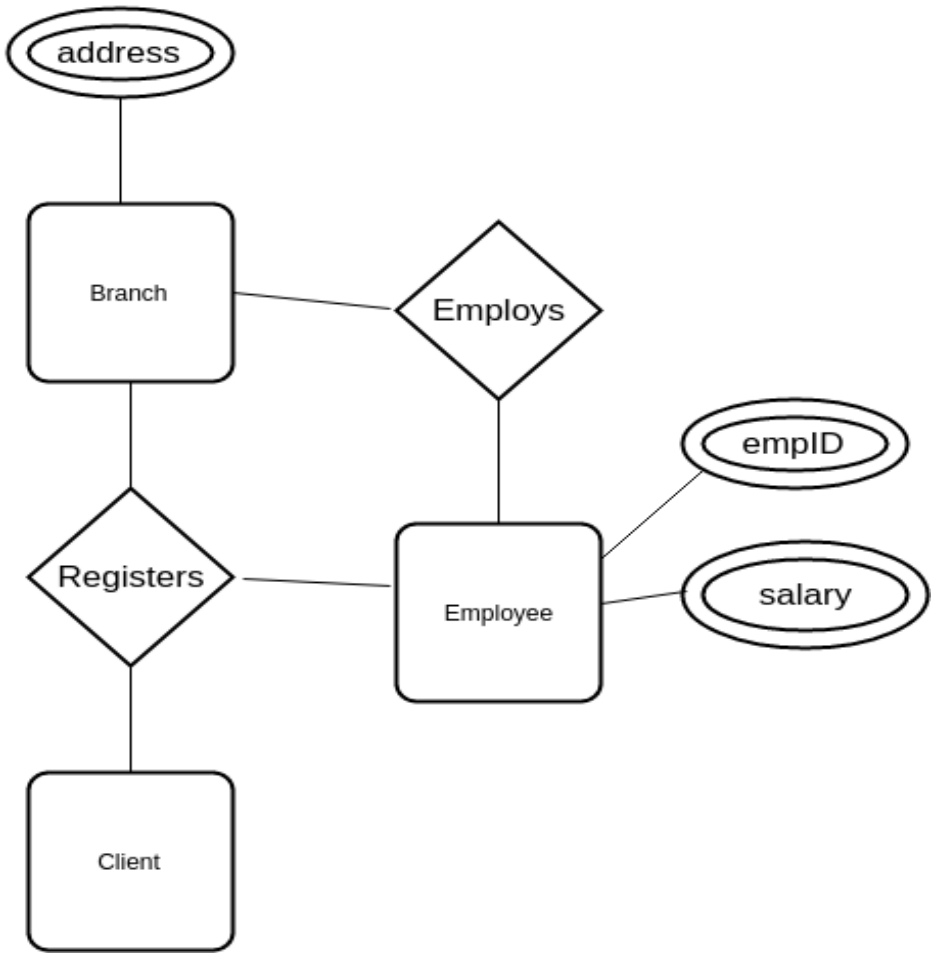
Relationship degree

The *degree* of a relationship is the number of entities that participate in the relationship. Most relationships are binary, for example, the Employs relationship in this diagram



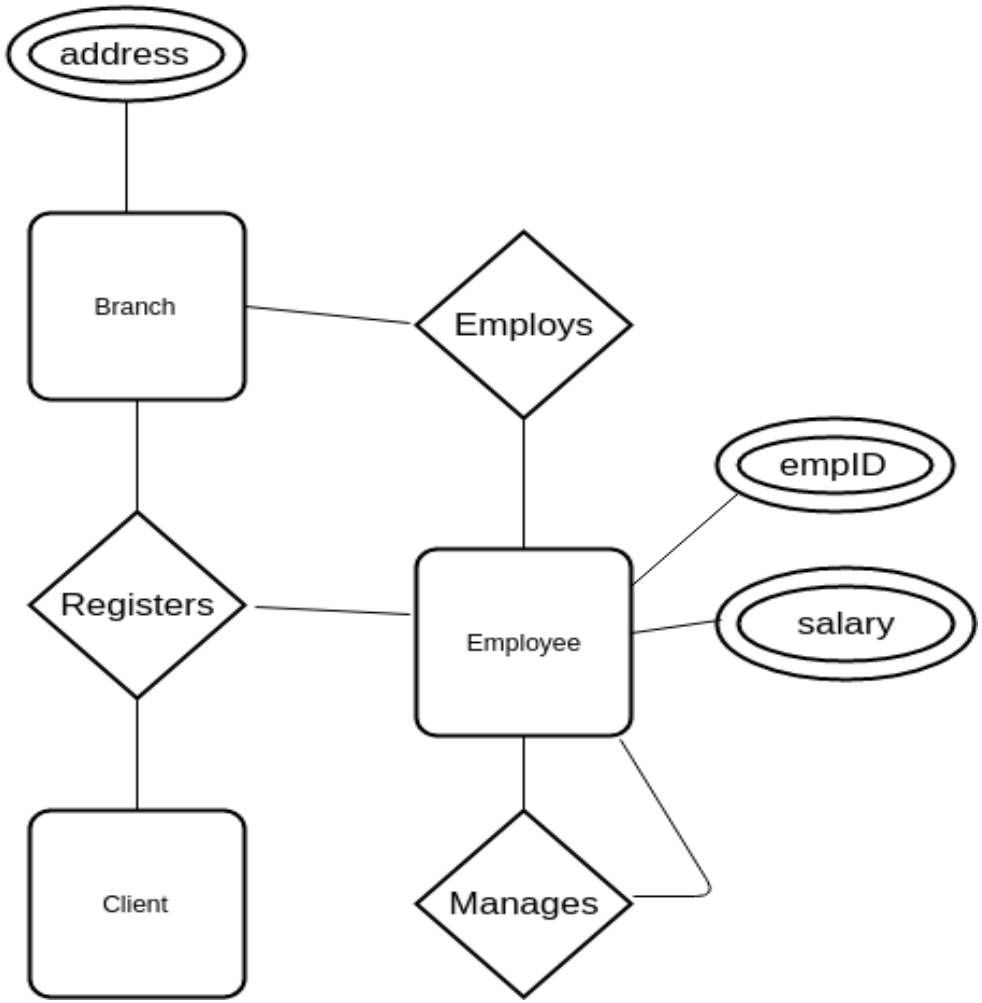
Higher degree relations

Higher degree relations are much less common. A relationship involved three participants is a *ternary* relationship. Degrees higher than three are usually called *complex* relationships. Here, Registers is a ternary relationship.



Recursive relationships

A relationship between an entity and itself is called a *recursive* relationship. Recursive relationships are *unary* relationships. For example, managers are employees but they also manage employees



Structural constraints: multiplicities

The real-world domain being modelled by a database system will place certain constraints on the relationships between entities. Some of these constraints will relate to *number* of entities of one type that may be related to another type for any given instance of the relationship.

Each relationship needs to have its multiplicities considered at each 'end point'. There will be two end points for a binary relationship. The multiplicities at each end of the relationship might be different.

(Note that on the following diagram the UML notation is used for multiplicities since it is easier to draw than the standard 'crow's foot' method.)

Multiplicities: one-to-one (1..1)

The most straightforward multiplicity is *one-to-one*. This means that for each instance of a relationship exactly one entity on one side of the relationship will be associated with exactly one entity on the other side of the relationship.

An example of such a relationship would be a relationship associating a Person and Passport in the context of an airline database system. For each instance of the relationship, there will be exactly one person associated with exactly one passport. Each person has one passport and more than one person cannot appear on the same passport.

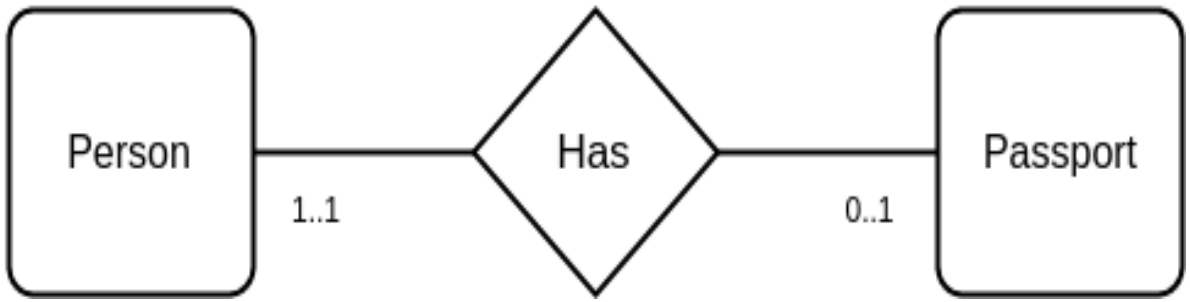
It is possible that such relationships can be removed. For example, perhaps *passportNumber* could simply be an attribute of the Person relation?

Multiplicities: zero-to-one (0..1)

In a zero-to-one relationship, one end of the relationship is *optional* and the other mandatory and involving exactly one entity.

An example of such a relationship would be one associating a Person and Passport in the more general context in which the law does not require a person to have a passport. In this context, one end of the relationship will be 0..1 since a person does not have to have a passport but, if they do, they will have exactly one passport. The other end of the relationship will still be 1..1 since each passport is owned by exactly one person.

0..1, 1..1 multiplicities

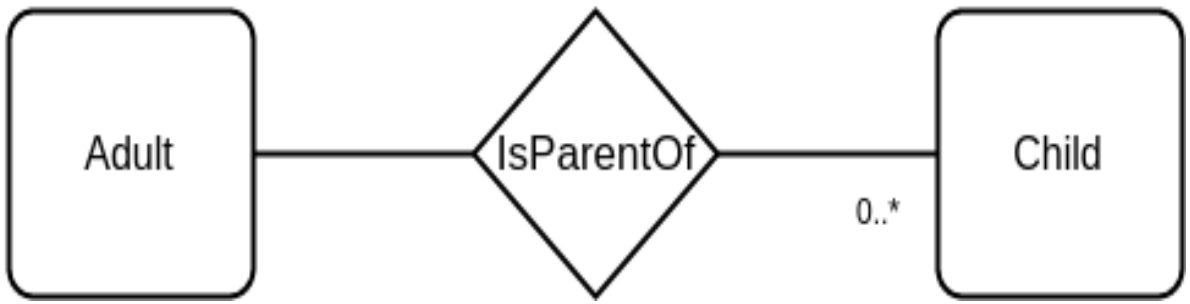


Multiplicities: one-to-many (1..*)

In a one-to-many relationship, at most one entity in a relationship is associate with many other entities in the relationship. An example of such a relationship might be the relationship between a project manager and projects, such as 'manages'. Each project manager may manage one-to-many projects (and each project has exactly one manager - where such a constraint applies).

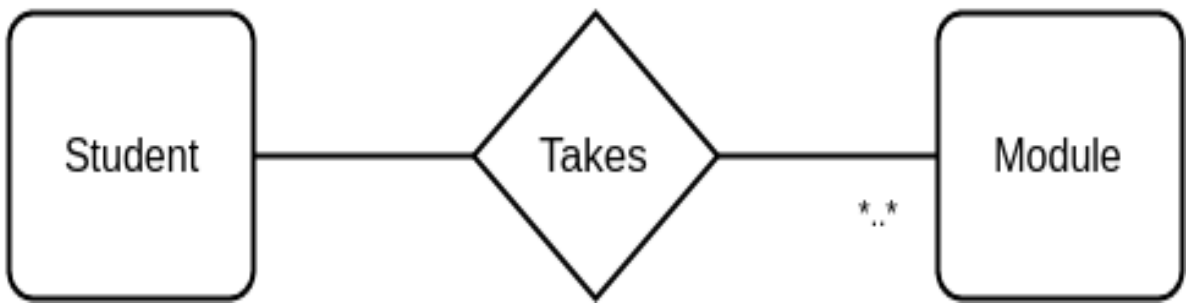
Multiplicities: zero-to-many (0..*)

In a zero-to-many relationship, zero entities in a relationship may be associated with many other entities in the relationship. An example of such a relationship would be the 'IsParentOf' relationship between adults and children.



Multiplicities: many-to-many (*..*)

In a many-to-many relationship, many entities in a relationship may be associated with many other entities in the relationship. An example of such a relationship would the relationship between modules and students. A student can take many modules and a module can be taken by many students.



Multiplicities: many-to-many (*..*)

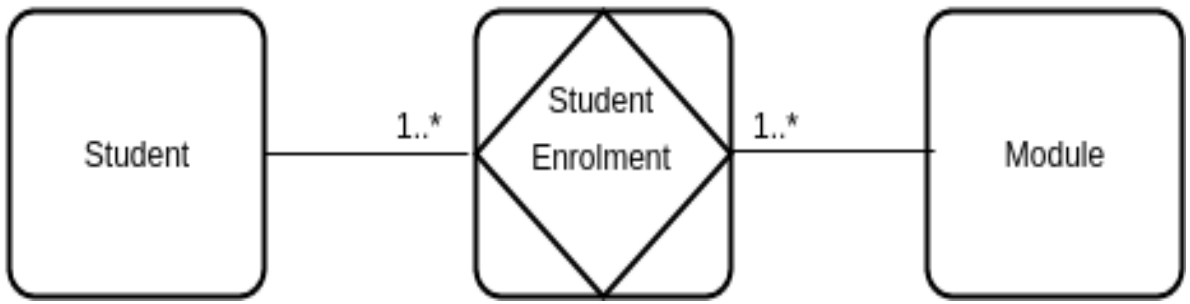
Many-to-many relationships can be hard to implement and are usually avoided. For example, how would one model the relationship on the previous slide using a relational DBMS?

One approach is to model such relationships with *Associative Entities*. An associative entity will exist solely to create an instance of a many-to-many-relationship, so that it may be modelled.

For example, with the Students/Module many-to-many relationship, we could create an associative entity called StudentEnrolment that represents each instance of a student being enrolled on a module.

Associative Entities

Associative entities can be used to resolve many-to-many relationships.



Next time

In the next session we will do a worked example of an ER diagram.