06-21923/21980 *Fundamentals/ICY: Databases* The University of Birmingham
Spring Semester 2013 School of Computer Science
originally derived from text by Achim Jung
(so he is sometimes the "I" in the text)

Handout and Exercises – Week 6

Entity-Relationship Modelling and Diagramming

1. **NOTES.** The style of diagram notation in these and later notes will differ somewhat from that used in the textbook(s) and in lectures. There are many different variants of ER diagramming.
Also, some alternative terminology (e.g., "multiplicities") is used in some places in this handout.
This document contains major enhancements by John Barnden.

2. **Identify entity types.** The first step is to find entity types that we wish (or that the customer asks us) to represent in the database.
Typical examples:

   - **People:** staff, clients/customers, patients, members, owners, contacts, other individuals, ...
   - **Objects:** stock items, real estate, offices, ...
   - **Organisations:** firms (suppliers or customers), departments, charities, clubs, committees, ...
   - **Object classes:** recordings, films, books, types of stock, biological species, work roles, ...
   - **Events:** concerts, examinations, lecture courses, consultations, sales, ...

   ⋮

   Often, the main entity types appear as *nouns* in a natural language specification. However, a *typical mistake* of beginners in ER-modelling is to include all nouns as database entity types. In each case, you must check whether there is more than one instance of the entity — otherwise, why use a table instead of a constant? Even if there are several instances, they may be totally fixed in advance, so for example, the days of the week are an unlikely entity set. Finally, you must check whether there is anyone really interested in the instances — otherwise we are cluttering the database with irrelevant information.

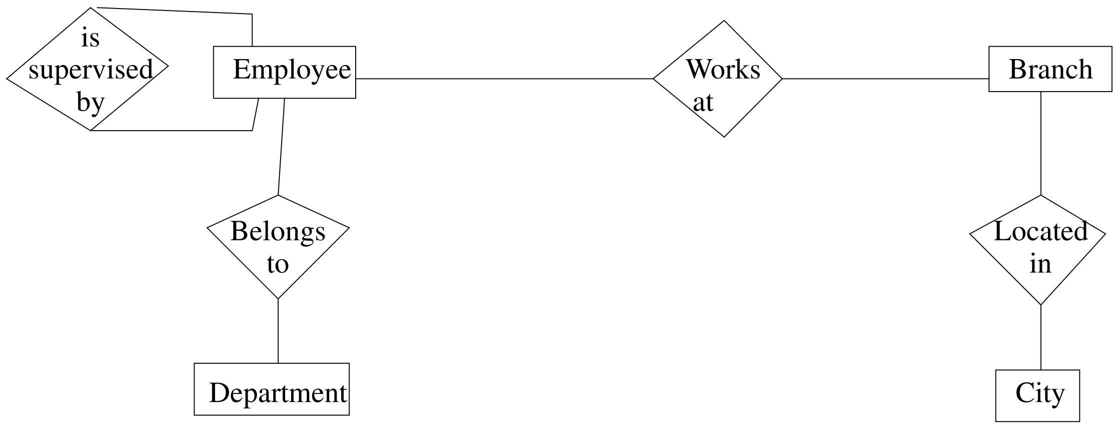
3. **Identify relationships.** Find relationships between entity types that need to be recorded:
Typical examples:

   - **Ownership:** person *owns* object
   - **Lines of command:** person *supervises* person
   - **Participation:** person *participates in* event
   - **Part of relationship:** item *is part of* order
     person *belongs to* organisation
   - **Location:** house *is located in* region
   - **Personal:** person *is married to* person
     person *is parent of* person

   ⋮

   Often, relationships are expressed by *verbs*.
   Relationships can involve more than two entity types, for example, an examination involves an examiner, a course, and a student; a property sale involves a seller, a buyer, a property, an estate agent, and two solicitors. These *higher-order* relationships are relatively rare, but important to know about.

4. **ER diagram.** The notation in these notes uses diagrams where entity types are represented as rectangles, relationships as diamonds. Small example:



Note: Beauty is not an objective here, so don't bother to waste time with a drawing program. Hand-drawn diagrams will always be fine. Clear layout, however, is very important, so try to avoid crossing lines etc.

5. **Collect attributes for entity types.**

   - Together, the set of attributes *must uniquely identify* the real-world entity that is represented. Typically, therefore, entity types have *many* attributes.
   - Additionally, you may wish to introduce an artificial identifier as the primary key.
   - Attributes can be complex: *date* consists of day of the month, month, and year; *address* consists of street, house number, parish, city, region, postcode, country, continent, planet, solar system, galaxy, etc.; *duration* consists of start date and end date. Don't worry about complex attributes at this stage in the design.
   - Attributes can be of varying structure: *children* can be a list of persons of varying length; *shipment* can consist of a varying number of items. Don't worry about this at this stage.

6. **Collect attributes for relationships that are represented as entity types.**

   - Often, the additional attributes refer to time: worked at branch *since* ...; was married to ...*from* ... *until*
   - It is normal to have *very few* attributes attached to a relationship. Indeed, it is often the sign of an erroneous model if you find that a lot of information should be recorded in a relationship table.

7. **Recording attributes in the design.**
   Some textbooks recommend to have attributes represented in the diagram; only do this if it clarifies the meaning of an entity (or relationship) name. Otherwise, you can list the attributes separately in the following form:

   Employee(<u>eid</u>, first_name, middle_name, last_name, date_of_birth, home_address,
   national_insurance_number, first_day_of_employment, ...)

   (Underline the attribute(s) that you wish to use as primary key.)
   Note that at a particular stage of ERM development we might not want to include attributes which refer to other entity types in the database, such as, department, line manager, etc., because we may not yet have decided how to handle relationships by means of attributes.

8. **Cardinalities in relationships.** For every entity and every relationship, record the (expected) number of times that a particular instance of that entity will take part in the relationship. We distinguish four different cases:

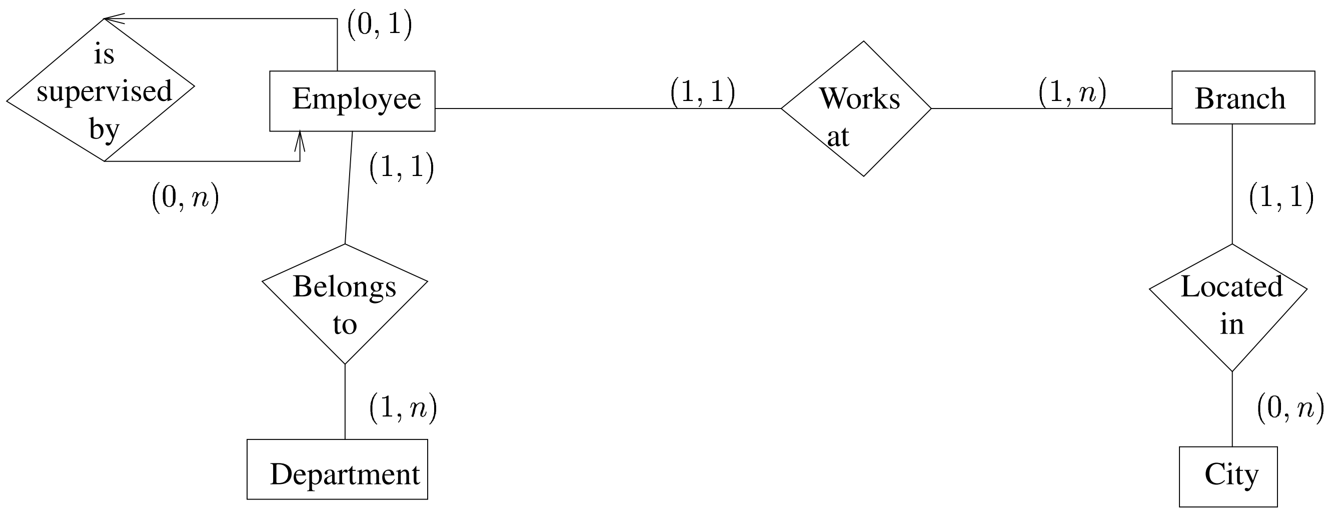
   (1,1) The instance must and will appear exactly once in the relationship. For example, every order was issued by exactly one customer. We record this multiplicity as $(1, 1)$, meaning, "at least once and at most once".

   (0,1) The instance appears at most once in the relationship. For example, although every person has exactly one father and one mother we may not be able to record their identities (even the full table of all of humanity would contain one missing entry, namely, that for the mother of Adam). We record this as $(0, 1)$, meaning, "possibly not, but at most once".

   (1,*n*) The instance appears at least once but can appear many times. As an example, every piece of music has at least one performer but there could be many musicians involved in its recording. Likewise, a bank account has at least one owner but often a spouse is recorded as second owner with equal rights. This is recorded as $(1, n)$ where we usually don't bother to make the n more specific even if we know that there is an upper bound. It would not make any difference to the tables generated.

(0,*n*) The instance may appear many times or not at all. This is the most common situation: A customer may have placed many or no order in a given time interval; a couple may have many or no children at all, etc. We record this as $(0, n)$.

Adding multiplicities to the diagram above we get:



Let's read these multiplicities and see whether they capture the real world situation correctly.

- An employee may or may not be supervised but if (s)he is, then there is only one supervisor. This is an *assumption* about this organisation which may not be valid in other companies.
- An employee may or may not be supervisor to other employees. If (s)he is, then there can be many subordinates.
- Every employee works at precisely one branch. A branch has at least one employee working there.
- A branch is located in precisely one city, and a city may have no branch or several branches of this company.
- Every employee works for a particular department, and a department has at least one member of staff.

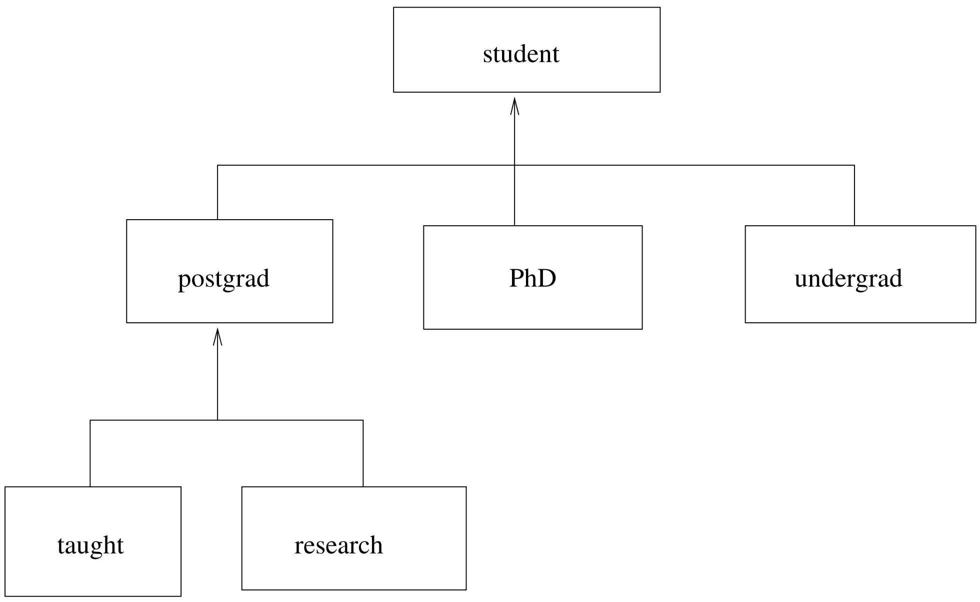
Important! Multiplicities will allow us to optimise the database but the price to be paid is a restriction on what the database can store. For example, in the organisation modelled above, it will not be possible to record a second supervisor for an employee.

9. **The future.** We will extend ER diagrams (ERDs) with further mechanisms later.

Extensions to the basic Entity-Relationship model

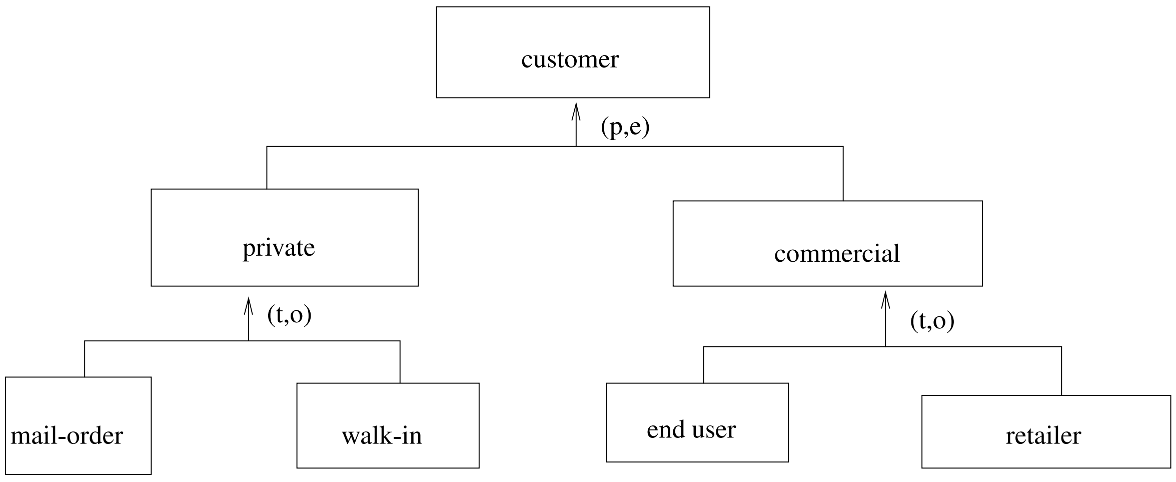
10. **Generalisation hierarchies.** Often we have to deal with situations where an entity type splits into subtypes in a natural way: A customer could be private or commercial; a student could be undergraduate, post-graduate taught, post-graduate research, or occasional; a vehicle could be a bike, a car, a lorry or something else; and on and on.
In each case we have to decide whether subtypes are interesting enough from a database point of view to warrant their own representation in the design. So for example, the distinction of human beings into males and females does not necessarily require separate entity types, in this case an attribute "`gender`" may well be enough. On the other hand, the separation of the "`student`" entity could be very useful because PhD students have supervisors (and so there is a relationship to members of staff) but undergraduates don't.
Another reason why you might want to make a sub- or superclass explicit is that there could be differences in the set of attributes that they support. In general, the subclass inherits all the attributes of the superclass but could have more. In the "`vehicle`" example, we might want to record engine size for cars but not for bicycles.
If you are unsure whether to make a subclass explicit or not, I recommend that you do. When we translate the diagram into actual tables there is an opportunity to consider this question again in the context of the whole design. If by that time it turns out that the subclass is not really necessary, then we will simply subsume the subclasses into the superclass and use attributes to distinguish instances from each other.

11. **Representing generalisation hierarchies in ER diagrams.** Sub- and supertypes are each represented as rectangles and the relationship between them is indicated by lines which end in an arrowhead towards the supertype:



As you can see, such a generalisation tree can have several levels.

12. **Annotating generalisation hierarchies.** There are two questions about sets of subtypes which will help us to decide on how they should be translated into tables. The first is whether the subtypes together cover all instances of the supertype. In our examples, we can say that males and females cover all humankind. On the other hand, our list of types of vehicles surely does not cover all possibilities. Let's reserve the letter "`t`" ("total") for the first case, and "`p`" ("partial") for the second.
The other issue is whether the subtypes are mutually exclusive or not. Again, we can state with confidence that a person can not be both male and female at the same time [Hmmm – you might wonder whether this is really true! – JAB]. On the other hand, a student on a PhD programme might at the same time take some additional classes in another subject. We record this fact with the letters "`e`" ("exclusive") and "`o`" ("overlapping").
The two letters are attached to the arrowhead in a generalisation hierarchy:



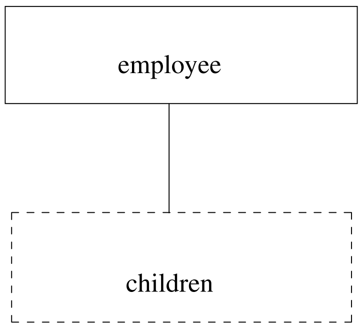
We have annotated the first distinction as "partial" because there is at least one other category of customer that we could think of, namely, government bodies. We have assumed this classification to be exclusive.
In the bottom two classifications we have assumed total coverage but with overlap between the subtypes; on the left because a mail-order customer may well walk into the shop one day and we are unlikely to want to create a new customer record in that situation. Similarly, on the right it may well be that a retailer uses some of the merchandise inside their own business while the majority is sold on (say, a car dealer may use some cars for the in-house rental business).

13. **Conflicting classifications.** There may be more than one way to classify a given entity. For example, while we can classify humans in to male and female, we can also classify them alternatively according to social status. It is all right to record these possibilities in the ER diagram without worrying too much about how they might eventually get translated into separate tables. Remember that ER modelling is all about adapting a design to the real world, not about adapting it to the needs of a computer system.

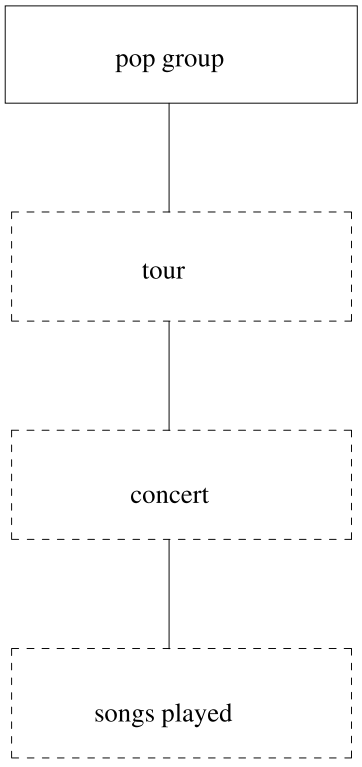
14. **Weak entity types.** We have emphasised the need for an entity to have a clear and verifiable link to an object (or concept) in the real world on several occasions before in Additional Notes. "Weak entity types" are called "weak" precisely because

this link is not of this nature. Instead, in order to relate an instance of a weak entity to the corresponding real world object, we need to have the context of another ordinary entity.
Let's look at an example. Consider a family-friendly employer who keeps a record of the children of his employees and their date of birth. This allows him to award a free afternoon to the parent whenever one of his or her children celebrates their birthday anniversary.[1] For this purpose it is enough to store first name and date of birth of each child of each employee. These two attributes, however, do not identify the children unambiguously because there may well be more than one "Mark" born on the same day. The information becomes identifying only in the context of the parent information, because we can safely assume that a person will not give the same name to identical twins.[2] Eventually, therefore, the table describing the children would have three attributes: first name, date of birth, and employee identifier (the primary key of the employee table).
Another way to view this situation is to start from the parent entity, and to consider the children as an attribute. Because the number of values of that attribute varies from one person to another (depending on the number of children they have), we can not simply have a fixed number of "`child_1`", ... , "`child_n`" attributes.

15. **Representing weak entity types in ER diagrams.** As with all elements of ER diagrams, you will find several conventions in the literature. Today I feel like representing them in the following way:



16. **Cascaded weak entity types.** It may well happen that a weak entity requires the context of another entity which itself requires the context of another entity. There is no limit to how far such a cascade may be nested, and the good news is that it does not cause any problems in the eventual translation into tables. Here is an example:



Instances at the bottom of this cascade refer to songs which were played at a certain concert during a certain tour of a certain pop group, for example, the interpretation of "Sounds of Silence" by Simon & Garfunkel in their Central Park concert on September 19 during their 1981 reunion tour.[3]

[1] Obviously, this example is entirely fictitious.

[2] On the other hand, a person may re-marry another divorcee and both may have children from their previous marriages. The real world really is a complicated place...

[3] This really shows my age ... says Achim Jung, not JAB!

Logical Design: Translating ER Diagrams into SQL "`CREATE`" Statements

NOTE 1: Table creation should ideally be done once you have made final decisions about what entity types and relationships should be present, and what attributes each entity type has. So, in particular, all replacement of M:N relationships by pairs of 1:M relationships, all re-representation of multivalued attributes, all representation of generalization heirarchies, all thought about 1:1 or symmetric relationships and how to represent them, and all apparently desirable normalization should have been done already. However, in practice it is possible that you will have a change of heart about conceptual design even while creating tables. Also, you may only be able to tell that you need to *de*normalize, or make other design changes, once you have created the database and tried it out under fairly realistic conditions. And ... your end users may suddenly realize they were wrong about what they wanted!!
NOTE 2: These notes should ideally be backed up by reading the parts of the textbook that deal with creating and updating tables.

17. Basics.
Basically, in order to translate an entity type into a table we proceed as follows:

(a) Determine attributes.

(b) Determine attribute types. Here are the main ones:

- `BOOLEAN` which has values `TRUE` and `FALSE`.
- `CHAR` which has values `'A'`, `'B'`, etc.
- `CHAR(`n`)` which has as values strings of length exactly n. If a shorter string is input then the system will pad it with blanks at the end.
- `TEXT` is the type of strings of arbitrary length (in practice, there is often an upper bound to the length as determined by the implementation of the DBMS but you are unlikely to come across this restriction).
- `VARCHAR(`n`)` is the type of strings of variable length but at most n. Longer strings will be rejected or truncated by the DBMS.
- `INT` (or `INTEGER`) is the type of integers in binary representation, just like Java's "`int`". In practice, integers have a maximum size, e.g. ±2 Billion for 32-bit integers. *Only use `INT` when you know that you will stay within these bounds!*
- `DECIMAL` or `NUMERIC` is the type of integers represented as strings of decimal digits. This will have some upper bound on the length of representations that the system is willing to tolerate. Although internally `DECIMAL`s are stored just like strings, they are interpreted as numbers. Therefore any leading zeros are removed. On the other hand, trailing zeros are preserved because they could indicate a level of precision.

  There is also a type `REAL` which behaves like Java's "`float`" but this should only be used when small rounding errors are acceptable in the manipulation of values. The point here is that floating point numbers are represented in binary format and rounding in that format can be at variance with the rounding that one would expect from numbers in their decimal format. *Always use `DECIMAL` for amounts of money!*
- `DECIMAL(`n`,`m`)` or `NUMERIC(`n`,`m`)` which is the type of decimal integers where the overall number of digits does not exceed n, and exactly m digits appear after the decimal point.
- `DATE` which is the type of calendar dates. These can be input and output according to many regional conventions.
- `TIME` which has values in the formal `'HH:MM:SS'`. Note that the numeric comparison operators $<$, etc., work for both `DATE` and `TIME`.
- `SERIAL` which provides a four-byte integer that is incremented automatically if a new tuple is inserted with `DEFAULT` as the corresponding value. This works even if several people work with the database concurrently. You should note, however, that this does not on its own entail that the corresponding field entries will be unique, as it is also possible to enter explicit values. In order to enforce uniqueness you must additionally declare the attribute to be a `PRIMARY KEY` or to be `UNIQUE` (see below).
- There are a number of additional types that you can learn about from the PostgreSQL documentation pages.

In case your data cannot be represented as a value of one of these types you may be dealing with a "complex attribute". These need to be broken down into attributes of primitive types. For example, to represent addresses as a field of type `TEXT` would be wrong in most cases because we would not be able to use the DBMS to retrieve individual aspects of an address, such as the city, or the country of residence.

(c) Determine a primary key. Usually, this requires the creation of an artificial attribute because real-world objects rarely carry unique identifiers with them. A (single-attribute) primary key could be an integer (and then often provuided by SERIAL as above), but if entity types naturally come with identifying numbers that start with leading zeroes, or come with identity "numbers" that contain non-numerical bits, such as National Insurance "numbers", then it is natural to use a string.

(d) Write out the SQL "`CREATE`" statement. For the "`staff`" table this would look as follows:

```
CREATE TABLE staff (sid       SERIAL,
                    title     VARCHAR(6),
                    firstname VARCHAR(15),
                    lastname  VARCHAR(20),
                    email     VARCHAR(40),
                    office    INT,
                    phone     INT
                   );
```

18. Foreign Keys.
The foreign key attributes should be declared as such so that the system can keep an eye on the consistency of the stored information. The basic way to do this, applicable when a foreign key consist of just one attribute, is shown by the following:

```
CREATE TABLE lecturing (cid     INT REFERENCES courses(cid),
                        sid     INT REFERENCES staff(sid),
                        year    INT,
                        numbers INT
                       );
```

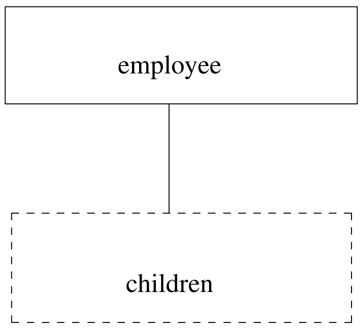
(Note: We will see later that it was a bad idea to include the attribute "`numbers`" into this table.)
Each use of `REFERENCES` above is an example for a **constraint** on the possible values in the database. In particular, it is a **referential integrity** constraint. There is more of REFERENCES constraints below.
If you have a REFERENCES constraint, then the attribute referred to must be declared as a PRIMARY KEY in the other table.
In the REFERENCES constraint for column `cid` above, the column refereed to in the other table has the same name, `cid`. Thus, you can omit the "`(cid)`" and just use "REFERENCES courses". Similarly for the other REFERENCES constraint. But you can refer to a differently named column by using that name within the parentheses: e.g. "`REFERENCES courses(course_id)`".

19. "`NOT NULL`" Constraint. When an attribute of an entity type E is never meant to have NULL values, it is advisable to declare it as being NOT NULL, as in the following example:

```
CREATE TABLE staff (staffID    INT PRIMARY KEY,
                    last_name  CHAR(20),
                    ...
                    branchID   INT NOT NULL
                                   REFERENCES branch(branchID),
                    start_date DATE
                   );
```

Then the system will complain if a "`NULL`" value is entered.
This constraint is, in particular, useful when the attribute is being used to represent a relationship from E to some entity type F, and is therefore typically a foreign key in E as in the example. Of course, the constraint is only appropriate if the relationship is *mandatory* (in the E-to-F direction).
Remember though that there are other reasons why you might want a NOT NULL constraint.

20. Weak Entities. Remember that instances of a weak entity can not be disambiguated through their own attributes alone but that information about the parent entity is needed as well. From this it is clear that the table for the weak entity should have an additional column which contains the primary key value of the controlling instance.
For an example, recall the situation we studied previously:



If the primary key of the "employee" table is called "employeeID", and we only want to record first name and date of birth for each individual child, then the "children" table should be created with the following SQL command:

```
CREATE TABLE children (first_name  CHAR(10),
                       dob         DATE,
                       employeeID  INT NOT NULL
                                       REFERENCES employee(employeeID)
                                       ON DELETE CASCADE,
                       PRIMARY KEY (first_name, employeeID)
                      );
```

Note that we require "employeeID" to be present for each child and that we want information about children to be deleted when the parent leaves the company. Also note that the primary key consists of more than one attribute, and that therefore we can not annotate an individual column declaration. Instead, we have made a separate `PRIMARY KEY` declaration *after* the last column has been declared.
The ON DELETE CASCADE is discussed in the next section.

21. More on Foreign Keys
The exact behaviour of a REFERENCES constraint can be stipulated in the `CREATE`-statement for the table in which the foreign key appears (NB: *not* in the specification of the table that the foreign key links to, i.e. the table where it is the primary key). The stipulation is by ON DELETE or ON UPDATE additions to the constraints. For instance,

- `ON DELETE NO ACTION` means that an entity record cannot be deleted as long as some other table still refers to it. This is the default behaviour.
- `ON DELETE CASCADE` means that when a record of an entity is deleted, then so are all entries in tables that refer to it.

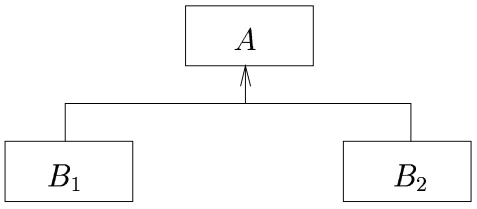
You can find out about ON UPDATE in the PostgreSQL manual or the textbook.

Having a REFERENCES constraint on a single attribute is only suitable for single-attribute foreign keys, not surprisingly. If you have a composite (i.e., multi-attribute) foreign key, then you need to do specify the foreign key separately as in the following abstract example:

```
CREATE TABLE xxx (a integer, b integer, c real, ...,
                  FOREIGN KEY(a,c) REFERENCES yyy ON DELETE CASCADE
                 );
```

where `yyy` is the name of the table into which the foreign key makes reference. In the example as shown, the attributes in table `yyy` are `a` and `c`. But different attribute names can be used, by having something like `FOREIGN KEY (a,c) REFERENCES yyy(d,e) ....`
A separate FOREIGN KEY constraint can also be used for a simple (single-attribute) key, as an alternative to using a constraint within the declaration of that attribute (column).

22. Generalisation Hierarchies. NOTE: This section mixes in design considerations which should ideally already have been sorted out before you come to table creation. However, I leave the material here for convenience of exposition. [John Barnden.]
Consider the following (generic) situation with one supertype and two subtypes:

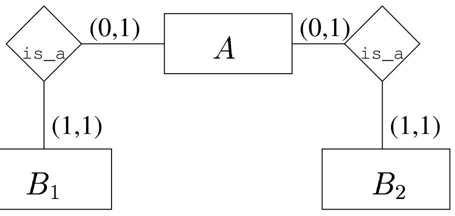


There are three possibilities for translating such a situation into tables:

(a) Drop B_1 and B_2 and only represent A as a table.

(b) Drop A and represent B_1 and B_2 as tables.

(c) Represent each entity as a separate table, getting three tables. From a modelling point of view, this amounts to explicit "`is_a`" relationships between subtypes and supertypes (read this as "every instance of B_1 *is a* instance of A):



Let's discuss the pros and cons of each solution:

(a) The classification is indicated in the table for A by two "`BOOLEAN`" attributes which have value "`TRUE`" if the current instance belongs to the corresponding subtype. If the classification is exclusive (no overlap) then a single attribute will suffice.

This solution is problematic if instances of subtypes have considerably differing attribute sets because this will necessitate entering "`NULL`" for those attributes that do not apply for a given instance. It is also problematic if one subtype takes part in a relationship but others don't. Again, null values may then have to be used.

(b) Dropping the supertype requires that the coverage by subtypes is total (exhasutive). Furthermore, we should also have that the subtypes do not overlap (are "disjoint", in other words, that the classification is exclusive); otherwise we will have at least partially repeated entries in different tables, giving us a dangerous form of redundancy.

From the querying point of view, this solution could also be problematic if it is common to ask questions that apply to all instances of A. In this case, the subtypes must each be queried independently and the results combined with "`UNION`". It is then important that no subtype be overlooked.

(c) Having tables for both supertype and subtypes is the most general solution. It can always be adopted and does not rely on any special assumptions about the classification. However, it is also the costly, both in number of tables (obviously) and in querying time for certain types of query: we may have to do lot of joining tables together when we query the system.

23. More on Database Constraints. We have already bumped into a number of ways of constraining what is accepted as a valid state of the database. Let's summarise this and complete the picture:

Column constraints The values that can legally appear under in single column can be constrained in various ways, notably:

(a) By assigning a type to the column. (This information is in fact required when creating a table.)

(b) By requiring the presence of a value: "`NOT NULL`" as in examples above.

(c) By requiring each value in a column to be used only once in it. This is done by including "`UNIQUE`" within the column declaration (after the type and along with other constraints if any).

NB: If a column is declared as the PRIMARY KEY then this implies uniqueness so no separate uniqueness constraint is required.

(d) By formulating a specific "`CHECK`" condition on the possible values. The syntax for this is illustrated in the following example:

```
CREATE TABLE staff(salary DECIMAL(10,2) NOT NULL
                                        CHECK(salary >= 0),
                  ...
                 );
```

A REFERENCES constraint included in a column declaration as above is also a form of column constraint.

Table constraints These are largely for specifying constraints that are not localized to a single attribute (column), although they can also be used for constraining single attributes. Table constraints are separate from the column declarations. They are usually put after the column declarations but can be mixed in with them. They include the following forms:

(a) A "`PRIMARY KEY`" declaration, as in an example above.

(b) A "`FOREIGN KEY ...  REFERENCES`" constraint, as in an example above.

(c) A "`UNIQUE`" constraint, e.g. "`UNIQUE (a,b,c)`". It means that it is not allowed for two records to agree on all attributes mentioned.

(Again, the constraint is implied by a "`PRIMARY KEY`" declaration.)

(d) "CHECK" constraints relating the values of two or more columns, as in:

```
CREATE TABLE borrowings(rental_start DATE,
                        rental_end   DATE,
                        ...
                        CHECK(rental_start <= rental_end)
                       );
```

Enterprise constraints These span one or several tables in a given database. These allow the database administrator (or designer) to run a whole query whenever a certain action is performed on the data (such as an insertion of a new record). The syntax is illustrated in the following example:

```
CREATE ASSERTION conflict_free
       CHECK( 1 >= (SELECT MAX(lectures_per_hour)
                    FROM (SELECT COUNT(*) AS lectures_per_hour
                          FROM timetable
                          GROUP BY lecturer, timetable_slot)));
```

PostgreSQL does not support enterprise constraints.

Another way to enforce enterprise constraints is to check data as and when it is entered into the database (or when records are updated or deleted) in the *application program* rather than relying on the DBMS to perform the check. Although this approach has the disadvantage that the task of keeping the data consistent is being split between two systems, it allows the designer to formulate conditions which require sophisticated processing and which are not easily expressed in SQL itself.

Caution In general, you should keep in mind that constraints are a double-edged sword; on the one hand, they help us to keep the data in the database meaningful, but on the other hand, they may stop us from adapting the database design to a changing world.

24. Miscellanea: A bug in PostgreSQL. The database "`fundamentals`", which we have used for exercises and examples throughout the course, was created by me and therefore I [Achim Jung] am the owner. Unfortunately, this does not stop anyone from creating tables in my database. Worse, I can not remove such tables because they will not belong to me.
Therefore, if you want to experiment with SQL `CREATE` statements, then **please do so in your own user space**. By default, `psql` will place you in your own database upon login anyway, so there is no excuse for inadvertently creating tables in "`fundamentals`".

EXERCISES

(all created by John Barnden)

Exercise 1

Draw a specialization/generalization hierarchy of entity types that might be in a database used by one of the following, depending on your student ID number:

0: A police force to help it solve crimes.

1: A large farming business.

2: NASA, to help it plan and record projects, voyages, or whatever.

Make choice N from the above if your ID number equals N mod 3 (i.e., your ID number divided by 3 leaves remainder N). I'm imposing this rule to ensure that the numbers of students tackling each example are roughly equal. However, if you've got a strong desire to do a different example, feel free to exchange with someone else if you can, stating in your answer the ID of the student with whom you exchanged.

Whichever of the above you do, provide an ERD fragment in at least TWO of the following notations:

- these notes' diagrammatic notation, complete with all details about the types of link
- the diagrammatic notation used in lectures
- the diagrammatic notation used in the textbook.

I would expect at least TEN entity types. In the police-force case, you could consider including different types of crime, different types of people, and different types of weapon, for example. Make sure that the hierarchy is at least three levels of entity types deep.

Don't show attributes or include relationships other than the subtype/supertype ones. (And don't worry about showing strength or weakness of entity types.)

Feel free to (but don't feel pressured to) make your own additions or other modifications to a notation if you think your changes are useful, but explain precisely what your additional devices mean and (briefly) why they are desirable.

Exercise 2

In your own default database (the one you are automatically in when you enter PostgreSQL), formulate and execute SQL CREATE statements for FIVE of the entity types you provided for Exercise 1. Show your CREATE statements in full and provide evidence that your creations worked, e.g. by using `\d` commands.

Make sure that at least two of the types are subtypes and that at least one is a supertype for at least one of *those* subtypes.

Include a few attributes of your own invention for each type. Also include any DEFAULT, NOT NULL, UNIQUE, CHECK, PRIMARY KEY, FOREIGN KEY and REFERENCES declarations/constraints that may be appropriate.

In the case of any DEFAULT, NOT NULL, UNIQUE and CHECK constraints, say why you are including them.

Exercise 3

Jupiter's moon Europa has an ice covering thought to be up to tens of kilometres thick. Under it there's thought to be a salt-water ocean. There's been a lot of speculation about whether there may be life in that ocean, and NASA is planning space probes to start investigating the matter.

Actually, my own secret sources inform me that there is indeed an advanced civilization living in the ocean. They call themselves the Snaic-Itilop. My undercover exobiology research team has determined that the Snaic-Itilop don't just have two sexes, but five, called in their language Yrot, Ruobal, Larebil, Neerg and Piku. This makes soap operas, sexual politics and dating agencies particularly interesting in Europa. If that's not enough, irrespective of their gender, the Snaic-Itilop come in three types: those with three buttocks, those with four, and those with seven. (So a member of any of these three types can be of any of the five genders.) These gender and anatomical factors complicate all areas of life in Europa, not least the provision of public lavatories. A further discovery about the Snaic-Itilop is that they can have any number of tentacles, and each one can have any number of colours randomly splodged on them. These colourings are economically and socially important, for instance in the Europan fashion industry.

The current ruler of Europa, namely Noremac the Smooth, wishes to construct a database recording details of all the Snaic-Itilop. This database is in particular to do a good job of representing the different sorts of individual, bearing in mind that, while all types of individual have features in common (e.g., a Panoceanic Identity Number, an age, a weight, a jelly/gristle ratio), different types have some special features. For instance, gender affects what internal and external organs an individual has. Gender and buttock-count affects abilities. For instance, only the Yrot and the Ruobal can bear children. The number of buttocks affects, for reasons still mysterious, what sensory organs an individual has. E.g., only the four-buttocked have vision. The database is going to have to contain complicated information about the sensory characteristics of individuals, such as, in the case of vision, the degree of acuity, short-sightnedness and astigmatism, and, in the case of hearing, the degree of acuity at high, medium and low frequencies and the accuracy of echolocation.

The database also needs to record who is the "mother" of whom and who is a "father" of whom. A child only has exactly one biological mother (an Yrot or a Ruobal), but has zero, one or two biological fathers, who must each be a Larebil or Neerg. The Piku are sterile. As on Earth, a mother or father can have any number of children.[4]

You are a member of Noremac the Smooth's database development team and are tasked to provide two ERMs suitable for that database, one at a high conceptual level and one at a lower, "logical" level. The former is not to concern itself with the detail of how finally to use tables to implement matters such as specialization/generalization hierarchies, multivalued attributes and M:N relationships. But the logical ERM does need to concern itself with that detail. So in the logical ERM, the entity types are to correspond exactly to the tables that will be used, and the relationships should all be of basic sorts and the attributes should all be single-valued. (However, in moving from the high conceptual to the logical level don't worry about "normalization,"an important matter that we'll be coming onto soon.)

If the description above leaves something vague or ambiguous, either make sensible assumptions or ask a demonstrator for their view of what should be done. (Attempting to contact Noremac the Smooth himself involves the danger of being bored to death by PR-speak.) In either case, state clearly how you resolved the problem. This matter reflects consultation you might have to do if you were really designing a database on Earth.

For each ERM, you should state in English what the entity types are, what their attributes are, what data types the attributes have (and whether they are multivalued), what the PKs are, what the relationships are and what characteristics they have. You should also provide an ERD (or set of related ERDs), using any style from the lectures, handouts or the textbook. As in Exercise 1, you can make creative but well-conceived and well-explained modifications to a notation. You don't need to include in an ERD all the attributes that are in the corresponding ERM, but do show the more interesting ones such as PKs and multivalued ones if any.

[4]One of the paragraphs in this Exercise up to this point is true.