Chapter 1

Exercise: Dennis does Exams

- Duplication of data *between* files:
  - wastes space
  - can lead to inconsistencies if, when a change must be made to student names, degree titles, course codes and choices of courses, any of of the following fails to happen: Dennis fails to update his own file but tells Danny to update his; Dennis forgets to tell Danny; Dennis types something different in his email from the change he makes to his file; the email link fails; Danny doesn't read his email; Danny forgets to make the change requested by the email; Danny types something different into his file from what is in the email
  - even if all the above are successful, there is obviously a time-lag during which the files are out of step anyway
  - there is further duplication in that Danny has Dennis' file, a file with asterisks and a neat file with exam codes: he needs to remember to either update the first and run the programs again in order to propagate the changes to the other files, or he must alter all 3 files manually (with the risk that he forgets to change one of them). (Manually altering the third file may be error prone because he needs to find student numbers and exam codes for courses from somewhere: he may again make errors when he looks these up and copies them over.)
  - a similar story to the above can be told about the further duplication of sending Helplesston Hall a paper version of the file: Danny must presumably 'phone through or post changes; all of the above problems may thus recur in this situation too.

- Duplication of data *within* the files: [once we have finished chapter 4 you will be able to use the terminology of "normal forms" in the following; but, even without that terminology, you should have spotted the following:] in Danny's file that has the exam codes a student's name and degree are repeated many times. If a female student gets married and changes her name, or if a student switches to a different degree, Danny needs to make multiple updates: there is a risk that he misses one or that he mistypes one of them.

- Lack of data independence [you don't know this term yet, but that shouldn't have stopped you from spotting the following:]

  – the programs must have the format of the files 'built into' them, which makes the system hard to maintain (a lot of work to add a field, for example, as many programs will need to be modified/recompiled/re-tested)

  – it leads to the need to have Danny's largely pointless first Pascal program which solely changes file formats.

- System cannot satisfy other needs: the level of the system is very 'operational'; we cannot, for example, extract trends about which courses have been growing in popularity (not least because there is no place for past years' data in the system).

- Other:

  – data standards clearly don't exist, e.g. different symbols used to separate data fields; Dennis identifies students by their name, Danny uses their number; Dennis separates names by a comma, Danny doesn't

  – poor data security due to use of email and duplication of data

  – no data validation done because all changes are made by text editors

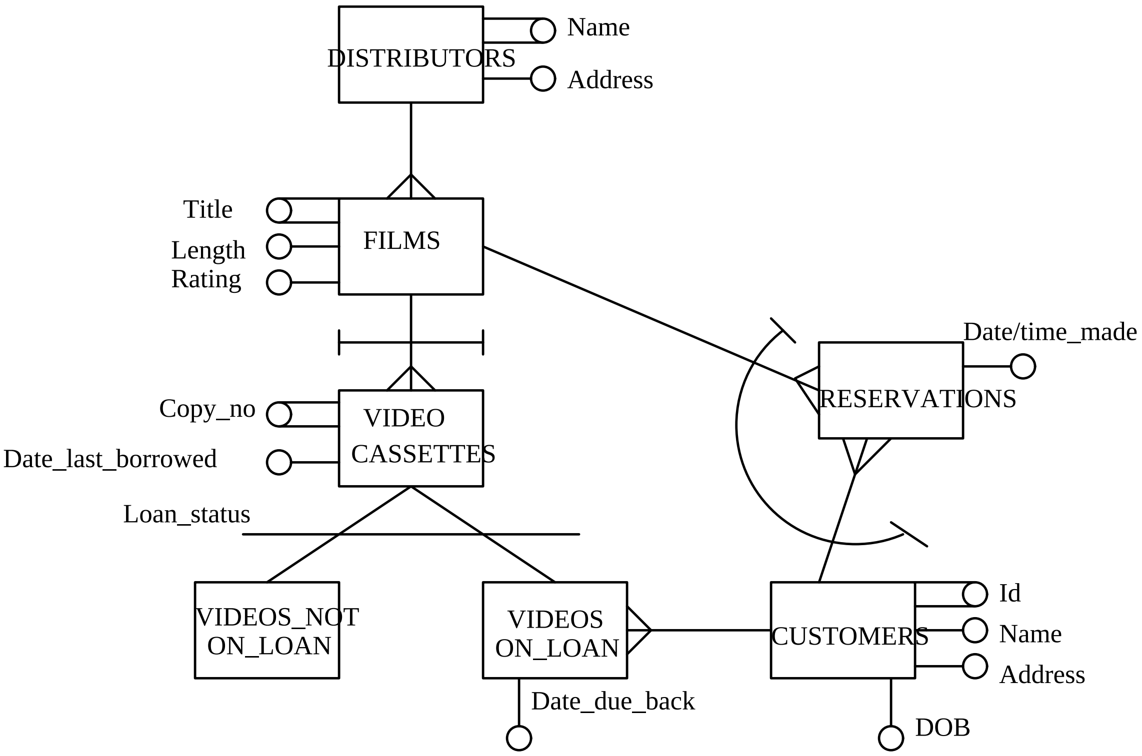
  – ... anything else that's sensible would also get some credit.

Chapter 3

Exercises B, C, D, E and F: Videos, Copiers, Buses, Nappies and Lard Sandwiches

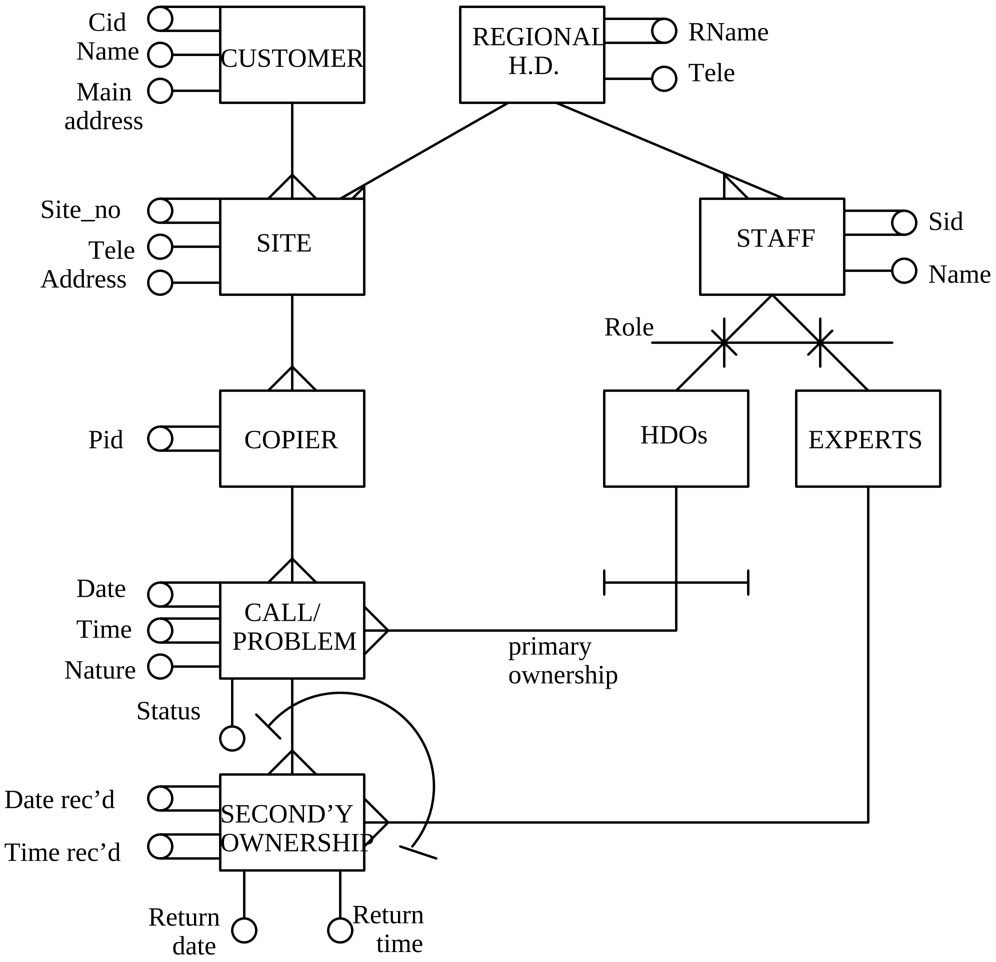
Note: Multiple answers are possible (e.g. what I think is redundant may not be what you think is redundant). I take this into account in the marking provided you *explain* what you're doing so that I can see where your reasoning diverges from mine.

If you don't understand why my diagrams below diverge from yours, visit me in my surgery hour.

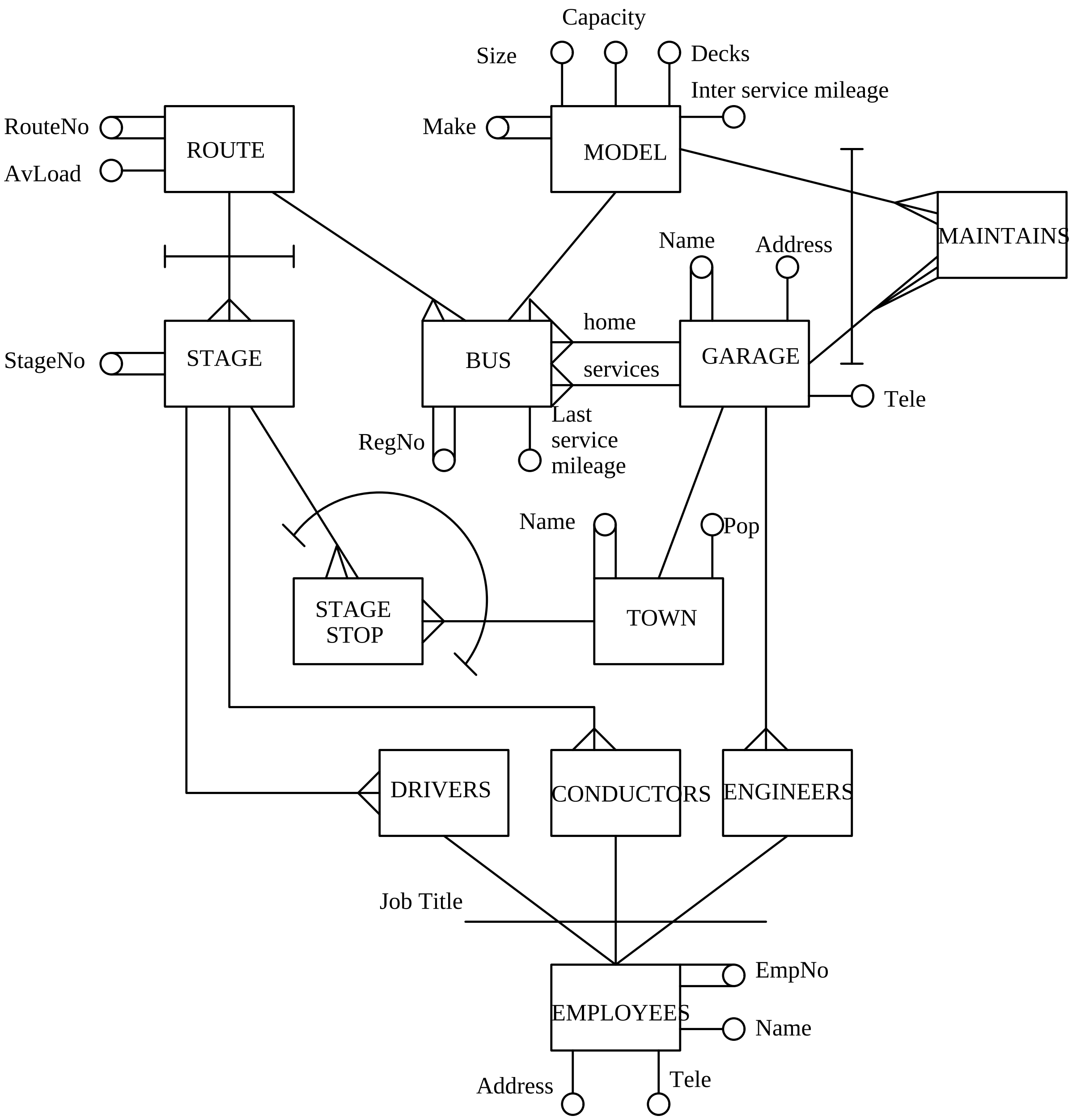


Assumptions: distributors can be identified by their name.

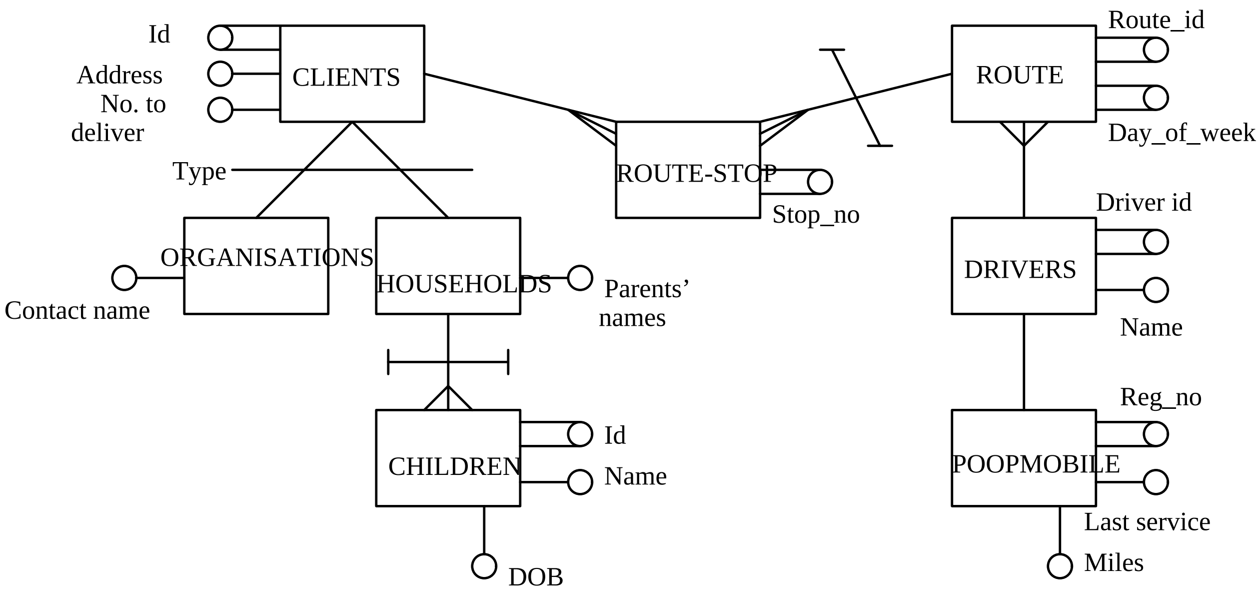
Other possible answers: if you assume that distributors have no properties other than names, then they are a redundant entity type and can become a property of films.



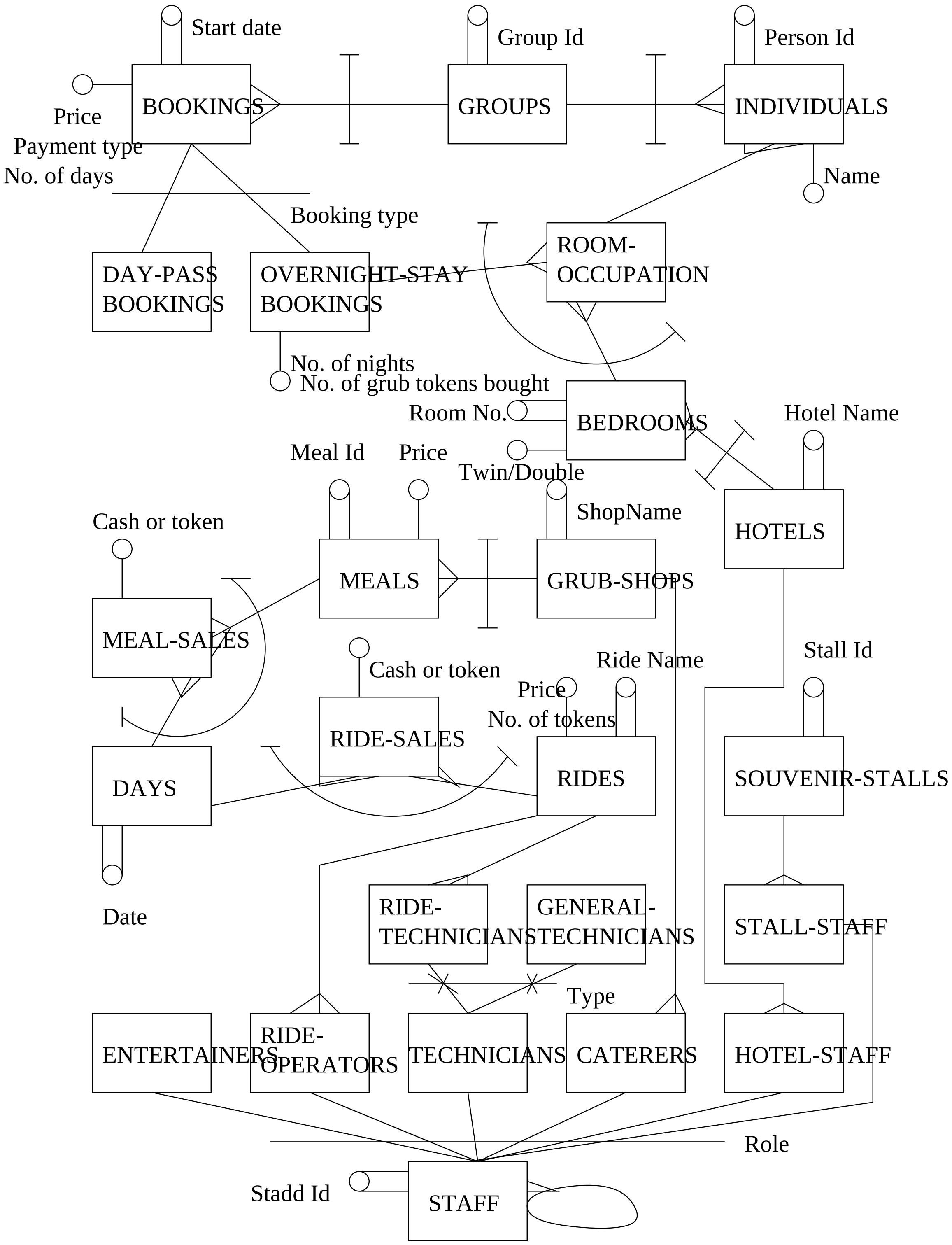
Assumptions: Only one photcopier per call. A variety of assumptions is possible about how to identify a call: I chose date, time and HDO's Sid. Other possibilities include date, time and Pid, or date, time, Sid and Pid (perhaps). Another assumption is that calls are always processed by their originating site's regional help desk, thus there is no need to have a relationship directly from call to help desk.



Assumptions: Towns and garages cannot be merged because some towns have no garage. A town can appear as a stop on more than one route. When a bus is linked to a garage by a 'services' relationship, then that bus's model is linked to at least that garage via the maintains entity type. (This does not make either redundant.)



Assumptions: I assume that two of the statements in the question, viz that drivers usually cover only 1 route per day and that clients on multiple routes usually have those visits on different days, will be handled by suitable applications code (i.e. these are not reflected in the diagram; they are to enforced by database programs). Drivers have ids and names. The children in a household are numbered (e.g. the kids in household 3 would be identified as $< 3, 1 >$, $< 3, 2 >$, etc.). Route stops are numbered, giving identifiers of the form <Tues, Route-id-1, stop-1>, <Tues, Route-id-1, stop-2>, etc.: this gives the ordering of stops that we need. (Thus, note that there is no need to use client number to do this identification of stops.) (And note that day of the week does not need to be part of the identification of routes if I were to assume that route ids were unique.) The subtypes of clients are exclusive.



Assumptions include:

1. Groups identified by GroupId.
2. GroupId and StartDate identify Bookings (Could invent a BookingNo).
3. HotelName identifies Hotels.
4. Bedrooms identified by RoomNo and an identification dependence on Hotels.
5. Individuals identified by PersonId and an identification dependence on Group.
6. No-of-grub-tokens can be zero in the case of bookings that aren't of the special offer kind.
7. Each meal has a possibly different price but costs just one Grub Token.
8. ShopName identifies Grub Shops.
9. RideName identifies Rides.
10. MealId identifies Meals.
11. StaffIf identifies staff.
12. StallId identifies souvenir stalls.

Integrity constraints include:

1. That groups contain at least two people, at least one of whom is an adult.
2. Room-occupation to Individuals is a one-to-two relationship.
3. No of night's stay is within one either way of no of day's stay.
4. Booking price is a function of the number of members of the group and the number of Grub Tokens (psso zero).
5. The allocation of rooms is related to the no. of members of the group.
6. Ride price is related to the number of tokens the ride costs.

Chapter 4

Exercises A, B, C, E and F: Videos, Copiers, Buses, Nappies, the Queen, Aubrey's French Fancies, Victorian Values and DJs

Exercise A

These answers are, as requested in the question, based on the *model* answers to the exercises from the end of Chapter 3.

Video Hire Shop

DISTRIBUTORS(**Name:** Address)
FILMS(**Title:** Length, Rating, DistribName)
VIDEO_CASSETTES(**FilmTitle, CopyNo:** Date_last_borrowed, Loan_status)
VIDEOS_ON_LOAN(**FilmTitle, CopyNo:** CustId, Date_due_back)
CUSTOMERS(**CustId:** CName, CAddress, DOB)
RESERVATIONS(**FilmTitle, CustId:** Date/time_made)

I have assumed that VIDEOS_NOT_ON_LOAN is redundant. All are in 3NF.

The full question, in this case, was to also list the referential integrity constraints. These are as follows:

1. FILMS.DistribName is referentially dependent on DISTRIBUTORS.Name
2. VIDEO_CASSETTES.FilmTitle is ref dep on FILMS.Title
3. VIDEOS_ON_LOAN.FilmTitle and VIDEOS_ON_LOAN.CopyNo are together ref dep on VIDEO_CASSETTES.FilmTitle and VIDEO_CASSETTES.CopyNo
4. VIDEOS_ON_LOAN.CustId is ref dep on CUSTOMERS.CustId
5. RESERVATIONS.FilmTitle is ref dep on FILMS.Title
6. RESERVATIONS.CustId is ref dep on CUSTOMERS.CustId

Help Desks

CUSTOMER(**Cid:** Name, Main_address)
REGIONAL_HD(**RName:** Tele)
SITE(**Site_no:** Address, Tele, Cid, RName)
COPIER(**Pid:** Site_no)
STAFF(**Sid:** Name, RName)
STAFF_ROLE(**Sid, Role:**)
CALL(**Date, Time, HDOSid:** Nature, Status, Pid)
SECONDY_OWNERSHIP(**Date_recd, Time_recd, CallDate, CallTime, CallHDOSid, ExpertSid:** ReturnDate, ReturnTime)

I have assumed that relations for HDOS and EXPERTS are redundant. All are in 3NF.

Buses

ROUTE(**RouteNo:** AvLoad)
MODEL(**Make:** Size, Capacity, Decks, InterServiceMileage)
GARAGE(**GName:** Address, Tele, TownName)
MAINTAINS(**GName, Make:**)
BUS(**RegNo:** LastServiceMileage, RouteNo, Make, HomeGName, ServiceGName)
TOWN(**TownName:** Pop)
STAGE_STOP(**RouteNo, StageNo, TownName:**)
EMPLOYEES(**EmpNo:** EName, Address, Tele, JobTitle)
DRIVERS(**EmpNo:** RouteNo, StageNo)
CONDUCTORS(**EmpNo:** RouteNo, StageNo)
ENGINEERS(**EmpNo:** GName)

I have assumed that a relation for STAGE is redundant (because STAGE_STOP captures the information already). Note that in capturing the relationship between TOWN and GARAGE I had to decide whether TownName should be a foreign key in GARAGE or whether GName should be a foreign key in TOWN. I chose the latter. If I had chosen the former, then I would need to use nulls in TOWNS.GName to handle towns that don't have garages. All are in 3NF.

Nappies

CLIENTS(**Client_id:** Address, No_to_deliver, Type)
ORGANISATIONS(**Client_id:** Contatct_name)
HOUSEHOLDS(**Client_id:** Parents'_names)
CHILDREN(**Client_id, Child_id:** Name, DOB)
ROUTES(**Route_id, Day_of_week:** Driver_id)
ROUTE_STOP(**Route_id, Day_of_week, Stop_no: Client_id**)
DRIVERS(**Driver_id:** Name)
POOPMOBILES(**RegNo:** Last_service, Miles, Driver_id)

(Note: either Driver_id must appear in POOPMOBILES or RegNo must appear in DRIVERS, but preferably not both.)

(In principle, DRIVERS and POOPMOBILES can be merged into a single relation. By not merging I get a more robust design. I can't think of any good reasons for merging them (efficiency arguments are out of place at this stage).)

Exercise B

(a) Your Majesty, with your present design you are unable to store museum names and addresses in an independent fashion. You can only store this info about a museum if you have loaned something to it. Thus it is impossible to insert a new museum, e.g. museum no. 12, the Fitzwilliam in Cambridge, until you have loaned something to it because to try to insert these details into the LOANS relation as it stands would require Item_no to take on a null value, which, as part of the primary key, it cannot. Equally, if you have loaned only one item to museum no. 3, the V&A in London, and then you recall that item, deletion of the tuple from LOANS will delete all mention of the V&A.

If a museum to which you have made multiple loans changes its name or address e.g. if the National Portrait Gallery is privatised and becomes the

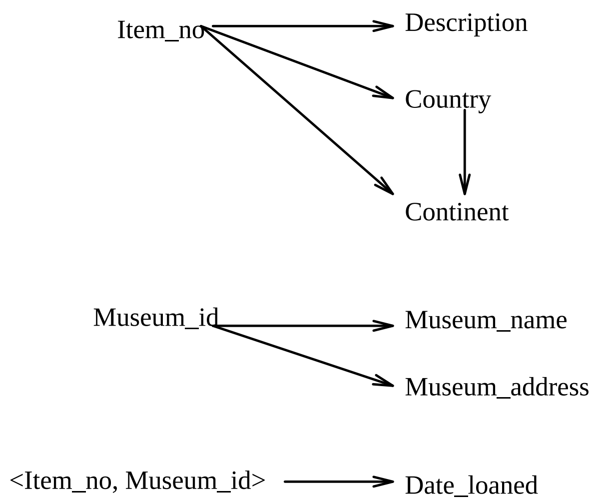
Murdoch Snapshot Emporium, then for every tuple in LOANS that records a loan to that museum, you will need to change the museum details. This is error prone since you might miss one of the tuples or mistype one of the changes.

The self same problems arise in connection with recording information about countries and their continents. Scotland, Europe cannot be recorded until the Scots make a gift to you. If the French have given you only one gift, and then they ask for it back, deletion of the tuple will lose all record of France. And if Australia changes its name to 'That-Stupid-Country-That-Rolf-Harris-Comes-From', then multiple changes may be needed. (These problems are less acute than those concerning museums.)

Is that clear, you scabby old bat?

(b) The relations are already in 1NF[1]

The functional dependencies are:



ITEMS is already in 2NF (it can't be otherwise because it does not have a composite key). But LOANS is not in 2NF because Museum_name and Museum_address are not functionally dependent on the whole of the key of LOANS; they are dependent only on Museum_id. I factor these attributes out of LOANS to give the following 2NF relations:

ITEMS(**Item_no:** Description, Country, Continent)
LOANS(**Item_no, Museum_id:** Date_loaned)
MUSEUMS(**Museum_id:** Museum_name, Museum_address)

[1] ...unless you assume the same item can be loaned to the same museum on multiple dates, in which case Date_loaned is a repeating group. I ignore this possibility here.

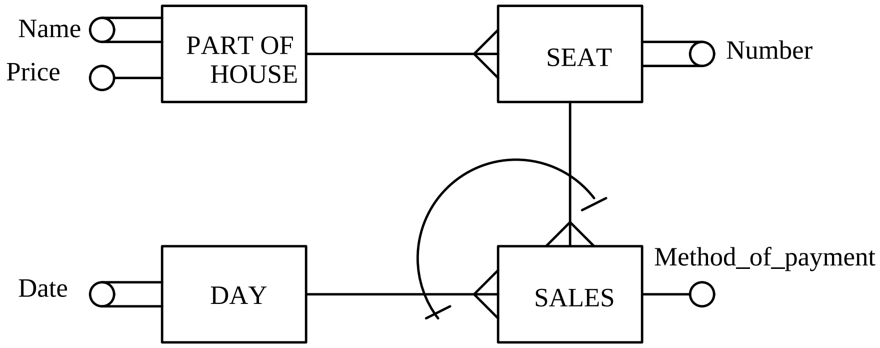
LOANS and MUSEUMS are in 3NF (there are no transitive dependencies). But ITEMS is not in 3NF because Continent is only transitively dependent on Item_no. I factor these attributes out of ITEMS to give the following 3NF relations:

ITEMS(**Item_no:** Description, Country)
COUNTRIES(**Country:** Continent)
LOANS(**Item_no, Museum_id:** Date_loaned)
MUSEUMS(**Museum_id:** Museum_name, Museum_address)

(Note: your answer might well be more complicated if you allowed for the fact that Istanbul spans Europe and Asia —but no one has ever considered that!)

Exercise C

(i) Entity-relationship model:



(Note: you could remove DAY from this and make Date an identification key property of SALES.)

(ii) Logical model:

PART_OF_HOUSE(**Name:** Price)
SEAT(**Seat_number:** Part_of_house_Name)
SALES(**Date, SeatNumber:** Method_of_payment)

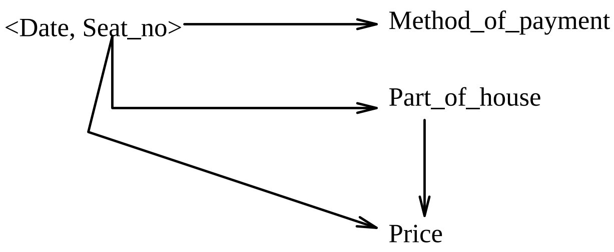
(If you didn't remove DAY from the ER diagram, then you should certainly have removed it by now, as I have done.)

(iii) Normalisation.

Aubrey's relation is not in 1NF: there is a repeating group of seats sold on each date. I will factor out the repeating group:

DAY(**Date:**) (redundant)
SALES(**Date, Seat_no:** Part_of_house, Price, Method_of_payment)

Now I show the functional dependencies:



The SALES relation is not in 2NF because Part_of_house and Price are dependent on only part of the key, namely Seat_no. I factor these out:

SALES(**Date, Seat_no:** Method_of_payment)
SEATS(**Seat_no:** Part_of_house, Price)

SALES is in 3NF (no transitive dependencies), but SEATS isn't (Price is determined by Part_of_house). I factor these out:

SALES(**Date, Seat_no:** Method_of_payment)
SEATS(**Seat_no:** Part_of_house)
HOUSE(**Part_of_house:** Price)

(iv) Briefly. . . The notion of normal forms can be used in two ways within database design. It can be used as an analysis method in its own right, as in (iii). The advantages of this are its apparent 'speed' and simplicity. The disadvantages are that it gives you no clues on how to get the original relations and functional dependencies.

Alternatively, it can be used as a validation technique within a more top-down analysis method, as in (ii). These top-down methods look like they are much more trouble, but many advantages accrue in taking the more measured approach. In particular, you get two chances to validate what you are doing: first, you can use the client-friendly ER diagrams for face-to-face validation; secondly, you use a check for normal forms as another check: problems found at this stage usually reveal misconceptions on the part of the analyst about the 'world' being modelled.

Exercise E

(i)

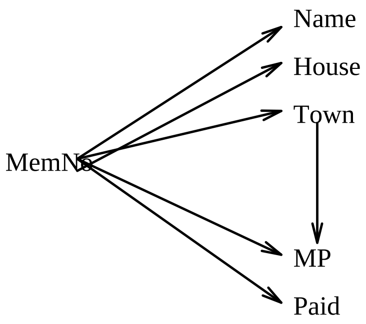
(a) You cannot record a town's MP unless a member resides in that town. You cannot record a campaign's name unless someone has made a donation to it.

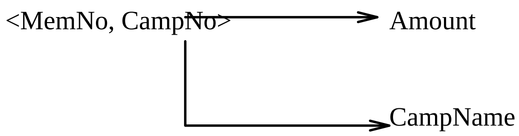
To change a MP for a town may require multiple changes. To change a campaign name may require multiple changes.

(b) The relation that corresponds to the file is not in 1NF because there is a repeating group. Decomposing to get 1NF gives:

MEMBERS(**MemNo:** Name, House, Town, MP, Paid)
DONATIONS(**MemNo, CampNo:** CampName, Amount)

The functional dependencies are:





DONATIONS is not in 2NF as CampName is dependent only on part of the key. Decomposing it gives us:

MEMBERS(**MemNo:** Name, House, Town, MP, Paid)
CAMPAIGN(**CampNo:** CampName)
DONATIONS(**MemNo, CampNo:** Amount)

MEMBERS is not in 3NF as MP is only transitively dependent on MemNo. Decomposing it gives us:

MEMBERS(**MemNo:** Name, House, Town, Paid)
TOWNS(**Town:** MP)
CAMPAIGN(**CampNo:** CampName)
DONATIONS(**MemNo, CampNo:** Amount)

These are now in 3NF.

(Note: I made a simplifying assumption. If a person makes more than one donation to the same campaign, I assumed those donations could be summed together and stored as a single tuple in the DONATIONS relation. Had I not made this assumption, i.e. if I had wanted to keep these multiple donations distinct, then I would need to invent a new attribute, DonNo, to keep track of them:

DONATIONS(**MemNo, CampNo, DonNo:** Amount))

(ii)

- There is duplication *between* files:
  - does Alfred remember to tell Betty to make changes
  - does Betty remember to make them

  A failure in this process leads to inconsistencies.
- There is a lack of data independence (this terminology is introduced in the next chapter, but you should still have been able to spot the following points):
  - programs 'know' the format of the data, which makes the system inflexible, i.e. it takes a lot of work to make changes, e.g. if Betty needs a new data field, Alfred's program may need to be altered throughout
  - as a consequence Betty has to carry out a largely pointless editing task —solely to resolve format incompatibilities between the two programs
- The system probably does not meet the needs of the organisation:
  - the level of the data is very 'operational': you cannot easily extract trends about the types of campaigns people prefer to support, etc.
- Other:

- the duplication of data files may lead to the organisation being more open to infiltration by pinko-commie-bastards and other extremists who happen to believe that a combination of market forces and prissy Victorian morals is the *cause* of the present mess, not the solution to it
- there is no data validation being done because all data entry is done using text editors
- data standards are not enforced; in particular, Betty's representation of names differs from Alfred's; a ramification of this is that Betty must invent people's titles; what will she invent for someone called 'Hilary'?
- program standards are not enforced, giving them the headache of maintaining programs in multiple languages
- etc.!

So, the future doesn't look too bright for Victorian Values. What a shame. (Hang the lot of 'em, that's what I say.)

Exercise F

(i)

(a) SUPERTYPE(**IdProp:** Prop1)
CLASSIFICATION(**IdProp, ClassProp:**)
SUBTYPE1(**IdProp:** Prop2)
SUBTYPE2(**IdProp:** Prop3)

Subtypes have the same key as their supertype. In the case of non-exclusive subtypes, to show which tuples in the supertype relation belong to which subtypes, we need a new relation; making ClassProp an attribute of SUPERTYPE wouldn't do; it would need to be multi-valued to show multiple membership, and this isn't allowed.

(b) ETYPE(**IdProp:** Prop, IdProp′)

Import the primary key from the one-end of a one-to-many relationship into the many-end; gives us two uses of IdProp here; the second is renamed to avoid name conflict.

(c) ETYPE1(**IdProp1:** Prop1)
ETYPE2(**IdProp1, IdProp2:** Prop2)
ETYPE3(**IdProp1, IdProp2, IdProp3:** Prop3)

Import the primary key and, due to identification dependence, make it part of importer's primary key.

(ii)

STATION(**StatName:** StatAddress)
RECORDING(**RecId:** RecName, Type)
PLAYED-LIST(**StatName, RecId:** No-of-times)
SONG(**SongId:** SongName, Lyricist, Composer)
VERSION(**RecId:** Artiste, SongId)
REMIX(**RecId:**)
SAMPLE(**VersionRecId, RemixRecId:** Mixer)

Referential integrity constraints:

- PLAYED-LIST.StatName on STATION.StatName
- PLAYED-LIST.RecId on RECORDING.RecId
- VERSION.RecId on RECORDING.RecID
- VERSION.SongID on SONG.SongID
- REMIX.RecId on RECORDING.RecId
- SAMPLE.VersionRecId on VERSION.RecID
- SAMPLE.RemixRecId on REMIX.RecID

(iii)

- VERSION violates first normal form (and, hence, also second normal form and third normal form) because in 1NF attributes must have atomic values but Artiste is now not atomic; it's a set.

  SAMPLE violates second normal form (and, hence, third normal form) because in 2NF attributes must be dependent on the whole key, but if Mixer properly belongs to REMIX it is dependent only on RecId.

- Here's some sample data:

  ```
  Vers1 Remix1: JazzyJazz
  Vers2 Remix1: JazzyJazz
  Vers1 Remix2: BoppyBop
  ```

It is impossible to insert the fact that a new Remix, Remix3, was mixed by JiggyJig, without also knowing which versions of songs he sampled. (To insert into the SAMPLE relation would require nulls in the first attribute, but this attribute is a key attribute.)

If Remix2 did not in fact contain a snatch of Vers1 we would delete the third tuple. But if this was the only remix made by BoppyBop and this is the only sample recorded we lose BoppyBop from the database.

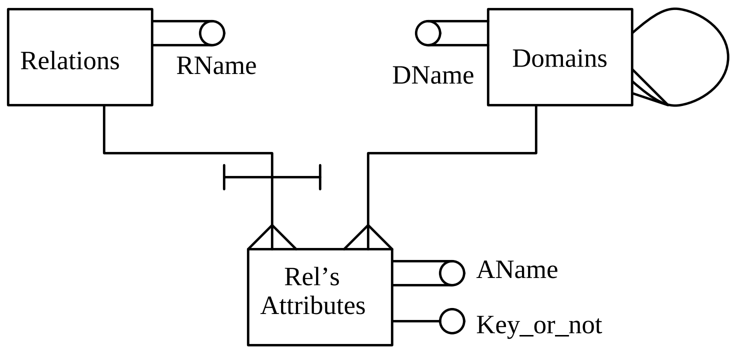
To change JazzyJazz's name to JazzieJazz requires three updates.

- Correcting these problems requires altering the conceptual model and repeating the process of mapping it to a logical data model. (Anyone who talks of normalisation gets no marks: the error is clearly stated to be in the conceptual model. And this is a point made *very* forcefully in lectures.)

Chapter 5

Data Dictionaries

(ii)



(iii)

RELATIONS(**RName:**)
REL_ATTRIBUTES(**RName, AName:** Key_or_not, DName)
DOMAINS(**DName:** Supertype_DName)

(RELATIONS is redundant for the following reason: since every relation must have at least one attribute, one can easily find out what relations there are by projecting RName out of REL_ATTRIBUTES.)

(iv) Referential integrity constraints:

REL_ATTRIBUTES.RName is ref dep on RELATIONS.RName
REL_ATTRIBUTES.DName is ref dep on DOMAINS.DName
DOMAINS.Supertype_DName is ref dep on DOMAINS.DName

Chapter 6

Sunday Soccer

Notes: There are often alternative (equivalent and hence equally correct) answers to those given below. Come to see me if you're having problems.

(A) You are being asked for integrity constraints *other than* entity integrity and referential integrity. Some examples are: there will be some relationship between the scores of the fixtures and the points which a club has; a club cannot play itself; if there is more than one match for a team on a particular day then the matches must be in the same town; etc. (i.e. any other plausible constraints which you can think of).

(B)

1. Get the names of all players that belong to the club 'Sudbury Strikers'.

    T1 := restrict(PLAYERS, Club_name = 'Sudbury Strikers')
    T2 := project(T1, {Player_name})

2. Find out who Billy The Fish plays for.

    T1 := restrict(PLAYERS, Player_name = 'Billy the Fish')
    T2 := project(T1, {Club_name})

3. Get the names of players who do *not* play for Fulchester Utd.

    T1 := restrict(PLAYERS, Club_name /= 'Fulchester Utd')
    T2 := project(PLAYERS, {Player_name})

4. Get the names of players who play for clubs based in Rossdale.

T1 := restrict(CLUBS, Town = 'Rossdale')
T2 := nat-join(T1, PLAYERS)
T3 := project(T2, {Player_name})

5. Get the names of players who do *not* play for clubs based in Rossdale.

T1, T2 and T3 := as in (4) above
T4 := project(PLAYERS, {Player_name})
T5 := difference(T4, T3)

6. Get the towns in which matches were played between 11/09/90 and 03/03/91.

T1 := restrict(FIXTURES, Fixture_date > 11/09/90 and Fixture_date < 03/03/91)
T2 := project(T1, {Home_club_name})
T3 := nat-join(T2, CLUBS, Home_club_name = Club_name)
T4 := project(T3, {Town})

7. Get the dates of all the fixtures that Fulchester Utd. have played.

T1 := restrict(FIXTURES, Home_club_name = 'Fulchester Utd')
T2 := restrict(FIXTURES, Away_club_name = 'Fulchester Utd')
T3 := union(T1, T2)
T4 := project(T3, {Fixture_date})

(A simpler alternative is to project fixture date from restrict(FIXTURES, Home_club_name = 'Ful. Utd' OR Away_club_name = 'Ful. Utd'))

8. Get the names and points of clubs who have won at least one match.

T1 := restrict(FIXTURES, Home_club_goals > Away_club_goals)
T2 := project(T1, {Home_club_name})
T3 := rename(T2, Home_club_name → Club_name)
T4 := restrict(FIXTURES, Away_club_goals > Home_club_goals)
T5 := project(T3, {Away_club_name})
T6 := rename(T5, Away_club_name → Club_name)
T7 := union(T3, T6)
T8 := nat-join(T7, CLUBS)
T9 := project(T8, {Club_name, Points})

9. Get the dates of all fixtures for which the two participating clubs are based in the same town.

T1 := rename(CLUBS, Club_name $\rightarrow$ Club_name1, etc.)
T2 := rename(CLUBS, Club_name $\rightarrow$ Club_name2, etc.)
T3 := nat-join(T1, T2, Town1 = Town2)
T4 := θ_join(T3, FIXTURES, (Club_name1 = Home_club_name and Club_name2 = Away_club_name) or (Club_name1 = Away_club_name and Club_name2 = Home_club_name))
T5 := project(T4, {Fixture_date})

One of the alternative solutions is:

T1 := nat_join(FIXTURES, CLUBS, Home_club_name = Club_name)
T2 := rename(CLUBS, Club_name $\rightarrow$ Club_name2, etc.)
T3 := nat-join(T1, T2, Away_club_name = Club_name2)
T4 := restrict(T3, Town = Town2)
T5 := project(T4, {Fixture_date})

Chapter 7

SQL for Films

A) Notes: There are often alternative (equivalent and hence equally correct) answers to those given below. Come to see me if you're having problems.

```
1. Films directed by Hitchcock
   select * from Films where dir_surname = 'Hitchcock'

2. Films made since 1970
   select * from Films where year_made > 1970

3. The film called A Man For All Seasons
   select * from Films where title = 'A Man For All Seasons'

4. All directors born in the USA
   select * from Directors where dir_bplace like '%USA%'

5. All directors born between 1940 and 1950
   select * from Directors
    where dir_yob >= 1940 and dir_yob <= 1950

6. All directors born in 1904, 1906 and 1908
   select * from Directors
    where dir_yob = 1904 or dir_yob = 1906 or dir_yob = 1908

  [OR: select * from Directors where dir_yob in (1904, 1906, 1908) ]

7. Years in which a film was made
   select distinct year_made from Films

8. All directors' first names
```

```
   select distinct dir_firstname from Directors
```

9. Title of films and corresponding years
```
   select title, year_made from Films
```

10. Films and the actors in them and years made
```
    select Actors.title, Actors,act_surname, Films.year_made
      from Actors, Films
     where Actors.title = Films.title
```

11. Films whose directors were born in England
```
    select Films.title from Directors, Films
     where Directors.dir_bplace like '%ENGLAND%' and
           Directors.dir_surname = Films.dir_surname
```

12. Directors of films made in the 1930s
```
    select distinct Directors.dir_surname, Directors.dir_firstname
      from Films, Directors
     where Films.year_made >= 1930 and Films.year_made < 1940
       and Films.dir_surname = Directors.dir_surname
```

13. Birthplaces of directors who directed films starring Woody Allen
```
    select distinct Directors.dir_bplace
      from Actors, Films, Directors
     where Actors.act_surname = 'Allen' and
           Actors.act_firstname = 'Woody' and
           Actors.title = Films.title and
           Films.dir_surname = Directors.dir_surname
```

14. Titles of films directed by directors born in Japan
```
    select Films.title from Directors, Films
     where Directors.dir_bplace like '%Japan%' and
           Directors.dir_surname = Films.dir_surname
```

15. Actors who appear in films directed by Minnelli
```
    select distinct Actors.act_surname, Actors.act_firstname
      from Films, Actors
     where Films.dir_surname = 'Minnelli' and
           Films.title = Actors.title
```

16. Actors who appear in films they directed themselves
```
    select Actors.act_surname, Actors.act_firstname
      from Films, Actors
```

```
 where Actors.act_surname = Films.dir_surname and
       Actors.title = Films.title
```

[While this is the best that can be done on the schema as it stands, why isn't it quite right? Obviously, your answer to (B) should be rectifying this sort of problem.]

17. Lengths of films made in England during the 1950s

```
select distinct film_length from Films
 where year_made >= 1950 and year_made < 1960 and
       country = 'England'
```

18. Titles of films directed by American-born directors which exceed one-and-a-half hours in length

```
select Films.title from Directors, Films
 where Directors.dir_bplace like '%USA%' and
       Films.film_length > 90 and
       Directors.dir_surname = Films.dir_surname
```

19. Birthplaces of directors associated with films of less than one-and-a-half hour's duration which were made in the USA and do not star Hitchcock

```
select distinct Directors.dir_bplace from Films, Directors
 where Films.film_length < 90 and
       Films.country = 'USA' and
       Films.title not in
        (select title from Actors
          where act_surname = 'Hitchcock') and
      Films.dir_surname = Directors.dir_surname
```

[This gives a two tuple result:

```
     GRAND FORKS, CANADA
     CHICAGO, ILLINOIS, USA
```

If you run it with Mitchum in place of Hitchcock, you should get a single tuple result:

```
     CHICAGO, ILLINOIS, USA
```

Be warned that simpler versions of this query are not right: they fail to recognise that a film may have no actors or that a film can have more than one actor. Again there are ramifications for your answer to (B).]

20. Average film length

```
select avg(film_length) from Films
```

21. Maximum film length and minimum film length
```
select max(film_length), min(film_length) from Films
```

22. Number of films made in each year
```
select year_made, count(*) from Films
 group by year_made
```

23. The oldest film
```
select title from Films
 where year_made in
   (select min(year_made) from Films)
```

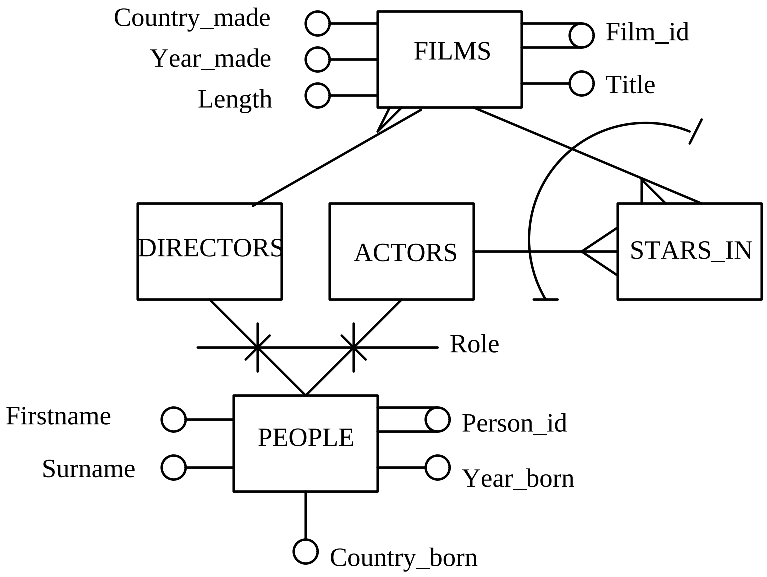
24. The number of films directed by Hitchcock
```
select count(*) from Films
 where dir_surname = 'Hitchcock'
```

25. Film lengths shared by >2 films
```
select film_length from Films
 group by film_length
having count(*) > 2
```

26. Pairs of directors born in the same town
```
select D1.dir_surname, D1.dir_firstname,
       D2.dir_surname, D2.dir_firstname
  from Directors D1, Directors D2
 where D1.dir_bplace = D2.dir_bplace and
       D1.dir_surname < D2.dir_surname
```

```
[The last condition is not strictly necessary but including it
 removes redundant answers:
   DISNEY   DISNEY
   DISNEY   MINNELLI
   MINNELLI DISNEY
   MINNELLI MINNELLI
 is reduced to simply:
   DISNEY   MINNELLI ]
```

B) Here's a possible ER diagram.



Here's the logical data model:

> FILMS(**Film_id:** Title, County_made, Year_made, Length, Director_person_id)
> STARS_IN(**Actor_person_id, Film_id:**)
> PEOPLE(**Person_id:** Firstname, Surname, Country_born, Year_born)
> DIRECTORS(**Director_person_id:**)
> ACTORS(**Actor_person_id:**)
> JOB(**Person_id, Role:**)

I believe that DIRECTORS, ACTORS and JOB are redundant: the same information (namely who acts and who directs) can be found from FILMS and STARS_IN.

Note the following improvements:

1. Since people cannot be identified by their surname alone, or by their firstname and surname, we now have a new artificial unique identifier, Person_id. In the old schema, because people weren't identified properly, the foreign keys didn't work properly: you could make incorrect matches, e.g. two directors sharing the surname 'WOOD' would appear to be related to the same films (incorrectly).

2. Since films get re-made, title is not enough to identify them; we now have a new artificial unique identifier, Film_id.

3. Actors and directors are subtypes of people, and share many properties by virtue of this. The old schema did not capture this at all.

4. Actors can be directors and directors can act, without confusion about who's who.

5. We are now making it very clear that a director can direct many films (although in this simple version we continue to assume that a film has only one director), and an actor can star in many films, and a film can have many actors in it.

6. It is now possible to find films with no actors in them (e.g. cartoons) more easily.

7. The new identifiers are better suited for use as indexes (see Chapter 9).

Chapter 8

Alice's Papers

(i)

(a) The number of copies of each type of newspaper (broadsheet/tabloid) that a deliverer must deliver.

(b) Identifiers of customers who take all the tabloids.

(ii)

(a)

```
select CustId, CustName from CUSTOMERS, SUBSCRIPTIONS, CUST-DELIVERERS
 where SUBSCRIPTIONS.PaperName = 'The Tory Rot'
   and CUST-DELIVERERS.PersonId = 3
   and CUSTOMERS.CustId = SUBSCRIPTIONS.CustId
   and CUSTOMERS.CustId = CUST-DELIVERERS.CustId
```

(Any way of equating the three CustIds is OK.)

(b)

```
select CustId from CUSTOMERS
 where CustId not in
       (select CustId from SUBSCRIPTIONS, NEWSPAPERS
         where Type = 'Tabloid'
           and SUBSCRIPTIONS.PaperName = NEWSPAPERS.PaperName)
```

(Using select 'CustId from SUBSCRIPTIONS, NEWSPAPERS where Type <> 'Tabloid, etc' is wrong!)

(c)

```
select T1.PaperName, T2.PaperName
  from NEWSPAPERS T1, NEWSPAPERS T2
 where T1.PaperName > T2.PaperName
   and T1.Price = T2.Price
```

(iii)

(a) The DBMS does nothing other than associate, in the data dictionary, the view name, TABLOIDS, with the SQL:

```
select * from NEWPSPAPERS
 where Type = 'Tabloid'
```

(b) The view definition is retrieved from the data dictionary and expanded into the query like a macro to give:

```
select * from NEWSPAPERS
 where Type = 'Tabloid'
```

This is executed as normal.

(c) The insertion is not done on the view; it is done to the underlying base table, NEWSPAPERS; it is NEWSPAPERS that gets a new tuple inserted (again thro' a macro expansion process).

(iv) If they both appear, then the user could try to insert a new tuple in which the values s/he supplied for the two attributes differed, thus allowing a view to be updated with a tuple that could never appear in the view. (Similarly with modification.)

Chapter 10

Mandy the Moorhen, Bathbun the Cat and Rats in the Kitchen

Exercise A

(ii) In (a), (b) and (c), replace "tuples" by "buckets" if you assume that the indexes are non-dense. For (d) it does not matter whether they are dense or not. For the solution proposed in (e) to work, the index must be dense.

(a) Use the index on A to get A = "32", then (given that the file is ordered) search backwards for all the tuples you need.

(b) Use the index on B to access the tuples where B = "12". Only these need to be checked for C > "37".

(c) Take the intersection of the index entries for B = "12" and for D = "97". These point to the required tuples.

(d) Nested loops are unavoidable here but having S as the outer loop reduces the number of disk accesses: $8 + (8 \times 12)$ compared with $12 + (12 \times 8)$.

(e) There is no need to access the file: simply print the printable bits of the index (if dense).

Exercise C

(i) Query tree:

```
project {SPONSORS.SName, ADOPTION.Cash}
                    |
                 restrict
    BIRDS.Bird_id = ADOPTION.Bird_id ∧
SPONSORS.Sponsor_id = ADOPTION.Sponsor_id ∧
       BIRDS.BName = "Stevie" ∧
       BIRDS.Species = "Swan"
                    |
                 product
                    |
           product     BIRDS
              |
     SPONSORS   ADOPTION
```

(ii)

(1) Break up complex restricts.

Using

$$\text{restrict}(R,\ \theta_1 \wedge \theta_2) \equiv \text{restrict}(\text{restrict}(R,\ \theta_2),\ \theta_1)$$

```
project {SPONSORS.SName, ADOPTION.Cash}
                    |
restrict BIRDS.Bird_id = ADOPTION.Bird_id
                    |
restrict SPONSORS.Sponsor_id = ADOPTION.Sponsor_id
                    |
restrict BIRDS.BName = "Stevie"
                    |
restrict BIRDS.Species = "Swan"
                    |
                 product
                    |
           product     BIRDS
              |
     SPONSORS   ADOPTION
```

(2) Move restrict down the tree as far as possible.

Using

restrict(nat-join(R, S), θ) $\equiv$ nat-join(restrict(R, θ_1), restrict(S, θ_2))

as long as θ can be rewritten as $\theta_1 \wedge \theta_2$ where θ_1 involves only attributes of R and θ_2 involves only attributes of S

(But here the nat-join is just a product.)

```
                  project {SPONSORS.SName, ADOPTION.Cash}
                                   |
                 restrict BIRDS.Bird_id = ADOPTION.Bird_id
                                   |
                                product
                                   |
                    +--------------+---------------+
                    |                              |
                 restrict          restrict BIRDS.BName = "Stevie"
SPONSORS.Sponsor_id = ADOPTION.Sponsor_id          |
                    |              restrict BIRDS.Species = "Swan"
                 product                           |
                    |                            BIRDS
              +-----+-----+
              |           |
           SPONSORS   ADOPTION
```

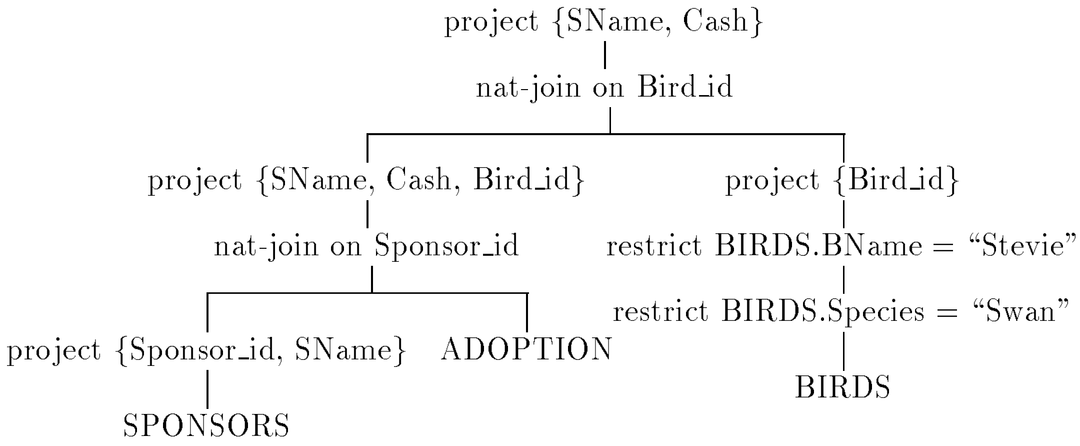
(3) Combine products followed by restricts into joins.

```
    project {SPONSORS.SName, ADOPTION.Cash}
                       |
              nat-join on Bird_id
                       |
          +------------+-------------+
          |                          |
nat-join on Sponsor_id   restrict BIRDS.BName = "Stevie"
          |                          |
    +-----+-----+        restrict BIRDS.Species = "Swan"
    |           |                    |
SPONSORS   ADOPTION                BIRDS
```

(4) Move projects down the tree as far as possible.

Using:

project(nat-join(R, S), A) $\equiv$
project(nat-join (project(R, $A_1 \cup A_2$}), (project(S, $A_3 \cup A_4$))
A)
where $A_1 \cup A_3 = A$ (the attributes we want) and A_2 and A_4
are not
in A but are needed in the join condition.



(iii) (a)

T1 := restrict(BIRDS, BName = "Mandy" $\wedge$ Species = "Moorhen")
T2 := nat-join(T1, ADOPTION)
T3 := restrict(T2, Cash > £3000)
T4 := restrict(BIRDS, Species = "Coot")
T5 := nat-join(T4, ADOPTION)
T6 := restrict(T5, Cash > 0)
T7 := intersect(T3, T6)
T8 := nat-join(T7, SPONSORS)
T9 := project(T8, {Sponsor_id, SName})

(iii) (b)

T1 := project(BIRDS, {Bird_id})
T2 := divide(ADOPTION, T1)
T3 := nat-join(T2, SPONSORS)
T4 := project(T3, {Sponsor_id, SName})

Exercise D

(i)

(a)

T1 := project(CATS, {Mother_catno})
T2 := project(CATS, {Father_catno})
T3 := union(T1, T2)
T4 := project(CATS, {Cat_no})
T5 := difference(T4, T3)
T6 := join(T5, CATS, T5.Cat_no = CATS.Cat_no)
T7 := project(T6, {Catname})

(b)

T1 := rename(CATS, Catno $\rightarrow$ Cat_no1, etc.)
T2 := rename(CATS, Catno $\rightarrow$ Cat_no2, etc.)
T3 := join(T1, CATS, Mother_catno1 = Cat_no)
T4 := join(T2, CATS, Father_catno2 = cat_no)
T5 := join(T3, T4, Mother_catno1 = Mother_catno2 $\wedge$
Father_catno2 = Father_catno1)
T6 := restrict(T5, Catname = 'Muffin' $\vee$ Catname = 'Crumpet'
$\vee$
Catname = 'Teacake')
T7 := project(T6, {Catname1, Catname2})

(ii)

(a) Transitive closure: requires $n-1$ joins, where n is the depth of the tree. But we don't know n in advance, and in this *sub*language we cannot write a loop that iterates to a fixpoint.

(b) `Count` is not among the 8 operators, and `Group by` is higher-order while the algebra is first-order.

(iii)

(a) Draw a query tree, break up restricts, move restricts down the tree, make products followed by restricts into joins, and move projects down the tree.

(b) Ψ = {Cat_no}; Φ = {Mother_catno1, Sex1}

(c) Use as an integrity constraint: on insertion or modification of the Mother_catno for a cat, you would want to find out whether that 'mother' cat is indeed female.

(d) θ can only use attributes in A.

If a project and a restrict on a relation are done together in a single pass through the relation, then the ordering in the query tree is irrelevant. But if they are done separately, then use this equivalence to do the restrict before the project: restrict might work fairly efficiently if it can exploit indexes, etc., and then it will feed a smaller relation into the project, which is better given the problems of removing duplicates in projects.

Exercise E

(i) project(R, {A, B})

(ii)

Query tree:

```
            project {REPORTS.Date_reported}
                          |
                       restrict
          VIOLATIONS.VId = REPORTS.Vid ∧
        RESTAURANTS.RId = REPORTS.RId ∧
             VIOLATIONS.VDesc = 'Rats' ∧
   RESTAURANTS.RName = 'The Cook's Tour'
                          |
                       product
                   ┌──────┴──────┐
                product     VIOLATIONS
           ┌───────┴───────┐
      RESTAURANTS       REPORTS
```

(1) Break up complex restricts.

Using

restrict(R, $\theta_1 \wedge \theta_2$) $\equiv$ restrict(restrict(R, θ_2), θ_1)

```
project {REPORTS.Date_reported}
                |
restrict VIOLATIONS.VId = REPORTS.Vid
                |
restrict RESTAURANTS.RId = REPORTS.RId
                |
restrict VIOLATIONS.VDesc = ‘Rats’
                |
restrict RESTAURANTS.RName = ‘The Cook’s Tour’
                |
             product
          ______|______
         |             |
      product     VIOLATIONS
    ____|____
   |         |
RESTAURANTSREPORTS
```

(2) Move restrict down the tree as far as possible.

Using

restrict(nat-join(R, S), θ) $\equiv$ nat-join(restrict(R, θ_1), restrict(S, θ_2))

as long as θ can be rewritten as $\theta_1 \wedge \theta_2$ where θ_1 involves only attributes of R and θ_2 involves only attributes of S
(But here the nat-join is just a product.)

```
                         project {REPORTS.Date_reported}
                                      |
                  restrict VIOLATIONS.VId = REPORTS.Vid
                                      |
                                   product
                                      |
                     ----------------------------------
                     |                                |
restrict RESTAURANTS.RId = REPORTS.RId              restrict
                     |                    VIOLATIONS.Vdesc = 'Rats'
                  product                             |
                     |                           VIOLATIONS
                -----------
                |         |
            restrict   REPORTS
RESTAURANTS.RName = 'The Cook's Tour'
                |
          RESTAURANTS
```

(3) Combine products followed by restricts into joins.

```
                project {REPORTS.Date_reported}
                             |
                       nat-join on VId
                             |
                 -------------------------
                 |                       |
          nat-join on RId             restrict
                 |          VIOLATIONS.Vdesc = 'Rats'
            -----------                  |
            |         |             VIOLATIONS
        restrict   REPORTS
RESTAURANTS.RName = 'The Cook's Tour'
            |
      RESTAURANTS
```

(4) Move projects down the tree as far as possible.

Using:

project(nat-join(R, S), A) $\equiv$
project(nat-join (project(R, $A_1 \cup A_2$}), (project(S, $A_3 \cup A_4$))
A)
where $A_1 \cup A_3 = A$ (the attributes we want) and A_2 and A_4
are not
in A but are needed in the join condition.