Name:	
Student#:	
Signature:	

The University Of New South Wales

Final Exam

November 2006

COMP9311

Database Systems

(1) Time allowed: **3 hours**

(2) Total number of marks: **110**

(3) Total number of questions: **8**

(4) Answer **all** questions

(5) Questions are **not** of equal value

(6) This paper may **not** be retained by the candidate

Fill in your details *before* you start work.

All answers must be written in ink. Except where they are expressly required, pencils may be used only for drawing, sketching or graphical work.

Textbooks, study notes, calculators, etc.
are **not** permitted in this exam.
Start each answer on a new page in a script book.
If you use more than one script book,
fill in your details on the front of *each* book.

Question 1

(20 marks) Consider the following set of requirements to model a bus timetable system:

- a *driver* has an employee id, a name and a birthday
- a *bus* has a make, model, registration number and capacity
  (e.g. a Volvo 425D bus which can carry 60 passengers, with registration MO-3235)
- a *bus* may also have features (e.g. air-conditioned, disabled access, video screens, etc.)
- a *bus-stop* (normally abbreviated to simply *stop*) is a defined place where a bus may stop to pick up or set down passengers
- each stop has a name, which is displayed on the timetable (e.g. "Central Station")
- each *stop* also has a location (street address) (e.g. "North side of Eddy Avenue")
- a *route* describes a sequence of one or more stops that a bus will follow
- each *route* has a number (e.g. route 372, from Coogee to Circular Quay)
- each *route* has a direction: "inbound" or "outbound"
  (e.g. 372 Coogee to Circular Quay is "inbound", 372 Circular Quay to Coogee is "outbound")
- for each stop on a route, we note how long it should take to reach that stop from the first stop
- the time-to-reach the first stop on a route is zero
- stops may be used on several routes; some stops may not (currently) be used on any route
- a *schedule* specifies an instance of a route (e.g. the 372 departing Circular Quay at 10:05am)
- schedules are used to produce the timetables displayed on bus-stops
- a *service* denotes a specific bus running on a specific schedule on a particular day with a particular driver
- services are used internally by the bus company to keep track of bus/driver allocations
- the number of minutes that each bus service arrives late at its final stop needs to be recorded

Using the information above, draw an ER diagram to model this scenario.

Note that you must:

a) underline all primary key attributes
b) clearly indicate relationship cardinalities
c) clearly indicate participation constraints
d) choose sensible names for attributes, entities and relationships
e) use a single, well-recognised ER notation for the entire diagram

Question 2

(10 marks) Give a suitable PostgreSQL `CREATE DOMAIN` statement to define each of the following attribute types:

a) `SmallInt` for integers in the range 1..100

b) `PosNum` for positive arbitrary-precision numbers

c) `String` for character strings up to 50 characters in length

d) `Code` for 3-char strings comprising only upper-case letters (e.g. `'ABC'`)

Use `check` constraints to ensure that only valid values can be stored in attributes based on these domains.

Question 3

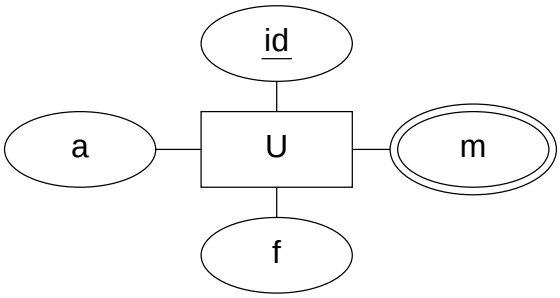
(10 marks) Convert each of the following ER diagram fragments into a relational schema expressed as a collection of PostgreSQL `CREATE TABLE` statements. Your schemas must show all primary key constraints, foreign key constraints and "not null" constraints suggested by the diagram. You may add any new attributes that you need to represent all aspects of the ER diagram. Also, document any semantic aspects suggested by the diagram which *cannot* be represented in the relational schema.

Use the domains from Q2 in answering this question, according to the following scheme:

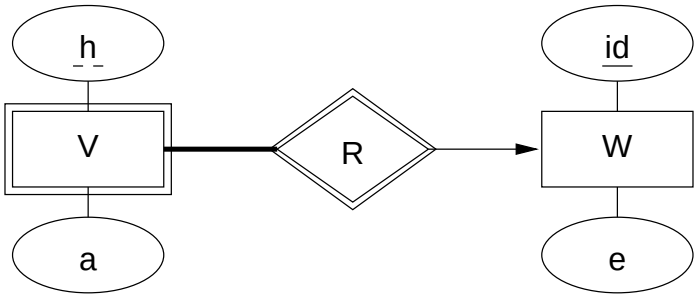
- any attribute called `id` is a `serial` attribute
- any attribute called `a` or `b` is a `SmallInt` attribute
- any attribute called `d` or `e` is a `PosNum` attribute
- any attribute called `f` or `g` is a `String` attribute
- any attribute called `m` or `n` is a `Code` attribute

Additional requirements for each mapping are given before the corresponding ER diagram.

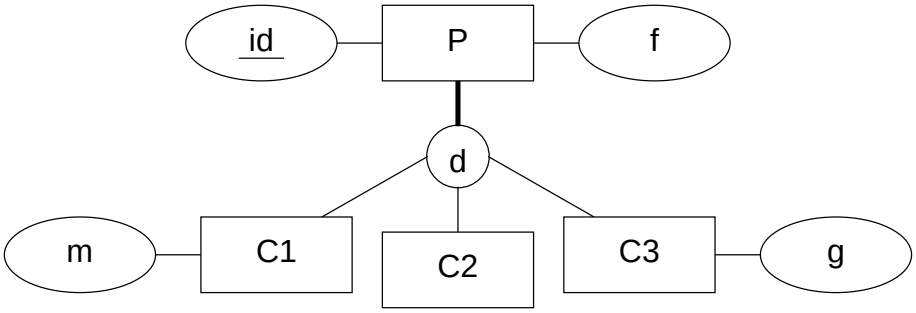
a) Map this entity and its attributes into one or more tables.



b) Map this weak/strong entity pair into multiple tables.



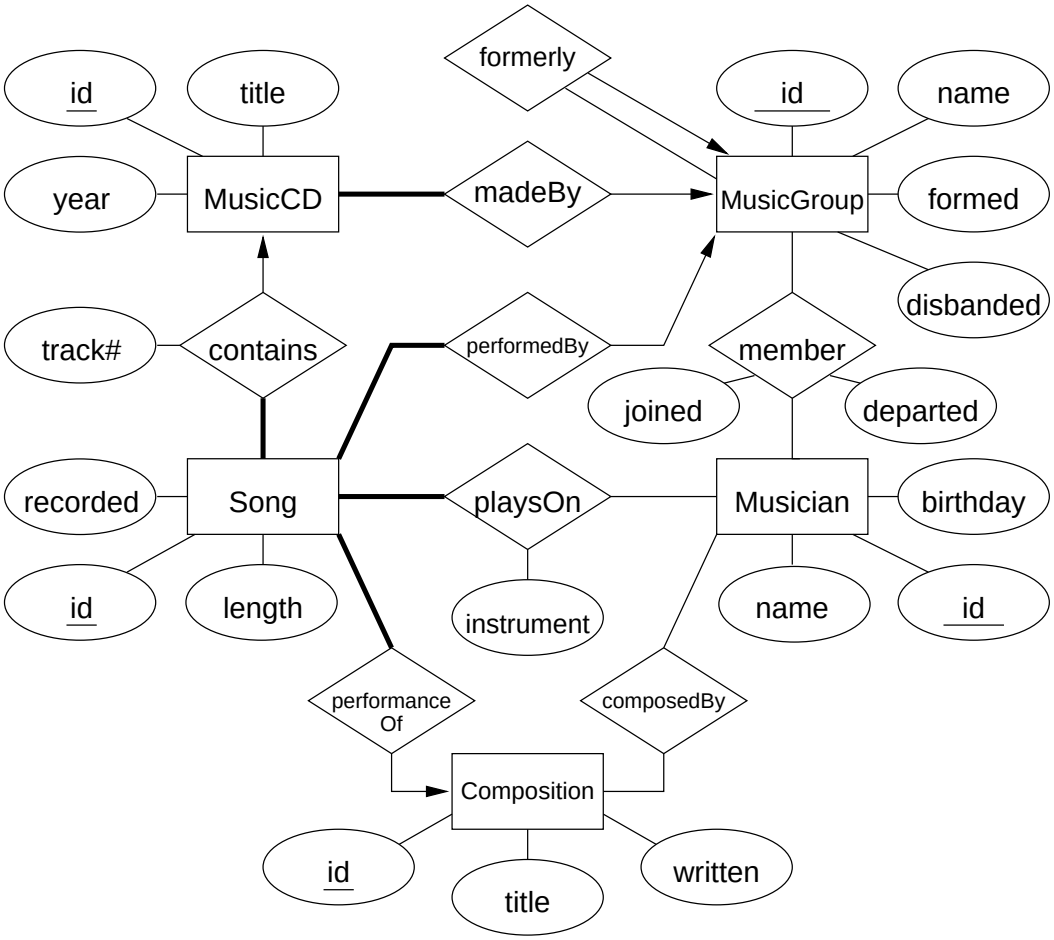
c) Convert this class hierarchy using the ER mapping.



Background for Questions 4-8.

The following questions deal with a database to represent information about CDs made by popular music groups. As well as containing some obvious data elements, the schema implements the requirements:

- a *group* has a number of members, who are *musicians*
- musicians write musical *compositions*; compositions may be anonymous
- a *song* is a particular performance of a composition
- groups record a collection of songs and assemble these on a *music CD*
- each music CD has songs by just one music group (i.e. no compilations)
- a composition may be performed many times, and by different groups
- every song must have at least one performer; performers can play on many songs
- performers may play several different instruments on a particular song
- each song performance appears on only one music CD
- no group ever releases more than one music CD in a given year
- a group is disbanded once all of its members have departed or it changes its name



The above ER design has been mapped into a relational database schema *(see over page)*.

```
create table MusicGroup (
        id          integer primary key,
        name        varchar(50) not null,
        formed      date not null, -- when the group formed
        disbanded   date, -- when they split up (null if still together)
        formerly    integer references MusicGroup(id)
);

create table MusicCD (
        id          integer primary key,
        title       varchar(100) not null,
        year        integer not null check (year >= 1970),
        madeBy      integer not null, -- which group made this CD
        foreign key (madeBy) references MusicGroup(id)
);

create table Musician (
        id          integer primary key,
        name        varchar(50) not null,
        birthday    date
);

create table Member (
        musicGroup  integer references MusicGroup(id),
        musician    integer references Musician(id),
        joined      date not null,
        departed    date,  -- null if still a member
        primary key (musicGroup,performer,joined)
                    -- allows them to join the group several times
                    -- e.g. join, leave, re-join, ...
);

create table Composition (
        id          integer primary key,
        title       varchar(100) not null,
        written     date
);

create table Composer (
        composition integer references Composition(id),
        musician    integer references Musician(id),
        primary key (composition,musician)
);
```

(continued over page)

```
create table Song (
        id          integer primary key,
        composition integer not null, -- which composition this is a performance of
        length      interval not null, -- duration of performance
        recorded    date not null,
        onCD        integer not null, -- which CD this song appears on
performedBy integer not null, -- which group performed this song
        trackNo     integer not null check (trackNo > 0), -- position
        foreign key (composition) references Composition(id),
        foreign key (onCD) references MusicCD(id),
        foreign key (performedBy) references MusicGroup(id)
);

create table PlaysOn (
        musician    integer references Musician(id),
        song        integer references Song(id),
        instrument  varchar(20),
        primary key (musician,song,instrument)
);
```

Question 4

(25 marks) For each of the following information requests, devise a *view* (or collection of views) to solve it. Give each "top-level" view the same name as the question part (e.g. `Q4a`, `Q4b`, etc.). You *must not* use SQL functions or PLpgSQL functions in solving these.

a) What is the most recent music CD made by the group 'Rolling Stones'?

b) Give the name of each group and the number of CDs that the group has made.

c) What is the title and length of the longest music CD ever made?

d) Which musician(s) played the most number of different instruments?

e) Which musicians are song-writers only? (i.e. compose but don't perform)

f) Which musicians have been members of more than one group during their careers?

g) Which member(s) of the group 'White Stripes' have played drums?
(Assume that "have played drums" is indicated by `PlaysOn.instrument = 'drums'`)

h) Which musicians played on all songs contained on the CDs made by 'The Cure'?

If the answer to a question is one or more music CDs, give the title(s). If the answer to a question is one or more music groups, give their name(s). If the answer to a question is one or more musicians, give their name(s).

Question 5

(5 marks) Write an SQL function to check whether the following assertion holds:

- each music CD has songs by just one music group

The function has is defined as:

```
create function oneGroupPerCD() returns boolean as $$ ... $$ language sql;
```

The function should return a value of `true` if, for every MusicCD in the database, all of the songs are performed by the group who made the CD, and return `false` otherwise.

You may define auxiliary views and SQL functions in implementing the `oneGroupPerCD` function. You *must not* use any PLpgSQL in solving this problem.

Question 6

(15 marks) Write a PLpgSQL function that takes the name of a music group and returns a summary report of all their music CDs as a text string. The function is defined as:

```
create function discography(groupName text) returns text
as $$ ... $$ language plpgsql;
```

If the `groupName` does not match that of any existing group, the function should raise an exception with the message: `No such group`.

The list of CDs should be produced in chronological order and should give for each CD: title, year made, list of group members who performed on the CD (in any song), list of other musicians who performed on the CD (in any song). The order in which musicians appear in lists does not matter; however, each name must appear only once. An example of the required format (for an imaginary music group):

```
MusicDB=# select discography('Pink Strawberries');

CD: We Love Strawberries (1978)
Group members: Joe Smith, John Smith, Jet Black
Other musicians: none

CD: Black is White (1980)
Group members: Joe Smith, John Smith, Jet Black, Dave Green
Other musicians: Harold White

CD: Mud Cream Pie (1982)
Group members: Joe Smith, John Smith, Dave Green
Other musicians: Jet Black, Prince, Ian Dury
```

(Note: Jet Black left the Pink Strawberries in 1981, and so appears as an "other" musician in 1982)

Question 7

(15 marks) Write triggers to ensure that `MusicGroup` data is correctly maintained when the following changes occur:

- when the final member of the group departs:
  - the group should be recorded as disbanded
  - using the departure date for the final member of the group
- when the name of a group is changed:
  - the existing `MusicGroup` tuple should *not* have its name changed; it should instead be marked as disbanded (using the `CURRENT_DATE` function to obtain the date)
  - a new `MusicGroup` entry should be made, with a link to the old `MusicGroup` tuple
  - all musicians who were members of the old group should be entered as members of the new group
  - all musicians where were members of the old group should be marked as having left the old group

For each case, you must write a suitable PostgreSQL `CREATE TRIGGER` statement and define a PLpgSQL function (or collection of functions) to carry out the appropriate actions. You must specify whether the trigger is executed `before` or `after` the modification event. Your triggers should be defined so as to avoid trigger-cycles; to do this, you are allowed to assume that PostgreSQL allows triggers to be restricted to updates of specified columns.

Question 8

(10 marks) Consider the following University-database scenario

- each course (C) runs in a given semester (E)
- all courses are "streamed" (multiple lectures, in different timeslots (T))
- each stream is taught by a different lecturer (L)
- a lecturer teaches only one course in a given semester
- each course is held in a given room (R) at a particular time
- students (S) enrol in one particular stream

This scenario suggests the following functional dependencies:

$$SCE \to L, \quad LE \to C, \quad CET \to L, \quad LET \to CR, \quad LECT \to R, \quad LET \to C$$

Produce a 3NF schema for this scenario, based on the above dependencies. Start by finding a minimal cover for the dependency set.