Lab Exercise 08
Review DBMS concepts and practice on an alternative DBMS

Aims

This exercise aims to introduce you to:

- practice SQL queries and DBMS meta-commands
- practice on an alternative DBMS

The goal of this lab practice is to try an alternative DBMS and know the difference between different DBMS. The DBMS used in this practice is SQLite, which is a light DBMS and already installed in the CSE servers.

This exercise contains a lot of reading and deals with some concepts in the background part. You can take the background section as a review of previous DBMS knowledge or you can also jump to the exercise part directly.

Here are some useful readings for you.

Comparison of different SQL implementations
Troels Arvin, on his home page

Comparison of relational database management systems
Wikipedia.

Background

Database Management Systems (DBMSs) are software systems that maintain and provide access to ***database instances*** and their ***meta-data***. So, what's a *database instance*? A useful working definition might be a "collection of inter-related data items". *Meta-data* is information that describes what the data items represent (i.e. it's data about data). In terms of these, a ***database*** would then be a pair (meta-data, database-instance). A DBMS allows you to describe the structure of a database by reading and storing meta-data, and allows you to "populate" the database by reading and storing data items that fit this structure.

In fact, there are several different "styles" of databases and DBMSs. In this course, we will be primarily concerned with one particular style of database, ***relational databases***, and their software systems, ***relational database management systems*** (RDBMSs). Relational databases are, in some senses, the "classical" style of database, but they are also extremely popular, extremely useful and have a simple, elegant theory which translates neatly to a practical implementation (a characteristic which is, unfortunately, rare in computer science).

But before we get too abstract, let's consider a concrete scenario...

Imagine that we are movie buffs and want to build a system that's better than IMDb: the Internet Movie Database. What kind of information do we need to store? Some obvious ones are movies and actors/actresses. What about others? Producers, directors, crew, plot, popularity, genre, bloopers, etc. etc. The list could go on almost endlessly, so let's focus on just the following: movies, genres, directors, actors, and information about the roles that actors play in movies.

The next question is: what kind of information do we want to store about movies? Once again, there are some obvious things like the movie's title and the year it was made. Storing the year would be useful to distinguish two movies with the same title e.g. the original 1933 version of "King Kong", the 1976 version (think: Jessica Lange), and Peter Jackson's 2005 version. It might also be useful to know what kind of movie it is (i.e. its genre, e.g. Sci-Fi). In fact, some movies could belong to several genres e.g. a Sci-Fi Horror movie (think "Alien").

What information do we want to store about actors? Clearly, their name, and perhaps their birthday and other biographical information (which, like a movie's year, might help to distinguish two actors with the

same name). If we're going to treat actors and actresses the same, we might not care about their gender ... but when it comes to Oscars time, it's important to know who's the Best Actress and who's the Best Actor, so we probably do need to represent the gender somehow. For directors, we probably want similar information to actors (after all, they're both people and, in fact, some actors (e.g. Clint Eastwood) go on to become successful directors).

Information about roles played by actors in movies is different to data objects like movies, actors and directors, because it involves a relationship between a movie and an actor. That is, an actors plays a particular role in the context of some movie. Note that they may play more than one role in a given movie, or might play the same role in several movies. Another similar example is describing the director of a movie; this is a relationship between the movie and the person who directed it. If we assume that each movie has only one director, then this relationship is actually simpler than that between actors, movies and roles.

Let's put all of this together into some kind of data structure. If we were to code the database in C, our meta-data might look something like:

```
typedef struct Movie {
        char *title;     // e.g. "Avatar"
        int   yearMade;  // sometime after 1850
        char *genre;     // what kind of movie is it
} Movie;

typedef struct Actor {
        char *name;      // e.g. "Clint Eastwood"
        int   yearBorn;  // sometime after 1850
        char  gender;    // i.e. 'm' or 'f'
} Actor;

typedef struct Director {
        char *name       // e.g. "Steve Spielberg" or "Clint Eastwood"
        int   yearBorn;  // sometime after 1850
} Director;
```

Note that have a gender for actors, but not for directors. Since there are not separate awards for directors and directoresses, the gender is not an issue in this context. (Although, if there were Best Director and Best Directoress awards, James Cameron would have rightfully gotten a Best Director award for “Avatar” :-)

The above doesn't deal with the relationships between actors and movies and directors and movies. If we consider the director/movie case, the relationship could potentially be implemented via:

```
typedef struct Movie {
        char *title;     // e.g. "Avatar"
        int   yearMade;  // sometime after 1850
        char *director;  // e.g. "James Cameron"
} Movie;
```

This works ok as long as each movie has only one director *and* all directors have different names. Ignoring multiple-director movies for the time being, here is an alternative which ensures that the movie is connected to exactly the right director:

```
typedef struct Movie {
        char *title;        // e.g. "Avatar"
        int   yearMade;     // sometime after 1850
        char *genre;        // what kind of movie is it
        Director *director; // e.g. ref to "James Cameron"
} Movie;
```

For the relationship describing an actor's role in a movie, we have a bigger challenge. Since an actor can have many roles in many movies, we need a data structure to represent multiple relationships. The following structure might work:

```
typedef struct Role {
        Movie *movie;  // e.g. ref to "Wall Street"
        char  *role;   // e.g. "Gordon Gekko"
        struct Role *next;
} Role;

typedef struct Actor {
        char *name;      // e.g. "Clint Eastwood"
        int   yearBorn;  // sometime after 1850
        char gender;     // i.e. 'm' or 'f'
        Role *roles;     // linked list of Roles
} Actor;
```

An alternative structure would be to put a linked list of (actor,role) pairs into each `Movie` struct. We have a similar issue if we want to allow a movie to be classified under several genres. We *could* implement this via a comma-separated string (e.g. `"Sci-Fi,Horror"`), but a more typical approach would be to have a linked list of genres for each movie:

```
typedef struct Genre {
        char *name;
        struct Genre *next;
} Genre;

typedef struct Movie {
        char *title;        // e.g. "Avatar"
        int   yearMade;     // sometime after 1850
        Genre *genre;       // linked list of Genres
        Director *director; // e.g. ref to "James Cameron"
} Movie;
```

If we wanted to allow the multiple-director case, that could be handled in a manner similar to genres, via a linked list of directors.

An important point to note is that, in this model, movies, actors and directors can be uniquely identified by using a pointer to a struct for that object. We don't need to use the year of their creation to disambuiguate objects with the same name.

Of course, the above discussion has nothing directly to do with relational databases, but it does raise some issues that they need to deal with, and we consider those now as we develop a relational data model for the above scenario.

First, relational databases have only two simple structuring mechanisms: relations and tuples. A ***relation*** is a set of ***tuples***, where each tuple contains a set of values related to a single object or relationship. A tuple is somewhat similar to a C struct in that it has a collection of heterogeneous data values, where each value has a name and a type. The elements of a tuple are called ***fields***. Relations can be conveniently thought of as tables, where each row of the table is a tuple, and each column contains the values for one field.

> Don't be confused by the two terms *relation* and *relationship*. They may sound similar, but they are not the same thing. A *relation* is as we defined it above: a set of tuples. A *relationship* represents an association between two objects e.g. John *works at* UNSW, you *are enrolled in* COMP9311. Now that that's clear, we'll muddy the waters again by pointing out that relations are often used to *represent* relationships ... but more on that below.

In the above example, some obvious potential relations are Movies, Actors, Directors. If we consider the fields for these relations, we could come up with a description like:

```
Movies(title, yearMade, director)
Actors(name, yearBorn, gender)
Directors(name, yearBorn)
```

Note that one of the fields `director` in the `Movies` relation is actually representing the relationship "director *directed* Movie". The other relationships, between actors and movies, cannot be represented in the relational model in the same way as they are in the C struct above, because the relational model has no equivalent to an array or linked list. Since all it has are relations and tuples, we end up introducing new relations to represent the relationships. And if we want to allow the more reaslistic possibility of a movie being directed by several people, we have a similar issue with the relationship between movies and directors.

The above considerations lead to the following data model:

```
// relations representing "objects"
Movies(title, yearMade)
Actors(name, yearBorn, gender)
Directors(name, yearBorn)
// relations representing relationships
BelongsTo(movie, genre)
AppearsIn(actor, movie, role)
Directs(director, movie)
```

We can now completely represent all of the different objects and relationships in our scenario. However, we're still not quite done, because the fields like `actor` in the `AppearsIn` relation (which we will denote from now on as `AppearsIn.actor`) requires that we specify a particular actor, but simply giving the name may not do this if their are two actors with the same name. Similar considerations apply to `AppearsIn.movie`, `Directs.director`, `Directs.movie` and `BelongsTo.movie`.

This issue arises because relational databases do not have pointers or object IDs, and so a value-based mechanism is used to establish the identity of objects. In RDBMSs, we can either use a subset of the data values or introduce a new field to contain an ID for the tuple which uniquely identifes that tuple within the table. Examples of this approach are common e.g. your student number, your tax file number, etc. If we call all the ID fields in the different tables `id`, just to simplify things, we end up with a data model like:

```
// relations representing "objects"
Movies(id, title, yearMade)
Actors(id, name, yearBorn, gender)
Directors(id, name, yearBorn)
// relations representing relationships
BelongsTo(movieID, genre)
AppearsIn(actorID, movieID, role)
Directs(directorID, movieID)
```

Once we can uniquely identify objects via an ID, the `yearMade` and `yearBorn` fields become less critical. We will now make the somewhat arbitrary decision to drop the `Actors.yearBorn` and `Directors.yearBorn` fields (maybe we're not interested in any biographical information), but to keep the `Movies.yearMade` field (because it's useful to distinguish "King Kong (1933)" from "King Kong (2005)" when talking about movies). This leads to the following refinement of our data model:

```
// relations representing "objects"
Movies(id, title, yearMade)
Actors(id, name, gender)
Directors(id, name)
// relations representing relationships
BelongsTo(movieID, genre)
AppearsIn(actorID, movieID, role)
Directs(directorID, movieID)
```

One final refinement to the model is also based on user interface considerations. If we ever need to present a list of people's names, it's useful to do it in alphabetical order on their family name. Unless the name is written as `"Spielberg, Steve"`, sorting based on family name is tricky to achieve, so we decide to break the names into two components:

```
// relations representing "objects"
Movies(id, title, yearMade)
Actors(id, familyName, givenName, gender)
Directors(id, familyName, givenName)
// relations representing relationships
BelongsTo(movieID, genre)
AppearsIn(actorID, movieID, role)
Directs(directorID, movieID)
```

The above data model is still somewhat abstract, but we can envisage tuples such as the following appearing in the database instance.

```
Movies(61,"Intolerable Cruelty",2003)
Actor(2999,"Zeta-Jones","Catherine","f")
AppearsIn(2999,61,"Marylin")
```

Before we can use this data model, we need to render it in a language that an RDBMS can understand. This requires us to make some concrete decisions about the precise types of the fields, and which combinations of fields are unique over all tuples in each relation. The ***SQL*** language is best known as a query language, but the SQL standard also defines a data definition language to allow us to describe data models for RDBMSs. SQL data models are typically called ***schemas*** (or *schemata* by linguistic pedants). We present an SQL implementation (schema) of the above data model, and then explain the bits that do not appear in abstract data model above.

```
create table Movies (
        id          integer,
        title       varchar(256),
        year        integer check (year >= 1900),
        primary key (id)
);

create table BelongsTo (
        movie       integer references Movies(id),
        genre       varchar(32),
        primary key (movie,genre)
);

create table Actors (
        id          integer,
        familyName  varchar(64),
        givenNames  varchar(64),
        gender      char(1),
        primary key (id)
);

create table AppearsIn (
        actor       integer references Actors(id),
        movie       integer references Movies(id),
        role        varchar(64),
        primary key (movie,actor,role)
);

create table Directors (
        id          integer,
        familyName  varchar(64),
        givenNames  varchar(64),
        primary key (id)
);

create table Directs (
        director    integer references Directors(id),
```

```
        movie       integer references Movies(id),
        primary key (director,movie)
);
```

A copy of this schema is available in the file

```
/home/cs9311/web/24T2/lab/08/schema.sql
```

The first thing to note is that the relations and fields have almost exactly the same names as in our abstract data model, except that we've dropped the `ID` from names like `movieID`. We introduce a new relation via the `create table` statement, which requires us to provide a name for the relation and then give the details of all of its fields. We can also specify which combination of fields is unique within the table via the `primary key` statement.

One clear difference with abstract relational model is that we need to specify types for each field. Note also that types can be specified more precisely than in C, where we can only state in the `struct` definition that the `yearMade` field is an integer value (although you should note that we changed the name of the field from `yearMade` to `year`). The `check` in SQL enforces that the `yearMade` value (actually `year`) is not only an integer, but it must be greater than 1899 (assuming that no movies were made before 1900).

String data types are different in SQL than they are in C. There are actually two different kinds of string: fixed-length (`char`) and variable-length (`varchar`). In both cases, we need to specify a maximum string length. Any `char` values that are shorter than the maximum length are blank padded to be exactly the specific length. Any `varchar` values that are shorter than the specified length are stroed in the DBMS exactly as written. Another important difference between C strings and SQL strings, is that SQL strings are written in single quotes (e.g. `'a string'`) and do not support C's escape characters (e.g. `\n`).

One final point of explanation on the schema ... a line such as

```
movie       integer references Movies(id),
```

tells the DBMS that there is a field called `movie` which contains an integer value. However, it cannot be just any old integer value; it must contain an integer that occurs in the `id` field of a tuple in the `Movies` table. Hopefully, it's reasonably clear that this is how we “link” tuples together.

We could load the above schema into an RDBMS, and it would give us a collection of empty tables. In order to do fun and useful things, we need some data. There is a large file containing SQL statements to add tuples to the tables in this database:

```
/home/cs9311/web/24T2/lab/08/data.sql
```

You can look at this if you want, but a more interesting way to interact with the data is via an RDBMS. In this prac, we are going to use the SQLite DBMS. Like most DBMSs it stores both meta-data and database instances. It also provides the SQL query language for looking at the data. And, like other DBMSs, it provides a set of meta-commands for doing things like reading SQL from files, looking at the meta-data, and so on. For the remainder of this prac exercise, we will largely be playing with an SQLite instance of the movies database described above.

Exercise

If you're logged in to a CSE workstation, you can start an SQLite interactive shell to work on our database via the command:

```
$ sqlite3 /home/cs9311/web/24T2/lab/08/movies.db
```

(The data is imported with *.db file. You can also try to figure out how to imported with data.sql and schema.sql.) This should produce a response something like:

```
SQLite version 3.7.13 2012-06-11 02:05:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

The specific vesion of SQLite may be different depending on which machine you happen to be using, but as long as it is version `3.7.x` the examples below should work.

> In the sample interactions in these prac exericses, we show anything that you are supposed to type in **`bold font`**. Anything that the computer is going to type at you will be in `normal font`. The Linux shell prompt is denoted by a `$`.

Note that you can't read the `movies.db` file using the `cat` command or by using a text editor, because it contains binary data representing the structures of the database, along with some text data representing the stored values. You normally interact with such files using the `sqlite3` command. The Linux `file` command will tell you what kind of file it is:

```
$ file /home/cs9311/web/24T2/lab/08/movies.db
movies.db: SQLite 3.x database
```

Returning to the SQLite command that you just ran, the `sqlite>` is SQLite's prompt. In response to this prompt, you can type either SQL statements or SQLite meta-commands. To find out what meta-commands are available type:

```
sqlite> .help
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail ON|OFF           Stop after hitting an error.  Default OFF
.databases             List names and files of attached databases
etc. etc. etc.
.trace FILE|off        Output each SQL statement as it is run
.vfsname ?AUX?         Print the name of the VFS stack
.width NUM1 NUM2 ...   Set column widths for "column" mode
.timer ON|OFF          Turn the CPU timer measurement on or off
sqlite>
```

Note that all of the meta-commands begin with a period (`'.'`) character. The most useful meta-command at this stage is `.quit`, which allows us to escape from SQlite:

```
sqlite> .quit
$
```

Now, restart SQLite and try this command:

```
sqlite> .schema
```

SQLite will give you a list of tables in the database. This list should match the list above, although the tables may not be in the same order as in the schema above. Another difference is that the `create table` is capitalized. The two words `create` and `table` are SQL keywords and SQL keywords are case-insensitive. Identifiers in SQL are also case insensitive, so we could write the name of the `Movies` table as `movies` or even as `MoViEs`.

Now let's try some data analysis. We know that there's a table called `Directors` in the database, but what directors do we actually know about? We can find out their names via the SQL statement:

```
sqlite> select givenNames,familyName from Directors;
James|Cameron
Lars|von Trier
Chan-wook|Park
```

```
Steven|Spielberg
David|Lynch
Joel|Coen
sqlite>
```

The above SQL statement asks SQLite to show you the two name fields from every tuple in the `Directors` table. The names look a bit strange because SQLite displays the field values in each tuple separated by a bar/pipe (`'|'`) character. If we wanted to make the names look more "normal", we could use SQL's string concatentation operator (`||`) to join the two name components together, separated by a space:

```
sqlite> select givenNames||' '||familyName from Directors;
James Cameron
Lars von Trier
Chan-wook Park
Steven Spielberg
David Lynch
Joel Coen
sqlite>
```

Remember that one reason we split the names was so that we could sort them alphabetically on family name, so let's do that:

```
sqlite> select givenNames||' '||familyName from Directors order by familyName;
James Cameron
Joel Coen
David Lynch
Chan-wook Park
Steven Spielberg
Lars von Trier
sqlite>
```

Now let's consider the entire contents of the `Movies` table. In SQL, we can ask for all fields in each tuple to be displayed by using a * instead of a list of field names, as in:

```
sqlite> select * from Directors;
1|Cameron|James
2|von Trier|Lars
3|Park|Chan-wook
4|Spielberg|Steven
5|Lynch|David
6|Coen|Joel
```

Each tuple now has all three fields displayed, the ID, the family name and the given names. Note that all of the ID values are distinct, so that a given ID identifies one specific director.

As well as giving information about individual tuples, SQL can also compute summary data on tables. For example, if I was too lazy to count that there were six directors from the above output, I could get SQL to tell me:

```
sqlite> select count(*) from Directors;
6
```

Before getting you to play around with some queries, a couple of other comments on the SQLite interactive interface are needed. You no doubt noticed that each of the SQL statements above ends with a semi-colon. Because SQL statements can extend over multiple lines, you need some way of indicating "Ok, this SQL statement is finished. Please execute it for me." That's what a semi-colon does.

If you end a line without a semi-colon, SQLite assumes that you have more to type for the current SQL statement and changes the prompt to "`...>`". For example:

```
sqlite> select
   ...> count(*)
   ...> from Directors;
6
```

This behaviour can be important if you get halfway through an SQL statement and then decide that you want to quit. SQLite will ignore meta-commands until you finish off the SQL statement with a semi-colon:

```
sqlite> select count(*)
   ...> from
   ...> .quit
   ...> .quit
   ...> ;
Error: near ".": syntax error
sqlite> .quit
$
```

Armed with above information, try to think of SQL queries to answer the following data retrieval problems:

1. How many movies are in the database?
2. What are the titles of all movies in the database?
3. What is the earliest year that film was made (in this database)? (Hint: there is a `min()` summary function)
4. How many actors are there (in this database)?
5. Are there any actors whose family name is “Zeta-Jones”? (case-sensitive)
6. What genres are there?
7. What movies did Spielberg direct? (title+year)
8. Which actor has acted in all movies (in this database)?
9. Are there any directors in the database who don't direct any movies?

[Solution]