COMP1004 Lab Week 3: SQL I– Part A

INTRODUCTION

Today we will start working with SQL. However, before that, we must focus on the problem description and identify the tables, columns, keys, etc. that will form your database.

Note: at the end of the lab today the following files should be submitted on the Moodle using the link "Coursework-Lab Week 3: ER-Diagram and SQL I Submission Link":

- The ER Diagram
- SQL query

Please let the two files be in pdf with your name, student ID number and username.

PROBLEM DESCRIPTION

A film house wants to create a database to store the details of its collections. Information to be stored about each movie includes their price, title, year, genre and number of scenes. Each movie will have a leading actor, and each leading actor may appear in several movies. Scenes will have a location name and running times (minutes). Each scene is supposed to belong to a single movie only. Actors have names associated with them, and it should be possible to search the database with the actor's name.

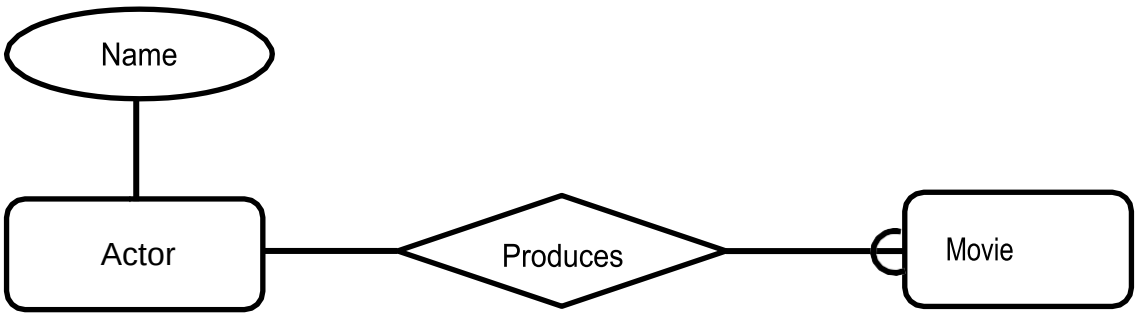
IDENTIFYING ENTITIES, ATTRIBUTES AND RELATIONSHIPS

The first step in database design is to identify entities, attributes and relationships from the problem description. It is also necessary to identify cardinality ratios of the relationships found. For example, Movie and Actor are two entities from the problem above. Actor has an attribute name. There is also a relationship between actors and movies; actors produce movies. This is a one-to-many relationship.

Exercise: Identify the other entities, attributes, relationships and cardinality ratios from the problem description above.

DRAWING AN E/R DIAGRAM

Once you have identified all the entities etc. from the description it is helpful to draw an E/R diagram. This can give you a useful picture of the layout of the database. Using the information above, we can produce a **partial** E/R diagram:



Exercise 1: Complete the E/R diagram with the additional information you have identified from the problem description. If any many-to-many relationship exists in your E/R diagram, split it into two one-to-many relationships. If there is any one-to-one relationship in your E/R diagram, consider merging the two entities involved into a single one **[Output 1].**

CREATING SQL TABLES

It is time now to start writing some SQL statements. Before this, if you're working on any of the lab machines, you must ensure you have set up a MySQL account on the School's servers, following the procedure outlined in the ***"Getting Started with MySQL"*** document.

To implement the database design in SQL you will need to create a table for each entity. A table representing an entity should have:

- A column for each attribute, with appropriate data type and NULL / NOT NULL specified. The default value is NULL if you don't specify anything here.
- A primary key and possibly some candidate keys. If you do not specify a primary key, MySQL may choose one for you if required. You should always specify one to avoid this problem. A primary key is always NOT NULL and UNIQUE, so you don't need to specify these values.

It is often useful to have:

- An ID column (INT) to act as a convenient primary key.
- A prefix for the table, which helps you relate columns to tables in queries.

For the `Actor` table we might choose the prefix act, so an ID column would be called `actID`. A name column might be called `actName`. The ID would likely be an integer value, and the name would be a variable length string, `VARCHAR(255)`.

Each column should be set as being `NULL`, meaning it is not required, or `NOT NULL`, indicating a value must be given. A primary key must be unique, so `actID` should be `NOT NULL`. For `actName` it is a little less clear; however it doesn't make much sense to have an actor with no name, so this can be `NOT NULL` too.

There are also a number of issues surrounding keys. `actID` has already been selected as the primary key, but we could have used `actName`, since every actor should have a unique name. This makes `actName` a candidate key.

In summary, to represent the entity actor:

- We will create a table called Actor.
- This will have columns called `actID` and `actName`.
- The `actID` column will be of type INT and NOT NULL.
- The `actName` column will be of type VARCHAR (255) and NOT NULL.
- `actID` is the primary key, so there will be a PRIMARY KEY constraint on it.
- `actName` is a candidate key, so there will be a UNIQUE constraint on it.

This means we can use the following SQL command to create the table:

```
CREATE TABLE Actor (
     actID INT NOT NULL,
     actName VARCHAR(255) NOT NULL,
     CONSTRAINT pk_actor PRIMARY KEY (actID),
```

```
    CONSTRAINT ck_actor UNIQUE (actName));
```

Note that there is a semicolon at the end of the statement. The semicolon signals that you've finished the command and want to run it.

Next we should create a `Movie` table. This will follow more or less the same procedure as the `Actor` table. However, we must include a foreign key since there is a one-to-many relationship between `Actor` and `Movie`. Creating a foreign key is usually done as follows:

- A foreign key column is put into the `Movie` table.
- If you are using prefixes, it is often helpful to prefix your column with that of the referenced table columns. This means it has exactly the same name as the column it references.
- A `FOREIGN KEY` constraint is used to tell the database about the relationship.
- You should also consider adding an `ON DELETE` clauses. `ON DELETE RESTRICT` is used by default.

Part of a table definition for the `Movie` entity is shown below. The foreign key represents the one-to-many relationship between `Actor` and `Movie`. Given a `Movie` entry, we can use the `Movie.actID` to find the corresponding `Actor`.

```
CREATE TABLE Movie (
    mvID INT NOT NULL,
    actID INT NOT NULL,
    -- Probably want some other attributes in here.
    CONSTRAINT pk_mv PRIMARY KEY (mvID),
    CONSTRAINT fk_mv_act FOREIGN KEY (actID) REFERENCES
    Actor (actID) -- Probably want some options here);
```

Note that to create a foreign key, the actor table must already exist. Often when a database is being created, referencing tables will be created before the referenced table. In this case, it is simpler to split up the process of table creation, adding the foreign keys later. Like this:

```
CREATE TABLE Movie (
    mvID INT NOT NULL,
    actID INT NOT NULL,
    -- Probably want some other attributes in here.
    CONSTRAINT pk_mv PRIMARY KEY (mvID));
```

Then, once Actor has been created:

```
ALTER TABLE Movie ADD CONSTRAINT fk_mv_act FOREIGN KEY (actID)
REFERENCES Actor (actID);
```

Using `ALTER TABLE` you can do a variety of other things, such as adding or removing constraints (more on ALTER TABLE during next lecture).

Exercise 2: Create tables to represent your whole database design. You should have a table for every entity, and a foreign key for every relationship. **[Output 2].**

MYSQL ENGINES

Unlike many DBMSs, MySQL comes equipped with a variety of storage engines. These operate on tables, and instruct the system how to process the data being added and removed. From MySQL Version 5.5 the default engine has been changed from `MyISAM` to `InnoDB`. This latter engine enforces foreign key constraints, so we will always use it. However, if you want to create a table using the `MyISAM` storage engine, simply add `ENGINE= MyISAM` to the end of your `CREATE TABLE` command. Like this:

```
CREATE TABLE Actor (
     actID INT NOT NULL,
     actName VARCHAR(255) NOT NULL,
     CONSTRAINT pk_actor PRIMARY KEY (actID),
     CONSTRAINT ck_actor UNIQUE (actName)) ENGINE= MyISAM;
```

In order to avoid issues with engines in the lab sessions and coursework, you should always work with the default engine (`InnoDB`).

USEFUL COMMANDS

Here are some SQL/MySQL commands you may find useful:

`DESCRIBE tableName;` This command will provide an overview of your table, including columns, names, data types and keys.

`SHOW tables;` This command shows a list of names all of your tables.

`SHOW CREATE TABLE tableName;` This command will show you the create table command for a given table. It will list all columns and constraints. This is useful if you can't remember what constraints you have on that table, or what they are called.