Group 15
243163 – Diego Antognini
244270 – Jason Racine
224295 – Alexandre Veuthey

EPFL
CS-322 Introduction to database systems
18.03.2015

IMDB Project

Milestone 1 partial report

0. Summary

1. Introduction

The goal of this report is to present our work for the project due to EPFL's CS-322 "Introduction to database systems" course. Realized application consists of a database designed and implemented on the basis of simple data files and short description given at the beginning of the project, coupled to a simple web application to manage and query this database. The stored data corresponds to some movies and series with simple relations such as actors, extracted from the *IMDB movies database*[1].

1.1. Technologies

As we had full choice about the tools and software we use for this project, here is a list of the main components we used and the reason of "why these and not others ?".

1.1.1. Database management system (DBMS)

We were offered to use some dedicated *Oracle*[2] server, internal to EPFL. The major inconvenient of this solution was that we won't be able to work on this project in locations where no easy Internet access is provided, like trains, and we would become dependent of the availability of the server.

Instead, we chose to locally install and use *MySQL Community Edition*[3], with the following argument about this choice.

- It is totally free and wide-used, essentially in the world of web applications, which kind of application we chosed to build (see [§1.1.2]) ;
- it comes with *MySQL Workbench*[4], a powerful dedicated tool to model schemas, manage graphically the DBMS parameters, check queries are working and so on ;
- it is available on all major OS, which can be useful as we have different kinds of computers with different OS families installed.

More precisely, we use the following versions of the DBMS and tools.

- MySQL 5.6.23
- MySQL Workbench 6.2.4
- MySQL Utilities 1.5.3

1.1.2. Graphical application development

We had full choice about the programming language and API technologies we want to use to develop the graphical application that has to come out as a result for this project.

We chosed to develop a web application written in PHP, potentially using Javascript to add more interactivity. The exact tools like webserver, PHP version and other details are not decided yet, because this is not part of the first milestone for this project.

[1] http://www.imdb.com
[2] http://www.oracle.com/fr/database/overview/index.html
[3] http://dev.mysql.com/downloads/
[4] http://dev.mysql.com/downloads/workbench/

2. Conceptual design

In this chapter, we analyse the data and metadata that are given at the start point of the project and mix them together into a Entity-Relational (ER) schema and a set of out-of-model constraints.

2.1. Metadata and data analysis

Here we do a hypothesis-based analysis of parts of the data and their metadata (in fact, just column names) we were given. This will be used in [§2.2] to justify parts of the ER schema.

2.1.1. "PERSON" and "ALTERNATIVE_NAME"

These 2 files are obviously linked by the fact that a person can have multiple alternative names, which is confirmed by the presence of the *"person_id"* field in *"ALTERNATIVE_NAME"*. In *"PERSON"* as well as in *"ALTERNATIVE_NAME"*, the *"name"* field seems to be always constructed in the same way, starting by the last name, followed by a comma, and then the first name(s). Sometimes, there is only one name without coma, suggesting it's just the "main name", let's say last name. This field is to be split into 2 distinct fields "lastname" and "firstname" by parsing during the data import.

We also remark that the field *"gender"* in *"PERSON"* contains always values "m" and "f", obviously describing if the person is a man or a woman. This is a candidate field to be exported as a separate entity set in ER schema.

2.1.2. "PRODUCTION" and "ALTERNATIVE_TITLE"

These 2 files are also linked in a similar way as "PERSON" and "ALTERNATIVE_NAME" (see [§2.1.1]).

By digging into "PRODUCTION", we see that there are 3 types of productions to consider.

- The single movies, that are registered once with all their informations ;
- the series, that are registered as a grouping element for episodes that they contain ;
- the episodes, linked to particular serie through the *"series_id"* field, and also registering a particular *"season_number"* and *"episode_number"* relative to the serie it belongs to.

This repartition of productions is a clear candidate to be modelled as a "ISA" hierarchy in the ER schema. More importantly, the *season* concept which is only present through the *"season_number"* field is a perfect candidate to be exported as a separate entity set in the ER schema, adding a level of hierarchy and thus helping to ensure consistency of the data.

The *"gender"* and "*kind*" fields of *"PRODUCTION"* are also good candidates for such export, as they contain highly-repeated values that cleary consist of a finite set of genders and kinds of movies, except for the "episode" and "tv_series" values of "*kind*" that are modelled through the "ISA" hierarchy.

2.1.3. "COMPANY" and "PRODUCTION_COMPANY"

The *"COMPANY"* file contains listing of companies. By digging into it, we can rapidly show the following particularities.

- The *"country_code"* field is highly-repeating, so it is a candidate to be exported in a separate entity set in the ER schema ;
- the *"company_type"* field is in a similar situation, thus also a candidate for such export.

With more attention, a more complex structure can be seen. Firstly, we remark that some keys of the field *"id"* are repeating with the same name of enterprise, only changing the *"company_type"* field.

This is an indication that a single company can have multiple types, giving an indication of the cardinality of the corresponding relation in the ER schema. Then, we see that a company can be present in multiple countries, but with changing *"id"*. A good example of all these variations can be found in the first 100 rows of the file, which is presented below.

Id	country_code	name	company_type
71	[be]	Sony Pictures Home Entertainment	distributors
72	[nl]	Sony Pictures Home Entertainment	distributors
72	[nl]	Sony Pictures Home Entertainment	production companies
73	[us]	Sony Pictures Home Entertainment	distributors
73	[us]	Sony Pictures Home Entertainment	production companies

The *"PRODUCTION_COMPANY"* file acts as an associative relationship between *"PRODUCTION"* and *"COMPANY"*. By digging into it, we easily see that a company can be involved in multiple productions, and that a production needs the participation of multiple companies, thus giving a N:N relationship in the incoming ER schema.

2.1.4. "PRODUCTION_CAST" and "CHARACTER"

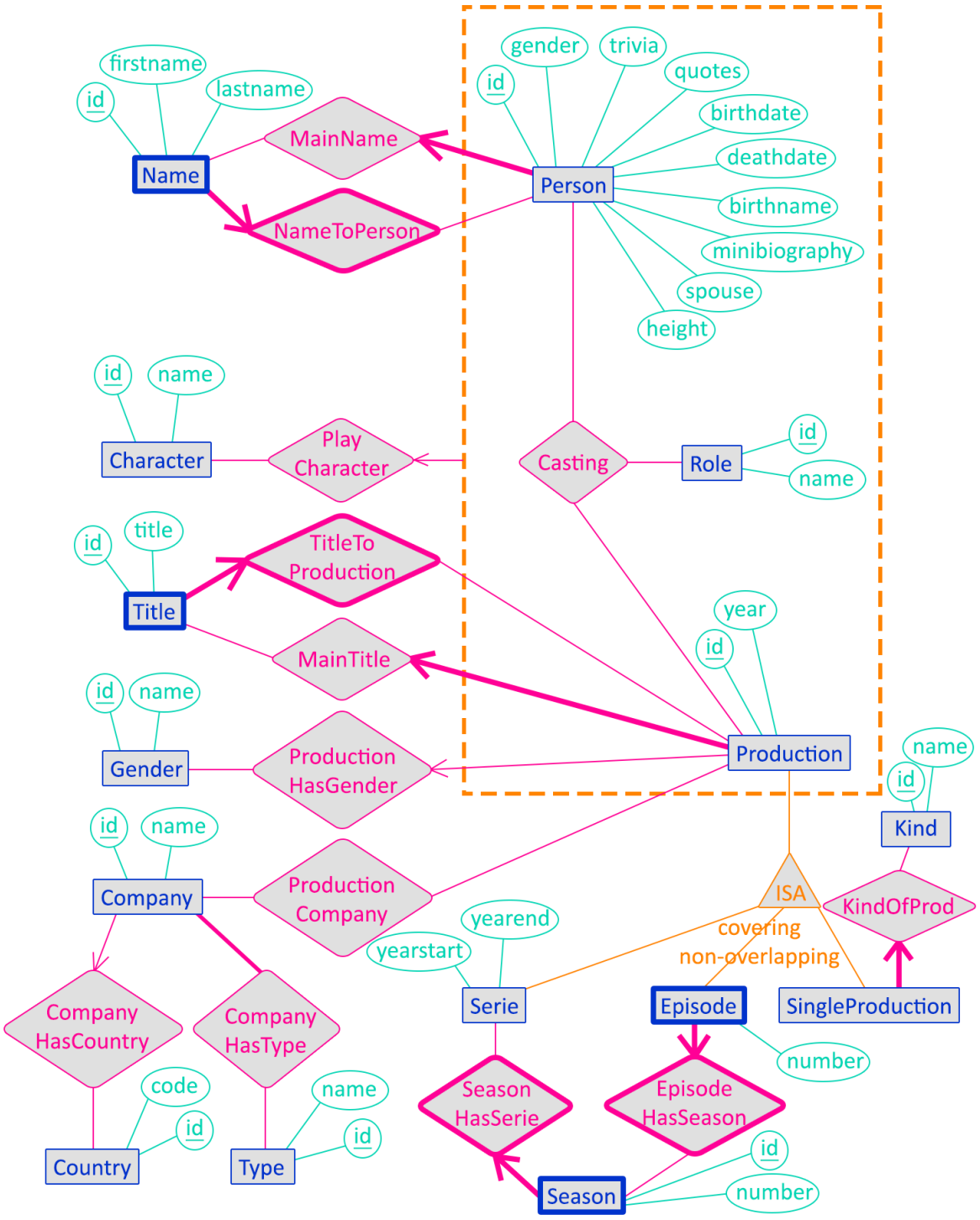
The *"PRODUCTION_CAST"* file acts as an associative relationship between *"PRODUCTION"*, "PERSON" and *"CHARACTER"*, with a supplementary *"role"* value. This value is clearly repeating, making it a good candidate to be modelled as a separate entity set in the ER schema. By exploring the file, we also see that the *"production_id"*, *"person_id"* and *"role"* are always filled, thus always telling that a person participated in a production as a particular role. This is to be modelled as a ternary relationship set in the ER model. But because the *"character_id"* is not always filled, we need to be careful and think about modelling it as a second relationship set, linking the aggregation of our first relationship set to the entity set corresponding to the file *"CHARACTER"*.

Finally, the *"CHARACTER"* file is just a listing of the characters played in movies. By making a quick test with the search function of a basic text editor into *"PRODUCTION_CAST"*, it seems that some characters appears in multiple productions, either referenced by the exact same key or by having multiple instances of it in the *"CHARACTER"* file with slightly different names (e.g. "James Bond", "James Bond 007", "007 James Bond", …). This will be hard to make this proper, but this is not the purpose of the first milestone of this project.

2.2. Entity-Relationship schema (ER)

The conceptual design of the database is achieved using the same ER formalism as the one seen in course. The schema can be found on the next page, followed by a set of out-of-model constraints, these are constraints that apply to the data model to ensure its correctness, but that can't be modelled by the chosen formalism (and more importantly, that any DBMS cannot ensure at all, it's up to the applicative part to deal later with these problems).

2.2.1. ER schema



2.2.2. Caption

Blue	entity sets
Turquoise	attributes
Magenta	relationship sets
- - - - - - -	aggregation
_____	ISA hierarchy

2.2.3. *Out-of-model constraints*

These are the constraints that cannot be modelled on the ER schema, and that will not be ensured by the DMBS but by the application.

1. For each instance of MainName, the referenced Name must belong to the referenced Person through an instance of NameToPerson ;
2. For each instance of MainTitle, the referenced Title must belong to the referenced Production through an instance of TitleToProduction.

2.2.4. *Account for design choices*

Hereafter, we talk about the main design choices we did and some particularities of the method we used to design this database.

Name, MainName and NameToPerson

The entity set Name corresponds to the *"ALTERNATIVE_NAME"* file described in [§2.1.1]. We removed the "alternative" part because the main name, originally the field *"name"* in *"PERSON"*, is also modelled through this table. The NameToPerson relationship is the main link between Person and Name. This makes Name a *weak entity*, because a name has no sense without its owner. The MainName relationship is the cleanest way of ensuring that each Person has *exactly* one main name. This leads to the first out-of-model constraint defined in [§2.2.3].

Title, MainTitle and TitleToProduction

The entity set Title corresponds to the *"ALTERNATIVE_TITLE"* file described in [§2.1.2]. We removed the "alternative" part in a similar fashion as for Name, described above. The TitleToProduction relationship is the main link between Production and Title. This makes Title a *weak entity*, because a title has no sense without its owning production. The MainTitle relationship is the cleanest way of ensuring that each Production has *exactly* one main title. This leads to the second out-of-model constraint defined in [§2.2.3].

field gender of Person

We chosed to not export this field as a separate entity set, despite what was described in [§2.1.1], because this is a single letter which can be managed by the applicative part of the project. This alternative is more efficient than setting a foreign key and the indexes it involves just for a single character. Thus, searching for all women in the Person entity set is just searching through the indexed column gender for all 'f'. If we did it with a separate entity set, we had to search through it for the 'id' of 'f' (let's say, 1), then doing a similar search than before but for all '1'.

aggregation around Casting

We made Casting a 3-ary relationship, linked to Character through PlayCharacter, instead of making Casting a 4-ary, because not all instances of Casting are linked to a Character.

ISA architecture

As suggested in [§2.1.2], the best way to model the different kinds of productions is to use a ISA hierarchy. Here we chosed to separate "singles movies" into SingleProduction and apply a covering constraint on the ISA hierarchy, to make things clearer into the resulting database. We also modelled the "seasons" concept as a separate entity set, not part of the ISA hierarchy (because in the original *"PRODUCTION"* file the seasons are just discerned by numbers in a column, they do not have rows for them), so things are clearly hierarchized, thus making the future queries easier to build.

3. Logical design

In this chapter, we translate the ER schema defined in [§2.2] into a purely relational schema, thus choosing data types (domains) for the attributes and determining whether relationship sets becomes foreign keys or associative tables. This schema is then implemented as a MySQL database.

3.1. Relational schema

Here, we describe the relational schema in a textual form, then justify our main choices about the translation of the ER schema into this relational schema.

3.1.1. Textual schema

This schema is given by alphabetic order of the relations names. Underlined attributes corresponds to the primary key for the corresponding table. Foreign keys are indicated under each relation description in green. Unicity constraints are indicated under each relation in blue. Attributes in bold are mandatory ones; ones not in bold can be omitted (NULL authorized).

casting(***id***: INT, ***person_id***: INT, ***production_id***: INT, ***role_id***: INT, *character_id*: INT)
person_id references *person.id*
production_id references *production.id*
role_id references *role.id*
character_id references *character.id*
(*person_id*, *production_id*, *role_id*) must be unique

character(***id***: INT, ***name***: VARCHAR(255))
(*name*) must be unique

company(***id***: INT, ***name***: VARCHAR(255), *country_id*: INT)
country_id references *country.id*
(*name*, *country_id*) must be unique

companytype(***id***: INT, ***company_id***: INT, ***type_id***: INT)
company_id references *company.id*
type_id references *type.id*
(*company_id*, *type_id*) must be unique

country(***id***: INT, ***code***: VARCHAR(2))
(*code*) must be unique

episode(***id***: INT, *number*: INT, ***season_id***: INT)
season_id references *season.id*
id references production.id (part of the ISA hierarchy)
(*number*, *season_id*) must be unique

gender(***id***: INT, ***name***: VARCHAR(255))
(*name*) must be unique

kind(***id***: INT, ***name***: VARCHAR(255))
(name) must be unique

name(***id***: INT, *firstname*: VARCHAR(255), ***lastname***: VARCHAR(255), ***person_id***: INT)
person_id references *person.id*
(*firstname*, *lastname*, *person_id*) must be unique

person(***id***: **INT**, ***gender***: **VARCHAR(1)**, *trivia*: *TEXT*, *quotes*: TEXT, *birthdate*: DATE, *deathdate*: DATE, *birthname*: TEXT, *minibiography*: TEXT, *spouse*: VARCHAR(255), *height*: FLOAT, ***name_id***: **INT**)
name_id references *name.id*
(*name_id*) must be unique

production(***id***: **INT**, *year*: YEAR, ***title_id***: **INT**, *gender_id*: INT)
title_id references *title.id*
gender_id references *gender.id*
(*title_id*) must be unique

productioncompany(***id***: **INT**, ***production_id***: **INT**, ***company_id***: **INT**)
production_id references *production.id*
company_id references *company.id*
(*production_id*, *company_id*) must be unique

role(***id***: **INT**, ***name***: **VARCHAR(255)**)
(*name*) must be unique

season(***id***: **INT**, *number*: INT, ***serie_id***: **INT**)
serie_id references *serie.id*
(*number*, *serie_id*) must be unique

serie(***id***: **INT**, *yearstart*: YEAR, *yearend*: YEAR)
id references *production.id* (part of the ISA hierarchy)

singleproduction(***id***: **INT**, ***kind_id***: **INT**)
id references *production.id* (part of the ISA hierarchy)
kind_id references kind.id

title(***id***: **INT**, ***title***: **VARCHAR(255)**, ***production_id***: **INT**)
production_id references *production.id*
(*title*, *production_id*) must be unique

type(***id***: **INT**, ***name***: **VARCHAR(255)**)
(*name*) must be unique

3.1.2. Account for translation choices

We made the following main choices when translating the conceptual schema into a relational one.

casting and its aggregation

The Casting relationship defined in the ER schema, and its aggregation used in the PlayCharacter relationship upon the Character entity, are translated to a single associative relation *casting* in the above relational schema, with the particularity that the *character_id* field can be NULL, meaning that no PlayCharacter instance exist for the given Casting instance. This seems more efficient to group them both in a single relation instead of having 2 different relations with more foreign keys to manage.

use of *id* fields in the associative relations

In the *casting*, *companytype* and *productioncompany* relations, an *id* field is always introduced, totally inexistent in the original data given as CSV files, declared as the primary keys for these relations. Thus, a unicity constraint is set over the foreign keys involved in the relation. It is generally

admitted that it is better when designing a database to associate each relation a single, one-field primary key than composing complex primary keys with multiple fields, even if this seems more logical to do. The reason for that is that when we want to improve the database, adding more relations into it, it is easier having only one field to use as a foreign key to reference another relation than to have several fields to use as a foreign key.

data types

For all key fields (primary and obviously foreign), the data type is INT, which seems logical and non-controversing in most cases. In particular, all "id" fields given in the CSV data files are numerical, and for the primary keys that are to be generated during data import (those of associative relations) MySQL has a functionality called "AUTO_INCREMENT" that acts only on INT type and its variants (UNSIGNED, BIGINT, TINYINT, ...) We can work only with INT UNSIGNED, having $\sim 2^{32} \cong 4 \cdot 10^9$ possible values, as all given data files have at most 10's of millions of entries.

Then, MySQL has special, dedicated data types for things like dates and years. In particular, by using the type DATE for data-like fields, it become easier in the queries to make calculations on those fields.

The textual fields are then divided into 3 groups.

- When the length can be arbitrarily long (biography, quotes, etc…) and no specific queries have to be executed on these fields, they are assigned the type TEXT, which can contain (theoretically) infinite texts, but for which indexation is hard to configure (because index must be specified on a maximum length, which cannot be infinite like the size of the corresponding field…) ;
- when the length is supposed to be quite limited (names, titles, etc…) and the fields appear to be used in specific queries of milestones 2 and 3, they are assigned the type VARCHAR(255), which can contain up to 255. It is then easy to put indexes on these fields. The exact value 255 is the more commonly used for such field, because it is quite large but is the larger number that can be encoded on 8 bits. As the length of a VARCHAR is stored with each instance of the corresponding attribute, 255 is thus the larger number of characters we can count without requiring adding a second metadata byte to count numbers >255 ;
- when the length is precisely known (1 for the gender of persons, 2 for country codes, …), the type VARCHAR(x) with x the precisely known length is assigned.

translation of the ISA hierarchy

We chosed to implement the ISA hierarchy described in the ER schema by keeping all entities involved into it (the parent as well as the children), linking them by their primary keys *id*. While it is a covering, non-overlapping ISA architecture, the parent entity set (Production) has its own attributes and is involved in external relationships, in which in particular ProductionCompany is of cardinality N:N, thus complicating things if we only keep the children relations.

3.1.3. Unacknowledged constraints

In this relational schema, lots of constraints previously specified are not present, and thus need to be acknowledged here before writing down the SQL code to create tables.

- The 2 out-of-model constraints declared for ER schema in [§2.2.3], which will be implemented by the application as no database mechanism can ensure that ;

- all weak entities of the ER schema (Name, Title, Season, Episode) will be ensured *weak* in the SQL code through the "ON DELETE CASCADE" directives ;
- the non-overlapping property of the ISA architecture will have to be checked by the application.

3.2. DDL SQL code

Hereafter is the SQL code that creates the tables and sets up the links (foreign keys) between them, compatible with the MySQL DBMS. The creation of the schema and database, as well as the management of the users and their rights to access database is not discussed here.

3.2.1. MySQL DDL code

```
-- 1. Create tables with unlinked columns, primary keys, unique indexes and simple indexes on
future foreign keys

CREATE TABLE `name` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `firstname` VARCHAR(255) NULL,
    `lastname` VARCHAR(255) NOT NULL,
    `person_id` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_first_last_person` (`firstname`, `lastname`, `person_id`)
);

CREATE TABLE `person` (
    `id` INT UNSIGNED,
    `gender` VARCHAR(1) NOT NULL,
    `trivia` TEXT NULL,
    `quotes` TEXT NULL,
    `birthdate` DATE NULL,
    `deathdate` DATE NULL,
    `birthname` TEXT NULL,
    `minibiography` TEXT NULL,
    `spouse` VARCHAR(255) NULL,
    `height` FLOAT NULL,
    `name_id` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_main_name` (`name_id`)
);

CREATE TABLE `role` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `name` VARCHAR(255) NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_name` (`name`)
);

CREATE TABLE `character` (
    `id` INT UNSIGNED,
    `name` VARCHAR(255) NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_name` (`name`)
);

CREATE TABLE `production` (
    `id` INT UNSIGNED,
    `year` YEAR NULL,
    `title_id` INT UNSIGNED NOT NULL,
    `gender_id` INT UNSIGNED NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_main_title` (`title_id`),
    KEY `idx_gender` (`gender_id`)
);

CREATE TABLE `casting` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `person_id` INT UNSIGNED NOT NULL,
    `production_id` INT UNSIGNED NOT NULL,
    `role_id` INT UNSIGNED NOT NULL,
    `character_id` INT UNSIGNED NULL,
    PRIMARY KEY (`id`),
```

```
    UNIQUE KEY `un_person_prod_role` (`person_id`, `production_id`, `role_id`)
);

CREATE TABLE `title` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `title` VARCHAR(255) NOT NULL,
    `production_id` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_title_production` (`title`, `production_id`)
);

CREATE TABLE `gender` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `name` VARCHAR(255) NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_name` (`name`)
);

CREATE TABLE `company` (
    `id` INT UNSIGNED,
    `name` VARCHAR(255) NOT NULL,
    `country_id` INT UNSIGNED NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_name_country` (`name`, `country_id`)
);

CREATE TABLE `country` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `code` VARCHAR(2) NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_code` (`code`)
);

CREATE TABLE `type` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `name` VARCHAR(255) NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_name` (`name`)
);

CREATE TABLE `companytype` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `company_id` INT UNSIGNED NOT NULL,
    `type_id` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_company_type` (`company_id`,`type_id`)
);

CREATE TABLE `singleproduction` (
    `id` INT UNSIGNED,
    `kind_id` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    KEY `idx_kind` (`kind_id`)
);

CREATE TABLE `kind` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `name` VARCHAR(255) NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_kind_name` (`name`)
);

CREATE TABLE `season` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `number` INT NULL,
    `serie_id` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_serie_number` (`serie_id`, `number`)
);

CREATE TABLE `serie` (
    `id` INT UNSIGNED,
    `yearstart` YEAR NULL,
    `yearend` YEAR NULL,
    PRIMARY KEY (`id`)
);
```

```
CREATE TABLE `episode` (
    `id` INT UNSIGNED,
    `number` INT NULL,
    `season_id` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_season_number` (`season_id`, `number`)
);

CREATE TABLE `productioncompany` (
    `id` INT UNSIGNED AUTO_INCREMENT,
    `production_id` INT UNSIGNED NOT NULL,
    `company_id` INT UNSIGNED NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `un_production_company` (`production_id`, `company_id`)
);


-- 2. add all foreign keys constraints that are on schema

ALTER TABLE `name`
    ADD CONSTRAINT `fk_nametoperson` FOREIGN KEY (`person_id`) REFERENCES `person` (`id`)
        ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `person`
    ADD CONSTRAINT `fk_mainname` FOREIGN KEY (`name_id`) REFERENCES `name` (`id`)
        ON DELETE RESTRICT ON UPDATE CASCADE;

ALTER TABLE `production`
    ADD CONSTRAINT `fk_maintitle` FOREIGN KEY (`title_id`) REFERENCES `title` (`id`)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    ADD CONSTRAINT `fk_productionhasgender` FOREIGN KEY (`gender_id`)
        REFERENCES `gender` (`id`) ON DELETE SET NULL ON UPDATE CASCADE;

ALTER TABLE `casting`
    ADD CONSTRAINT `fk_casting_person` FOREIGN KEY (`person_id`) REFERENCES `person` (`id`)
        ON DELETE CASCADE ON UPDATE CASCADE,
    ADD CONSTRAINT `fk_casting_role` FOREIGN KEY (`role_id`) REFERENCES `role` (`id`)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    ADD CONSTRAINT `fk_casting_production` FOREIGN KEY (`production_id`)
        REFERENCES `production` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
    ADD CONSTRAINT `fk_casting_character` FOREIGN KEY (`character_id`)
        REFERENCES `character` (`id`) ON DELETE SET NULL ON UPDATE CASCADE;

ALTER TABLE `title`
    ADD CONSTRAINT `fk_titletoproduction` FOREIGN KEY (`production_id`)
        REFERENCES `production` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `company`
    ADD CONSTRAINT `fk_companyhascountry` FOREIGN KEY (`country_id`)
        REFERENCES `country` (`id`) ON DELETE SET NULL ON UPDATE CASCADE;

ALTER TABLE `companytype`
    ADD CONSTRAINT `fk_companyhastype_company` FOREIGN KEY (`company_id`)
        REFERENCES `company` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
    ADD CONSTRAINT `fk_companyhastype_type` FOREIGN KEY (`type_id`) REFERENCES `type` (`id`)
        ON DELETE RESTRICT ON UPDATE CASCADE;

ALTER TABLE `season`
    ADD CONSTRAINT `fk_seasonhasserie` FOREIGN KEY (`serie_id`) REFERENCES `serie` (`id`)
        ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `episode`
    ADD CONSTRAINT `fk_episodehasseason` FOREIGN KEY (`season_id`) REFERENCES `season` (`id`)
        ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `singleproduction`
    ADD CONSTRAINT `fk_singleproduction_has_kind` FOREIGN KEY (`kind_id`)
        REFERENCES `kind` (`id`) ON DELETE RESTRICT ON UPDATE CASCADE;

ALTER TABLE `productioncompany`
    ADD CONSTRAINT `fk_productioncompany_production` FOREIGN KEY (`production_id`)
        REFERENCES `production` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
    ADD CONSTRAINT `fk_productioncompany_company` FOREIGN KEY (`company_id`)
        REFERENCES `company` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;


-- 3. add foreign keys constraints relative to "ISA" architecture
```

```
ALTER TABLE `serie`
    ADD CONSTRAINT `fk_serie_isa_production` FOREIGN KEY (`id`) REFERENCES `production` (`id`)
        ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `episode`
    ADD CONSTRAINT `fk_episode_isa_production` FOREIGN KEY (`id`)
        REFERENCES `production` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `singleproduction`
    ADD CONSTRAINT `fk_singleproduction_isa_production` FOREIGN KEY (`id`)
        REFERENCES `production` (`id`) ON DELETE CASCADE ON UPDATE CASCADE;
```

3.2.2. Method for writing SQL DDL code

The above SQL code is highlighted by colours according to the following list.

Keywords — keywords of the language, defining tables and constraints

Identificators — names of tables, attributes, constraints, etc…
They are “backticks quoted”, as this is the way to ensure that if an identificator accidentally (or voluntarily…) contain a keyword of MySQL, it will not be interpreted as a keyword at all.

Data types — MySQL data types for each attributes
They are set according to the relational schema (see [§3.1.1]). Keys (primary and foreign) are defined as UNSIGNED to gain 1 bit, because keys are never negative.

Null cond. — These are constraints that specify whether a field can be set to NULL (no value) or not. MySQL has a default value when it is not specified, but assuming default values when building a script is always a bad idea, as default values might change from a version to another. For “id” fields, it is not necessary to specify “NOT NULL”, because it is a mandatory implication of the use of the “PRIMARY KEY” clause.

ON … clauses — These clauses specify what to do when a primary key referenced by a foreign is updated or its corresponding row is deleted from the referenced table.

AUTO_INCREMENT clauses are set on primary keys of associative tables, as well as exported tables that were originally redundant attributes, for which no data exist at all in the given CSV data files.

The “ON UPDATE” clauses are always set to CASCADE as this enable us (DBA) to migrate the database to other servers with possible implications need on the numerical format of the keys with some ease of data update.
The “ON DELETE” clauses are set according to the following method :

- If the foreign key is defined in a weak entity, referencing the weak relationship according to the ER schema, we use “ON DELETE CASCADE” because the weak entity cannot exist without the entity it is linked on, by definition ;
- if the foreign key is referencing a table that acts as a limited list of values (genders of productions, for example), we use “ON DELETE RESTRICT”, thus preventing the deletion of a value from the list when it is in use ;
- if the foreign key is defined for the purpose of an ISA hierarchy, we use “ON DELETE CASCADE”, as when the parent goes away, its children (representing other aspects of the same entity) must also go away ;

- when the foreign key can be set to NULL, we generally use “ON DELETE SET NULL”.

The creation of the foreign key constraints is entirely done separately at the end of the script for 2 major reasons.

- There are some circular relationships, that are a table A referencing a table B while the table B references the table A. As a foreign key cannot be created until the referenced table exist, at least one of the 2 foreign key in the case of a circular relationship must be created afterwards ;
- As for the above reason we need to create few foreign keys after corresponding tables have been created, it is then more readable to create them all after all tables have been created, otherwise we have lots of places to look at to find all foreign keys while reading the script.

The “UNIQUE KEY” clauses are created according to the relational schema (see [§3.1.1]).

For all fields that will be declared as foreign keys after the creation of the tables, and that are not already referenced in a “UNIQUE KEY” index, we create a simple “KEY” index during the table creation. We do that to ensure retro-compatibility of the script, because older versions of MySQL may not accept the creation of a foreign key if no index is defined on the corresponding field (for performance reasons[5]). Newer versions of MySQL automatically create this index if it does not exist, but by setting it explicitly we make our script compatible also with older versions.

3.2.3. Extended note about AUTO_INCREMENT clauses

It is said above that the AUTO_INCREMENT clause is set on all primary keys of tables which represents associative relationships of the ER schema, and on tables that have been created to suppress redundancy from some attribute fields, because such keys does not exist at all in the given CSV data files. The AUTO_INCREMENT clause thus ensure that distinct, continuous values are assigned automatically at all rows of the considered table when these rows are added.

For all other *id* fields (primary keys), there is not AUTO_INCREMENT clause because the value will be entered to correspond to what is in the CSV data files. This is the best way to design the first version of the database so that import of all CSV data is easy to do in the next milestone.

This advantage will thus turn into an inconvenient when the database is ready, because when wanting to add a row (for example a production) through the application interface, we will have to manually enter a primary key, or at least implement the application such that it automatically find a non-used value for the primary key and use it immediately.

To correct the situation, we will use DDL commands of the following form AFTER the data have been imported (to be done concretely in milestone 2…), so that all primary key fields of the database are then controlled by a AUTO_INCREMENT clause, facilitating management of data through the application.

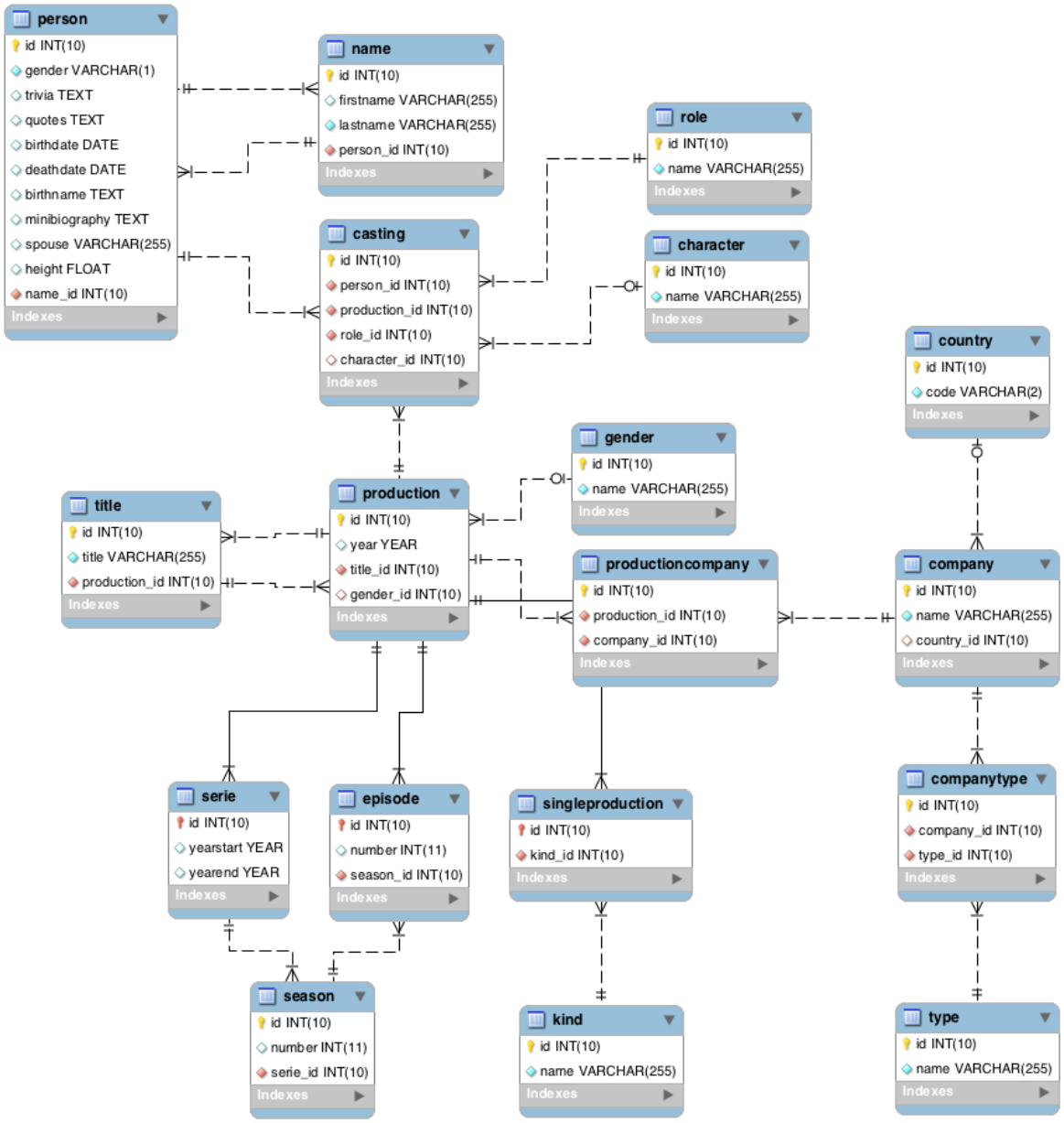
```
ALTER TABLE `table_name` CHANGE COLUMN `id` `id` INT UNSIGNED AUTO_INCREMENT;
```

The effect of such command on the primary key field *id* of the table *table_name* is to put on it a AUTO_INCREMENT controller, which starting value is automatically initialized to the current maximum value of the table plus one.

[5] http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html

3.2.4. Verification of the correctness of the script

After running the script to create all elements, we can use MySQL Workbench to generate[6] the corresponding visual relational schema in the MySQL formalism. The obtained schema is the following, visually confirming that everything seems right.



[6] http://dev.mysql.com/doc/workbench/en/wb-reverse-engineer-live.html