

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

EPFL  
CS-322 Introduction to database systems  
20.05.2015

## **IMDB Project**

---

**Final report**

## 0. Summary

---

<b>1. Introduction.....</b>	<b>3</b>
1.1. <i>Technologies.....</i>	3
<b>2. Conceptual design.....</b>	<b>5</b>
2.1. <i>Metadata and data analysis.....</i>	5
2.2. <i>Entity-Relationship schema (ER).....</i>	6
<b>3. Logical design .....</b>	<b>9</b>
3.1. <i>Relational schema .....</i>	9
3.2. <i>DDL SQL code.....</i>	12
<b>4. Data import .....</b>	<b>18</b>
4.1. <i>Changes made in database schema .....</i>	18
<b>5. Web application.....</b>	<b>20</b>
5.1. <i>Search functionality.....</i>	20
5.2. <i>Main pages .....</i>	24
5.3. <i>Inserting, updating and deleting entities .....</i>	27
<b>6. Simple queries (milestone 2) .....</b>	<b>36</b>
6.1. <i>SQL code .....</i>	36
6.2. <i>Application dedicated page.....</i>	40
<b>7. Interesting queries (milestone 3) .....</b>	<b>42</b>
7.1. <i>SQL code .....</i>	42
7.2. <i>Application dedicated page.....</i>	47
<b>8. Detailed queries analysis.....</b>	<b>51</b>
8.1. <i>Necessity of indexes.....</i>	51
8.2. <i>Distribution of the cost.....</i>	58
8.3. <i>Running times of all queries .....</i>	59
<b>9. Appendix A – Data import detailed report .....</b>	<b>61</b>
9.1. <i>Imported files.....</i>	61
9.2. <i>SQL DDL code before milestone 2 modification.....</i>	65
<b>10. Appendix B – Web application detailed.....</b>	<b>69</b>
10.1. <i>Major components .....</i>	69
10.2. <i>Main pages queries .....</i>	70
10.3. <i>Insert, update and delete procedures.....</i>	74

# 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*<sup>1</sup>.

## 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*<sup>2</sup> 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*<sup>3</sup>, 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*<sup>4</sup>, 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.

As already having experience in such kind of development, we decided to create a web application based on the following servers and tools.

- *Apache*<sup>5</sup> 2.4.12, as the web server
- *PHP*<sup>6</sup> 5.5.0, as the server-side programming technology
- *jQuery*<sup>7</sup> 2.1.3, as the client-side programming framework

---

<sup>1</sup> <http://www.imdb.com>

<sup>2</sup> <http://www.oracle.com/fr/database/overview/index.html>

<sup>3</sup> <http://dev.mysql.com/downloads/>

<sup>4</sup> <http://dev.mysql.com/downloads/workbench/>

<sup>5</sup> <http://httpd.apache.org>

<sup>6</sup> <http://php.net>

<sup>7</sup> <https://jquery.com>

- *Twitter's bootstrap*<sup>8</sup> 3.3.4, as the graphical enhancement framework

The application is strongly based on *AJAX requests*<sup>9</sup> to gather heavy data sets and execute long-running requests while informing user on the current progress status.

The development is enhanced by the ILARIA toolset, a powerful PHP framework developed by Jason Racine, one of our team's member. This framework provides the following operations to speed up development.

- Full MVC (Model-View-Controller) paradigm
- Internal management and handling of asynchronous requests
- MySQL high-level abstraction layer to ease SQL commands building and management
- Enhanced security and errors protection

---

<sup>8</sup> <http://getbootstrap.com>

<sup>9</sup> [http://fr.wikipedia.org/wiki/Ajax\\_\(informatique\)](http://fr.wikipedia.org/wiki/Ajax_(informatique))

## 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 clearly 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 that 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 “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, with a supplementary attribute for the type of company, that describes

some sort of “role” the company played in the production (production company or distributor). This attribute is thus also a good candidate to be exported as a separate entity set to reduce redundancy.

#### 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*”.

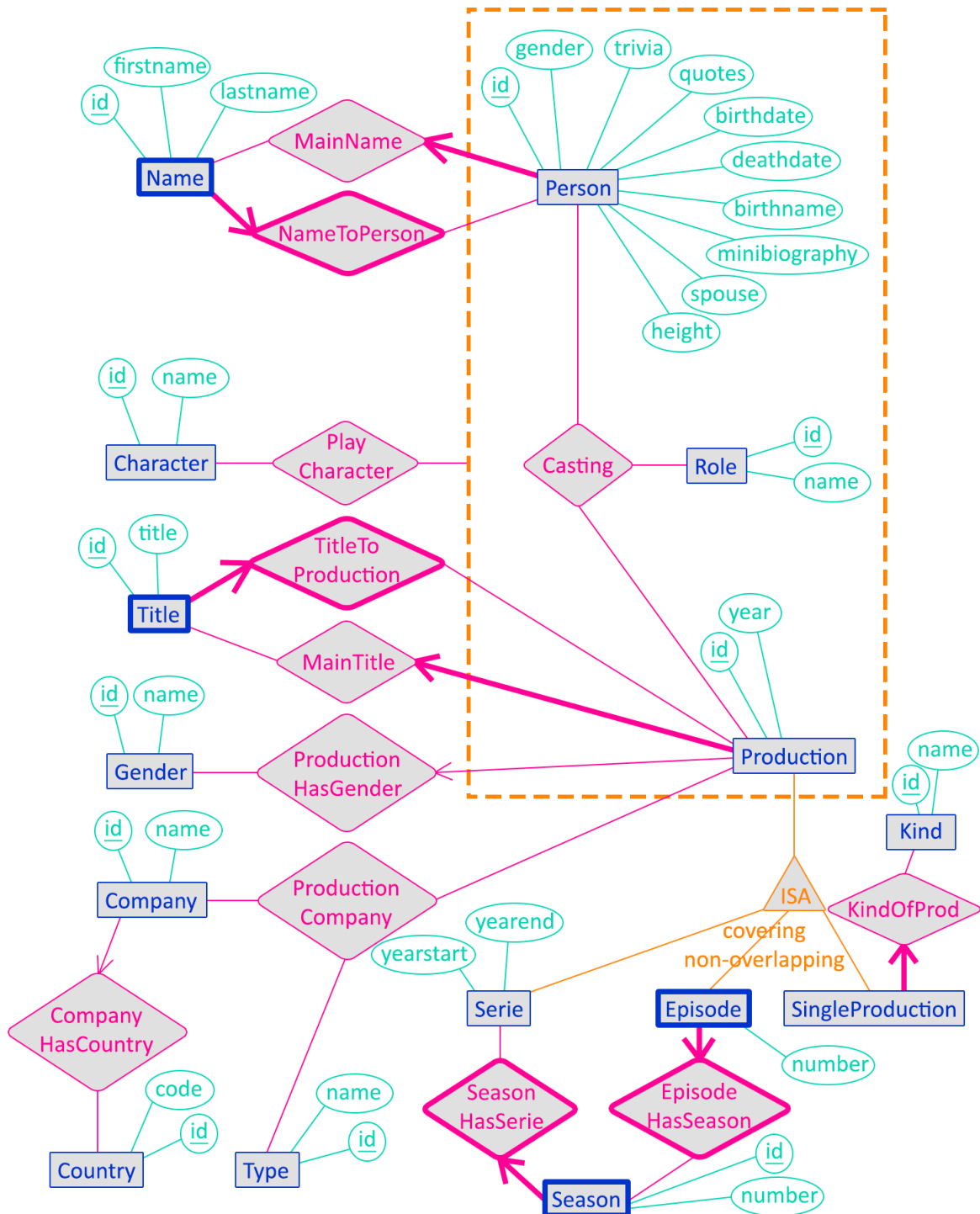
Finally, the “*CHARACTER*” file is just a listing of the characters played in movies. By making a quick test with the search function of a basic text editor into “*PRODUCTION\_CAST*”, it seems that some characters appears in multiple productions, either referenced by the exact same key or by having multiple 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 chose 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 chose 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

**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: INT, **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, **type\_id**: INT)

*production\_id* references *production.id*

*company\_id* references *company.id*

*type\_id* references *type.id*

(*production\_id*, *company\_id*, *type\_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: INT, yearend: INT)

*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* and *companytype* 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 that do not appear directly in CSV files) 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. When storing years, we originally used the YEAR data type, but as discussed in [§4.1.3], it was impossible to store pre-1900 values, so we changed it to INT to ensure everything is fine.

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) CHARACTER SET utf8 COLLATE utf8_bin NULL,  
  `lastname` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin 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) 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) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `un_name` (`name`)  
);  
  
CREATE TABLE `character` (  
  `id` INT UNSIGNED,  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `un_name` (`name`)  
);  
  
CREATE TABLE `production` (  
  `id` INT UNSIGNED,  
  `year` INT UNSIGNED 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_character` (`person_id`, `production_id`, `role_id`,  
    `character_id`)  
);
```

```
CREATE TABLE `title` (  
  `id` INT UNSIGNED AUTO_INCREMENT,  
  `title` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin 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) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `un_name` (`name`)  
);  
  
CREATE TABLE `company` (  
  `id` INT UNSIGNED,  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin 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(10) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `un_code` (`code`)  
);  
  
CREATE TABLE `type` (  
  `id` INT UNSIGNED AUTO_INCREMENT,  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL,  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `un_name` (`name`)  
);  
  
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) CHARACTER SET utf8 COLLATE utf8_bin 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` INT UNSIGNED NULL,  
  `yearend` INT UNSIGNED 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,
```

```
`production_id` INT UNSIGNED NOT NULL,  
`company_id` INT UNSIGNED NOT NULL,  
`type_id` INT UNSIGNED NOT NULL,  
PRIMARY KEY (`id`),  
UNIQUE KEY `un_production_company` (`production_id`, `company_id`, `type_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 `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,  
  ADD CONSTRAINT `fk_productioncompany_type` FOREIGN KEY (`type_id`)  
  REFERENCES `type` (`id`) ON DELETE RESTRICT 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.

<b>Keywords</b>	keywords of the language, defining tables and constraints
<b>Identifiers</b>	names of tables, attributes, constraints, etc... They are “backticks quoted”, as this is the way to ensure that if an identifier accidentally (or voluntarily...) contain a keyword of MySQL, it will not be interpreted as a keyword at all.
<b>Data types</b>	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.
<b>Null cond.</b>	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.
<b>ON ... clauses</b>	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.
<b>AUTO_INCREMENT</b>	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<sup>10</sup>). 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.

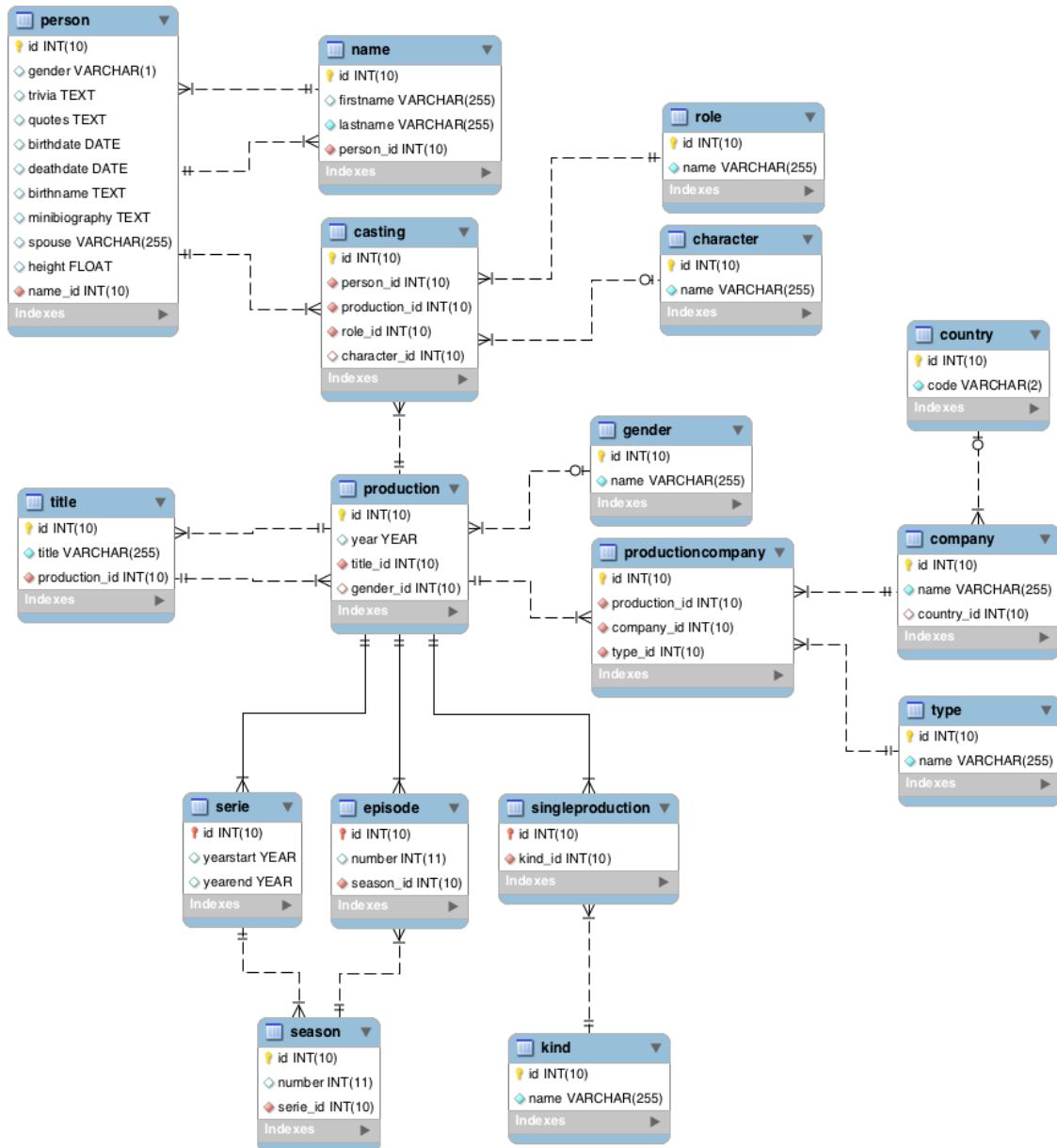
---

<sup>10</sup> <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<sup>11</sup> the corresponding visual relational schema in the MySQL formalism. The obtained schema is the following, visually confirming that everything seems right.



<sup>11</sup> <http://dev.mysql.com/doc/workbench/en/wb-reverse-engineer-live.html>

## 4. Data import

---

The data are initially given in CSV format. To import them, we built some PHP scripts reading these CSV files, computing intermediate transformations (like exploding a single CSV line into multiple relations records) and issuing SQL DML commands.

The detailed report for data importation can be found in Appendix A [\[§9.1\]](#).

### 4.1. Changes made in database schema

---

Some simple tests were done on small samples of each CSV file before any real import attempt, to detect most common error cases and build the scripts based on these observation. All these preliminary tests leaded to some database schema modifications and corrections, detailed below.

The modifications described below as executed during development are already implemented in the final DDL SQL code given in [\[§3.2\]](#). To show the DDL SQL code as it was before these changes were applied, refer to Appendix A [\[§9.2\]](#).

#### 4.1.1. Data types for indexed texts

---

As created by default, the VARCHAR fields appearing in indexes in order to be “searchable” suffered from a problem. The lower/upper case was taken in consideration when searching, which is not what we wanted. The following script was run, adding a particular collation to the concerned fields, in order to fix this problem.

```
ALTER TABLE `name` CHANGE COLUMN `firstname`  
  `firstname` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NULL;  
ALTER TABLE `name` CHANGE COLUMN `lastname`  
  `lastname` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;  
ALTER TABLE `role` CHANGE COLUMN `name`  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;  
ALTER TABLE `character` CHANGE COLUMN `name`  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;  
ALTER TABLE `title` CHANGE COLUMN `title`  
  `title` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;  
ALTER TABLE `gender` CHANGE COLUMN `name`  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;  
ALTER TABLE `company` CHANGE COLUMN `name`  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;  
ALTER TABLE `country` CHANGE COLUMN `code`  
  `code` VARCHAR(2) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;  
ALTER TABLE `type` CHANGE COLUMN `name`  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;  
ALTER TABLE `kind` CHANGE COLUMN `name`  
  `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;
```

#### 4.1.2. Length of country codes

---

By looking at the very first lines of the COMPANY.CSV file when designing database a few weeks ago, it seemed that all country codes were 2-characters long, ignoring the brackets. But slightly longer codes have been found by the automated inspection of this file before the import attempt, thus the size of this field has been enlarged to 10 by the following script.

```
ALTER TABLE `country` CHANGE COLUMN `code`  
  `code` VARCHAR(10) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL;
```

#### 4.1.3. Handling of pre-1900 values in YEAR-typed fields

---

It appeared that some years of productions (both the field **year** of the **production** table, and the fields **yearstart** and **yearend** of the **serie** table) were before 1900, thus cannot be stored by the specific **YEAR** MySQL format previously defined for these fields. This format has been changed to simple **INT UNSIGNED** by the following script.

```
ALTER TABLE `production` CHANGE COLUMN `year`  
  `year` INT UNSIGNED NULL;  
ALTER TABLE `serie` CHANGE COLUMN `yearstart`  
  `yearstart` INT UNSIGNED NULL;  
ALTER TABLE `serie` CHANGE COLUMN `yearend`  
  `yearend` INT UNSIGNED NULL;
```

#### 4.1.4. *Changed UNIQUE index in casting*

---

By going through the casting table during inspection before import attempt, it was discovered that the same person could act as multiple characters in the same production. Thus, the unique key `un_person_prod_role` in the casting table was extended to the `character_id` field by the following script.

```
ALTER TABLE `casting` DROP INDEX `un_person_prod_role`;  
ALTER TABLE `casting` ADD UNIQUE INDEX `un_person_prod_role_character`  
  (`person_id`, `production_id`, `role_id`, `character_id`);
```

## 5. Web application

---

This chapter explains how the web application is built, what are the main queries used to make it work, and details the functionalities offered to the user.

In Appendix B [§10] can be found all detailed information that is not crucial to the project comprehension but that is still interesting and useful to understand the work we did to develop the application.

Major components offered by the ILARIA framework are described in Appendix B [§10.1].

### 5.1. Search functionality

---

With a so large database to manage, the search functionality is considered as the core operation of the whole application. Thus it is designed to be as efficient as possible on a single laptop, reducing the primary search time over all “searchable areas” to a couple of seconds.

#### 5.1.1. Search queries

---

In this chapter we present with full details the search queries used.

##### Productions

The primary search query for finding productions is as follow (here we are searching for “hunger games”).

```
SELECT DISTINCT PR.`id`, TI_MAIN.`title`, PR.`year`, GE.`name` AS `gender`
FROM `production` PR
INNER JOIN (
    SELECT TI.`id`, TI.`production_id` AS `prod_id`
    FROM `title` TI
    WHERE TI.`title` COLLATE UTF8_GENERAL_CI LIKE "%hunger games%"
) TI_SEARCH ON PR.`id` = TI_SEARCH.`prod_id`
INNER JOIN `title` TI_MAIN ON PR.`title_id` = TI_MAIN.`id`
LEFT JOIN `gender` GE ON PR.`gender_id` = GE.`id`
GROUP BY PR.`id`
ORDER BY PR.`year` DESC, TI_MAIN.`title` ASC;
```

By returning the production’s ID in the result, it is then possible to add a link on the graphical result lines, redirecting to the production’s main page. The search is done on all titles (main and alternative), but the resulting productions are displayed by their main titles only.

##### Persons

The primary search query for finding persons is as follow (here we are searching for “jennifer lawrence”).

```
SELECT DISTINCT PE.`id`, NA_MAIN.`lastname`, NA_MAIN.`firstname`, PE.`birthdate`,
PE.`deathdate`
FROM `person` PE
INNER JOIN `name` NA_SEARCH ON PE.`id` = NA_SEARCH.`person_id`
INNER JOIN `name` NA_MAIN ON PE.`name_id` = NA_MAIN.`id`
WHERE (
    NA_SEARCH.`lastname` COLLATE UTF8_GENERAL_CI LIKE "%jennifer%"
    OR NA_SEARCH.`lastname` COLLATE UTF8_GENERAL_CI LIKE "%lawrence%"
) AND (
    NA_SEARCH.`firstname` COLLATE UTF8_GENERAL_CI LIKE "%jennifer%"
    OR NA_SEARCH.`firstname` COLLATE UTF8_GENERAL_CI LIKE "%lawrence%"
)
GROUP BY PE.`id`
ORDER BY NA_MAIN.`lastname` ASC, NA_MAIN.`firstname` ASC;
```

By returning the person's ID in the result, it is then possible to add a link on the graphical result lines, redirecting to the person's main page. The search is done on all names (main and alternative), but the resulting persons are displayed by their main names only.

In the above example, the original search text was "jennifer lawrence". In the live application, any input string is broken down by using space as separator, and a customized query is built dynamically, based on the above sample structure, adding a "OR" clause to each part of the main "AND" clause for every additional name. When only a single word is specified, the query is simplified to a simple OR between **firstname** and **lastname** on this single word.

## Characters

The primary search query for finding characters is as follow (here we are searching for "everdeen").

```
SELECT DISTINCT CH.`id`, CH.`name`, COUNT(DISTINCT CA.`person_id`) AS `persons_count`,  
COUNT(DISTINCT CA.`production_id`) AS `productions_count`  
FROM `character` CH  
INNER JOIN `casting` CA ON CH.`id`=CA.`character_id`  
WHERE CH.`name` COLLATE UTF8_GENERAL_CI LIKE "%everdeen%"  
GROUP BY CH.`id`  
ORDER BY CH.`id`;
```

When characters are found, we use their IDs on a button click to open a modal window showing a list of persons that played that character, retrieved by the following secondary search query. Here the ID is 2604958 corresponding to Katniss Everdeen.

```
SELECT DISTINCT PE.`id`, NA.`firstname`, NA.`lastname`  
FROM `person` PE  
INNER JOIN `name` NA ON PE.`name_id` = NA.`id`  
INNER JOIN `casting` CA ON PE.`id` = CA.`person_id`  
WHERE CA.`character_id` = 2604958;
```

In a similar way, we can open a modal window showing a list of productions in which this character appeared, retrieved by the following secondary search query.

```
SELECT DISTINCT PR.`id`, TI.`title`, PR.`year`  
FROM `production` PR  
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`  
INNER JOIN `casting` CA ON PR.`id` = CA.`production_id`  
WHERE CA.`character_id` = 2604958;
```

The 2 above queries are returning person's, respectively production's ID, allowing us to put buttons on interface leading to the corresponding person's or production's main page.

## Companies

The primary search query for finding companies is as follow (here we are searching for "lionsgate").

```
SELECT DISTINCT COM.`id` AS `id`, COM.`name` AS `name`, COU.`code` AS `country`,  
COUNT(DISTINCT PC_PROD.`production_id`) AS `produced_count`, COUNT(DISTINCT  
PC_DIST.`production_id`) AS `distributed_count`  
FROM `company` COM  
LEFT JOIN `country` COU ON COM.`country_id` = COU.`id`  
INNER JOIN `productioncompany` PC_PROD ON COM.`id` = PC_PROD.`company_id`  
AND PC_PROD.`type_id` = (  
SELECT `id` FROM `type` WHERE `name`="production companies"  
)  
INNER JOIN `productioncompany` PC_DIST ON COM.`id` = PC_DIST.`company_id`  
AND PC_DIST.`type_id` = (  
SELECT `id` FROM `type` WHERE `name`="distributors"  
)  
WHERE COM.`name` COLLATE UTF8_GENERAL_CI LIKE "%lionsgate%"  
GROUP BY COM.`id`  
ORDER BY COM.`name`;
```

When companies are found, we use their IDs on a button click to open a modal window showing a list of movies in which the company is involved as a production company, retrieved by the following secondary search query. Here the ID is 3293, corresponding to Lionsgate [us].

```
SELECT DISTINCT PR.`id` AS `id`, TI.`title` AS `title`, PR.`year` AS `year`
FROM `production` PR
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`
INNER JOIN `productioncompany` PC ON PR.`id` = PC.`production_id`
INNER JOIN `type` TY ON PC.`type_id` = TY.`id`
WHERE TY.`name`="production companies"
      AND PC.`company_id`=3293
ORDER BY PR.`year` DESC, TI.`title` ASC;
```

In the application, the same query is used a second time for a twin button, leading to the list of movies in which the company is involved as a distributor. The search parameter “production companies” is simply replaced by “distributors” in the above query.

## Genders

The primary search query for finding genders of movies is as follows (here we are searching for “action”).

```
SELECT DISTINCT GE.`id` AS `id`, GE.`name` AS `name`, COUNT(DISTINCT PR.`id`) AS
`count_prod`
FROM `gender` GE
INNER JOIN `production` PR ON GE.`id` = PR.`gender_id`
WHERE GE.`name` COLLATE UTF8_GENERAL_CI LIKE "%action%"
GROUP BY GE.`id`
ORDER BY GE.`name`;
```

When genders are found, we use their IDs on a button click to open a modal window showing a list of movies having this gender, retrieved by the following secondary search query. Here the ID is 19, corresponding to action.

```
SELECT DISTINCT PR.`id` AS `id`, TI.`title` AS `title`, PR.`year` AS `year`
FROM `production` PR
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`
INNER JOIN `gender` GE ON PR.`gender_id` = GE.`id`
WHERE GE.`id`=19
ORDER BY PR.`year` DESC, TI.`title` ASC;
```

### 5.1.2. Graphical interface

---

The search form is composed of 2 parts.

- A “simple query” can be done, searching simultaneously through all the primary search queries presented above.
- An “advanced query” can be done, specifying which primary search queries must be launched.

These 2 parts are presented to the user in a collapsible accordion for better visual comprehension of the search possibilities.

The image shows two search forms. The 'Simple search' form has a 'Search for' label, a text input field with placeholder 'Text to search', and a 'Search' button. The 'Advanced search' form has a 'Search for' label, a text input field with placeholder 'Text to search', a 'Search in' label, and a list of checkboxes: 'productions', 'persons', 'characters', 'companies', and 'genres'. There is also a 'Search' button at the bottom.

*The 2 search forms expanded for the purpose of this report*

When a simple search is done, a screen like the following is displayed.

Search results for "everdeen"

Productions			results 1-1 (total 1)
Title	Year	Genre	
<a href="#">Make-up Tutorial: Katniss Everdeen Look</a>	2010		

Persons			results 1-1 (total 1)
Name	Birthdate	Deathdate	
<a href="#">Larina Jean Adamson</a>			

Characters			results 1-7 (total 7)
Name	Actors	Movies	
Fatness Everdeen	<a href="#">1</a>	<a href="#">1</a>	
Mr. Everdeen	<a href="#">1</a>	<a href="#">1</a>	
Katniss Everdeen	<a href="#">8</a>	<a href="#">14</a>	
Victoria Everdeen	<a href="#">1</a>	<a href="#">1</a>	
Primrose Everdeen	<a href="#">4</a>	<a href="#">8</a>	
Katniss Everdeen in 'The Hunger Games: Catching Fire'	<a href="#">1</a>	<a href="#">4</a>	
Fatness Foreverdeen	<a href="#">2</a>	<a href="#">2</a>	

Companies				results 0-0 (total 0)
Name	Country	Produced	Distributed	

Genres		results 0-0 (total 0)
Name	Movies	

*A result screen sample for "everdeen"*

Each of the 5 result sections is an AJAX-loaded data array, linked to the corresponding primary search query.

In the case of an advanced search, only sections that are checked on the advanced form will be displayed on the result page and thus loaded by AJAX queries.

In the characters, companies and genres result arrays, grey buttons with counts (for actors and movies, in the case of characters as shown on above picture) are calling secondary search queries and loading the result in the modal window.

In the persons, productions and companies result arrays, blue buttons with arrows are redirecting to the main pages for these elements.

## 5.2. Main pages

The application offers main pages for viewing the details of productions, persons and companies. Other entities don't have dedicated main pages because the search functionality's various modal windows are sufficient to accessing all their details efficiently and conveniently.

### 5.2.1. Person's main page

The following picture presents the main page of a person.

### Jennifer Lawrence

Gender	Woman
Birthdate	1990-08-15
Deathdate	-
Birthname	Lawrence, Jennifer Shrader
Spouse	-
Height	1.75 m

Alternative names results 1-3 (total 3) 1

+ insert

Last name	First name	update	delete
Jen			
JLaw			
Lawrence	Jennifer Shrader		

#### Trivia

She has English, German, Irish, Scottish, and remote French ancestry.

#### Quotes

[on her experiences on the set of \_The Hunger Games: Catching Fire (2013)\_ (qv)] Everybody told me there were no spiders, so when I saw three, I started crying. Jungles are not easy when you're afraid of everything. I think I am a legitimate alcoholic. No, what's it called? An arachnophobic.

### Biography

Academy Award-winning actress Jennifer Lawrence, best-known for playing Katniss Everdeen in \_The Hunger Games (2012)\_ (qv), was born in Louisville, Kentucky on August 15, 1990, to Karen (Koch), who manages a children's camp, and Gary Lawrence, who works in construction. She has two older brothers, Ben and Blaine, and has English, as well as some German, Irish, and Scottish, ancestry. Before Jennifer became an actress, she was involved in cheer-leading, field hockey, softball, and modelling, none of which she held a passion for. Her career began when she traveled to Manhattan at the age of 14. After conducting her first cold read, agents told her mother that "it was the best cold read by a 14- year-old they had ever heard", and tried to convince her mother that she needed to spend the summer in Manhattan. After leaving the agency, Jen was spotted by an agent in the midst of shooting an H&M ad and asked to take her picture. The next day, that agent followed up with her and invited her to the studio for a cold read audition. Again, the agents were highly impressed and strongly urged her mother to allow her to spend the summer in New York City. As fate would have it, she did, and subsequently appeared in commercials such as MTV's "My Super Sweet 16" and played a role in the movie, \_The Devil You Know (2013)\_ (qv). Shortly thereafter, her career forced her and her family to move to Los Angeles, where she was cast in the TBS sitcom \_"The Bill Engvall Show" (2007)\_ (qv), and in smaller movies like \_The Poker House (2008)\_ (qv) and \_The Burning Plain (2008)\_ (qv). Her big break came when she played Ree in \_Winter's Bone (2010)\_ (qv), which landed her Academy Award and Golden Globe nominations. Shortly thereafter, she secured the role of Mystique in franchise reboot \_X-Men: First Class (2011)\_ (qv), which went on to be a hit in Summer 2011. Around this time, Lawrence scored the role of a lifetime when she was cast as Katniss Everdeen in the big-screen adaptation of literary sensation \_The Hunger Games (2012)\_ (qv). That went on to become one of the highest-grossing movies ever with over \$407 million at the domestic box office, and instantly propelled Lawrence to the A-list among young actors/actresses. Three Hunger Games sequels are scheduled for release in November 2013, 2014, and 2015, with Lawrence reprising her role at least for the first one (\_The Hunger Games: Catching Fire (2013)\_ (qv)). In 2012 the romantic comedy Silver Linings Playbook earned her the Academy Award, Golden Globe Award, Screen Actors Guild Award, Satellite Award and the Independent Spirit Award for Best Actress, amongst other accolades, making her the youngest person ever to be nominated for two Academy Awards for Best Actress and the second-youngest Best Actress winner Lawrence can also be seen in \_The Beaver (2011)\_ (qv), \_Like Crazy (2011)\_ (qv), \_House at the End of the Street (2012)\_ (qv), and \_American Hustle (2013)\_ (qv).

### Played in movies

results 1-20 (total 70) 1 2 3 4

Role	Character	Title	Year	Genre	Kind
actress	Raven	X-Men: Apocalypse	2016	Adventure	movie
actress	Mystique	X-Men: Apocalypse	2016	Adventure	movie
actress	Joy Mangano	Joy	2015	Drama	movie
actress	Katniss Everdeen	The Hunger Games: Mockingjay - Part 2	2015	Sci-Fi	movie
actress	Herself - Nominee	19th Annual Critics' Choice Movie Awards	2014		tv movie
actress	Herself - Interviewee	2014 Golden Globe Arrivals Special	2014	Talk-Show	tv movie
actress	Herself - Winner	2014 MTV Movie Awards	2014	Comedy	tv movie

Jennifer Lawrence's main page, truncated vertically

The above picture is truncated for practical reasons. The "Played in movies" is 20 elements long, with pagination visible in the upper right corner. A similar list, with the explicit title "Played in series" is present just below it, thus clearly separating these 2 kinds of elements. Episodes are implicitly included in the series, with modal windows to accessing corresponding lists.

The "Played in movies", "Played in series" and "Alternative names" lists are AJAX-loaded, thus speeding up the main page loading and apparition of the main elements (name, birthdate, etc...).

Queries used to load this page's data are given in Appendix B [§10.2.1].



The following picture presents the main page of a production.

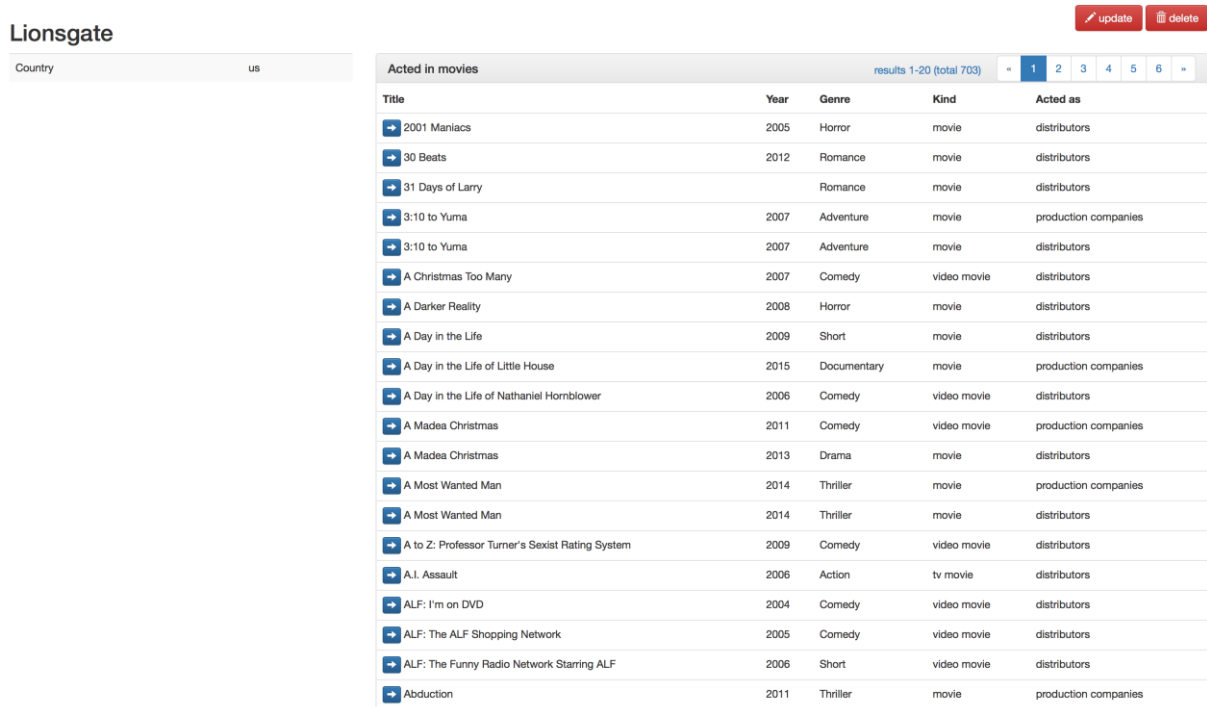
*The Hunger Game's main page*

The above main page presents the graphical configuration of a single production kind of production. In the case of an episode, the corresponding serie's name, season and episode numbers would appear in the upper right general infos tab. In the case of a serie, a fourth AJAX-loaded list is added, with the list of seasons and buttons to load using AJAX the lists of episodes in a season in the modal window.

Queries used to load this page's data are given in Appendix B [§10.2.2].

### 5.2.3. Company's main page

The following picture presents the main page of a company.



Lionsgate		Acted in movies				results 1-20 (total 703)	«	1	2	3	4	5	6	»
Country	us	Title	Year	Genre	Kind	Acted as								
		→ 2001 Maniacs	2005	Horror	movie	distributors								
		→ 30 Beats	2012	Romance	movie	distributors								
		→ 31 Days of Larry		Romance	movie	distributors								
		→ 3:10 to Yuma	2007	Adventure	movie	production companies								
		→ 3:10 to Yuma	2007	Adventure	movie	distributors								
		→ A Christmas Too Many	2007	Comedy	video movie	distributors								
		→ A Darker Reality	2008	Horror	movie	distributors								
		→ A Day in the Life	2009	Short	movie	distributors								
		→ A Day in the Life of Little House	2015	Documentary	movie	production companies								
		→ A Day in the Life of Nathaniel Horriblower	2006	Comedy	video movie	distributors								
		→ A Madea Christmas	2011	Comedy	video movie	production companies								
		→ A Madea Christmas	2013	Drama	movie	distributors								
		→ A Most Wanted Man	2014	Thriller	movie	production companies								
		→ A Most Wanted Man	2014	Thriller	movie	distributors								
		→ A to Z: Professor Turner's Sexist Rating System	2009	Comedy	video movie	distributors								
		→ A.I. Assault	2006	Action	tv movie	distributors								
		→ ALF: I'm on DVD	2004	Comedy	video movie	distributors								
		→ ALF: The ALF Shopping Network	2005	Comedy	video movie	distributors								
		→ ALF: The Funny Radio Network Starring ALF	2006	Short	video movie	distributors								
		→ Abduction	2011	Thriller	movie	production companies								

Lionsgate main page, vertically truncated

The above picture is truncated for practical reasons. The “Acted in movies” is 20 elements long, with pagination visible in the upper right corner. A similar list, with the explicit title “Acted in series” is present just below it, thus clearly separating these 2 kinds of elements. Episodes are implicitly included in the series, with modal windows to accessing corresponding lists.

The “Acted in movies” and “Acted in series” data arrays are AJAX-loaded.

Queries used to load this page's data are given in Appendix B [§].

### 5.3. Inserting, updating and deleting entities

In this chapter, we present the data management functionalities offered by the application. Full SQL queries are not given, mainly to save place on this document, but also because they are highly repeating and simple. Instead, we describe the procedures followed by application to insert complex things, including coherence verifications and error management.

Before any insertion is possible, we have to take a look to primary keys fields of the whole database. In chapter [§3.2.3] we discussed about the necessity of some primary keys to not be auto incremented during import phase, due to the fact that keys were already present in the CSV files.

Here is given the status of each primary key field in the database.

Table.field	Status
casting.id	Already has AUTO_INCREMENT
character.id	Need to be defined as AUTO_INCREMENT
company.id	Need to be defined as AUTO_INCREMENT
country.id	Already has AUTO_INCREMENT
episode.id	Do not need AUTO_INCREMENT, references production.id (ISA)
gender.id	Already has AUTO_INCREMENT
kind.id	Already has AUTO_INCREMENT
name.id	Already has AUTO_INCREMENT
person.id	Need to be defined as AUTO_INCREMENT
production.id	Need to be defined as AUTO_INCREMENT
productioncompany.id	Need to be defined as AUTO_INCREMENT
role.id	Already has AUTO_INCREMENT
season.id	Already has AUTO_INCREMENT
serie.id	Do not need AUTO_INCREMENT, references production.id (ISA)
singleproduction.id	Do not need AUTO_INCREMENT, references production.id (ISA)
title.id	Already has AUTO_INCREMENT
type.id	Already has AUTO_INCREMENT

The following script was used to apply changes on the database.

```
SET FOREIGN_KEY_CHECKS=0;
ALTER TABLE `character` CHANGE COLUMN `id` `id` INT UNSIGNED AUTO_INCREMENT;
ALTER TABLE `company` CHANGE COLUMN `id` `id` INT UNSIGNED AUTO_INCREMENT;
ALTER TABLE `person` CHANGE COLUMN `id` `id` INT UNSIGNED AUTO_INCREMENT;
ALTER TABLE `production` CHANGE COLUMN `id` `id` INT UNSIGNED AUTO_INCREMENT;
ALTER TABLE `productioncompany` CHANGE COLUMN `id` `id` INT UNSIGNED AUTO_INCREMENT;
SET FOREIGN_KEY_CHECKS=1;
```

The above script has to be run only as is when all data have been imported from the CSV files. Having these changes made on the original database before importing CSV files can lead to error, bad ID references and broken foreign keys.

### 5.3.1. Persons

The persons have a statistics page, from which it is possible to insert a new person.

#### Persons

Count

4857853

+ insert

*Statistics page for the persons, with the insert button*

When inserting a person, a form is presented to the user to fill in the data.

Last name	<input type="text" value="last name"/>
First name	<input type="text" value="first name"/>
Gender	<input type="text" value="-unknown-"/>
Birthdate	<input type="text" value="AAAA-MM-JJ"/>
Deathdate	<input type="text" value="AAAA-MM-JJ"/>
Birthname	<input type="text" value="birthname"/>
Trivia	<input type="text" value="short trivia"/>
Quotes	<input type="text" value="quotes"/>
Minibiography	<input type="text" value="short and concise biography"/>
Spouse	<input type="text" value="name of spouse"/>
Height	<input type="text" value="height in meters"/>
	<input type="button" value="Cancel"/> <input type="button" value="Validate"/>

*Form for filling in a person's details*

The update button is located in the upper right corner of the person's main page. The form used for this update is exactly the same as the one used for the insertion. Fields are automatically populated by the ILARIA framework, using basic SELECT queries.

The delete button is located in the upper right corner of the person's main page. It shows a modal window asking for the user to confirm what he wants to do.

Confirmation required

The person "Jennifer Lawrence" will be deleted. Are you sure ?

*Modal window asking for confirmation when trying to delete a person*

The insertion, update and deletion procedures can be found in Appendix B [\[§10.3.1\]](#).

### 5.3.2. Alternative names

When located on a person's main page, it is possible to manage the alternative names of this person.

Alternative names		results 1-3 (total 3)	«	1	»
		<a href="#">+ insert</a>			
Last name	First name				
Jen		<a href="#">update</a>	<a href="#">delete</a>		
JLaw		<a href="#">update</a>	<a href="#">delete</a>		
Lawrence	Jennifer Shrader	<a href="#">update</a>	<a href="#">delete</a>		

The alternative names list on the Jennifer Lawrence's main page

When inserting an alternative name, a form is presented to the user to fill in data.

Lastname	<input type="text" value="last name"/>
Firstname	<input type="text" value="first name"/>
<a href="#">Cancel</a>	<a href="#">Validate</a>

Form for filling in an alternative name

The form used for this update is exactly the same as the one used for the insertion. Fields are automatically populated by the ILARIA framework, using basic SELECT queries.

A modal window is used to ask for user confirmation.

Confirmation required

The alternative name " Jen " will be deleted. Are you sure ?

[Cancel](#) [Delete](#)

Modal window asking for confirmation when trying to delete an alternative name

The insertion, update and deletion procedures can be found in Appendix B [\[§10.3.2\]](#).

### 5.3.3. Productions

The productions have a statistics page, from which it is possible to insert a new singleproduction or serie.

Productions	
Kind	Count
Movies	1075664 <a href="#">+ insert</a>
Series	112648 <a href="#">+ insert</a>
Episodes	1991013 <a href="#">go to its serie's details page to add a new episode</a>

Statistics page for the production, with the insert buttons for singleproductions (movies) and series

When inserting a movie (singleproduction), the following form is presented to the user to fill in data.

**Title**

**Year**

**Genre**

**Kind**

*Form for filling in the basic details of a singleproduction*

When inserting a serie, the following form is presented to the user to fill in data.

**Title**

**Start year**

**End year**

**Genre**

*Form for filling in the basic details of a serie*

The insertion button for an episode is located in the upper right corner of the serie main page. Thus, this inserts an episode in a pre-chosen serie. The following form is presented to the user to fill in data.

**Title**

**Year**

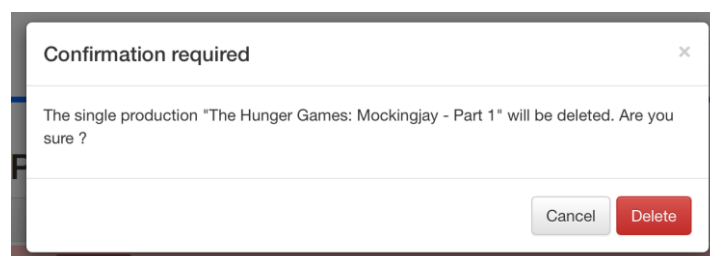
**Season #**

**Episode #**

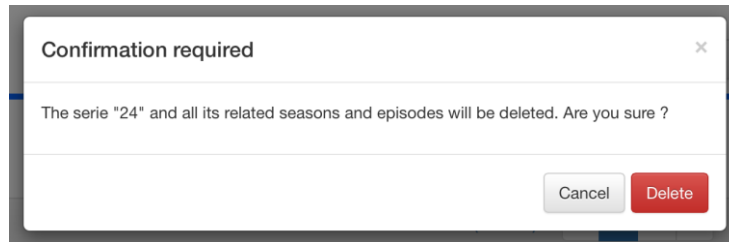
*Form for filling in the basic details of an episode*

The update button is located in the upper right corner of the production's main page. The form used for this update is exactly the same as the one used for the insertion of the corresponding kind of production (movie, serie or episode). Fields are automatically populated by the ILARIA framework, using basic SELECT queries.

The delete button is located in the upper right corner of the production's main page. It shows a modal window asking for the user to confirm what he wants to do.



*Modal window asking for confirmation when trying to delete a singleproduction*

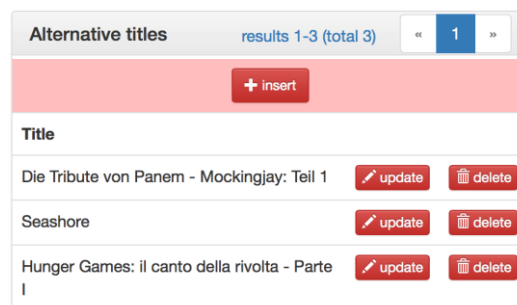


*Modal window asking for confirmation when trying to delete a serie*

The insertion, update and deletion procedures can be found in Appendix B [§10.3.3].

#### 5.3.4. Alternative titles

When located on a production's main page, it is possible to manage the alternative titles of this production.



*The alternative titles list on the Hunger Games Mockingjay main page*

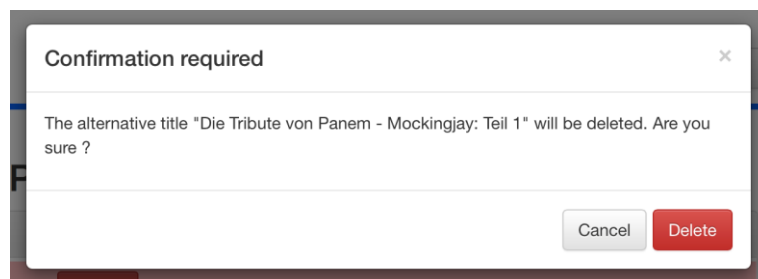
When inserting an alternative title, a form is presented to the user to fill in data.

**Title**

*Form for filling in an alternative title*

The form used for this update is exactly the same as the one used for the insertion. Fields are automatically populated by the ILARIA framework, using basic SELECT queries.

A modal window is used to ask for user confirmation.

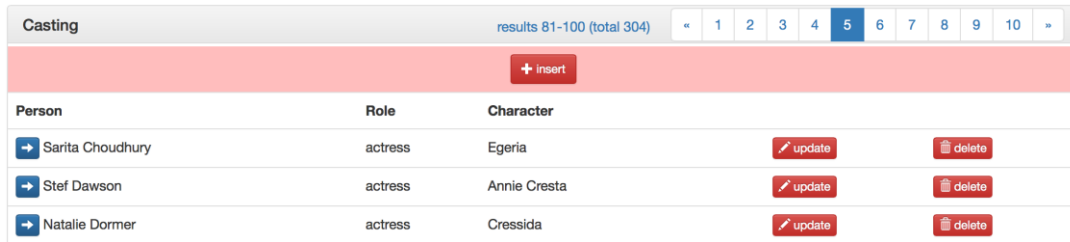


*Modal window asking for confirmation when trying to delete an alternative title*

The insertion, update and deletion procedures can be found in Appendix B [§10.3.4].

### 5.3.5. Casting

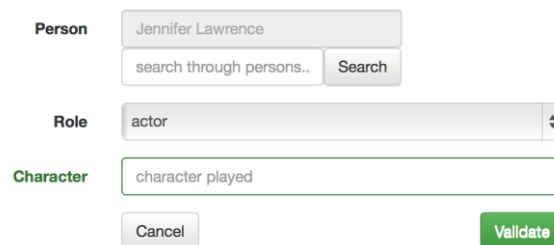
When located on a production's main page, it is possible to manage the casting of this production.



Person	Role	Character		
→ Sarita Choudhury	actress	Egeria		
→ Stef Dawson	actress	Annie Cresta		
→ Natalie Dormer	actress	Cressida		

*A part of the casting array on the Hunger Games Mockingjay main page*

When inserting a casting record, a form is presented to the user to fill in data.



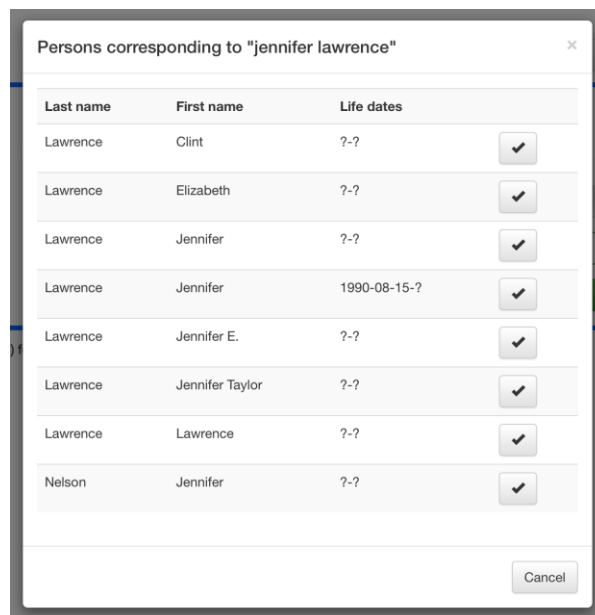
**Person**

**Role**

**Character**

*Form for filling in a casting record*

The field "Person" is special. The upper read-only box indicates the currently selected person (here Jennifer Lawrence). The "search through persons..." box and its "Search" button are used to search another person for this role, and triggers a modal windows to present results.



Persons corresponding to "jennifer lawrence"			
Last name	First name	Life dates	
Lawrence	Clint	?-?	<input checked="" type="checkbox"/>
Lawrence	Elizabeth	?-?	<input checked="" type="checkbox"/>
Lawrence	Jennifer	?-?	<input checked="" type="checkbox"/>
Lawrence	Jennifer	1990-08-15-?	<input checked="" type="checkbox"/>
Lawrence	Jennifer E.	?-?	<input checked="" type="checkbox"/>
Lawrence	Jennifer Taylor	?-?	<input checked="" type="checkbox"/>
Lawrence	Lawrence	?-?	<input checked="" type="checkbox"/>
Nelson	Jennifer	?-?	<input checked="" type="checkbox"/>

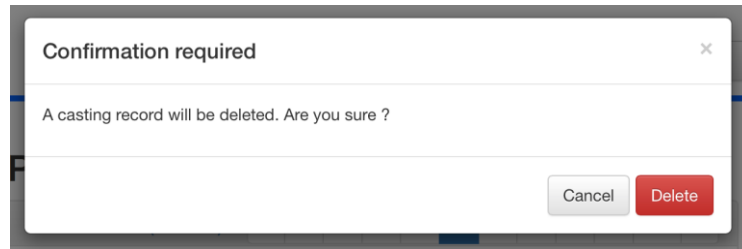
*Modal window showing persons corresponding to "jennifer lawrence"*

When clicking on a ☒ button in this modal window, the corresponding person is loaded in the form and the modal window is closed. This globally works like an intelligent drop-down menu.

The form used for this update is exactly the same as the one used for the insertion. Fields are automatically populated by the ILARIA framework, using basic SELECT queries.

A modal window is used to ask for user confirmation.





Modal window asking for confirmation when trying to delete a casting record

The insertion, update and deletion procedures can be found in Appendix B [§10.3.5].

### 5.3.6. Companies involved in a production










When located on a production's main page, it is possible to manage the companies involved in this production.

Companies involved

results 1-20 (total 21)

« 1 2 »

+ insert

Name	Country	Type		
 Cathay-Keris Films	sg	distributors	 update	 delete
 StudioCanal	de	distributors	 update	 delete
 Lionsgate	us	distributors	 update	 delete

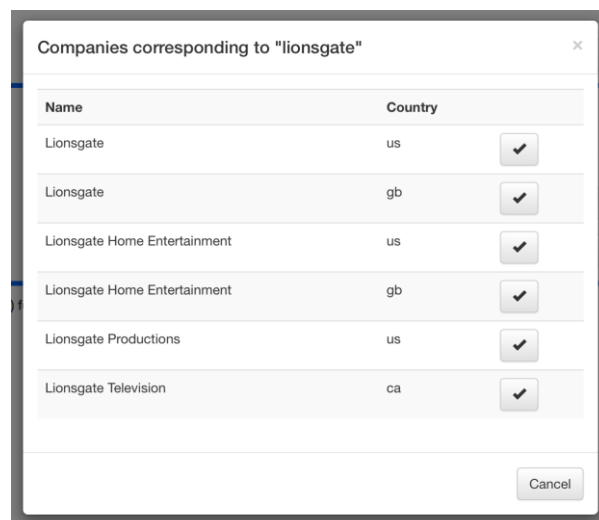
A part of the companies array on the Hunger Games Mockingjay main page

When inserting a productioncompany record, a form is presented to the user to fill in data.


Company:   
   
 Type:

Form for filling in a productioncompany record

The field "Company" is special. The upper read-only box indicates the currently selected company (here Lionsgate located in the United States). The "search through companies..." box and its "Search" button are used to search another company, and triggers a modal windows to present results.

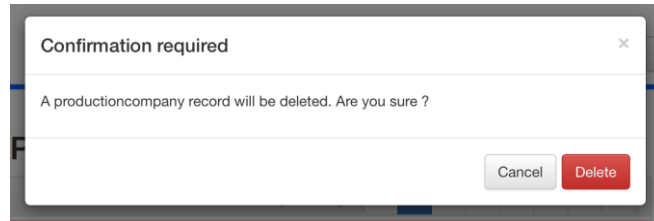


Modal window showing companies corresponding to "lionsgate"

When clicking on a  button in this modal window, the corresponding company is loaded in the form and the modal window is closed. This globally works like an intelligent drop-down menu.

The form used for this update is exactly the same as the one used for the insertion. Fields are automatically populated by the ILARIA framework, using basic SELECT queries.

A modal window is used to ask for user confirmation.

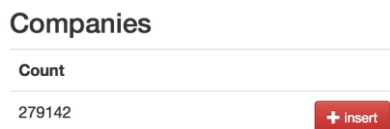


*Modal window asking for confirmation when trying to delete a productioncompany record*

The insertion, update and deletion procedures can be found in Appendix B [\[§10.3.6\]](#).

### 5.3.7. Companies

The companies have a statistics page, from which it is possible to insert a new company.



*Statistics page for the companies, with the insert button*

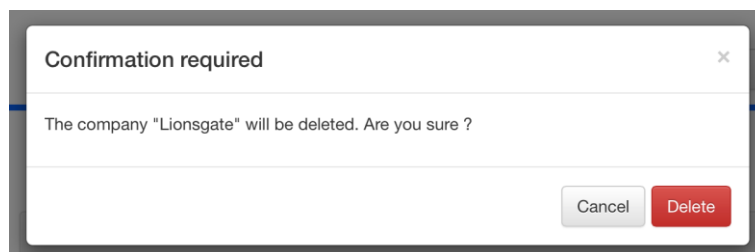
When inserting a company, a form is presented to the user to fill in the data.

A form with two input fields. The first field is labeled 'Name' in red and contains the placeholder text 'company name'. The second field is labeled 'Country' in green and contains the placeholder text 'country code'. Below the fields are two buttons: 'Cancel' and 'Validate'.

*Form for filling in a company's details*

The update button is located in the upper right corner of the company's main page. The form used for this update is exactly the same as the one used for the insertion. Fields are automatically populated by the ILARIA framework, using basic SELECT queries.

The delete button is located in the upper right corner of the company's main page. It shows a modal window asking for the user to confirm what he wants to do.



*Modal window asking for confirmation when trying to delete a company*

The insertion, update and deletion procedure can be found in Appendix B [\[§10.3.7\]](#).

### 5.3.8. Miscellaneous

A specific page is dedicated to the management of miscellaneous entities : roles, genres, types and kinds.

Person's roles	Movie's genres	Company's types	Movie's kinds
results 1-11 (total 11)	results 1-20 (total 27)	results 1-2 (total 2)	results 1-4 (total 4)
<div>+ insert</div>	<div>+ insert</div>	<div>+ insert</div>	<div>+ insert</div>
actor	Action	distributors	movie
actress	Adventure	production companies	tv movie
cinematographer	Animation		video game
<div>update delete</div>	<div>update delete</div>	<div>update delete</div>	<div>update delete</div>

*Management page for miscellaneous entities*

As the corresponding forms are very simple (always one single field to fill in), the details are not shown here. It is basically forbidden by the database itself ("ON DELETE RESTRICT" constraints) to delete one of these miscellaneous entities record when it is used by other records. Application will show a modal deny message when trying to do so.

## 6. Simple queries (milestone 2)

---

In this chapter, we present the SQL queries developed for the second milestone (called “simple query”).

### 6.1. SQL code

---

Here are the SQL codes for the simple queries.

#### 6.1.1. Query (a)

---

Original assignment was as follows.

*“Compute the number of movies per year. Make sure to include tv and video movies”*

Here is the corresponding SQL query we developed.

```
SELECT P.`year`, COUNT(P.`id`)
FROM `production` P
INNER JOIN `singleproduction` S ON P.`id` = S.`id`
WHERE S.`kind_id` IN (
    SELECT K.`id`
    FROM `kind` K
    WHERE K.`name` IN ("tv movie", "video movie", "movie")
)
GROUP BY P.`year`;
```

#### 6.1.2. Query (b)

---

Original assignment was as follows.

*“Compute the ten countries with most production companies”*

Here is the corresponding SQL query we developed.

```
SELECT COU.`id`, COU.`code`, SUB.`number`
FROM (
    SELECT COM.`country_id`, COUNT(DISTINCT COM.`id`) AS `number`
    FROM `company` COM
    INNER JOIN (
        SELECT PC.`company_id`
        FROM `productioncompany` PC
        INNER JOIN `type` TY ON PC.`type_id` = TY.`id`
        WHERE TY.`name` = "production companies"
    ) PC ON PC.`company_id` = COM.`id`
    GROUP BY COM.`country_id`
    HAVING COM.`country_id` IS NOT NULL
    ORDER BY `number` DESC
    LIMIT 10
) SUB
INNER JOIN `country` COU ON SUB.`country_id` = COU.`id`;
```

### 6.1.3. Query (c)

Original assignment was as follows.

*“Compute the min, max and average career duration”*

We developed an all-in-one SQL query that used to work but was too slow (about 6-8 minutes on a modern laptop).

```
SELECT MIN(T.`careerDuration`) AS `min`, MAX(T.`careerDuration`) AS `max`,  
       AVG(T.`careerDuration`) AS `avg`  
FROM (   
    SELECT (MAX(P.`year`) - MIN(P.`year`)) AS `careerDuration`  
    FROM (   
        SELECT DISTINCT C.`person_id`, C.`production_id`  
        FROM `casting` C  
    ) C  
    INNER JOIN (   
        SELECT P.`id`, P.`year`  
        FROM `production` P  
        WHERE P.`year` IS NOT NULL  
    ) P ON C.`production_id` = P.`id`  
    GROUP BY C.`person_id`  
 ) T;
```

To improve performances, we chose to design and exploit a materialized view. The following code is used to create the table.

```
CREATE TABLE m2c_careerduration (   
    person_id INT UNSIGNED,  
    careerDuration INT UNSIGNED,  
    PRIMARY KEY (`person_id`)  
 );
```

The following code is used to create the procedure that update this table.

```
DELIMITER $$  
CREATE PROCEDURE refresh_m2c_careerduration ()  
BEGIN  
    TRUNCATE TABLE m2c_careerduration;  
    INSERT INTO m2c_careerduration (`person_id`, `careerDuration`)  
    SELECT C.`person_id`, (MAX(P.`year`) - MIN(P.`year`)) AS `careerDuration`  
    FROM (   
        SELECT DISTINCT C.`person_id`, C.`production_id`  
        FROM `casting` C  
    ) C  
    INNER JOIN (   
        SELECT P.`id`, P.`year`  
        FROM `production` P  
        WHERE P.`year` IS NOT NULL  
    ) P ON C.`production_id` = P.`id`  
    GROUP BY C.`person_id`;  
END;  
$$  
DELIMITER ;
```

Then, the following call has to be made every time we want the materialized view to be updated.

```
CALL refresh_m2c_careerduration;
```

Finally, here is the SQL query that has to be launched, and run in very little time (couple of seconds), when we want to obtain data.

```
SELECT MIN(T.`careerDuration`) AS `min`, MAX(T.`careerDuration`) AS `max`,  
       AVG(T.`careerDuration`) AS `avg`  
FROM m2c_careerduration T;
```

#### 6.1.4. Query (d)

Original assignment was as follow.

*“Compute the min, max and average number of actors in a production”*

We developed an all-in-one SQL query that used to work but was too slow (about 2-3 minutes on a modern laptop).

```
SELECT MIN(T.`number`) AS `min`, MAX(T.`number`) AS `max`, AVG(T.`number`) AS `avg`
FROM (
  SELECT COUNT(C.`id`) AS `number`
  FROM `casting` C
  INNER JOIN `role` R ON C.`role_id` = R.`id`
  WHERE R.`name` = "actor"
  GROUP BY C.`production_id`
) T;
```

To improve performances, we chosed to design and exploit a materialized view. The following code is used to create the table.

```
CREATE TABLE m2d_nbactorproduction (
  production_id INT UNSIGNED,
  nb_actor INT UNSIGNED,
  PRIMARY KEY (`production_id`)
);
```

The following code is used to create the procedure that update this table.

```
DELIMITER $$
CREATE PROCEDURE refresh_m2d_nbactorproduction ()
BEGIN
  TRUNCATE TABLE m2d_nbactorproduction;
  INSERT INTO m2d_nbactorproduction (`production_id`, `nb_actor`)
  SELECT C.`production_id`, COUNT(C.`id`) AS `number`
  FROM `casting` C
  INNER JOIN `role` R ON C.`role_id` = R.`id`
  WHERE R.`name` = "actor"
  GROUP BY C.`production_id`;
END;
$$
DELIMITER ;
```

Then, the following call has to be made every time we want the materialized view to be updated.

```
CALL refresh_m2d_nbactorproduction;
```

Finally, here is the SQL query that has to be launched, and run in very little time (couple of seconds), when we want to obtain data.

```
SELECT MIN(T.`nb_actor`) AS `min`, MAX(T.`nb_actor`) AS `max`, AVG(T.`nb_actor`) AS `avg`
FROM m2d_nbactorproduction T;
```

#### 6.1.5. Query (e)

Original assignment was as follow.

*“Compute the min,max and average height of female persons”*

Here is the corresponding SQL query we developed.

```
SELECT MIN(P.`height`) AS `min`, MAX(P.`height`) AS `max`, AVG(P.`height`) AS `avg`
FROM `person` P
WHERE P.`height` IS NOT NULL
  AND P.`gender` = "f";
```

### 6.1.6. Query (f)

---

Original assignment was as follow.

*“List all pairs of persons and movies where the person has both directed the movie and acted in the movie. Do not include tv and video movies.”*

Here is the corresponding SQL query we developed.

```
SELECT DISTINCT C.person_id, C.production_id
FROM `casting` C
WHERE EXISTS
(
    SELECT CC.id
    FROM `casting` CC
    INNER JOIN `role` R
    ON CC.role_id = R.id
    WHERE CC.person_id = C.person_id
    AND CC.production_id = C.production_id
    AND R.name = "director"
    AND EXISTS
    (
        SELECT CC.id
        FROM `casting` CC
        INNER JOIN `role` R
        ON CC.role_id = R.id
        INNER JOIN `singleproduction` S
        ON CC.production_id = S.id
        INNER JOIN `kind` K ON S.`kind_id` = K.`id`
        WHERE K.`name` = "movie"
        AND CC.person_id = C.person_id
        AND CC.production_id = C.production_id
        AND R.name = "actor"
    )
)
);
```

### 6.1.7. Query (g)

---

Original assignment was as follow.

*“List the three most popular character names”*

Here is the corresponding SQL query we developed.

```
SELECT CH.`name`
FROM (
    SELECT CA.`character_id`, COUNT(CA.`id`) AS `number`
    FROM `casting` CA
    WHERE CA.`character_id` IS NOT NULL
    GROUP BY CA.`character_id`
    ORDER BY `number` DESC
    LIMIT 0,3
) T
INNER JOIN `character` CH ON T.`character_id` = CH.`id`;
```

## 6.2. Application dedicated page

The web application offers a complete page dedicated to the execution and the results visualization of these 7 simple queries.

### 6.2.1. Launch page

The launch page gives the original assignment linked to each query, and a button to launch it.

Query A  
Compute the number of movies per year. Make sure to include tv and video movies.  
Execute

Query B  
Compute the ten countries with most production companies.  
Execute

Query C  
Compute the min, max and average career duration. (A career length is implied by the first and last production of a person)  
Execute Refresh (~8 minutes)

Query D  
Compute the min, max and average number of actors in a production.  
Execute Refresh (~3 minutes)

Query E  
Compute the min, max and average height of female persons.  
Execute

Query F  
List all pairs of persons and movies where the person has both directed the movie and acted in the movie. Do not include tv and video movies.  
Execute

Query G  
List the three more popular character names.  
Execute

*Launch page for the simple queries of milestone 2*

### 6.2.2. Queries results

The queries execution is asynchronous, using AJAX call to fill data arrays similar to those used in other parts of application (see [§10.1.1]).

The execution of query A leads to a result array with the following appearance.

Number of movies per year results 121-140 (total 140)		« 2 3 4 5 6 7 »							
Year	Count								
2004	25588								
2005	29623								
2006	31155								

*Results array for the simple query A*



The execution of query B leads to a result array with the following appearance.

10 countries with most production companies		results 1-10 (total 10)	«	1	»
Country	Count				
us	93682				
gb	17796				
ca	10863				

*Results array for the simple query B*

The execution of query C leads to a result array with the following appearance.

Career duration			results 1-1 (total 1)	«	1	»
Min	Max	Avg				
0	130	3.8130				

*Results array for the simple query C*

The execution of query D leads to a result array with the following appearance.

Number of actors per production			results 1-1 (total 1)	«	1	»
Min	Max	Avg				
1	940	6.4909				

*Results array for the simple query D*

The execution of query E leads to a result array with the following appearance.

Height of female persons			results 1-1 (total 1)	«	1	»
Min	Max	Avg				
0.014999999664723873	3.049999952316284	1.6628801188523565				

*Results array for the simple query E*

The execution of query F leads to a result array with the following appearance.

Productions with actor being actual producer		results 1-20 (total 1280806)	«	1	2	3	4	5	6	»
Person ID	Production ID									
20	808953									
20	1583982									
20	1756649									
246	2134039									
246	146447									

*Results array for the simple query F*

The execution of query G leads to a result array with the following appearance.

Most popular characters		results 1-3 (total 3)	«	1	»
Name					
Himself					
Herself					
Himself - Host					

*Results array for the simple query G*

## 7. Interesting queries (milestone 3)

---

In this chapter, we present the SQL queries developed for the third milestone (called “interesting queries”).

### 7.1. SQL code

---

Here are the SQL codes for the interesting queries.

#### 7.1.1. Query (a)

---

Original assignment was as follows.

*“Find the actors and actresses (and report the productions) who played in a production where they were 55 or more year older than the youngest actor/actress playing.”*

Here is the corresponding SQL query we developed.

```
SELECT DISTINCT C.`person_id`, C.`production_id`
FROM `casting` C
INNER JOIN (
    SELECT P.`id`, P.`birthdate`
    FROM `person` P
    WHERE P.`birthdate` IS NOT NULL
) P ON C.`person_id` = P.`id`
INNER JOIN `role` R ON C.`role_id` = R.`id`
WHERE R.`name` IN ("actor", "actress")
AND EXISTS (
    SELECT MIN(PP.`birthdate`) AS `min_birthdate`
    FROM `casting` CC
    INNER JOIN (
        SELECT PP.`id`, PP.`birthdate`
        FROM `person` PP
        WHERE PP.`birthdate` IS NOT NULL
    ) PP ON CC.`person_id` = PP.`id`
    INNER JOIN `role` R ON CC.`role_id` = R.`id`
    WHERE CC.`production_id` = C.`production_id`
    AND R.`name` IN ("actor", "actress")
    HAVING TIMESTAMPDIFF(YEAR, `min_birthdate`, P.`birthdate`) >= 55
);
```

#### 7.1.2. Query (b)

---

Original assignment was as follows.

*“Given an actor, compute his most productive year.”*

Here is the corresponding SQL query we developed.

```
SELECT P.`year`, COUNT(*) AS number
FROM (
    SELECT C.`production_id`
    FROM `casting` C
    WHERE C.`person_id` = ###
) T
INNER JOIN `production` P ON T.`production_id` = P.`id`
WHERE P.`year` IS NOT NULL
GROUP BY P.`year`
ORDER BY number DESC
LIMIT 0,1;
```

#### 7.1.3. Query (c)

---

Original assignment was as follows.

*“Given a year, list the company with the highest number of productions in each genre.”*

Here is the corresponding SQL query we developed.

```
SELECT T.`gender_id`, T.`company_id`, T.`number`
FROM (
  SELECT PP.`gender_id`, P.`company_id`, COUNT(P.`company_id`) AS `number`
  FROM `productioncompany` P
  INNER JOIN `production` PP ON PP.`id` = P.`production_id`
  INNER JOIN `type` T ON P.`type_id` = T.`id`
  WHERE T.`name` = "production companies"
  AND PP.`year` = ###
  AND PP.`gender_id` IS NOT NULL
  GROUP BY PP.`gender_id`, P.`company_id`
  ORDER BY PP.`gender_id` ASC, `number` DESC, P.`company_id`
) T
GROUP BY T.`gender_id`;
```

#### 7.1.4. Query (d)

Original assignment was as follows.

*“Compute who worked with spouses/children/potential relatives on the same production. You can assume that the same real surname (last name) implies a relation.”*

Here is the corresponding SQL query we developed.

```
SELECT DISTINCT C.person_id, C.production_id
FROM `casting` C
INNER JOIN `person` P ON C.person_id = P.id
INNER JOIN `name` N ON P.name_id = N.id
WHERE EXISTS (
  SELECT CC.id
  FROM `casting` CC
  INNER JOIN `person` PP ON CC.person_id = PP.id
  INNER JOIN `name` NN ON PP.name_id = NN.id
  WHERE CC.production_id = C.production_id
  AND CC.person_id <> C.person_id
  AND N.lastname = NN.lastname
);
```

#### 7.1.5. Query (e)

Original assignment was as follows.

*“Compute the average number of actors per production per year.”*

Here is the corresponding SQL query we developed.

```
SELECT P.`year`, AVG(T.`number`) AS `number`
FROM (
  SELECT C.`production_id`, COUNT(DISTINCT C.`person_id`) AS number
  FROM `casting` C
  INNER JOIN `role` R ON C.`role_id` = R.`id`
  WHERE R.`name` = "actor"
  GROUP BY C.`production_id`
) T
INNER JOIN `production` P ON P.`id` = T.`production_id`
WHERE P.`year` IS NOT NULL
GROUP BY P.`year`;
```

#### 7.1.6. Query (f)

Original assignment was as follows.

*“Compute the average number of episodes per season.”*

Here is the corresponding SQL query we developed.

```
SELECT AVG(T.`number`) AS `number`  
FROM (  
  SELECT E.`season_id`, COUNT(E.`id`) AS `number`  
  FROM `episode` E  
  GROUP BY E.`season_id`  
) T;
```

### 7.1.7. Query (g)

---

Original assignment was as follows.

*“Compute the average number of seasons per serie.”*

Here is the corresponding SQL query we developed.

```
SELECT AVG(T.`number`) AS `number`  
FROM (  
  SELECT S.`serie_id`, COUNT(S.`id`) AS `number`  
  FROM `season` S  
  GROUP BY S.`serie_id`  
) T;
```

### 7.1.8. Query (h)

---

Original assignment was as follows.

*“Compute the top ten tv-series (by number of seasons).”*

Here is the corresponding SQL query we developed.

```
SELECT S.`id`, T.`number`  
FROM (  
  SELECT S.`serie_id`, COUNT(S.`id`) AS `number`  
  FROM `season` S  
  GROUP BY S.`serie_id`  
  ORDER BY `number` DESC  
  LIMIT 0,10  
) T  
INNER JOIN `serie` S ON S.`id` = T.`serie_id`;
```

### 7.1.9. Query (i)

---

Original assignment was as follows.

*“Compute the top ten tv-series (by number of episodes per season).”*

Here is the corresponding SQL query we developed.

```
SELECT S.`serie_id`, AVG(T.`number`) AS `number`  
FROM (  
  SELECT E.`season_id`, COUNT(E.`id`) AS `number`  
  FROM `episode` E  
  GROUP BY E.`season_id`  
) T  
INNER JOIN `season` S ON T.`season_id` = S.`id`  
GROUP BY S.`serie_id`  
ORDER BY `number` DESC  
LIMIT 0,10;
```

### 7.1.10. Query (j)

---

Original assignment was as follows.

*“Find actors, actresses and directors who have movies (including tv movies and video movies) released after their death.”*

Here is the corresponding SQL query we developed.

```
SELECT DISTINCT Per.`id`
FROM `casting` C
INNER JOIN `role` R ON C.`role_id` = R.`id`
INNER JOIN `production` P ON C.`production_id` = P.`id`
INNER JOIN `singleproduction` S ON P.`id` = S.`id`
INNER JOIN `kind` K ON S.`kind_id` = K.`id`
INNER JOIN `person` Per ON C.`person_id` = Per.`id`
WHERE R.`name` IN ("actor", "actress", "director")
AND K.`name` IN ("movie", "tv movie", "video movie")
AND Per.`deathdate` IS NOT NULL
AND P.`year` > EXTRACT(YEAR FROM Per.`deathdate`);
```

#### 7.1.11. Query (k)

Original assignment was as follows.

*“For each year, show three companies that released the most movies.”*

After giving a try to standard SQL queries, we determined that only a materialized view would permit running times not exceeding tens of seconds for the main query. The following code is used to create the table.

```
CREATE TABLE m3k_moviesbycompanyperyear (
  year INT UNSIGNED,
  company_id INT UNSIGNED,
  nb_movie INT UNSIGNED,
  PRIMARY KEY (`year`, `company_id`)
);
```

The following code is used to create the procedure that update this table.

```
DELIMITER $$
CREATE PROCEDURE refresh_m3k_moviesbycompanyperyear ()
BEGIN
  TRUNCATE TABLE m3k_moviesbycompanyperyear;
  INSERT INTO m3k_moviesbycompanyperyear (`year`, `company_id`, `nb_movie`)
  SELECT P.`year`, PC.`company_id`, COUNT(P.`id`) AS `number`
  FROM `casting` C
  INNER JOIN `production` P ON C.`production_id` = P.`id`
  INNER JOIN `productioncompany` PC ON P.`id` = PC.`production_id`
  INNER JOIN `type` T ON PC.`type_id` = T.`id`
  WHERE T.`name` = "production companies"
  AND P.`year` IS NOT NULL
  GROUP BY P.`year`, PC.`company_id`
  ORDER BY P.`year`, `number` DESC;
END;
$$
DELIMITER ;
```

Then, the following call has to be made every time we want the materialized view to be updated.

```
CALL refresh_m3k_moviesbycompanyperyear;
```

Finally, here is the SQL query that has to be launched, and run in very little time (couple of seconds), when we want to obtain data.

```
SELECT T.`year`, T.`company_id`, T.`nb_movie`
FROM (
  SELECT T.`year`, T.`company_id`, T.`nb_movie`,
    @year_rank := IF(@current_year = T.`year`, @year_rank + 1, 1) AS `year_rank`,
    @current_year := T.`year`
  FROM m3k_moviesbycompanyperyear T
) T
WHERE year_rank <= 3;
```

#### 7.1.12. Query (l)

Original assignment was as follows.

*“List all living people who are opera singers ordered from youngest to oldest.”*

Here is the corresponding SQL query we developed.

```
SELECT P.`id`  
FROM `person` P  
WHERE P.`birthdate` IS NOT NULL  
AND P.`deathdate` IS NULL  
AND (  
    P.`trivia` LIKE "%opera singer%"  
    OR P.`minibiography` LIKE "%opera singer%"  
)  
ORDER BY P.`birthdate` DESC;
```

#### 7.1.13. Query (m)

Original assignment was as follows.

*“List 10 most ambiguous credits (pairs of people and productions) ordered by the degree of ambiguity. A credit is ambiguous if either a person has multiple alternative names or a production has multiple alternative titles. The degree of ambiguity is a product of the number of possible names (real name + all alternatives) and the number of possible titles (real + alternatives).”*

Here is the corresponding SQL query we developed.

```
SELECT DISTINCT C.person_id, C.production_id, N.`number`*T.`number` AS `number`  
FROM `casting` C  
INNER JOIN (  
    SELECT N.person_id, COUNT(N.id) AS `number`  
    FROM `name` N  
    GROUP BY N.person_id  
    HAVING `number` > 1  
) N ON C.person_id = N.person_id  
INNER JOIN (  
    SELECT T.production_id, COUNT(T.id) AS `number`  
    FROM `title` T  
    GROUP BY T.production_id  
    HAVING `number` > 1  
) T ON C.production_id = T.production_id  
ORDER BY number DESC  
LIMIT 0,10;
```

#### 7.1.14. Query (n)

Original assignment was as follows.

*“For each country, list the most frequent character name that appears in the productions of a production company (not a distributor) from that country.”*

After giving a try to standard SQL queries, we determined that only a materialized view would permit running times not exceeding tens of seconds for the main query. The following code is used to create the table.

```
CREATE TABLE m3n_mostcharactercountry (  
    country_id INT UNSIGNED,  
    character_id INT UNSIGNED,  
    number INT UNSIGNED,  
    PRIMARY KEY (`country_id`, `character_id`)  
);
```

The following code is used to create the procedure that update this table.

```
DELIMITER $$
CREATE PROCEDURE refresh_m3n_mostcharactercountry ()
BEGIN
    TRUNCATE TABLE m3n_mostcharactercountry;
    INSERT INTO m3n_mostcharactercountry (`country_id`, `character_id`, `number`)
    SELECT Comp.`country_id`, C.`character_id`, COUNT(C.`character_id`) AS `number`
    FROM `casting` C
    INNER JOIN `productioncompany` PC ON C.`production_id` = PC.`production_id`
    INNER JOIN `type` T ON PC.`type_id` = T.`id`
    INNER JOIN `company` Comp ON PC.`company_id` = Comp.`id`
    WHERE T.`name` = "production companies"
    AND Comp.`country_id` IS NOT NULL
    AND C.`character_id` IS NOT NULL
    GROUP BY Comp.`country_id`, C.`character_id`
    ORDER BY Comp.`country_id`, `number` DESC, C.`character_id` ASC;
END;
$$
DELIMITER ;
```

Then, the following call has to be made every time we want the materialized view to be updated.

```
CALL refresh_m3n_mostcharactercountry;
```

Finally, here is the SQL query that has to be launched, and run in very little time (couple of seconds), when we want to obtain data.

```
SELECT T.`country_id`, T.`character_id`
FROM m3n_mostcharactercountry T
GROUP BY T.`country_id`;
```

## 7.2. Application dedicated page

The web application offers a complete page dedicated to the execution and the results visualization of these 14 interesting queries.

### 7.2.1. Launch page

The launch page gives the original assignment linked to each query, and a button to launch it. It looks very similar to the one used for the simple queries (see [§6.2.1]).

In the particular case of queries B and C, which require input parameters, text field on the web page are allowing the user to chose the value he wants for parameters.

The image shows two sections of a web application interface. The top section is titled 'Query B' and contains the text 'Given an actor, compute his most productive year.' and 'This query can be launched directly from an actor's page.' Below this text is a text input field labeled 'Actor ID' and a blue button labeled 'Execute'. The bottom section is titled 'Query C' and contains the text 'Given a year, list the company with the highest number of productions in each genre.' Below this text is a text input field labeled 'Year' and a blue button labeled 'Execute'.

Part of the launch page for queries of milestone 3, showing the input fields for parameterized queries

### 7.2.2. Queries results

The queries execution is asynchronous, using AJAX call to fill data arrays similar to those used in other parts of application (see [§10.1.1]).

The execution of query A leads to a result array with the following appearance.

Actors 55 year older than youngest ones in production results 1-20 (total 1000)		«	1	2	3	4	5	6	»
Person ID	Production ID								
4	2676684								
305	1445631								
305	1445930								

Results array for the interesting query A

The execution of query B leads to a result array with the following appearance.

Most productive year for given actor results 1-1 (total 1)		«	1	»
Year	Productions			
2014	54			

Results array for the interesting query B

The execution of query C leads to a result array with the following appearance.

Most productive companies for each genre results 1-20 (total 26)			«	1	2	»
Genre	Company	Number of productions				
1	49	22				
2	88564	10				
3	115716	8				

Results array for the interesting query C

The execution of query D leads to a result array with the following appearance.

Persons who worked with spouses/children/relatives on same productions results 1-6 (total 6)		«	1	2	3	4	5	6	»
Person	Production								
3251893	2129160								
3251894	2129160								
3901301	2447623								

Results array for the interesting query D

The execution of query E leads to a result array with the following appearance.

Average number of actors per production per year results 1-20 (total 136)		«	1	2	3	4	5	6	»
Year	Actors per production								
1880	1.0000								
1888	1.0000								
1890	1.0000								

Results array for the interesting query E

The execution of query F leads to a result array with the following appearance.

Average number of episodes per season results 1-1 (total 1)		«	1	»
Number				
16.6765				

Results array for the interesting query F



The execution of query G leads to a result array with the following appearance.

Average number of seasons per serie		results 1-1 (total 1)	«	1	»
Number					
1.8001					

Results array for the interesting query G

The execution of query H leads to a result array with the following appearance.

Top-ten tv-series (by number of seasons)		results 1-10 (total 10)	«	1	»
Serie	Number				
775230	76				
774229	76				
352482	73				

Results array for the interesting query H

The execution of query I leads to a result array with the following appearance.

Top-ten tv-series (by number of episodes per season)		results 1-10 (total 10)	«	1	»
Serie	Number				
1225561	9502.0000				
341096	8644.0000				
1166486	7050.0000				

Results array for the interesting query I

The execution of query J leads to a result array with the following appearance.

Actors, actresses and directors having movies released after their death (1000)		«	1	2	3	4	5	6	»
Person ID									
76									
76									
886									

Results array for the interesting query J

The execution of query K leads to a result array with the following appearance.

Three companies that released most movies per year			results 1-20 (total 1000)	«	1	2	3	4	5	6	»
Year	Company	Number of movies									
1880	241276	4									
1888	121133	10									
1890	14673	18									

Results array for the interesting query K

The execution of query L leads to a result array with the following appearance.

Living opera singers from youngest to oldest		results 1-20 (total 386)	«	1	2	3	4	5	6	»
Person ID										
647837										
3175809										
429326										

Results array for the interesting query L

The execution of query M leads to a result array with the following appearance.

10 most ambiguous credits			results 1-10 (total 10)	«	1	»
Person	Production	Degree of ambiguity				
616486	2588294	1659				
2365776	3002320	1600				
616486	3112661	1106				

*Results array for the interesting query M*

The execution of query N leads to a result array with the following appearance.

Most frequent character name from production companies for each country		«	1	2	3	4	5	6	»
Country ID	Character ID								
1	1								
2	1								
3	1								

*Results array for the interesting query N*

## 8. Detailed queries analysis

In this chapter we present our analysis for some queries selected in chapter 7 (milestone 3). The 3 queries we chose to analyse are E, J and M.

### 8.1. Necessity of indexes

We are going to explain the necessity of indexes by analyzing queries E, J and M of the milestone 3 and comparing the gain obtained using indexes.

Unfortunately *MySQL* doesn't propose a tool like *Oracle* to show the exact execution plan where we can see exactly which kind of join is used and others interesting details. However, we have used the command `EXPLAIN EXTENDED` and the visualization tool of *MySQLWorkbench*. More informations about those outputs can be found on <https://dev.mysql.com/doc/refman/5.5/en/explain-output.html#explain-join-types> and <https://dev.mysql.com/doc/refman/5.5/en/explain-extended.html>.

The running time obtained for those 3 queries were measured using a *MacBook Pro mid-2010*.

#### 8.1.1. Query(e)

First, we recall the Query(e) :

```
SELECT P.`year`, AVG(T.`number`) AS `number`
FROM (
  SELECT C.`production_id`, COUNT(DISTINCT C.`person_id`) AS number
  FROM `casting` C
  INNER JOIN `role` R ON C.`role_id` = R.`id`
  WHERE R.`name` = "actor"
  GROUP BY C.`production_id`
) T
INNER JOIN `production` P ON P.`id` = T.`production_id`
WHERE P.`year` IS NOT NULL
GROUP BY P.`year`;
```

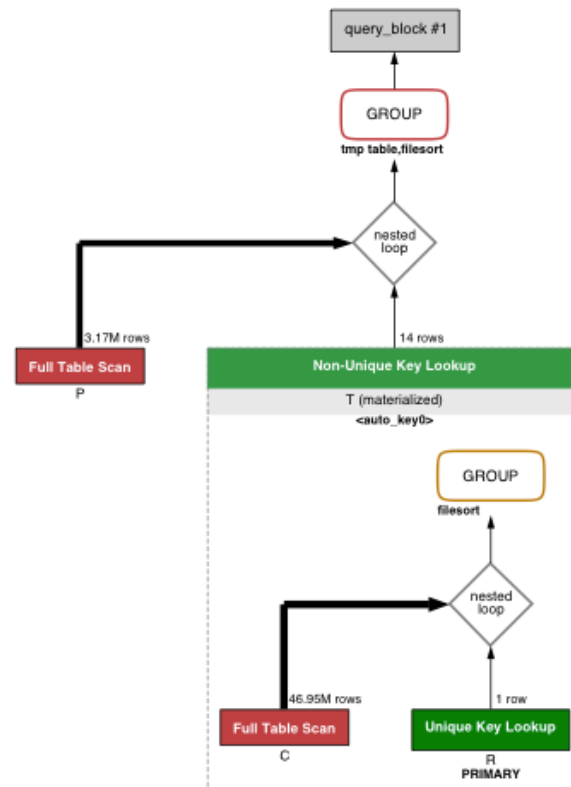
We can see that we have a temporary table (which is called a derived in *MySQL*) containing a join between the tables *casting* and *role*. Once the temporary table is created, we join it with the table *production*.

#### Without any indexes

Let's have a check of the execution plan without any indexes except the primary keys.

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	P	ALL	PRIMARY,idx_production_year	NULL		NULL	3165691	100.00	Using where; Using temporary; Using filesort
1	PRIMARY	<derived2>	ref	<auto_key0>	<auto_key0>	4	imbd.P.id	14	100.00	NULL
2	DERIVED	C	ALL	un_person_prod_role_character,fk_casting_production	NULL		NULL	46953826	100.00	Using filesort
2	DERIVED	R	eq_ref	PRIMARY	PRIMARY	4	imbd.C.role_id	1	100.00	Using where

Execution plan seen as a table for the query E of milestone 3, without indexes



Execution plan seen as a schema for the query E of milestone 3, without indexes

For the table *role*, we have to do a look-up to find the *id* corresponding to an actor. We can create an index on the field *role.name* to reduce the time to know which *id* is an actor. However the gain won't be so much comparing to the other indexes.

For the table *casting*, we have to do a full-scan because the rows aren't clustered by *role\_id*. The join operation will be high due to the lack of index on this field because we have to check every row (~45 millions) to proceed the join.

We can see that for the table *production*, we have to do a full-scan because we don't have any information about the location of the production which have a year. Moreover, the group by year is costly because we have to create a temporary table to sort the production by year.

With those observation, we can create indexes to solve those problems. Without any indexes, we obtains a time of **212 seconds**. For the rest, using the primary key helps, because otherwise the time would be incredibly high !

### With indexes

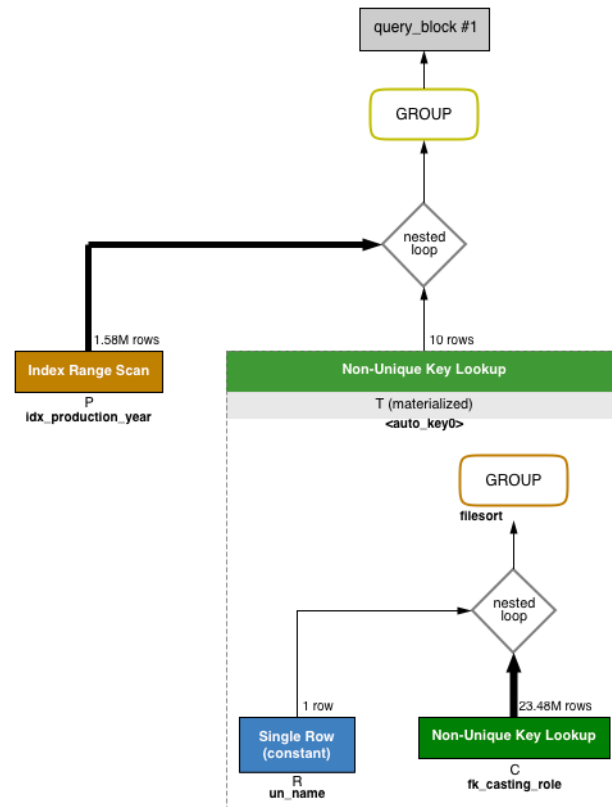
We have create the following indexes on :

- *casting.role\_id* to improve the join operation
- *role.name* to improve the look-up to see which *id* corresponds to an actor
- *production.year* to improve the group by part

With those indexes, we obtain the following execution plan :

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	P	range	PRIMARY,idx_production_year	idx_producti...	5	NULL	1582845	100.00	Using where; Using index
1	PRIMARY	<derived2>	ref	<auto_key0>	<auto_key0>	4	imbd.P.id	10	100.00	NULL
2	DERIVED	R	const	PRIMARY,un_name	un_name	767	const	1	100.00	Using index; Using filesort
2	DERIVED	C	ref	un_person_prod_role_character,fk_casting_role,fk_ca...	fk_casting_role	4	const	23476913	100.00	Using where

Execution plan seen as a table for the query E of milestone 3, with proper indexes set



Execution plan seen as a schema for the query E of milestone 3, with proper indexes set

Let's first talk about the index *production.year*. Before, we had a full-scan, filesort and temporary table (for the group by). Now, we have a range-scan (which is more efficient) and don't need to use a filesort with a temporary table for the groupby because the field *year* is indexed. Using the primary key wouldn't have been a good idea because we still need a full scan to obtain the production which has a year !

For the temporary table (derived2), it doesn't change anything because we didn't have indexes before (because this table doesn't exist) and it doesn't worthwhile to create a table with indexes during the query processing.

If we compare the row for *role*, we can observe we passed from *eq\_ref* (which means that we use an equality operator to find the corresponding row, which was a good join) to *const* (because of the index on *role.name*) which is much better because there is only one row, it can be considered as constant by the optimized and so reducing the access time to read this value.

Finally, for the table *casting*, the index *casting.role\_id* helps a lot for the join operation, passing from a full-scan to a ref, which is used for the operator "*=*" or "*< = >*", which is much better than a full-scan. Remember that casting has 45 millions of rows so we gain a lot in performances.

Another interesting detail is the number of rows checked ! You can see the differences between the case without and with the indexes. For *production* table, we seek 2 times less rows and 2 times less rows for *casting*, which correspond to ~25 millions of rows in total !

With all those indexes, we have reached a time of **102 seconds**, which is a **speed up of 2.08** from the initial time. We might ask ourselves if it is better to use an index on *casting.production\_id* rather than *casting.role\_id*, improving the group-by but not the join part. We measure the time and we didn't wait until it finished because 10 minutes was already left, which is catastrophic comparing to the index on the *role\_id* field !

Finally, we can ask ourselves whether other indexes could help. With this manner of writing query, we don't think we can find better indexes (to convince ourselves, it is enough to see the query plan). Maybe it is better to obtain better performances without using a temporary table, we have tried and have obtained worse performances.

### 8.1.2. Query(j)

First, we recall the Query(j) :

```
SELECT DISTINCT Per.`id`
FROM `casting` C
INNER JOIN `role` R ON C.`role_id` = R.`id`
INNER JOIN `production` P ON C.`production_id` = P.`id`
INNER JOIN `singleproduction` S ON P.`id` = S.`id`
INNER JOIN `kind` K ON S.`kind_id` = K.`id`
INNER JOIN `person` Per ON C.`person_id` = Per.`id`
WHERE R.`name` IN ("actor", "actress", "director")
AND K.`name` IN ("movie", "tv movie", "video movie")
AND Per.`deathdate` IS NOT NULL
AND P.`year` > EXTRACT(YEAR FROM Per.`deathdate`);
```

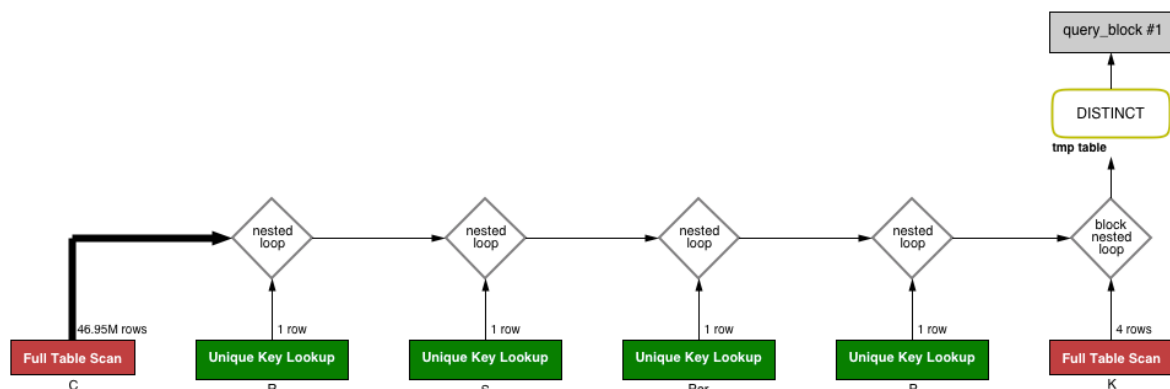
Because we have normalized everything, we have a lot of joins. However, the primary keys are used which help a lot for the efficiency. Finally, we have 4 “where conditions” which help reducing the running time of the query.

#### Without indexes

Let's have a check of the execution plan without any indexes except the primary keys.

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	C	ALL	NULL	NULL	NULL	NULL	46953826	100.00	Using temporary
1	SIMPLE	R	eq_ref	PRIMARY	PRIMARY	4	imbd.C.role_id	1	100.00	Using where
1	SIMPLE	S	eq_ref	PRIMARY	PRIMARY	4	imbd.C.production_id	1	100.00	Using where
1	SIMPLE	Per	eq_ref	PRIMARY,un_main_name,idx_person_height,idx_per...	PRIMARY	4	imbd.C.person_id	1	100.00	Using where
1	SIMPLE	P	eq_ref	PRIMARY	PRIMARY	4	imbd.C.production_id	1	100.00	Using where; Distinct
1	SIMPLE	K	ALL	PRIMARY	NULL	NULL	NULL	4	75.00	Using where; Distinct; Using join buffer (Block Nested Loop)

Execution plan seen as a table for query J of milestone 3, without indexes



Execution plan seen as a schema for query J of milestone 3, without indexes

For the table *singleproduction*, *person* and *production*, the primary keys are used and so we cannot improve this part of the join. However, for the table *casting*, we don't have any information about the foreign keys for *casting*. Moreover, as before, it is more efficient to use an index on *role.name* rather than its primary key. For the table *kind*, we will add an index on the field *name*, because we can observe that only 75% of the data are returned using the primary key.

To improve this part, it is necessary to use an index on the field *role.name* (rather than the primary key) and we should choose the index on the field *casting.role\_id* for the same reason as before, rather than choosing an index on *casting.production\_id*.

With those observations, we can create indexes to solve those problems. Without any indexes, we obtain a time of **30 seconds**.

### With indexes

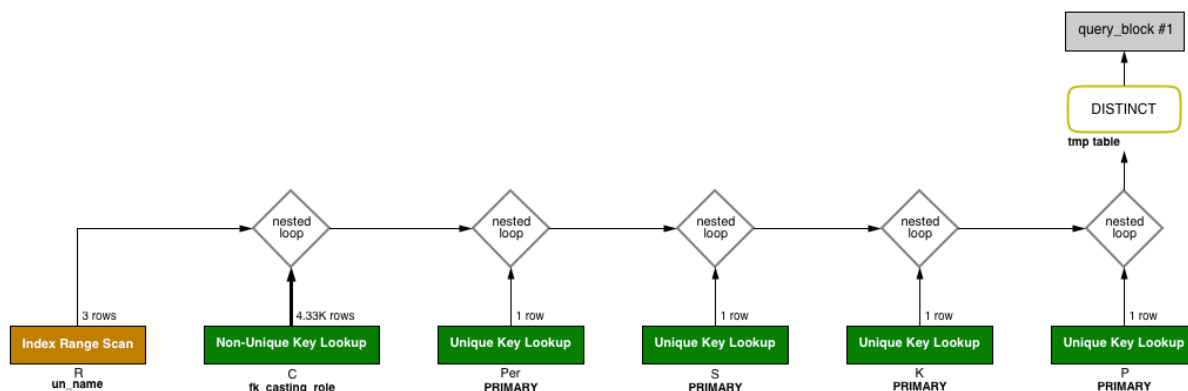
We have to create the following indexes on :

- *casting.role\_id* to improve the join operation
- *role.name* to improve the look-up to see which *id* corresponds to an actor/actress/director
- *kind.name* to improve the look-up to see which *id* corresponds to the specified kind of movies

With those indexes, we obtain the below execution plan.

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	R	range	PRIMARY,un_name	un_name	767	imdb.R.id	3	100.00	Using where; Using index; Using temporary
1	SIMPLE	C	ref	un_person_prod_role_character,fk_casting_role,fk_ca...	fk_casting_role	4	imdb.R.id	4326	100.00	Using where
1	SIMPLE	Per	eq_ref	PRIMARY,un_main_name,idx_person_height,idx_per...	PRIMARY	4	imdb.C.person_id	1	100.00	Using where
1	SIMPLE	S	eq_ref	PRIMARY,idx_kind	PRIMARY	4	imdb.C.production_id	1	100.00	Distinct
1	SIMPLE	K	eq_ref	PRIMARY,un_kind_name	PRIMARY	4	imdb.S.kind_id	1	100.00	Using where; Distinct
1	SIMPLE	P	eq_ref	PRIMARY,idx_production_year	PRIMARY	4	imdb.C.production_id	1	100.00	Using where; Distinct

Execution plan seen as a table for query J of milestone 3, with proper indexes set



Execution plan seen as a schema for query J of milestone 3, with proper indexes set

We can see that we avoid the full-scan of *casting* and only 4326 rows are checked (reduced by a factor 10'000) which is a very big gain !

For the table *role* we don't use anymore the primary key and use the index on *role.name* which leads to a range-scan. Because *role* is very small (we checked before 1 row and now 3 rows, which is very small), it won't be worse to use it and it will be better for the same reason as said before.

The table *person* and *production* haven't changed their execution. The table *singleproduction* kept the same index but has to use a distinct now. We didn't find an explanation for that.

Finally, the table *kind* didn't use our index. The reason is surely the order of the join and the optimized one has seen that only 1 row was possible at this step. Maybe with different data in the table *kind* (in fact we have only 4 entries and we look up for 3), the index on *kind.name* may be used for the same reason as *role.name* (where *role* has 11 entries).

With all those indexes, we have reached a time of **9 seconds**, which is a **speed up of 3.33**. Those indexes were quite trivial because a lot of primary keys are used. However with those 2 indexes (and 1 not used), we obtain a more efficient query.

### 8.1.3. Query(m)

First, we recall the Query(m) :

```
SELECT DISTINCT C.person_id, C.production_id, N.`number`*T.`number` AS `number`
FROM `casting` C
INNER JOIN (
    SELECT N.person_id, COUNT(N.id) AS `number`
    FROM `name` N
    GROUP BY N.person_id
    HAVING `number` > 1
) N ON C.person_id = N.person_id
INNER JOIN (
    SELECT T.production_id, COUNT(T.id) AS `number`
    FROM `title` T
    GROUP BY T.production_id
    HAVING `number` > 1
) T ON C.production_id = T.production_id
ORDER BY number DESC
LIMIT 0,10;
```

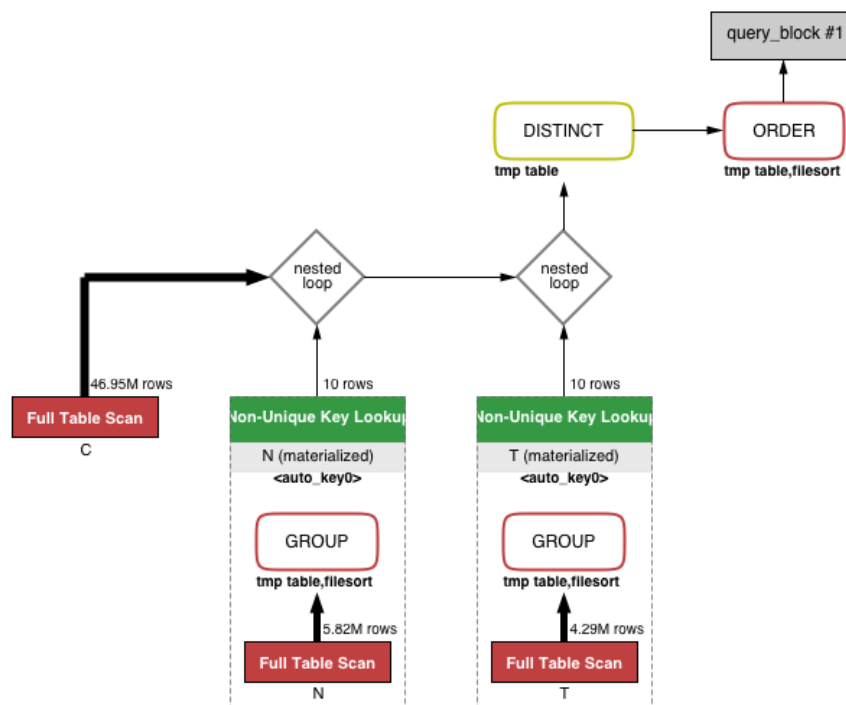
Because tables *name*, *title* and *casting* are big (6 millions, 45 millions, 3.5 millions), we cannot do directly a join on those 3 tables (as again due to our massive normalization). So, it is necessary to create temporary tables which will have less rows !

#### Without indexes

Let's have a check of the execution plan without any indexes except the primary keys.

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	C	ALL	NULL	NULL	NULL	NULL	46953826	100.00	Using temporary; Using filesort
1	PRIMARY	<derived2>	ref	<auto_key0>	<auto_key0>	4	imbd.C.person_id	10	100.00	NULL
1	PRIMARY	<derived3>	ref	<auto_key0>	<auto_key0>	4	imbd.C.production_id	10	100.00	NULL
3	DERIVED	T	ALL	un_title_production,fk_titledtoproduction	NULL	NULL	NULL	4291248	100.00	Using temporary; Using filesort
2	DERIVED	N	ALL	un_first_last_person,fk_nametoperson	NULL	NULL	NULL	5824649	100.00	Using temporary; Using filesort

Execution plan seen as a table for query M of milestone 3, without indexes



Execution plan seen as a schema for query M of milestone 3, without indexes

The first thing we can observe is that we have a full-scan for the 3 tables, which is enormous (55 millions in total). No keys at all are used in this query !



Let's begin with the table *casting*. We join directly from *casting.person\_id* to *name.person\_id* and we do the same for the titles. It should be very interesting to add a double index on the fields (*casting.person\_id*, *casting.production\_id*), as it will help a lot for the join operation which is very big.

For the temporary tables, we won't create any indexes because it will be faster to use them directly rather than the time required to create indexes. The number of rows for the temporary table of *name* and *title* are respectively 500'000 and 200'00.

For the tables *name* and *title*, their behaviour is similar. To count the number of rows, we have to use a temporary table which is costly. Interesting indexes should be on *name.person\_id* and *title.production\_id*. This will help a lot for the grouping part (which needs a temporary table with a filesort).

Unfortunately, for the last distinct and order, we cannot use indexes to improve this part.

With those observations, we can create indexes to solve those problems. Without any indexes, we obtain a time of **301 seconds**.

### With indexes

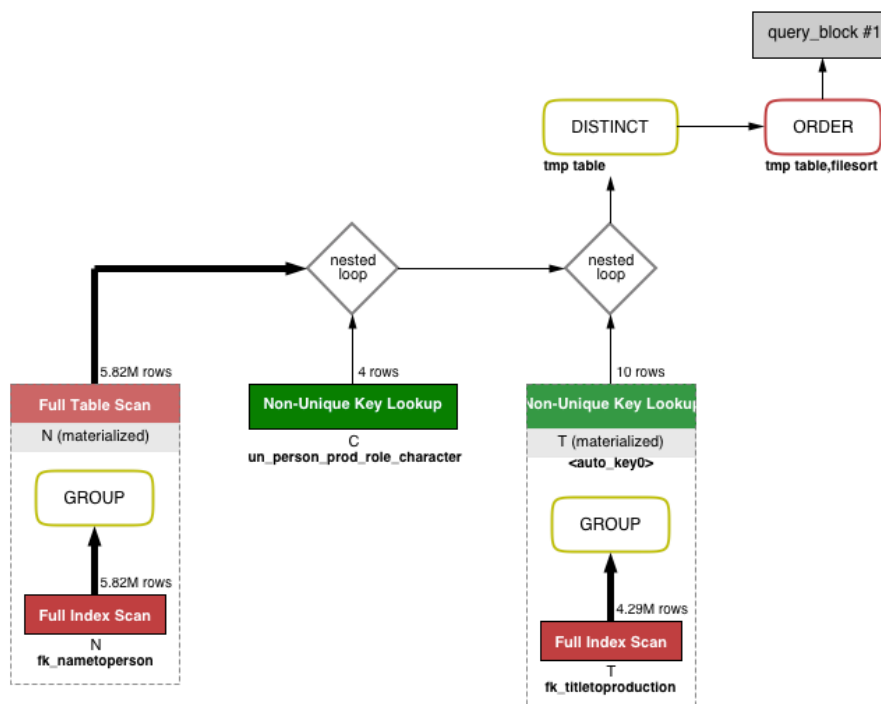
We have to create the following indexes on :

- *casting.(person\_id, production\_id)* to improve the join operation
- *name.person\_id* to improve the group by part
- *title.production\_id* to improve the group by part

With those indexes, we obtain the below execution plan.

id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	5824649	100.00	Using temporary; Using filesort
1	PRIMARY	C	ref	un_person_prod_role_character, fk_casting_production	un_person_...	4	N.person_id	4	100.00	Using index
1	PRIMARY	<derived3>	ref	<auto_key0>	<auto_key0>	4	imbd.C.production_id	10	100.00	Using index
3	DERIVED	T	index	un_title_production, fk_titletoproduction	fk_titletopro...	4	NULL	4291248	100.00	Using index
2	DERIVED	N	index	un_first_last_person, fk_nametoperson	fk_nametop...	4	NULL	5824649	100.00	Using index

Execution plan seen as a table for query M of milestone 3, with proper indexes set



Execution plan seen as a schema for query M of milestone 3, with proper indexes set

First thing we can see is that we are avoiding a full-scan to the table *casting*, which will be a big gain in the execution time. Moreover the number of rows has been a lot reduced to reach the number of 4.

For the tables *name* and *title* with their indexes, we have improved the group-by part which doesn't need anymore a temporary table with a filesort, which is also a good thing for the running time.

The table *derived3* hasn't changed at all. However, table *derived2* now needs a full-scan (and still the temporary table and the filesort to do the last distinct and order). We didn't find an explanation but if we analyze the total number of rows, it is ~15 millions compared to the 55 millions before.

With all those indexes, we have reach a time of **42 seconds**, which is a **speed up of 7.17**.

## 8.2. Distribution of the cost

---

After improving the queries E, J and M, we are going to try giving a rational explanation about the distribution of the cost. When we talk about times in this chapter, we are referencing the time obtained when we described the necessity of indexes in previous chapter.

### 8.2.1. Query(e)

---

Let's start with the sub-query. The final temporary table has 2.3 millions rows. It is much better to have this temporary table rather than directly doing a join between *casting*, *role* and *production*. Creating indexes for this temporary isn't worthwhile to its size.

Due to the index on *casting.production\_id*, the grouping part is quite fast alone but because we have a join on the table *role*, we need to do a filesort. The other index on *role.name* is very useful because the *id* of the role actor is considered as constant by the optimizer. This sub-query (if executed alone) is very fast (less than few seconds).

The place where we loose most of the time (we would say about 90%) is writing the result of this subquery inside a temporary table and then doing a join with the table *production*. The reason is that the last *where* clause is not strong enough to reduce the number of joins to do. The productions having a NULL year represent only 4.23% of the productions. After the join, it represents only 1.6% over all the productions in the join which is clearly insignificant. However, if the percentage was higher (~20-40%) the time will be a lot reduced (remind of the index on *production.year*).

Finally, we still have the *group by* but it is optimized because we don't need to use a temporary table nor a filesort (due to the index on *production.year*).

### 8.2.2. Query(j)

---

This query uses 5 joins which is quite big (especially with tables like *casting*, *production* and *person*). However, we have 4 *where* conditions which will limit a lot the number of final output (10'000).

Due to primary keys on tables *person*, *singleproduction*, *kind*, *production* and the indexes on *casting.role\_id* and *role.name*, the time of the join is a lot reduced.

We still have the distinct at the end which need a temporary table to eliminate the duplicate, which takes also time but less than the join of course. We have tried to replaced it with a group by but it was more efficient to use distinct rather than *group by* because the *group by* needs a temporary table and a file sort

### 8.2.3. Query(m)

First thing we can see which takes a lot of time are our 2 temporary tables (so no indexes) due to the sub-queries. Remember they contains 500'000 and 200'000 rows with which we have to do a join operation with *casting*. Those steps are costly but less than the distinct/order part, it takes approximately 10% of the time (4.6 seconds).

Hopefully, those temporary table use indexes on *name.person\_id* and *production.title\_id* which reduces their execution time (as said earlier in this chapter) especially for the group by part.

Finally, we have to distinct them (which needs a temporary table) and last step is to sort them which needs another temporary table and a filesort. The bottleneck is clearly here because we have 3 millions of entries for the distinct (which takes few seconds), which becomes 2.7 millions and finally the order part which is the bottleneck which takes the rest of the time.

Unfortunately, to improve the order part, we cannot make any assumption about the temporary tables (such that the person/production with the highest score has a high score for the name and the title).

## 8.3. Running times of all queries

All queries have been executed on a *MacBook Pro Retina late 2013*. 2 kinds of runs have been used : the first one is an execution without any cache and the second one with cache (which is more realistic in a website due to the number of requests done by visitors and the ability for DBA to schedule "cold-start" procedures to heat the base at first launch). For the rest of details, please refer to [\[§1.1.1\]](#).

### 8.3.1. Simple queries of milestone 2

Here are the running times for the queries of milestone 2.

Query / Time [sec]	Without cache	With cache
Query a	1.7	2.0
Query b	7.3	5.0
Query c	435.0	360.0
Query d	96.0	88.0
Query e	1.8	0.7
Query f	1.6	0.7
Query g	15.3	5.7

Queries C and D have been improved to use materialized views which can be updated once every day, in order to have an execution time (simply a fetch in the materialized views) of a few seconds when actually executing the query. However, we indicate here the running time of the equivalent all-in-one SQL query. In the case of materialized views, it is a bit longer because we have to store the results in a new table.

### 8.3.2. Simple queries of milestone 3

---

Here are the running times for the queries of milestone 3.

Query / Time [sec]	Without cache	With cache
Query a	88.900	45.000
Query b	0.002	0.001
Query c	3.850	2.100
Query d	3.920	3.600
Query e	49.630	52.000
Query f	1.277	0.500
Query g	1.040	0.070
Query h	0.060	0.038
Query i	0.900	0.600
Query j	5.660	6.000
Query k	93.162	75.000
Query l	5.412	4.000
Query m	3.840	25.000
Query n	1067.365	934.000

Queries K and N have been improved to use materialized views which can be updated once every day, in order to have an execution time (simply a fetch in the materialized views) of a few seconds when actually executing the query. However, we indicate here the running time of the equivalent all-in-one SQL query. In the case of materialized views, it is a bit longer because we have to store the results in a new table.

## 9. Appendix A – Data import detailed report

This appendix presents complete information about the work we did to import CSV files into the database.

### 9.1. Imported files

Each of the following chapter details the importation of a particular CSV file, the general ideas implemented in the corresponding PHP scripts, and the results obtained after the importation (number of imported rows, essential warnings, etc...).

Each script is build such that all lines subject to an error of parsing or data format during the import is written back in a separate CSV file. By doing this, only the failing lines of an import attempt have to be imported again after the errors have been corrected in the script.

#### 9.1.1. Persons

Persons are listed in the PERSONS.CSV file. For each row in this file, the script adds 2 records in database.

- A record is inserted in the table **person**, with all fields of CSV except the name.
- A record is inserted in the table **name**, representing the main name of the person. The ID of this record is generated by the **AUTO\_INCREMENT** clause on this table.

Because the name and person tables are loop-linked by 2 foreign keys that cannot be **NULL**, we temporarily authorize the **name\_id** attribute in table person to be NULL during the import, so we can import the **person** and set its **name\_id** to **NULL**, then import the main **name** linked to the **person**, gather its ID and set it back as **name\_id** in **person**. Instead of doing this, we could just have disabled the foreign keys check for the entire database, but it has the inconvenient that foreign keys values are not fully checked again when constraints are re-enabled.

Here is the numeric result of the first import attempt.

Treated rows	Success	Success with warning	Errors
4'857'852	4'857'845	0	7

The 7 errors have been identified as follows.

CSV line	ID	Manual operation
819'536	824190	removed (175 cm) from size
2'571'426	2586222	removed a space in birthdate between december and the year
2'710'402	2726001	replaced 1/2 by 0.5 in centimeters
2'848'752	2865228	rewritten 5'9 1/2 as 5' 9 1/2"
3'068'990	3086716	removed (160cm) from size
3'708'910	3730653	removed /98 in birthdate
4'049'865	4073747	replaced 6", 0' by 6'

After being manually corrected, these 7 lines were successfully imported during the second attempt.

Treated rows	Success	Success with warning	Errors
7	7	0	0

### 9.1.2. Alternative names

Alternative names of persons are listed in the ALTERNATIVE\_NAME.CSV file. For each row in this file, the script adds 1 record in database.

- A record is inserted in the table **name**, representing the alternative name and directly linked to the **person**'s ID through the **person\_id** field.

All rows were correctly imported at first attempt, with 10'109 duplicate entries that were deleted (some of them were colliding with another identical alternative name for the same person, others were colliding with their corresponding person's main name).

Treated rows	Success	Success with warning	Errors
869'697	859'588	0	10'109

### 9.1.3. Companies

Companies are listed in COMPANY.CSV file. For each row in this file, the script adds up to 2 records in database.

- If it doesn't already exist due to previously imported companies, a record is inserted in the table **country**, representing the country in which the company is established.
- A record is inserted in the table **company**, representing the company.

All rows were correctly imported at first attempt, with no error nor warning ever.

Treated rows	Success	Success with warning	Errors
279'142	279'142	0	0

### 9.1.4. Productions

Productions are listed in PRODUCTION.CSV file. For each row in this file, the script adds at least 3 records in database.

- A record is inserted in the table **production**, representing the parent element in the ISA hierarchy for productions.
- A record is inserted in one of the **singleproduction**, **serie** or **episode** table, depending on the kind of production, representing the child element in the ISA hierarchy.
- A record is inserted in the table **title**, representing the main title of the production.

In addition to this, records may be added in the following cases.

- When a row seems to be an alternative title for another episode (having same season and episode number in a given serie), it is not added like described above, but only generates a record in the table **title**, linked to the existing duplicate episode found as one of its alternative titles.
- When a singleproduction is added, a record may be added in the table **kind** if the corresponding kind does not already exist.
- When a production is added, a record may be added in the table **gender** if the corresponding gender does not already exist.
- When an episode is added, a record may be added in the table **season** if the corresponding season does not already exist in the given serie.

As for the persons import (see [§9.1.1]), the `title_id` field in the table `production` was temporarily `NULL`-authorized during the import operations.

Here is the result of the first import attempt.

Treated rows	Success	Success with warning	Errors
3'180'098	3'178'788	778	532

773 warning are generated for rows that generated an alternative title for an existing episode, as explained above. In addition to being added as alternative titles, these rows also generated an output line in a particular “duplicate ref” CSV file, to prevent other elements (productioncompany, casting) pointing to them from crashing when being imported. This CSV file thus contains association of their ID to the ID of episodes they are finally linked to as alternative titles.

The 5 remaining warnings are due to very long titles that have been truncated to 255 characters.

Despite investigation, no obvious reason was found for the 532 errors, so a second import attempt has been executed without changing anything to the script. During this second attempt, all these errors were eliminated. The final explanation is that they were episodes linked to series that appeared further in the CSV file, and were not already inserted during the first attempt.

Treated rows	Success	Success with warning	Errors
532	532	0	0

#### 9.1.5. Alternative titles

Alternative titles of productions are listed in the file `ALTERNATIVE_TITLE.CSV`. For each row in this file, the script adds 1 record in database.

- A record is added in the table `title`, representing the alternative title and directly linked to the `production`'s ID through the `production_id` field.

When importing a line, the script checks in the “duplicate ref” CSV file generated during productions import for alternative titles pointing to episodes that were originally converted themselves as alternative titles for other episodes, automatically correcting the `production_id` field when necessary so no row ever crash or is evicted.

All rows were correctly imported at first attempt, with 29'744 duplicate entries that were deleted (some of them were colliding with another identical alternative title for the same production, others were colliding with their corresponding production's main title). 3 titles were truncated to 255 characters, and generated a warning for this reason.

Treated rows	Success	Success with warning	Errors
407'441	377'694	3	29'744

#### 9.1.6. Characters

The characters are listed in the file `CHARACTER.CSV`. For each row in this file, the script adds 1 record in database.

- A record is added in the table `character`, representing the character.

When a character being imported has the exact same name as another already imported character, it is not imported. Instead, a line is added in a separate “duplicate ref” CSV file, associating its ID to the ID of the already existing character, so that casting importation won’t fail and won’t miss any record.

All rows were correctly imported at first attempt, with 704 duplicate entries that were added to the “duplicate ref” CSV file instead of being imported into database. 5 names were truncated to 255 characters, and generated a warning for this reason.

Treated rows	Success	Success with warning	Errors
3'589'274	3'588'565	5	704

### 9.1.7. Castings

The castings informations are listed in the file PRODUCTION\_CAST.CSV. For each row in this file, the script adds up to 2 records in database.

- If it doesn’t already exist due to previously imported casting records, a record is inserted in the table **role**, representing the role played by the person in the production.
- A record is added in the table **casting**, representing the casting record.

When importing a line, the script checks in the “duplicate ref” CSV file generated during characters import for IDs pointing to other characters, automatically correcting the **character\_id** field when necessary so no row ever crash or is evicted.

At first import attempt, 20'186 rows were corrected by the “duplicate ref” CSV file to point to the correct character.

Treated rows	Success	Success with warning	Errors
44'046'417	44'024'572	20'186	1'659

Some errors are due to an error in the script, not correcting **production\_id** field according to its “duplicate ref” CSV file. After this script was corrected, the second import attempt generated the following result.

Treated rows	Success	Success with warning	Errors
1'659	0	1'272	387

The 1'272 warnings corresponds to the previously generated errors due to the leak of **production\_id** field correction according to the “duplicate ref” CSV file. The remaining 387 errors are pure duplicate rows in the PRODUCTION\_CAST.CSV file violating the unicity columns over the whole attributes set of the table **casting**, thus simply not imported.

### 9.1.8. Association between production and company

The last CSV file, PRODUCTION\_COMPANY.CSV, contains association between companies and productions, giving each tuple a particular type (distributor or production company). For each row in this file, the script adds up to 2 records in database.

- If it doesn’t already exist due to previously imported productioncompany records, a record is inserted in the table **type**, representing the type of association.
- A record is added in the table **productioncompany**, representing the association.



When importing a line, the script checks in the “duplicate ref” CSV file generated during productions import for alternative titles pointing to episodes that were originally converted themselves as alternative titles for other episodes, automatically correcting the `production_id` field when necessary so no row ever crash or is evicted.

At first import attempt, 20 rows were corrected as described right above, thus generating a warning. 80'261 rows were not imported at all, violating the unicity constraint over whole set of attributes of the table `productioncompany`.

Treated rows	Success	Success with warning	Errors
3'407'851	3'327'570	20	80'261

## 9.2. SQL DDL code before milestone 2 modification

Below you can find the DDL SQL code for the database creation as it was before milestone 2 changes [§4.1] were executed.

```
-- 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) 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 `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,  
  `production_id` INT UNSIGNED NOT NULL,  
  `company_id` INT UNSIGNED NOT NULL,  
  `type_id` INT UNSIGNED NOT NULL,  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `un_production_company` (`production_id`, `company_id`, `type_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 `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,  
  ADD CONSTRAINT `fk_productioncompany_type` FOREIGN KEY (`type_id`)  
  REFERENCES `type` (`id`) ON DELETE RESTRICT 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;
```

## 10. Appendix B – Web application detailed

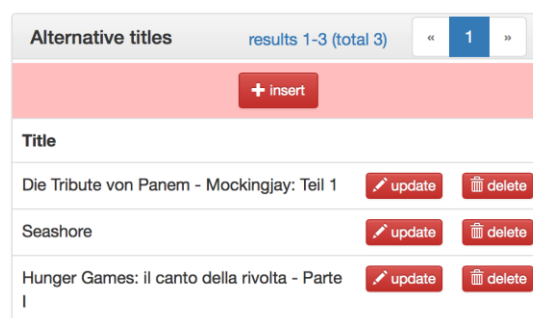
This appendix presents advanced and redundant information about the web application backbone and development process.

### 10.1. Major components

In this chapter, we first describe the main automatic components proposed by the ILARIA framework that are used through the entire application to present data in a smart way, mostly using AJAX.

#### 10.1.1. AJAX-loaded data lists

Data lists are implemented in an AJAX-loading fashion that makes them beautiful, instantly displaying a “loading” GIF while queries are executed. Pagination is automatically managed, and buttons for inserting, updating and deleting rows are easily added by few lines of codes.



The screenshot shows a web interface for 'Alternative titles'. At the top, there's a header with the title 'Alternative titles', a status 'results 1-3 (total 3)', and pagination controls showing '1' as the current page. Below the header is a red bar with a '+ insert' button. The main content is a table with three rows of data. Each row has a 'Title' column and two action buttons: 'update' (with a pencil icon) and 'delete' (with a trash icon).

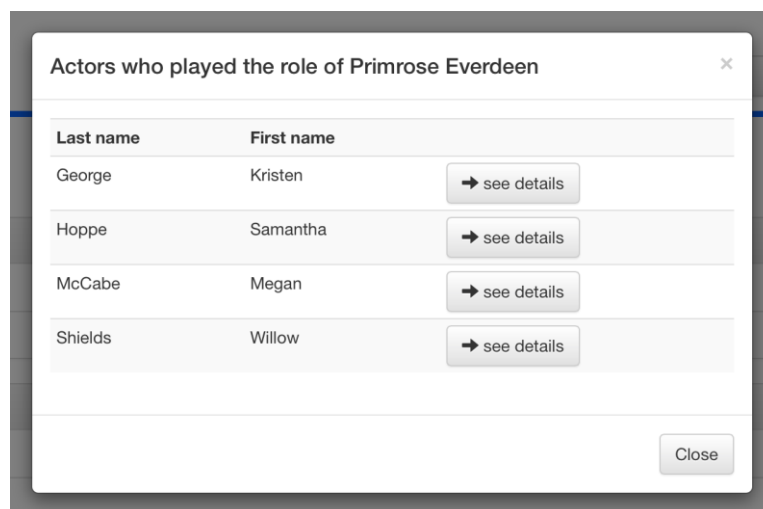
Alternative titles		results 1-3 (total 3)	« 1 »
<div>+ insert</div>			
Title		update	delete
Die Tribute von Panem - Mockingjay: Teil 1		update	delete
Seashore		update	delete
Hunger Games: il canto della rivolta - Parte 1		update	delete

*An AJAX-loaded data list with 3 elements and management buttons*

Every data list having a similar header with paging buttons as the one here is AJAX loaded. The entire dataset is loaded once by AJAX after query was executed, and the pagination is done locally in Javascript, offering the user a more fluid experience.

#### 10.1.2. AJAX modal window

Using bootstrap generic functionality, a modal window is present in the HTML template code of the application and is actively used to display subpages in the application, like the “are you sure ?” windows when trying to delete an element, or result lists for subqueries when doing search.



The screenshot shows a modal window titled 'Actors who played the role of Primrose Everdeen'. It contains a table with four rows of actor data. Each row has columns for 'Last name', 'First name', and a 'see details' button. The modal has a 'Close' button at the bottom right.

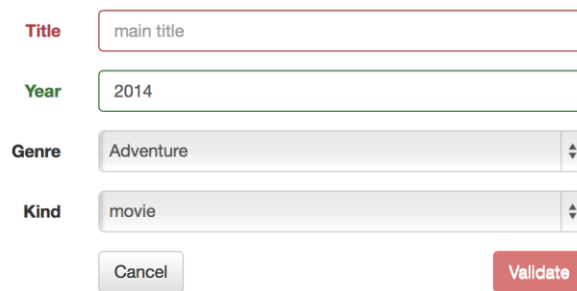
Last name	First name	
George	Kristen	→ see details
Hoppe	Samantha	→ see details
McCabe	Megan	→ see details
Shields	Willow	→ see details

*The modal window filled with some actors playing the role of Primrose Everdeen in The Hunger Games saga*

The content of this modal window can (and actually *is*) loaded through AJAX requests. The request is issued when the window is called to be opened, and the window displays a “loading” GIF until the result becomes available to be displayed.

### 10.1.3. Dynamic forms

Forms are graphically built using bootstrap components. Javascript bunches of code are automatically generated by the ILARIA framework to validate the content of the forms on the client side, checking things like “this field must not be empty” and “this field must have a unique value across all records”. When necessary, AJAX requests are issued to do live validation when editing things.



The image shows a web form with four input fields: 'Title' (text input, empty, red border), 'Year' (text input, '2014', green border), 'Genre' (dropdown menu, 'Adventure', grey border), and 'Kind' (dropdown menu, 'movie', grey border). Below the fields are two buttons: 'Cancel' (grey) and 'Validate' (red, disabled).

A form with the “title” field in error state due to its emptiness

Validation button of forms are automatically disabled and red-colored when a problem relative to the validation of a field is detected in the corresponding form. By using the ILARIA form component, all validations are automatically done on client and server sides for maximum security and problem prevention.

## 10.2. Main pages queries

In the following chapters we give details about the exact SQL queries that are used through the application to display main pages of persons, productions and companies.

### 10.2.1. Person’s main page

These queries are used to display the content of a person’s main page, as described in [\[§5.2.1\]](#).

The following query is used to load the main elements of the page (no-AJAX).

```
SELECT DISTINCT PE.`id`, PE.`gender`, PE.`trivia`, PE.`quotes`, PE.`birthdate`,  
                PE.`deathdate`, PE.`birthname`, PE.`minibiography`, PE.`spouse`, PE.`height`,  
                PE.`name_id`, NA.`lastname`, NA.`firstname`  
FROM `person` PE  
INNER JOIN `name` NA ON PE.`name_id` = NA.`id`  
WHERE PE.`id` = 2682115;
```

The following query is used by AJAX loading to gather the alternative names.

```
SELECT DISTINCT NA.`id`, NA.`lastname`, NA.`firstname`  
FROM `name` NA  
WHERE NA.`person_id` = 2682115  
      AND NA.`id` != 2666765;
```

The final AND clause is used to evict the main name from the result set.

The following query is used by AJAX loading to gather the roles of a person in singleproductions.

```
SELECT DISTINCT SP.`id` AS `prod_id`, TI.`title` AS `prod_title`, KI.`name` AS `prod_kind`,  
                GE.`name` AS `prod_gender`, PR.`year` AS `prod_year`, CH.`name` AS `char_name`, RO.`name`  
                AS `role_name`  
FROM `casting` CA  
INNER JOIN `production` PR ON CA.`production_id` = PR.`id`
```

```
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`
LEFT JOIN `gender` GE ON PR.`gender_id` = GE.`id`
INNER JOIN `singleproduction` SP ON PR.`id` = SP.`id`
INNER JOIN `kind` KI ON SP.`kind_id` = KI.`id`
LEFT JOIN `character` CH ON CA.`character_id` = CH.`id`
INNER JOIN `role` RO ON CA.`role_id` = RO.`id`
WHERE CA.`person_id` = 2682115
ORDER BY PR.`year` DESC, TI.`title` ASC;
```

The following query is used by AJAX loading to gather the roles of a person in series.

```
(
SELECT SER.`id` AS `prod_id`, TI.`title` AS `prod_title`, COUNT(DISTINCT EP.`id`) AS
`episode_count`, COUNT(DISTINCT SEA.`id`) AS `season_count`, SER.`yearstart` AS
`prod_yearstart`, SER.`yearend` AS `prod_yearend`, GE.`name` AS `prod_gender`, CH.`name`
AS `char_name`, RO.`name` AS `role_name`
FROM `casting` CA
INNER JOIN `production` PR ON CA.`production_id` = PR.`id`
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`
LEFT JOIN `gender` GE ON PR.`gender_id` = GE.`id`
INNER JOIN `serie` SER ON PR.`id` = SER.`id`
INNER JOIN `season` SEA ON SER.`id` = SEA.`serie_id`
INNER JOIN `episode` EP ON SEA.`id` = EP.`season_id`
LEFT JOIN `character` CH ON CA.`character_id` = CH.`id`
INNER JOIN `role` RO ON CA.`role_id` = RO.`id`
WHERE CA.`person_id` = 1831321
GROUP BY PR.`id`, RO.`name`, CH.`name`
) UNION DISTINCT (
SELECT SER.`id` AS `prod_id`, TI.`title` AS `prod_title`, COUNT(DISTINCT EP.`id`) AS
`episode_count`, COUNT(DISTINCT SEA.`id`) AS `season_count`, SER.`yearstart` AS
`prod_yearstart`, SER.`yearend` AS `prod_yearend`, GE.`name` AS `prod_gender`, CH.`name`
AS `char_name`, RO.`name` AS `role_name`
FROM `casting` CA
INNER JOIN `production` PR_EP ON CA.`production_id` = PR_EP.`id`
INNER JOIN `episode` EP ON PR_EP.`id` = EP.`id`
INNER JOIN `season` SEA ON EP.`season_id` = SEA.`id`
INNER JOIN `serie` SER ON SEA.`serie_id` = SER.`id`
INNER JOIN `production` PR_SER ON SER.`id` = PR_SER.`id`
INNER JOIN `title` TI ON PR_SER.`title_id` = TI.`id`
LEFT JOIN `gender` GE ON PR_SER.`gender_id` = GE.`id`
LEFT JOIN `character` CH ON CA.`character_id` = CH.`id`
INNER JOIN `role` RO ON CA.`role_id` = RO.`id`
WHERE CA.`person_id` = 1831321
GROUP BY PR_SER.`id`, RO.`name`, CH.`name`
)
ORDER BY `prod_yearstart` DESC, `prod_title` ASC;
```

The first subquery searches for series in which the person is directly linked, while the second subquery is searching for series containing one or more episodes, with person linked to these episodes.

### 10.2.2. Production's main page

These queries are used to display the content of a production's main page as described in [§0].

The following query helps to determine the kind of production in the ISA hierarchy (singleproduction, serie or episode).

```
(
SELECT SP.`id` AS `single`, NULL AS `serie`, NULL AS `episode`
FROM `singleproduction` SP
WHERE SP.`id`=2998449
) UNION (
SELECT NULL AS `single`, SE.`id` AS `serie`, NULL AS `episode`
FROM `serie` SE
WHERE SE.`id`=2998449
) UNION (
SELECT NULL AS `single`, NULL AS `serie`, EP.`id` AS `episode`
FROM `episode` EP
WHERE EP.`id`=2998449
);
```

Then, in the case of a singleproduction, the following query is used to gather general informations about it.

```
SELECT PR.`id` AS `prod_id`, TI.`title` AS `prod_title`, PR.`year` AS `prod_year`, KI.`name`  
      AS `prod_kind`, GE.`name` AS `prod_gender`, PR.`title_id` AS `maintitle_id`  
FROM `production` PR  
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`  
LEFT JOIN `gender` GE ON PR.`gender_id` = GE.`id`  
INNER JOIN `singleproduction` SP ON PR.`id` = SP.`id`  
INNER JOIN `kind` KI ON SP.`kind_id` = KI.`id`  
WHERE PR.`id`=2998449;
```

In the case of a serie, the following query is used to gather general informations about it.

```
SELECT SER.`id` AS `prod_id`, TI.`title` AS `serie_title`, SER.`yearstart` AS  
      `serie_yearstart`, SER.`yearend` AS `serie_yearend`, GE.`name` AS `serie_gender`, TI.`id`  
      AS `maintitle_id`, COUNT(DISTINCT SEA.`id`) AS `season_count`, COUNT(DISTINCT EP.`id`) AS  
      `episode_count`  
FROM `serie` SER  
INNER JOIN `production` PR ON SER.`id` = PR.`id`  
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`  
LEFT JOIN `gender` GE ON PR.`gender_id` = GE.`id`  
LEFT JOIN `season` SEA ON SER.`id` = SEA.`serie_id`  
LEFT JOIN `episode` EP ON SEA.`id` = EP.`season_id`  
WHERE SER.`id`=11306  
GROUP BY SER.`id`;
```

In the case of an episode, the following query is used to gather general informations about it.

```
SELECT EP.`id` AS `prod_id`, EP_TI.`title` AS `episode_title`, EP.`number` AS  
      `episode_number`, EP_PR.`year` AS `episode_year`, EP_TI.`id` AS `maintitle_id`,  
      SEA.`number` AS `season_number`, SER.`id` AS `serie_id`, SER_TI.`title` AS `serie_title`,  
      SER_GE.`name` AS `serie_gender`  
FROM `episode` EP  
INNER JOIN `production` EP_PR ON EP.`id` = EP_PR.`id`  
INNER JOIN `title` EP_TI ON EP_PR.`title_id` = EP_TI.`id`  
INNER JOIN `season` SEA ON EP.`season_id` = SEA.`id`  
INNER JOIN `serie` SER ON SEA.`serie_id` = SER.`id`  
INNER JOIN `production` SER_PR ON SER.`id` = SER_PR.`id`  
INNER JOIN `title` SER_TI ON SER_PR.`title_id` = SER_TI.`id`  
LEFT JOIN `gender` SER_GE ON SER_PR.`gender_id` = SER_GE.`id`  
WHERE EP.`id`=11316;
```

In the case of a serie, the list of seasons it contains is loaded by the following query.

```
SELECT SEA.`id` AS `season_id`, SEA.`number` AS `season_number`, COUNT(DISTINCT EP.`id`) AS  
      `episode_count`  
FROM `season` SEA  
INNER JOIN `episode` EP ON SEA.`id` = EP.`season_id`  
WHERE SEA.`serie_id`=11306  
GROUP BY SEA.`id`  
ORDER BY SEA.`number`;
```

In the case of a serie, when clicking on the “see episodes” button of a given season, the following query is used to load the content of the modal window.

```
SELECT EP.`id` AS `episode_id`, TI.`title` AS `episode_title`, EP.`number` AS `episode_number`  
FROM `episode` EP  
INNER JOIN `production` PR ON EP.`id` = PR.`id`  
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`  
WHERE EP.`season_id`=920  
ORDER BY EP.`number`;
```

In the case of an episode, the information about the related season and serie can be obtained via the following query.

```
SELECT EP.`id` AS `episode_id`, TI.`title` AS `episode_title`, EP.`number` AS `episode_number`  
FROM `episode` EP  
INNER JOIN `production` PR ON EP.`id` = PR.`id`  
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`  
WHERE EP.`season_id`=920  
ORDER BY EP.`number`;
```



For all kinds of productions, the following query is used to gather the casting.

```
SELECT PE.`id` AS `person_id`, NA.`lastname` AS `person_lastname`, NA.`firstname` AS  
`person_firstname`, RO.`name` AS `role_name`, CH.`name` AS `char_name`  
FROM `casting` CA  
INNER JOIN `person` PE ON CA.`person_id` = PE.`id`  
INNER JOIN `name` NA ON PE.`name_id` = NA.`id`  
INNER JOIN `role` RO ON CA.`role_id` = RO.`id`  
LEFT JOIN `character` CH ON CA.`character_id` = CH.`id`  
WHERE CA.`production_id`=2998449;
```

For all kinds of productions, the following query is used to gather the companies involved in the production.

```
SELECT COM.`id` AS `id`, COM.`name` AS `name`, COU.`code` AS `country`, TY.`name` AS `type`  
FROM `productioncompany` PC  
INNER JOIN `company` COM ON PC.`company_id` = COM.`id`  
INNER JOIN `country` COU ON COM.`country_id` = COU.`id`  
INNER JOIN `type` TY ON PC.`type_id` = TY.`id`  
WHERE PC.`production_id`=2998449;
```

For all kinds of productions, the following query is used to gather the alternative titles.

```
SELECT TI.`title`  
FROM `title` TI  
WHERE TI.`production_id`=2998449  
AND TI.`id`!=2998451;
```

The final AND clause is used to evict the main title from the result set.

### 10.2.3. Company's main page

These queries are used to display the content of a company's main page as described in [§5.2.3].

The following query is used to gather the general informations about a company.

```
SELECT TI.`title`  
FROM `title` TI  
WHERE TI.`production_id`=2998449  
AND TI.`id`!=2998451;
```

The following query is used to gather the list of singleproductions in which the company is involved.

```
SELECT SP.`id` AS `id`, TI.`title` AS `title`, KI.`name` AS `kind`, PR.`year` AS `year`,  
GE.`name` AS `gender`, TY.`name` AS `type`  
FROM `productioncompany` PC  
INNER JOIN `production` PR ON PC.`production_id` = PR.`id`  
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`  
LEFT JOIN `gender` GE ON PR.`gender_id` = GE.`id`  
INNER JOIN `singleproduction` SP ON PR.`id` = SP.`id`  
INNER JOIN `kind` KI ON SP.`kind_id` = KI.`id`  
INNER JOIN `type` TY ON PC.`type_id` = TY.`id`  
WHERE PC.`company_id`=3293  
ORDER BY TI.`title`, TY.`id`;
```

The following query is used to gather the list of series in which the company is involved.

```
(  
SELECT SER.`id` AS `id`, TI.`title` AS `title`, COUNT(DISTINCT EP.`id`) AS `episode_count`,  
COUNT(DISTINCT SEA.`id`) AS `season_count`, SER.`yearstart` AS `yearstart`, SER.`yearend`  
AS `yearend`, GE.`name` AS `gender`, TY.`name` AS `type`  
FROM `productioncompany` PC  
INNER JOIN `production` PR ON PC.`production_id` = PR.`id`  
INNER JOIN `title` TI ON PR.`title_id` = TI.`id`  
LEFT JOIN `gender` GE ON PR.`gender_id` = GE.`id`  
INNER JOIN `serie` SER ON PR.`id` = SER.`id`  
INNER JOIN `season` SEA ON SER.`id` = SEA.`serie_id`  
INNER JOIN `episode` EP ON SEA.`id` = EP.`season_id`  
INNER JOIN `type` TY ON PC.`type_id` = TY.`id`  
WHERE PC.`company_id`=3293  
GROUP BY PR.`id`, TY.`id`  
)
```

```
) UNION DISTINCT (  
SELECT SER.`id` AS `id`, TI.`title` AS `title`, COUNT(DISTINCT EP.`id`) AS `episode_count`,  
COUNT(DISTINCT SEA.`id`) AS `season_count`, SER.`yearstart` AS `yearstart`, SER.`yearend`  
AS `yearend`, GE.`name` AS `gender`, TY.`name` AS `type`  
FROM `productioncompany` PC  
INNER JOIN `production` PR_EP ON PC.`production_id` = PR_EP.`id`  
INNER JOIN `episode` EP ON PR_EP.`id` = EP.`id`  
INNER JOIN `season` SEA ON EP.`season_id` = SEA.`id`  
INNER JOIN `serie` SER ON SEA.`serie_id` = SER.`id`  
INNER JOIN `production` PR_SER ON SER.`id` = PR_SER.`id`  
INNER JOIN `title` TI ON PR_SER.`title_id` = TI.`id`  
LEFT JOIN `gender` GE ON PR_SER.`gender_id` = GE.`id`  
INNER JOIN `type` TY ON PC.`type_id` = TY.`id`  
WHERE PC.`company_id`=3293  
GROUP BY PR_SER.`id`, TY.`id`  
)  
ORDER BY `title`, `type`;
```

The first subquery searches for series in which the company is directly linked, while the second subquery is searching for series containing one or more episodes, with company linked to these episodes.

### 10.3. Insert, update and delete procedures

The following chapters present the insert, update and delete procedure, written in pseudo-code to save time and space as they are all basic DML SQL queries.

#### 10.3.1. Persons

These are the procedures used to modify tables corresponding to a “Person”.

##### Insertion

At the form validation, the insertion procedure is triggered. During this procedure, the foreign key checks are disabled on the database (SET FOREIGN\_KEY\_CHECKS=0). They are re-enabled at the end of the procedure. It is no longer an option to authorize the NULL value on one of the foreign key fields as it was done during import (see [§9.1.1]), because executing the corresponding query takes about 10 seconds with millions of records already in the table.

The insertion procedure follows this schema.

1. Start a transaction
2. Insert a record in the **person** table, with value 0 as **name\_id**
3. Insert a record in the **name** table, with the new person record ID as **person\_id**
4. Update the **person** record to set **name\_id** to the correct value
5. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

##### Update

Despite the insertion procedure needs it, the update one doesn't necessitate any constraint to be wiped out.

The update procedure follows this schema.

1. Start a transaction
2. Update the corresponding record in the **person** table
3. Find the main name ID from this record
4. Update the corresponding record in the **name** table

## 5. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

### Deletion

Like the insertion procedure, the deletion one temporarily disables the foreign keys constraints while deleting a person.

The delete procedure follows this schema.

1. Start a transaction
2. Delete the **name** records with matching **person\_id**
3. Delete the **person** record
4. Delete the **casting** records with matching **person\_id**
5. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

#### 10.3.2. Alternative names

---

These are the procedures used to modify tables corresponding to a “Alternative name”.

### Insertion

The insertion procedure follows this schema.

1. Insert a record in the **name** table

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

### Update

The update procedure follows this schema.

1. Update the corresponding record in the **name** table

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

### Deletion

The delete procedure follows this schema.

1. Delete the corresponding record from the **name** table

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

#### 10.3.3. Productions

---

These are the procedures used to modify tables corresponding to a “Production”.

### Insertion of a movie

At the form validation, the insertion procedure is triggered. During this procedure, the foreign key checks are disabled on the database (SET FOREIGN\_KEY\_CHECKS=0). They are re-enabled at the end of the procedure. It is no longer an option to authorize the NULL value on one of the foreign key fields as it was done during import (see [§9.1.4]), because executing the corresponding query takes about 10 seconds with millions of records already in the table.

The insertion procedure follows this schema.

1. Start a transaction
2. Insert a record in the **production** table, with value 0 as **title\_id**
3. Insert a record in the **title** table, with the new production record ID as **production\_id**
4. Update the production record to set **title\_id** to the correct value
5. Insert a record in the **singleproduction** table, thus completing the ISA hierarchy covering constraint
6. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

### Insertion of a serie

Like for the insertion of a movie, the foreign keys constraints are temporarily disabled during insertion of a serie.

The insertion procedure follows this schema.

1. Start a transaction
2. Insert a record in the **production** table, with value 0 as **title\_id**
3. Insert a record in the **title** table, with the new production record ID as **production\_id**
4. Update the production record to set **title\_id** to the correct value
5. Insert a record in the **serie** table, thus completing the ISA hierarchy covering constraint
6. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

### Insertion of an episode

Like for the insertion of a movie, the foreign keys constraints are temporarily disabled during insertion of an episode.

The insertion procedure follows this schema.

1. Start a transaction
2. Insert a record in the **production** table, with value 0 as **title\_id**
3. Insert a record in the **title** table, with the new production record ID as **production\_id**
4. Update the production record to set **title\_id** to the correct value
5. Search through the **season** table for the season having **serie\_id** and **number** correctly set
6. If no season is found, insert it in the **season** table
7. Insert a record in the **episode** table, pointing to the (newly inserted) season, thus completing the ISA hierarchy covering constraint
8. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

### Updating a movie, a serie or an episode

Despite the insertion procedures needs it, the update ones doesn't necessitate any constraint to be wiped out.

The update procedures are similar and ordered like the insertion ones for each kind of production. Transactions are always used, and rollback is a systematic action in case of problem.

### Deleting a movie or an episode

Like the insertion procedures, the deletion ones temporarily disables the foreign keys constraints while deleting a production.

The delete procedure follows this schema.

1. Start a transaction
2. Delete the **title** records with matching **production\_id**
3. Delete the **production** record
4. Delete the **singleproduction/episode** record
5. Delete the **casting** records with matching **production\_id**
6. Delete the **productioncompany** records with matching **production\_id**
7. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

### Deleting a serie

Like the insertion procedure, the deletion one temporarily disables the foreign keys constraints while deleting a production.

The delete procedure follows this schema.

1. Start the transaction
2. Delete the **title** records with matching **production\_id**
3. Delete the **production** record
4. Delete the **serie** record
5. Delete the **casting** records with matching **production\_id**
6. Delete the **productioncompany** records with matching **production\_id**
7. Select and memorize the **season** records with matching **serie\_id**
8. Delete these **season** records
9. Select and memorize the **episode** records linked to a **season** record previously memorized
10. Delete these **episode** records
11. Delete the **production** records matching any of the previously memorized **episode**
12. Delete the **title** records with matching **production\_id** on any memorized **episode**
13. Delete the **casting** records with matching **production\_id** on any memorized **episode**
14. Delete the **productioncompany** records with matching **production\_id** on any memorized **episode**
15. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

#### 10.3.4. Alternative titles

---

These are the procedures used to modify tables corresponding to a “Alternative title”.

##### **Insertion**

The insertion procedure follows this schema.

1. Insert a record in the **title** table

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

##### **Update**

The update procedure follows this schema.

1. Update the corresponding record in the **title** table

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

##### **Deletion**

The delete procedure follows this schema.

1. Delete the corresponding record from the **title** table

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

#### 10.3.5. Casting

---

These are the procedures used to modify tables corresponding to a “Casting”.

##### **Insertion**

The insertion procedure follows this schema.

1. Start a transaction
2. Search through the **character** table for character having **name** correctly set
3. If no character is found, insert it in the **character** table
4. Insert a record in the **casting** table, pointing to the (newly inserted) character
5. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

##### **Update**

The update procedure follows this schema.

1. Start a transaction
2. Search through the **character** table for character having **name** correctly set
3. If no character is found, insert it in the **character** table

4. Update the corresponding record in the **casting** table, pointing to the (newly inserted) character
5. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

### Deletion

The deletion procedure follows this schema.

1. Delete the **casting** record

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

#### *10.3.6. Companies involved in a production*

---

These are the procedures used to modify tables corresponding to the participation of a company in a production.

### Insertion

The insertion procedure follows this schema.

1. Insert a record in the **productioncompany** table

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

### Update

The update procedure follows this schema.

1. Update the **productioncompany** record

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

### Deletion

The deletion procedure follows this schema.

1. Delete the **productioncompany** record

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.

#### *10.3.7. Companies*

---

These are the procedures used to modify tables corresponding to a “Company”.

### Insertion

The insertion procedure follows this schema.

1. Start a transaction
2. Search through the **country** table for country having **code** correctly set
3. If no country is found, insert it in the **country** table

4. Insert a record in the **company** table
5. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

### Update

The update procedure follows this schema.

1. Start a transaction
2. Search through the **country** table for country having **code** correctly set
3. If no country is found, insert it in the **country** table
4. Update the corresponding record in the **company** table
5. Commit the transaction

If anything fails during this procedure, a rollback is immediately triggered, guaranteeing that no inconsistent state of the database could be ever reached.

### Deletion

The delete procedure follows this schema.

1. Delete the **company** record

No transaction is used because there is only one SQL query to execute, thus fully succeeding or failing by itself.