



UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA

*La Universidad Católica de Loja*

Informe Preliminar

Fundamentos de Bases de Datos

**Autor:** Kevin Bustamante

Octubre 2022 – Febrero 2023

Introducción .....	3
Modelos .....	4
Dependencia Funcional.....	4
.....	4
Modelo Conceptual.....	4
Modelo Lógico .....	7
Dependencias Funcionales -Tabla Universal-.....	9
Modelo Físico .....	10
Normalización .....	11
First Normal Form (1NF).....	11
Second Normal Form (2NF) .....	12
Third Normal Form (3NF) .....	12
Pasos Para Realizar el Proyecto.....	18
LIMPIEZA COLUMNA CREW .....	40
Conclusiones .....	43

## Introducción

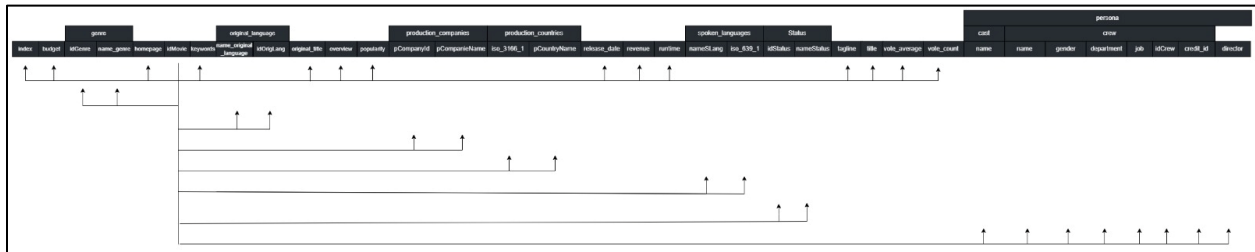
Las bases de datos son una parte fundamental de los sistemas informáticos y han demostrado ser una herramienta crucial a lo largo del tiempo. Nos brindan la oportunidad de almacenar, manipular, ingresar y eliminar una gran cantidad de información. Este informe tiene como objetivo poner en práctica los conocimientos adquiridos durante nuestro estudio de la materia de Base de Datos, a través de la importación de un archivo CSV desde una base de datos usando el sistema MySQL y pasar por diferentes fases, como la inserción de datos, la limpieza de datos, la carga y la explotación de los datos.

Este informe también pretende identificar y diferenciar los métodos utilizados en cada una de las fases mencionadas anteriormente. Además, nuestra intención es generar resultados orientados a la gestión eficiente y uso de la información relevante que se encuentra en el conjunto de datos con el que trabajamos.

A lo largo de este informe, también nos permitió refrescar nuestros conocimientos sobre temas importantes que estudiamos en la materia, como diagramas de Entidad/Relación, modelos conceptuales, lógicos y físicos, creación de consultas SQL y otros temas importantes. Esto nos ayudó a consolidar nuestra comprensión y habilidades en el área de bases de datos.

# Modelos

## Dependencia Funcional



id → { index, budget, homepage, keywords, original\_title, overview, popularity, release\_date, revenue, runtime, tagline, title, vote\_average, vote\_count }

id → { idGenre, name\_Genre }

id → { name\_original\_language, idOrigLang }

id → { pcompanyId, pcompanyName }

id → { iso\_3166\_1, pcountryName }

id → { nameSLang, iso\_639\_1 }

id → { idStatus, nameStatus }

id → { name, gender, department, job, idCrew, credit\_id, director }

## Modelo Conceptual

Después de haber establecido los datos mediante la tabla universal, normalizamos los mismos utilizando la metodología Entidad-Relación, identificando los atributos correspondientes a cada relación. Para ello, utilizamos la herramienta draw.io, lo que nos permitió crear diagramas de flujo efectivamente.

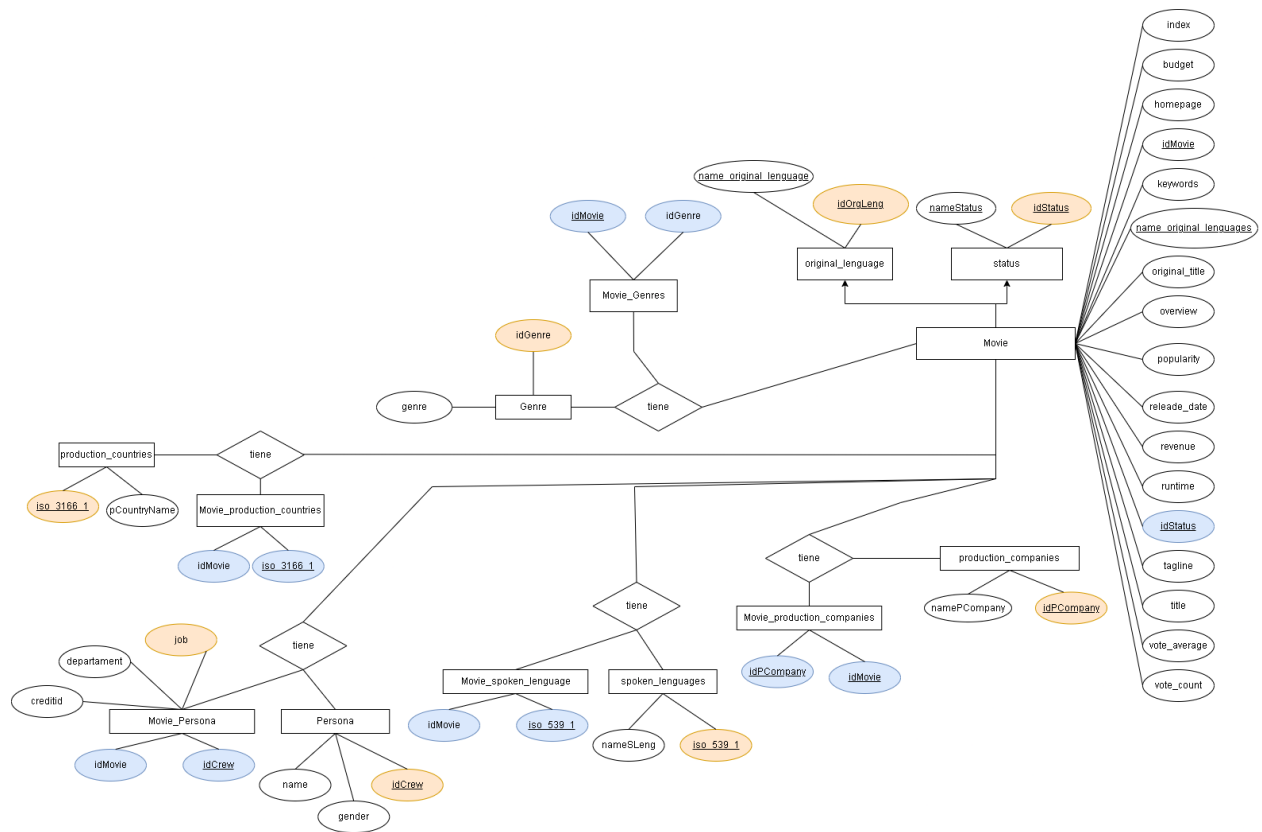


Figura1: Representación del Modelo Conceptual(versión 1)

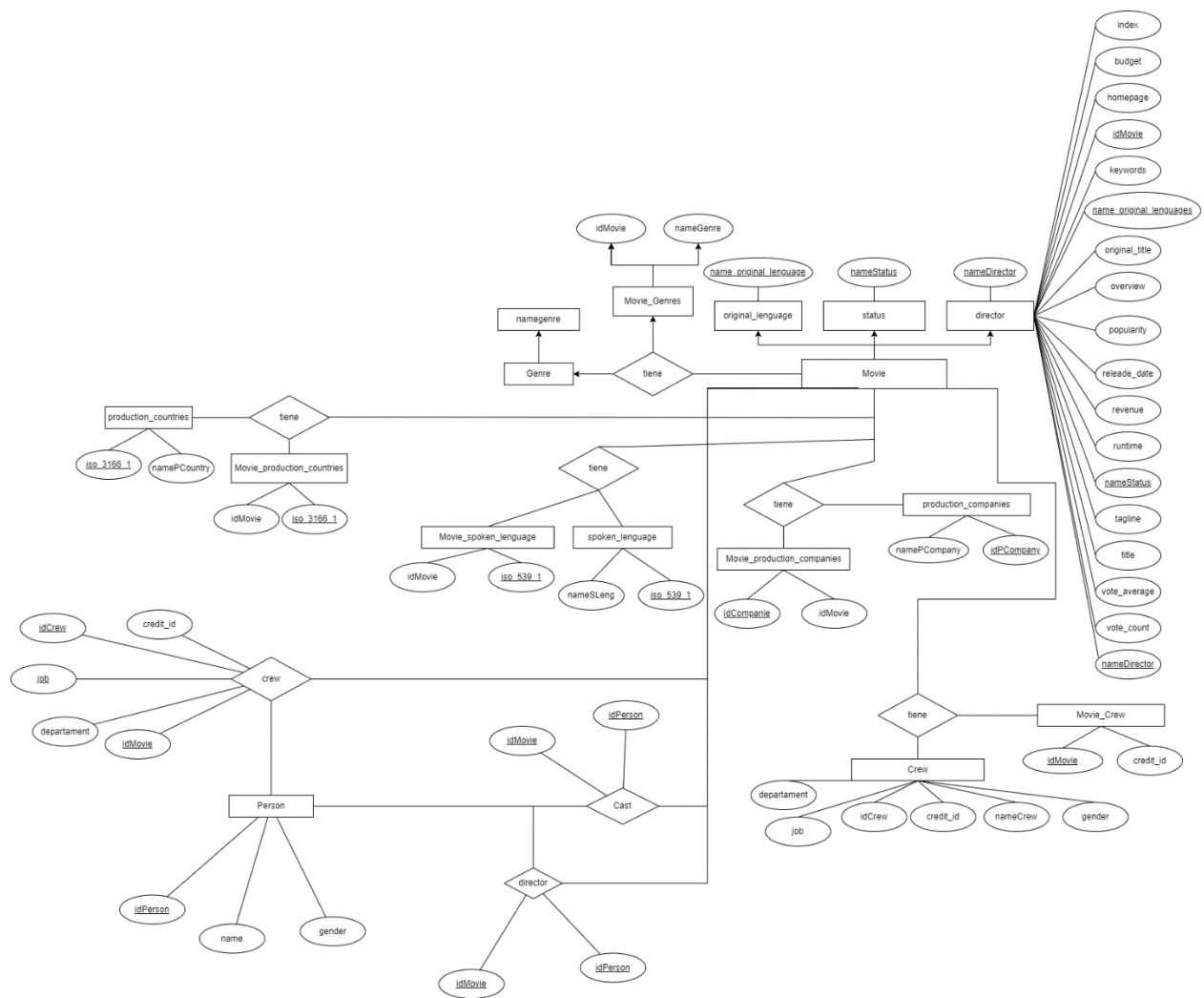


Figura1: Representación del Modelo Conceptual (versión 2)

## Modelo Lógico

Con el modelo conceptual ya realizado, se seleccionan los atributos apropiados para cada tabla y se crea un prototipo de estas basado en las entidades y relaciones identificadas. En cada tabla, se declara una clave primaria y, en su caso, una clave foránea, esto con el objetivo de tener una base sólida para desarrollar el esquema de la base de datos.

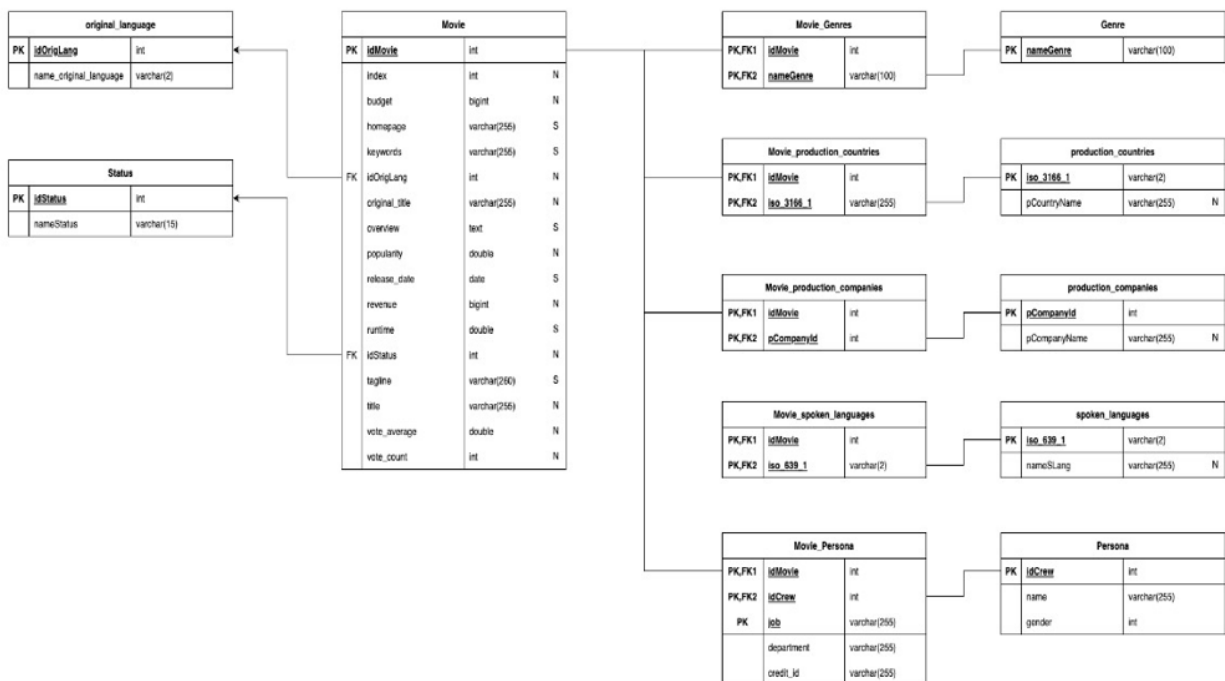


Figura 2: Representación gráfica del Modelo Lógico (versión 1)

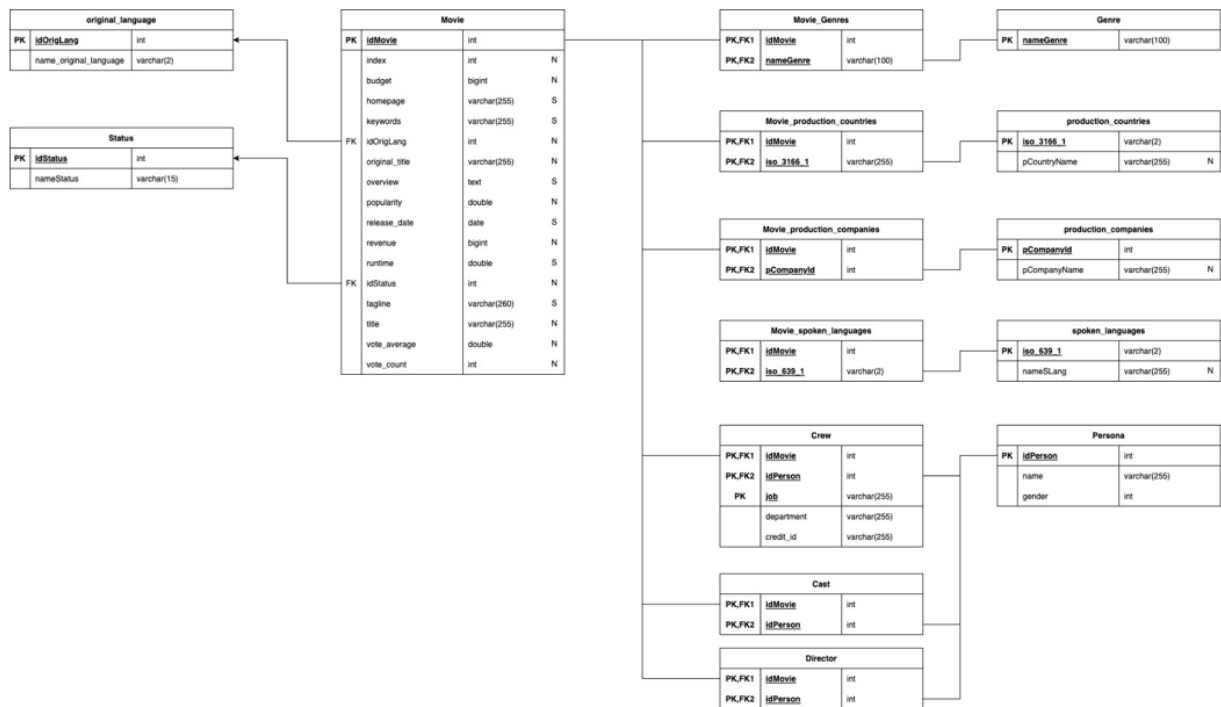


Figura 2: Representación gráfica del Modelo Lógico (versión 2)



## Dependencias Funcionales -Tabla Universal-

Una vez establecido el modelo lógico y conceptual se creó las dependencias funcionales existentes en la tabla universal.

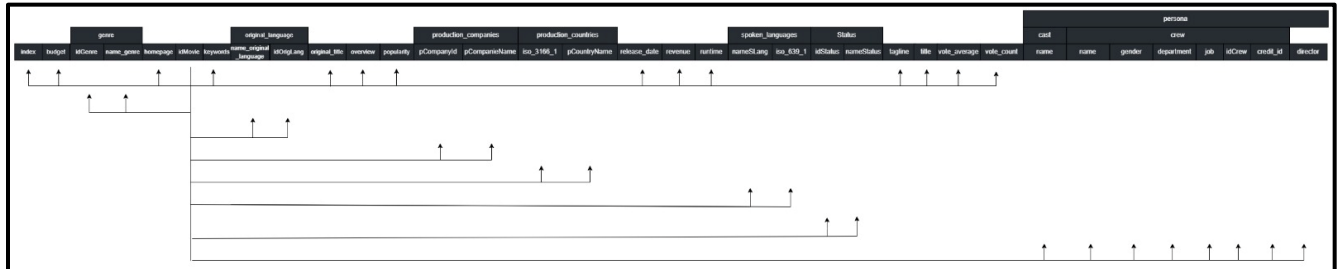


Figura 3: La imagen muestra una descripción gráfica de las dependencias existentes en las columnas del dataset

`id → { index, budget, homepage, keywords, original_title, overview, popularity, release_date, revenue, runtime, tagline, title, vote_avarage, vote_count }`

`id → { idGenre, name_Genre }`

`id → { name_original_language, idOrigLang }`

`id → { pcompanyId, pcompanyName }`

`id → { iso_3166_1, pcountryName }`

`id → { nameSLang, iso_639_1 }`

`id → { idStatus, nameStatus }`

`id → { name, gender, department, job, idCrew, credit_id, director }`

Figura 4: Dependencias Funcionales textualizadas para un mejor entendimiento

## Modelo Físico

Una vez terminado el modelo lógico pasamos a la fase final que es la creación del modelo físico donde se especificó las tablas, columnas, claves principales y a su vez sus claves externas o foráneas con sus respectivas relaciones. Con este modelo podemos pasar a generar las respectivas sentencias DDL

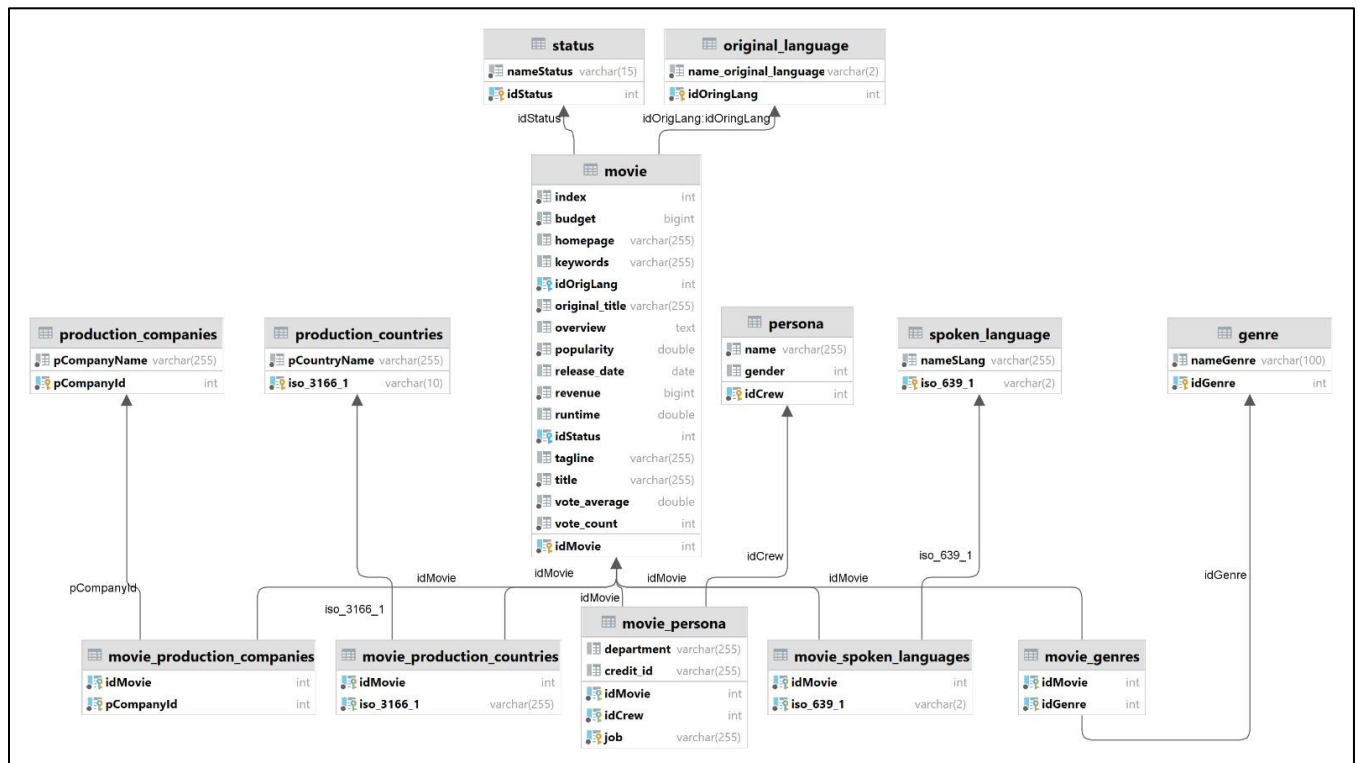


Figura 5: Representación gráfica del Modelo Físico (versión 1)

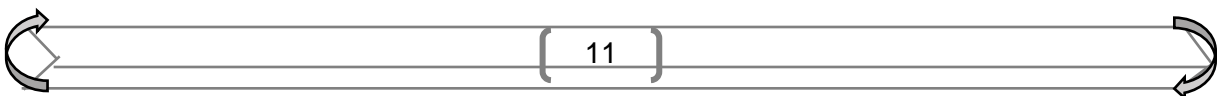
## Normalización

El proceso de normalización se aplica a la estructura de las bases de datos para mejorar en temas de eficiencia, organización e integridad. El objetivo de la normalización es eliminar redundancias y dependencias funcionales, y se logra a través de la división de la información en diferentes tablas relacionadas entre sí. Es importante tener en cuenta el concepto de la normalización antes de profundizar en su aplicación en un proyecto en particular.

Seguimos los siguientes pasos para normalizar los datos del archivo CSV:

### First Normal Form (1NF)

- Una relación en donde la intersección de cada fila y columna contiene un y solo un valor.
- La fase antes de 1NF es Unnormalized Form (UNF) la cual es una tabla que contiene uno más grupos repetidos.
- Para transformar la tabla no normalizada a la primera forma normal, identificamos y eliminamos los grupos que se repiten dentro de la tabla.
- Un grupo repetitivo es un atributo, o grupo de atributos, dentro de una tabla que ocurre con múltiples valores para una sola ocurrencia de los atributos clave designados para esa tabla.



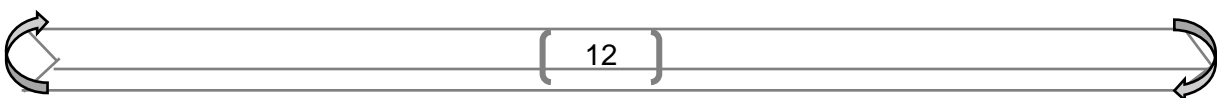
## Second Normal Form (2NF)

- Una relación que está en la primera forma normal y cada atributo que no es de clave principal depende funcionalmente de la clave principal.
- La normalización de las relaciones 1NF a 2NF implica la eliminación de dependencias parciales. Si existe una dependencia parcial, eliminamos los atributos parcialmente dependientes de la relación colocándolos en una nueva relación junto con una copia de su determinante.

## Third Normal Form (3NF)

- Una relación que está en primera y segunda forma normal y en la que ningún atributo que no sea de clave principal depende transitivamente de la clave principal.
- La normalización de las relaciones 2NF a 3NF implica la eliminación de las dependencias transitivas.
- Si existe una dependencia transitiva, eliminamos los atributos transitivamente dependientes de la relación colocando los atributos en una nueva relación junto con una copia del determinante.

Primero identificamos y eliminamos grupos repetitivos dentro de la tabla. Un grupo repetitivo es un atributo, o grupo de atributos, dentro de una tabla que ocurre con múltiples valores para una sola ocurrencia de los atributos clave designados para esa tabla.



### Tabla Uno a Muchos (1 : N)

Status es una columna con tres posibles valores (released, rumored, post-production)

- Una película puede tener un Status, pero un Status puede representar a muchas películas.
- Para solucionar, se crea una tabla separada que se llame Status.
- La llave primaria es statusName.

Como solución, utilizamos el statusName como llave foránea en Movie. Director es una columna con el nombre de un único director.

- Una película puede tener un Director, pero un Director puede tener muchas películas.
- Para solucionar, se crea una tabla separada que se llame Director.
- La llave primaria es directorName.
- Como solución, utilizamos el directorName como llave foránea en Movie.

Original\_language es una columna con el nombre del lenguaje original de la Movie.

- Una película tiene un Original\_language, pero un Original\_language puede tener muchas películas.
- Para solucionar, se crea una tabla separada que se llame original\_language.
- La llave primaria es origLanName.
- Como solución, utilizamos el origLanName como llave foránea en original\_language.

### Tabla Muchos a Muchos (N : N)

En nuestro caso, ninguna de las 4803 entradas se repite, cada película es diferente (única).

Cada Movie tiene un index y un id diferente. Como llave primaria utilizamos id el cual nombramos idMovie para no confundirlo con otros id que existan en otras columnas.

- Genres contiene un String de géneros que puede contener 0 a n géneros.
  - Para solucionar, cada columna debe contener un solo valor.
  - En este caso tenemos una lista de géneros. Existen múltiples valores.
  - La solución es crear una tabla separada que se llame Genres.
  - La llave primaria es genreName
  - Una Movie puede tener muchos géneros, y un género puede estar en muchas Movie.
  - La relación (muchos a muchos) se soluciona con una tabla llamada Movie\_Genres utilizando la llave primaria de cada uno.
  
- Keywords contiene un String de keywords que puede contener (0 a N) keywords.
  - Para solucionar, cada columna debe contener un solo valor.
  - En este caso tenemos una lista de keywords. Existen múltiples valores.
  - La solución es crear una tabla separada que se llame Keywords.
  - La llave primaria es keywordName
  - Una Movie puede tener muchos keywords, y un keyword puede aparecer en muchas Movie.
  - La relación (muchos a muchos) se soluciona con una tabla llamada Movie\_Keywords utilizando la llave primaria de cada uno.

- Production\_companies contiene un String de JSON que contiene un name y un id que puede contener 0 a n production\_companies.
  - Para solucionar, cada columna debe contener un solo valor.
  - En este caso tenemos una lista de production\_companies. Existe múltiples valores.
  - La solución es crear una tabla separada que se llame Production\_companies.
  - Tiene dos atributos, name e id los cuales los nombramos prodCompName y prodCompId el cual es la primary key.
  - Una Movie puede tener muchos production\_companies, y un production\_companies puede aparecer en muchas Movie.
  - La relación (muchos a muchos) se soluciona con una tabla llamada Movie\_Production\_companies utilizando la llave primaria de cada uno.
- Production\_countries contiene un String de JSON que contiene un iso\_3166\_1 y un name que puede contener 0 a n production\_countries.
  - Para solucionar, cada columna debe contener un solo valor.
  - En este caso tenemos una lista de production\_countries. Existen múltiples valores.
  - La solución es crear una tabla separada que se llame Production\_countries.
  - Tiene dos atributos, iso\_3166\_1 y name los cuales los nombramos, iso\_3166\_1 el cual es la primary key y prodCompName.

- Una Movie puede tener muchos production\_countries, y un production\_countries puede aparecer en muchas Movie.
  - La relación (muchos a muchos) se soluciona con una tabla llamada Movie\_Production\_countries utilizando la llave primaria de cada uno.
- spoken\_languages contiene un String de JSON que contiene un iso\_639\_1 y un name que puede contener 0 a n spoken\_languages.
    - Para solucionar, cada columna debe contener un solo valor.
    - En este caso tenemos una lista de spoken\_languages. Existen múltiples valores.
      - La solución es crear una tabla separada que se llame spoken\_languages.
      - Tiene dos atributos, iso\_639\_1 y name los cuales los nombramos, iso\_639\_1 el cual es la primary key y spokLangName.
      - Una Movie puede tener muchos spoken\_languages, y un spoken\_languages puede aparecer en muchas Movie.
      - La relación (muchos a muchos) se soluciona con una tabla llamada Movie\_Spoken\_languages utilizando la llave primaria de cada uno.
  - **'Cast', 'Crew' y 'Director'** contienen los nombres de integrantes que participan de alguna forma en la película. Estos necesitan ser normalizados en una tabla con datos comunes, en este caso llamada **Persona**.
  - Se crea una tabla Persona que incluya los atributos comunes a todas las entidades, entre los cuales están: gender, idCrew y name.



- Ya que la relacion Persona-Movie es muchos a muchos, por consecuencia se necesita otra tabla que contenga el idMovie e idCrew.
- Es momento de crear tablas temporaes de '*Cast*', '*Crew*' y '*Director*', para la migracion de datos hacia la tabla persona.
- Antes de migrar los datos a la tabla Persona, es importante realizar una limpieza de datos para garantizar que los datos sean precisos y coherentes.

## Pasos Para Realizar el Proyecto

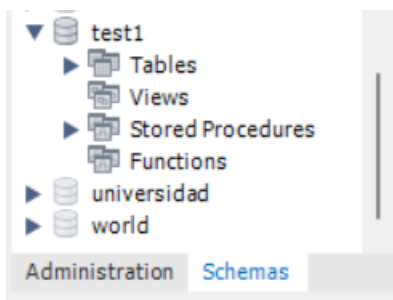
### 1. Creación del DDL

Al ya tener claro como estará diseñado las dependencias funcionales, el modelo conceptual, lógico y físico, procedemos a materializar las tablas en el DDL, donde comenzaremos creando una database(figura 6).

- `DROP DATABASE IF EXISTS test1;`
- `CREATE DATABASE test1 CHARACTER SET utf8mb4;`
- `USE test1;`

(Figura 6): creación de la base de datos

Al ejecutar este comando en la parte de ‘schemas’ ya aparecerá nuestra base de datos(figura 7)



(Figura 7): Base de datos visible

El siguiente paso sería crear todas las tablas a las que se ha concluido con los modelos formulados anteriormente

```

9 • DROP TABLE IF EXISTS `Status` ;
10
11 • CREATE TABLE IF NOT EXISTS `Status` (
12     `idStatus` INT NOT NULL AUTO_INCREMENT,
13     `status` VARCHAR(255) NULL,
14     PRIMARY KEY (`idStatus`))
15     ENGINE = InnoDB;
16
17
18 -----
19 -- Table `Original_language`
20 -----
21 • DROP TABLE IF EXISTS `Original_language` ;
22 • CREATE TABLE IF NOT EXISTS `Original_language` (
23     `idOriginal_language` INT AUTO_INCREMENT,
24     `language` VARCHAR(255) NULL,
25     PRIMARY KEY (`idOriginal_language`))
26     ENGINE = InnoDB;
27

```

```

28 -----
29 -- Table `Movies`
30 -----
31
32 • DROP TABLE IF EXISTS `Movies` ;
33 • CREATE TABLE IF NOT EXISTS `Movies` (
34     `idMovie` INT NOT NULL AUTO_INCREMENT,
35     `index` INT NOT NULL,
36     `budget` INT NULL,
37     `homepage` VARCHAR(1000) NULL,
38     `original_title` VARCHAR(1000) NULL,
39     `overview` VARCHAR(5000) NULL,
40     `popularity` DOUBLE NULL,
41     `release_date` DATE NULL,
42     `revenue` INT NULL,
43     `runtime` DOUBLE NULL,
44     `tagline` VARCHAR(1000) NULL,
45     `title` VARCHAR(1000) NULL,
46     `keywords` TEXT(1000) NULL,
47     `vote_count` INT NULL,
48     `vote_average` DOUBLE NULL,
49     `idStatus` INT NOT NULL,
50     `idOriginal_language` INT NOT NULL,
51     PRIMARY KEY (`idMovie`),
52     FOREIGN KEY (`idStatus`)
53     REFERENCES `Status` (`idStatus`),
54     FOREIGN KEY (`idOriginal_language`)
55     REFERENCES `Original_language` (`idOriginal_language`))
56     ENGINE = InnoDB;

```

```

-----
-- Table `Production_companie`
-----
DROP TABLE IF EXISTS `Production_companie` ;
CREATE TABLE IF NOT EXISTS `Production_companie` (
    `idProduction_companie` INT NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(255) NULL,
    PRIMARY KEY (`idProduction_companie`))
ENGINE = InnoDB;

-----
-- Table `Companie_movie`
-----
DROP TABLE IF EXISTS `Companie_movie` ;
CREATE TABLE IF NOT EXISTS `Companie_movie` (
    `idProduction_companie` INT NOT NULL,
    `idMovie` INT NOT NULL,
    PRIMARY KEY (`idProduction_companie`, `idMovie`),
    FOREIGN KEY (`idProduction_companie`)
    REFERENCES `Production_companie` (`idProduction_companie`),
    FOREIGN KEY (`idMovie`)
    REFERENCES `Movies` (`idMovie`))
ENGINE = InnoDB;

```

```

DROP TABLE IF EXISTS `Production_countrie` ;
> CREATE TABLE IF NOT EXISTS `Production_countrie` (
  `isoCountrie` VARCHAR(2) NOT NULL,
  `name` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`isoCountrie`))
ENGINE = InnoDB;

```

```

-- Table `Countrie_movie`

DROP TABLE IF EXISTS `Countrie_movie` ;
> CREATE TABLE IF NOT EXISTS `Countrie_movie` (
  `isoCountrie` VARCHAR(2) NOT NULL,
  `idMovie` INT NOT NULL,
  PRIMARY KEY (`isoCountrie`, `idMovie`),
  FOREIGN KEY (`isoCountrie`)
    REFERENCES `Production_countrie` (`isoCountrie`),
  FOREIGN KEY (`idMovie`)
    REFERENCES `Movies` (`idMovie`))
ENGINE = InnoDB;

```

```

DROP TABLE IF EXISTS `Spoken_language` ;
CREATE TABLE IF NOT EXISTS `Spoken_language` (
  `isoLang` VARCHAR(2) NOT NULL,
  `name` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`isoLang`))
ENGINE = InnoDB;

```

```

-- Table `Language_movie`

DROP TABLE IF EXISTS `Language_movie` ;
CREATE TABLE IF NOT EXISTS `Language_movie` (
  `idMovie` INT NOT NULL,
  `isoLang` VARCHAR(2) NOT NULL,
  PRIMARY KEY (`idMovie`, `isoLang`),
  FOREIGN KEY (`idMovie`)
    REFERENCES `Movies` (`idMovie`),
  FOREIGN KEY (`isoLang`)
    REFERENCES `Spoken_language` (`isoLang`))
ENGINE = InnoDB;

```

- DROP TABLE IF EXISTS `Genre` ;
- CREATE TABLE IF NOT EXISTS `Genre` (
  - `idGenre` INT NOT NULL AUTO\_INCREMENT,
  - `name` VARCHAR(255) NOT NULL,
  - PRIMARY KEY (`idGenre`))
 ENGINE = InnoDB;

```

-- Table `Genre_movie`

```

- DROP TABLE IF EXISTS `Genre\_movie` ;
- CREATE TABLE IF NOT EXISTS `Genre\_movie` (
  - `idMovie` INT NOT NULL,
  - `idGenre` INT NOT NULL,
  - PRIMARY KEY (`idMovie`, `idGenre`),
  - FOREIGN KEY (`idMovie`)
    - REFERENCES `Movies` (`idMovie`),
  - FOREIGN KEY (`idGenre`)
    - REFERENCES `Genre` (`idGenre`))
 ENGINE = InnoDB;

- DROP TABLE IF EXISTS `Person` ;
- CREATE TABLE IF NOT EXISTS `Person` (
  - `idPerson` INT NOT NULL,
  - `name` VARCHAR(255) NULL,
  - `gender` INT NULL,
  - PRIMARY KEY (`idPerson`))
 ENGINE = InnoDB;



```

DROP TABLE IF EXISTS `Crew` ;
CREATE TABLE IF NOT EXISTS `Crew` (
  `idMovie` INT NOT NULL,
  `job` VARCHAR(255) NOT NULL,
  `idPerson` INT NOT NULL,
  `department` VARCHAR(255) NULL,
  `credit_id` VARCHAR(255) NULL,
  PRIMARY KEY (`idMovie`, `job`, `idPerson`),
  FOREIGN KEY (`idMovie`)
    REFERENCES `Movies` (`idMovie`),
  FOREIGN KEY (`idPerson`)
    REFERENCES `Person` (`idPerson`))
ENGINE = InnoDB;

DROP TABLE IF EXISTS `Cast` ;
CREATE TABLE IF NOT EXISTS `Cast` (
  `idPerson` INT NOT NULL,
  `idMovie` INT NOT NULL,
  PRIMARY KEY (`idPerson`, `idMovie`),
  FOREIGN KEY (`idPerson`)
    REFERENCES `Person` (`idPerson`),
  FOREIGN KEY (`idMovie`)
    REFERENCES `Movies` (`idMovie`))
ENGINE = InnoDB;

-----
-- Table `Director`
-----
DROP TABLE IF EXISTS `Director` ;
CREATE TABLE IF NOT EXISTS `Director` (
  `idPerson` INT NOT NULL,
  `idMovie` INT NOT NULL,
  PRIMARY KEY (`idPerson`, `idMovie`),
  FOREIGN KEY (`idPerson`)
    REFERENCES `Person` (`idPerson`),
  FOREIGN KEY (`idMovie`)
    REFERENCES `Movies` (`idMovie`))
ENGINE = InnoDB;

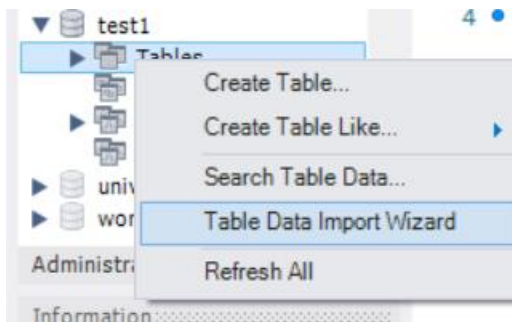
```

## 2. Importar el dataset a la base de datos

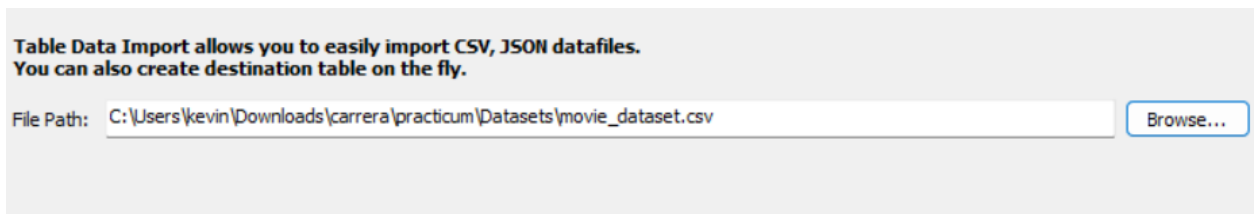
- El siguiente paso a tener en cuenta es que necesitamos la información dentro de nuestra base de datos para poder aplicar los modelos anteriores, para esto se hará clic derecho a donde dice tables dentro de nuestra base de datos y haremos clic en table data import wizard(figura 8).

- Se nos abrirá una ventana donde tendremos que poner la localización de nuestro dataset (figura 9):
- A continuación nos saldrá una pestaña que nos indicara en que database queremos que se cree la tabla (figura 10)
- En la siguiente pestaña configuraremos todo en base a nuestro dataset (figura 11).

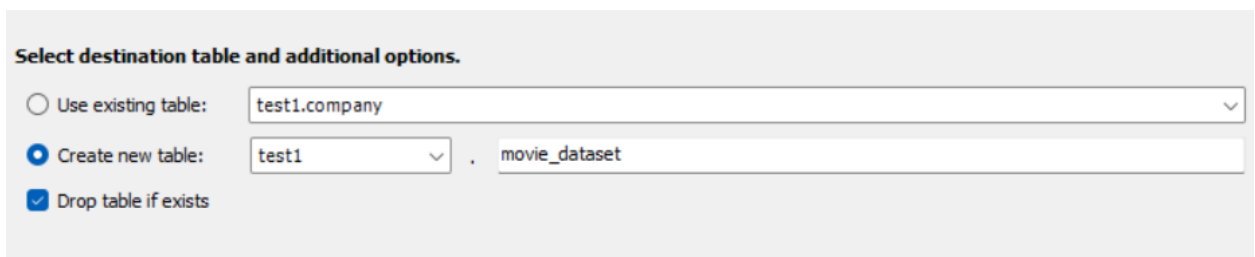
- Y con esto ya podremos tener la última pestaña que solo nos indicara cuando se importe el dataset(figura 12). Y podremos observar nuestra tabla creada y poblada con la información del dataset(figura13 y 14).



(figura 8)




(figura 9): ubicación del dataset



(figura 10): creación de la tabla en una database seleccionada

### Configure Import Settings

Detected file format: csv 

Options:

Field Separator	,
Line Separator	LF
Enclose Strings in	"
null and NULL word as SQL keyword	YES

Encoding: utf-8

Columns:		Field Type
<input checked="" type="checkbox"/> Source Column		
<input checked="" type="checkbox"/> index		int
<input checked="" type="checkbox"/> budget		int
<input checked="" type="checkbox"/> genres		text
<input checked="" type="checkbox"/> homepage		text
<input checked="" type="checkbox"/> id		int
<input checked="" type="checkbox"/> keywords		text

< Back   Next >   Cancel

(figura 11): ajustando datos del dataset para la inserción en tabla

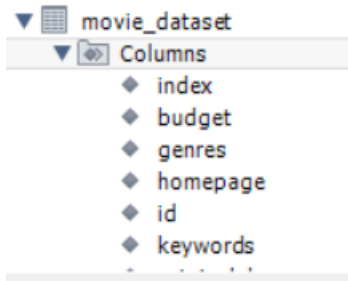
The following tasks will now be performed. Please monitor the execution.

☐ Prepare Import

☐ Import data file

Click [Next >] to execute.

(Figura12)



(figura 13)

index	budget	genres	homepage	id	keywords	original_language	original_title
0	237000000	Action Adventure Fantasy Science Fiction	http://www.avatarmovie.com/	19995	culture clash future space war space colony soc...	en	Avatar
1	300000000	Adventure Fantasy Action	http://disney.go.com/disneypictures/pirates/	285	ocean drug abuse exotic island east india tradin...	en	Pirates of the Caribbean: At World's End
2	245000000	Action Adventure Crime	http://www.sonypictures.com/movies/spectre/	206647	spy based on novel secret agent sequel mi6	en	Spectre
3	250000000	Action Crime Drama Thriller	http://www.thedarkknightrises.com/	49026	dc comics crime fighter terrorist secret identity ...	en	The Dark Knight Rises

(figura 14)

### 3. Creación de funciones que Permitan Extraer y Limpiar los Datos.

- Tratar la tabla original\_language

```
USE test1;

DROP PROCEDURE IF EXISTS TablaOriginalLanguage;

DELIMITER $$
CREATE PROCEDURE TablaOriginalLanguage()
BEGIN

DECLARE done INT DEFAULT FALSE;
DECLARE nameOL VARCHAR(100);

-- Declarar el cursor
DECLARE CursorOL CURSOR FOR
    SELECT DISTINCT original_language AS names from movie_dataset;

-- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

-- Abrir el cursor
OPEN CursorOL;
CursorOL_loop: LOOP
    FETCH CursorOL INTO nameOL;

    -- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
    IF done THEN
        LEAVE CursorOL_loop;
    END IF;
    IF nameOL IS NULL THEN
        SET nameOL = '';
    END IF;
    SET @oStatement = CONCAT('INSERT INTO original_languageCURSOR (name) VALUES (\',
    nameOL, '\');');
    PREPARE sent1 FROM @oStatement;
    EXECUTE sent1;
    DEALLOCATE PREPARE sent1;

    CLOSE CursorOL;
END $$
DELIMITER ;

CALL TablaOriginalLanguage();

DROP TABLE IF EXISTS original_languageCURSOR

CREATE TABLE original_languageCURSOR (
    name varchar(255) PRIMARY KEY
);

SELECT * FROM original_languageCURSOR;
```



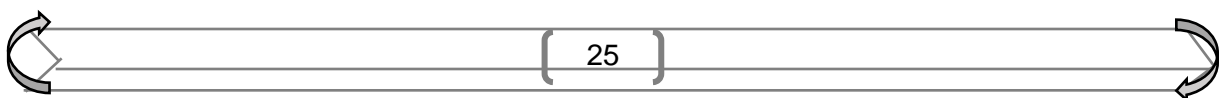
Esta es una descripción de un procedimiento que crea una tabla llamada "original\_language" y la llena con valores de la tabla "movie\_dataset". Primero, se verifica si ya existe un procedimiento con el nombre "TablaOriginal\_language", y en caso de que sí, se borra. Después, se establece un delimitador personalizado "\$\$" en lugar del delimitador estándar ";".

Luego, se define el procedimiento "TablaOriginal\_language". Se crea una variable llamada "namelanguage" de tipo VARCHAR (100), que se utilizará para almacenar los valores obtenidos de un cursor. Se crea un cursor llamado "Cursorlanguage" que selecciona los valores únicos de la columna "original\_language" de la tabla "movie\_dataset". También se debe crear un controlador de continuación llamado "NOT FOUND" que se activará cuando el cursor haya alcanzado su final.

Se abre el cursor y se entra en un bucle "CursorDirector\_loop". Dentro de este bucle, se extrae la siguiente fila del cursor y se almacena en la variable "namelanguage". Si se alcanza el final del cursor, se sale del bucle. Se verifica si la variable "namelanguage" es NULL y en caso de ser así, se establece en una cadena vacía.

Luego, se concatena una consulta SQL para insertar el valor de "namelanguage" en la tabla "original\_language". La consulta se prepara y se ejecuta, y luego se dealloca. Finalmente, se cierra el cursor, se termina el procedimiento y se establece de nuevo el delimitador estándar ";". La línea "CALL TablaOriginal\_language();" ejecuta el procedimiento. La tabla "status" indica el estado final.

- **Tratar status**



```

DROP PROCEDURE IF EXISTS TablaStatus;

DELIMITER $$
CREATE PROCEDURE TablaStatus()
) BEGIN

DECLARE done INT DEFAULT FALSE;
DECLARE nameStatus VARCHAR(100);

-- Declarar el cursor
DECLARE CursorStatus CURSOR FOR
SELECT DISTINCT CONVERT(status USING UTF8MB4) AS names from movie_dataset;

-- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

-- Abrir el cursor
OPEN CursorStatus;

OPEN CursorStatus;
CursorStatus_loop: LOOP
    FETCH CursorStatus INTO nameStatus;

-- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
    IF done THEN
        LEAVE CursorStatus_loop;
    END IF;
    IF nameStatus IS NULL THEN
        SET nameStatus = '';
    END IF;
    SET @_oStatement = CONCAT('INSERT INTO statusCURSOR (name) VALUES (\'',
    nameStatus, '\');');
    PREPARE sent1 FROM @_oStatement;
    EXECUTE sent1;
    DEALLOCATE PREPARE sent1;

END LOOP;

CALL TablaStatus();

DROP TABLE IF EXISTS statusCURSOR;

CREATE TABLE statusCURSOR (
    name varchar(255) PRIMARY KEY
);

SELECT * FROM statusCURSOR;

```

Para este procedimiento se usó lo mismo que con la tabla original languages, la única diferencia son las variables que se manejan.

- Tratar tabla movie

```
DROP PROCEDURE IF EXISTS TablaMovie;

DELIMITER $$

CREATE PROCEDURE TablaMovie()
BEGIN

DECLARE done INT DEFAULT FALSE;
DECLARE Movindex INT;
DECLARE Movbudget BIGINT;
DECLARE Movhomepage VARCHAR(1000);
DECLARE MovidMovie INT;
DECLARE Movkeywords TEXT;
DECLARE Movoriginal_language VARCHAR(255);
DECLARE Movoriginal_title VARCHAR(255) ;
DECLARE Movoverview TEXT;
DECLARE Movpopularity DOUBLE;
DECLARE Movrelease_date VARCHAR(255);
DECLARE Movrevenue BIGINT;
DECLARE Movruntime DOUBLE;
DECLARE Movstatus VARCHAR(255);
DECLARE Movtagline VARCHAR(255);
DECLARE Movtitle VARCHAR(255);
DECLARE Movvote_average DOUBLE;
DECLARE Movvote_count INT;
DECLARE nameDirector VARCHAR(255);

DECLARE Director_nameDirector varchar(255);
```

```

DECLARE Director_nameDirector varchar(255);
DECLARE Director_nameStatus varchar(255);
DECLARE Director_nameOriginal_language varchar(255);

-- Declarar el cursor
DECLARE CursorMovie CURSOR FOR
    SELECT 'index', budget, homepage, id, keywords, original_language, original_title, overview, popularity, release_date, revenue, runtime, 'status',
        tagline, title, vote_average, vote_count, director FROM movie_dataset;

-- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

-- Abrir el cursor
OPEN CursorMovie;

| CursorMovie_loop: LOOP
    FETCH CursorMovie INTO Movindex, Movbudget, Movhomepage, MovidMovie, Movkeywords, Movoriginal_language, Movoriginal_title, Movoverview,
        Movpopularity, Movrelease_date, Movrevenue, Movruntime, Movstatus, Movtagline, Movtitle, Movvote_average, Movvote_count, nameDirector;

● DROP TABLE IF EXISTS MovieCURSOR;

● CREATE TABLE MovieCURSOR (
    'index' int,
    budget bigint,
    homepage varchar(1000),
    id int PRIMARY KEY,
    keywords TEXT,
    original_language varchar(255),
    original_title varchar(255),
    overview TEXT,
    popularity double,
    release_date varchar(255),
    revenue bigint,
    runtime double,
    'status' varchar(255),
    tagline varchar(255),
    title varchar(255),
    vote_average double,
    vote_count int,
    director varchar(255),
    FOREIGN KEY (original_language) REFERENCES original_languageCURSOR(name),
    FOREIGN KEY (status) REFERENCES statusCURSOR(name),
    FOREIGN KEY (director) REFERENCES directorCURSOR(name)
);

● DROP TABLE IF EXISTS MovieCURSOR;

```

Este procedimiento comienza definiendo variables para almacenar los diferentes aspectos de una película, tales como el ID, título original, popularidad, fecha de lanzamiento, ganancias, duración, resumen, entre otros. Después, se crea un cursor llamado "CursorMovie" para seleccionar todos los atributos de las películas en la tabla "movie\_dataset". También se establece un controlador de continuación "NOT FOUND" para marcar el final del cursor. Luego, se abre el cursor y se inicia un bucle para recorrer todas las películas. En cada iteración, se utiliza el

comando "FETCH" para extraer los valores de los atributos y almacenarlos en las variables correspondientes. Además, se realizan consultas para obtener los IDs de la película en las tablas de estado y idioma original. Finalmente, con todos los valores recopilados, se inserta una nueva fila en la tabla "movies" con todos los atributos de la película. El bucle continúa hasta que se haya alcanzado el final del curso, el cual es el momento en el que se cierra.

- **Tratar Production companies**

```
USE test1;

DROP PROCEDURE IF EXISTS TablaProduction_companies;

DELIMITER $$
CREATE PROCEDURE TablaProduction_companies ()

BEGIN

    DECLARE done INT DEFAULT FALSE ;
    DECLARE jsonData json ;
    DECLARE jsonId varchar(250) ;
    DECLARE jsonLabel varchar(250) ;
    DECLARE resultSTR LONGTEXT DEFAULT '';
    DECLARE i INT;

    -- Declarar el cursor
    DECLARE myCursor
    CURSOR FOR

        SELECT JSON_EXTRACT(CONVERT(production_companies USING UTF8MB4), '$[*]') FROM movie_dataset ;

    -- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
    DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET done = TRUE ;

    -- Abrir el cursor
    OPEN myCursor ;
    drop table if exists production_companietem;
    SET @sql_text = 'CREATE TABLE production_companietem ( id int, nameCom VARCHAR(100));';
    PREPARE stmt FROM @sql_text;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;

    cursorLoop: LOOP
        FETCH myCursor INTO jsonData;

        -- Controlador para buscar cada uno de los arrays
        SET i = 0;

        -- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
    IF done THEN
        LEAVE cursorLoop ;
    END IF ;

    WHILE(JSON_EXTRACT(jsonData, CONCAT('$[', i, ']')) IS NOT NULL) DO
        SET jsonId = IFNULL(JSON_EXTRACT(jsonData, CONCAT('$[', i, '].id')), '');
        SET jsonLabel = IFNULL(JSON_EXTRACT(jsonData, CONCAT('$[', i, '].name')), '');
        SET i = i + 1;

        SET @sql_text = CONCAT('INSERT INTO production_companietem VALUES (' , REPLACE(jsonId, '\'', '') , ', ' , jsonLabel, '); ');
        PREPARE stmt FROM @sql_text;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;

    END LOOP ;

    select distinct * from production_companietem;
    INSERT INTO production_companiesCURSOR
    SELECT DISTINCT id, nameCom
    FROM production_companietem;
    drop table if exists production_companietem;
    CLOSE myCursor ;

ENDSS
DELIMITER ;

call TablaProduction_companies();

CREATE TABLE production_companiesCURSOR (
    id INT PRIMARY KEY,
    name varchar(100)
);
```

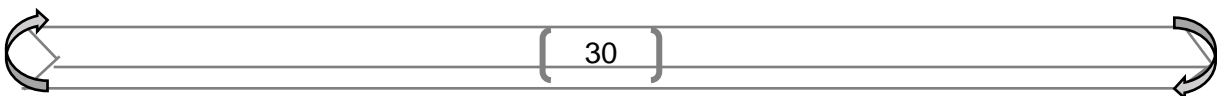
Este procedimiento tiene como objetivo crear una tabla llamada "production\_companie" a partir de la información contenida en la columna "production\_companies" de la tabla "movie\_dataset". Para llevar a cabo este proceso, se definen una serie de variables, incluyendo un cursor denominado "myCursor". Este cursor se utilizará para recorrer los diferentes registros contenidos

en la columna "production\_companies". Además, se define un handler llamado "continue", que indicará cuándo se ha llegado al final del cursor.

Posteriormente, se crea una tabla temporal llamada "production\_companieTem" para almacenar los valores que se extraen de la columna "production\_companies". Después, se abre el cursor y se inicia un loop para recorrer cada uno de los registros contenidos en la columna "production\_companies". En cada iteración, se utiliza la función JSON\_EXTRACT para obtener los valores de "id" y "name" de cada registro. Estos valores se insertan en la tabla temporal "production\_companieTem".

Una vez que se han recorrido todos los registros en la columna "production\_companies", se cierra el cursor y se insertan en la tabla "production\_companie" solo los valores distintos. Finalmente, se elimina la tabla temporal "production\_companieTem" una vez que se han completado todas las operaciones.

- **TratarProduction countries**



```

USE test1;

DROP PROCEDURE IF EXISTS TablaProduction_countries;

DELIMITER $$
CREATE PROCEDURE TablaProduction_countries ()

BEGIN

    DECLARE done INT DEFAULT FALSE ;
    DECLARE jsonData json ;
    DECLARE jsonId varchar(250) ;
    DECLARE jsonLabel varchar(250) ;
    DECLARE resultSTR LONGTEXT DEFAULT '';
    DECLARE i INT;

    -- Declarar el cursor
    DECLARE myCursor
    CURSOR FOR
        SELECT JSON_EXTRACT(CONVERT(production_countries USING UTF8MB4), '$[*]') FROM movie_dataset ;

    -- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
    DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET done = TRUE ;

    -- Abrir el cursor
    OPEN myCursor ;
    drop table if exists production_countriesTem;
    SET @sql_text = 'CREATE TABLE production_countriesTem ( iso_3166_1 varchar(2), nameCountry VARCHAR(100));';
    PREPARE stmt FROM @sql_text;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
) cursorLoop: LOOP
    FETCH myCursor INTO jsonData;

    -- Controlador para buscar cada uno de los arrays
    SET i = 0;

    -- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
) IF done THEN
    LEAVE cursorLoop ;

    WHILE(JSON_EXTRACT(jsonData, CONCAT('$[', i, ']')) IS NOT NULL) DO
    SET jsonId = IFNULL(JSON_EXTRACT(jsonData, CONCAT('$[', i, '].iso_3166_1')), '');
    SET jsonLabel = IFNULL(JSON_EXTRACT(jsonData, CONCAT('$[', i, '].name')), '');
    SET i = i + 1;

    SET @sql_text = CONCAT('INSERT INTO production_countriesTem VALUES (' , REPLACE(jsonId, '\', ''), ', ' , jsonLabel, '); ' );
    PREPARE stmt FROM @sql_text;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;

    END WHILE;

END LOOP ;

select distinct * from production_countriesTem;
INSERT INTO production_countriesCURSOR
SELECT DISTINCT iso_3166_1, nameCountry
FROM production_countriesTem;
drop table if exists production_countriesTem;

        SELECT DISTINCT iso_3166_1, nameCountry
        FROM production_countriesTem;
        drop table if exists production_countriesTem;
        CLOSE myCursor ;

    END$$
DELIMITER ;

call TablaProduction_countries();

SELECT * FROM production_countriesCURSOR;

) CREATE TABLE production_countriesCURSOR (
    iso_3166_1 varchar(2) PRIMARY KEY,
    name varchar(100)
);

DROP TABLE IF EXISTS production_countriesCURSOR;

SELECT COUNT(*) FROM production_countriesCURSOR;

```

Este procedimiento ya ha sido explicado antes (Production\_companies), solo que aquí se le ha cambiado las variables.

- **Tratar SpokenLanguage**

```

DROP PROCEDURE IF EXISTS TablaSpokenLanguages;

DELIMITER $$
CREATE PROCEDURE TablaSpokenLanguages ()
|
BEGIN

    DECLARE done INT DEFAULT FALSE ;
    DECLARE jsonData json ;
    DECLARE jsonId varchar(250) ;
    DECLARE jsonLabel varchar(250) ;
    DECLARE resultSTR LONGTEXT DEFAULT '';
    DECLARE i INT;

    -- Declarar el cursor
    DECLARE myCursor
    CURSOR FOR
        SELECT JSON_EXTRACT(CONVERT(spoken_languages USING UTF8MB4), '$[*]') FROM movie_dataset ;

    DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET done = TRUE ;

    -- Abrir el cursor
    OPEN myCursor ;
    drop table if exists spokenLanguagesTem;
    SET @sql_text = 'CREATE TABLE spokenLanguagesTem ( iso_639_1 varchar(2), nameLang VARCHAR(100));';
    PREPARE stmt FROM @sql_text;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
) cursorLoop: LOOP
    FETCH myCursor INTO jsonData;

    -- Controlador para buscar cada uno de los arrays
    SET i = 0;

    -- Si alcanza el final del cursor entonces salir del ciclo repetitivo
    IF done THEN
        LEAVE cursorLoop ;
    )

    IF done THEN
        LEAVE cursorLoop ;
    END IF ;

    WHILE(JSON_EXTRACT(jsonData, CONCAT('$[', i, ']')) IS NOT NULL) DO
        SET jsonId = IFNULL(JSON_EXTRACT(jsonData, CONCAT('$[', i, '].iso_639_1')), '');
        SET jsonLabel = IFNULL(JSON_EXTRACT(jsonData, CONCAT('$[', i, '].name')), '');
        SET i = i + 1;

        SET @sql_text = CONCAT('INSERT INTO spokenLanguagesTem VALUES (' , REPLACE(jsonId, '\', ''), ', ', jsonLabel, '); ');
        PREPARE stmt FROM @sql_text;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;

    END WHILE;

    END LOOP ;

    select distinct * from spokenLanguagesTem;
    INSERT INTO spoken_languagesCURSOR

```

```

        SELECT DISTINCT iso_639_1, nameLang
        FROM spokenLanguagesTem;
        drop table if exists spokenLanguagesTem;
        CLOSE myCursor ;

    END$$
DELIMITER ;

call TablaSpokenLanguages();

SELECT * FROM spoken_languagesCURSOR;

CREATE TABLE spoken_languagesCURSOR (
    iso_639_1 varchar(2) PRIMARY KEY,
    name varchar(100)
);

DROP TABLE IF EXISTS spoken_languagesCURSOR;

SELECT COUNT(*) FROM spoken_languagesCURSOR;

```

Este procedimiento ya ha sido explicado antes (Production\_countries), solo que aquí se le ha cambiado las variables.

- **Tratar Genre**



```

DROP PROCEDURE IF EXISTS TablaGenre;
DELIMITER $$
CREATE PROCEDURE TablaGenre()
BEGIN
DECLARE done INT DEFAULT FALSE;
DECLARE nameGenre VARCHAR(100);
-- Declarar el cursor
DECLARE Cursorgenre CURSOR FOR
    SELECT DISTINCT CONVERT(REPLACE(REPLACE(genres, 'Science Fiction', 'Science-Fiction'
    'TV Movie', 'TV-Movie') USING UTF8MB4) FROM movie_dataset;
-- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
-- Abrir el cursor
OPEN Cursorgenre;
drop table if exists temperolgenre;
SET @sql_text = 'CREATE TABLE temperolgenre (name VARCHAR(100));';
PREPARE stmt FROM @sql_text;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;

END IF;
-- Separar los géneros en una tabla temporal
DROP TEMPORARY TABLE IF EXISTS temp_genres;
CREATE TEMPORARY TABLE temp_genres (genre VARCHAR(50));
SET @_genres = nameGenre;
WHILE (LENGTH(@_genres) > 0) DO
    SET @_genre = TRIM(SUBSTRING_INDEX(@_genres, ' ', 1));
    INSERT INTO temp_genres (genre) VALUES (@_genre);
    SET @_genres = SUBSTRING(@_genres, LENGTH(@_genre) + 2);
END WHILE;
-- Insertar los géneros separados en filas individuales
INSERT INTO temperolgenre (name)
SELECT genre FROM temp_genres;
END LOOP CursorDirector_loop;
select distinct * from temperolgenre;
INSERT INTO genre (name)
SELECT DISTINCT name
FROM temperolgenre;
done table if exists temperolgenre;

```

El procedimiento inicia chequeando si la tabla "TablaGenre" ya existe y en caso de ser así, la elimina. Después se delimita el límite entre el código SQL y el procedimiento almacenado. Se sigue con la definición del procedimiento TablaGenre(). Dentro de este, se establece una variable "done" para marcar cuándo el cursor ha alcanzado su final, una variable "nameGenre" para guardar el nombre del género y un cursor "Cursorgenre" para obtener los géneros distintos de la base de datos de películas. El cursor es abierto y una tabla temporal "temperolgenre" es creada para guardar temporalmente los géneros. Luego, se inicia un ciclo repetitivo "CursorDirector\_loop" que recoge cada nombre de género del cursor y lo procesa. Cuando el cursor ha alcanzado su final, se sale del ciclo repetitivo.

- **Tratar Person**



FETCH para recuperar cada fila de la tabla "movie\_dataset" y almacenarla en una variable llamada "jsonData". Después, se inicia otro bucle que procesará todos los arrays dentro de "jsonData".

Dentro de este segundo bucle, la función JSON\_EXTRACT se utiliza para extraer el id, nombre y género de cada objeto JSON en el array. Esta información se asigna a variables locales llamadas "jsonId", "jsonLabel" y "jsongenre". Se crea una tabla temporal llamada "personTem" para almacenar esta información extraída. La información se inserta en la tabla temporal mediante una sentencia preparada y una sentencia EXECUTE. Finalmente, se cierra el cursor y se termina el procedimiento. La información almacenada en la tabla temporal se inserta en la tabla "person" con una sentencia INSERT.

## • Tratar CompanieMovie

```
DROP PROCEDURE IF EXISTS TablaCompaMov;
DELIMITER $$
CREATE PROCEDURE TablaCompaMov ()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE idMovie int;
    DECLARE idProdComp JSON;
    DECLARE idJSON text;
    DECLARE i INT;
    -- Declarar el cursor
    DECLARE myCursor
    CURSOR FOR
        SELECT id, production_companies FROM movie_dataset;
    -- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
    DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET done = TRUE ;
    -- Abrir el cursor
    OPEN myCursor ;
    drop table if exists CompaMovTem;
    SET @sql_text = 'CREATE TABLE CompaMovTem ( id int, idGenre int );';
    PREPARE stmt FROM @sql_text;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
    cursorLoop: LOOP
        FETCH myCursor INTO idMovie, idProdComp;
        -- Controlador para buscar cada uno de los arrays
        SET i = 0;
        -- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
        IF done THEN
            LEAVE cursorLoop ;
        END IF ;
        WHILE (JSON_EXTRACT(idProdComp, CONCAT('$[', i, '].id')) IS NOT NULL) DO
            SET idJSON = JSON_EXTRACT(idProdComp, CONCAT('$[', i, '].id')) ;
            SET i = i + 1;
            SET @sql_text = CONCAT('INSERT INTO CompaMovTem VALUES (', idMovie, ', ',
                REPLACE(idJSON, '\\', '\\\\'), ', ');
            PREPARE stmt FROM @sql_text;
            EXECUTE stmt;
            DEALLOCATE PREPARE stmt;
        END WHILE;
    END LOOP ;
    select distinct * from CompaMovTem;
    INSERT INTO companie_movie
    SELECT DISTINCT id, idGenre
    FROM CompaMovTem;
    DROP TABLE if exists CompaMovTem;
    CLOSE myCursor ;
END$$
DELIMITER ;
```

El texto describe un procedimiento que comienza con la creación de un cursor llamado "myCursor". Este cursor selecciona dos columnas, "id" y "production\_companies", de la tabla "movie\_dataset". Después, se crea una tabla temporal llamada "CompaMovTem" que servirá para almacenar los resultados de la consulta.

El procedimiento luego entra en un bucle llamado "cursorLoop" que se ejecutará mientras haya filas disponibles en el cursor. En cada iteración, se extrae la siguiente fila y se almacena en las variables "idMovie" y "idProdComp". La columna "idProdComp" contiene información en formato JSON, por lo que se utiliza la función "JSON\_EXTRACT" para extraer los valores y almacenarlos en la variable "idJSON".

Después, se construye una consulta SQL dinámica usando la función "CONCAT" para insertar los valores extraídos en la tabla temporal "CompaMovTem". Cuando se han extraído todos los datos de todas las filas, se cierra el cursor y se hace una consulta "SELECT DISTINCT" para obtener valores únicos de la tabla temporal y almacenarlos en la tabla "companie\_movie". Finalmente, se elimina la tabla temporal.

El procedimiento puede ser ejecutado en la consola de MySQL o desde un script, y se asegura de que se elimine antes de ser creado en caso de que ya exista.

- **Tratar CountriesMovies**

```

DROP PROCEDURE IF EXISTS TablaCounMov;
DELIMITER $$
CREATE PROCEDURE TablaCounMov ()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE idMovie int;
    DECLARE idProdCoun text;
    DECLARE idJSON text;
    DECLARE i int;
    -- Declarar el cursor
    DECLARE myCursor
    CURSOR FOR
        SELECT id, production_countries FROM movie_dataset;
    -- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
    DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET done = TRUE ;
    -- Abrir el cursor
    OPEN myCursor ;
    drop table if exists MovCounTemp;
    SET @sql_text = 'CREATE TABLE MovCounTemp ( id int, idGenre varchar(255) );';
    PREPARE stmt FROM @sql_text;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
    cursorLoop: LOOP
        FETCH myCursor INTO idMovie, idProdCoun;
        -- Controlador para buscar cada uno de los arrays
        SET i = 0;

        -- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
    IF done THEN
        LEAVE cursorLoop ;
    END IF ;
    WHILE (JSON_EXTRACT(idProdCoun, CONCAT('$[', i, '].iso_3166_1')) IS NOT NULL) DO
        SET idJSON = JSON_EXTRACT(idProdCoun, CONCAT('$[', i, '].iso_3166_1')) ;
        SET i = i + 1;
        SET @sql_text = CONCAT('INSERT INTO MovCounTemp VALUES (', idMovie, ', ',
            REPLACE(idJSON, '\', ''), '); ');
        PREPARE stmt FROM @sql_text;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
    END WHILE;
    END LOOP ;
    select distinct * from MovCounTemp;
    INSERT INTO countrie_movie
    SELECT DISTINCT idGenre, id
    FROM MovCounTemp;
    DROP TABLE MovCounTemp;
    CLOSE myCursor ;
END$$
DELIMITER ;

```

Se utiliza el mismo procedimiento que en company movie, pero con diferentes variables.

- **Tratar LanguageMovie**

```

DROP PROCEDURE IF EXISTS LanguageMovie;
DELIMITER $$
CREATE PROCEDURE LanguageMovie ()
) BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE idMovie int;
    DECLARE idSpokLang text;
    DECLARE idJSON text;
    DECLARE i INT;
    -- Declarar el cursor
    DECLARE myCursor
    CURSOR FOR
        SELECT id, spoken_languages FROM movie_dataset;
    -- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
    DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET done = TRUE ;
    -- Abrir el cursor
    OPEN myCursor ;
) cursorLoop: LOOP
    FETCH myCursor INTO idMovie, idSpokLang;
    -- Controlador para buscar cada uno de los arrays
    SET i = 0;
    -- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
) IF done THEN
    LEAVE cursorLoop ;
END IF ;
) WHILE(JSON_EXTRACT(idSpokLang, CONCAT('$[', i, '].iso_639_1')) IS NOT NULL) DO

    WHILE(JSON_EXTRACT(idSpokLang, CONCAT('$[', i, '].iso_639_1')) IS NOT NULL) DO
    SET idJSON = JSON_EXTRACT(idSpokLang,  CONCAT('$[', i, '].iso_639_1')) ;
    SET i = i + 1;
    SET @sql_text = CONCAT('INSERT INTO language_movie VALUES (', idMovie, ', ',
        REPLACE(idJSON, '\\', '\\\\'), '); ');
    PREPARE stmt FROM @sql_text;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
    END WHILE;
END LOOP ;
CLOSE myCursor ;
END$$
DELIMITER ;

```

Se utiliza el mismo procedimiento que yase ha visto antes, pero con diferentes variables.

## • Tratar Director

```

-- Tabla Director ---

DROP PROCEDURE IF EXISTS Director;
DELIMITER $$
CREATE PROCEDURE Director()
) BEGIN
    DECLARE done INT DEFAULT FALSE ;
    DECLARE idPersonas INT;
    DECLARE MovId INT;
    DECLARE MovDirector VARCHAR(100);

    -- Declarar el cursor
    DECLARE CursorDirector CURSOR FOR
        SELECT id,director FROM movie_dataset;
    -- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado a su fin)
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    -- Abrir el cursor
    OPEN CursorDirector;
    drop table if exists personTem;

    drop table if exists personTem;
    SET @sql_text = 'CREATE TABLE directorTemp ( idPer int,
    idMov int);';
    PREPARE stmt FROM @sql_text;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;
    CursorMovie_loop: LOOP
        FETCH CursorDirector INTO MovId,MovDirector;
        -- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
    IF done THEN
        LEAVE CursorMovie_loop;
    END IF;
    SELECT idPerson INTO idPersonas FROM person WHERE person.name=MovDirector;
    INSERT INTO directorTemp VALUES (idPersonas,MovId);
    END LOOP;
    CLOSE CursorDirector;
    select distinct * from directorTemp;
    INSERT INTO director
        SELECT DISTINCT idPer,idMov

```

Se utiliza el mismo procedimiento que yase ha visto antes, pero con diferentes variables.

- **Tratar Crew**

```
DROP PROCEDURE IF EXISTS TablaCrew;
DELIMITER $$
CREATE PROCEDURE TablaCrew ()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE idMovie int;
    DECLARE idCrew text;
    DECLARE idJSON text;
    DECLARE jobJSON text;
    DECLARE departmentJSON text;
    DECLARE credit_idJSON text;
    DECLARE i INT;
    -- Declarar el cursor
    DECLARE myCursor
    CURSOR FOR
    SELECT id, CONVERT(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE
    (REPLACE(crew, ' ', '\ '), '\ ', '{'), '{', '}'),
    '\: \', ' ': '), '\ ', '\ ', ' ', ' '), '\: ', ' ': '), '\ ', ' ')
    USING UTF8mb4 ) FROM movie_dataset;
    -- Declarar el handler para NOT FOUND (esto es marcar cuando el cursor ha llegado
    DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET done = TRUE ;
    -- Abrir el cursor
    OPEN myCursor ;
    cursorLoop: LOOP
        FETCH myCursor INTO idMovie, idCrew;
        -- Controlador para buscar cada uno de los arrays
        SET i = 0;
        -- Si alcanzo el final del cursor entonces salir del ciclo repetitivo
        IF done THEN
            LEAVE cursorLoop ;
        END IF ;
        WHILE (JSON_EXTRACT(idCrew, CONCAT('$[', i, '].id')) IS NOT NULL) DO
            SET jobJSON = JSON_EXTRACT(idCrew, CONCAT('$[', i, '].job')) ;
            SET idJSON = JSON_EXTRACT(idCrew, CONCAT('$[', i, '].id')) ;
            SET departmentJSON = JSON_EXTRACT(idCrew, CONCAT('$[', i, '].department')) ;
            SET i = i + 1;
            SET @sql_text = CONCAT('INSERT Ignore INTO Crew VALUES (', idMovie, ', ',
            REPLACE(idJSON, '\ ', ' '), ', ', REPLACE(idJSON, '\ ', ' '), ', ',
            REPLACE(departmentJSON, '\ ', ' '), ', ', REPLACE(credit_idJSON, '\ ', ' '), ');')
            PREPARE stmt FROM @sql_text;
            EXECUTE stmt;
            DEALLOCATE PREPARE stmt;
        END WHILE;
    END LOOP ;
    CLOSE myCursor ;
END$$
DELIMITER ;
call TablaCrew();
```

El objetivo de este procedimiento es insertar datos en una tabla llamada "Crew" a partir de una tabla llamada "movie\_dataset". Antes de crear el procedimiento, se revisa si ya existe un procedimiento con el mismo nombre utilizando "DROP PROCEDURE IF EXISTS TablaCrew". Luego, se cambia el delimitador a "\$\$" para que MySQL pueda procesar correctamente el código dentro de la definición del procedimiento.

El procedimiento comienza declarando variables como "idMovie", "idCrew", "idJSON", "jobJSON", "departmentJSON", y "credit\_idJSON", todas para almacenar valores temporales

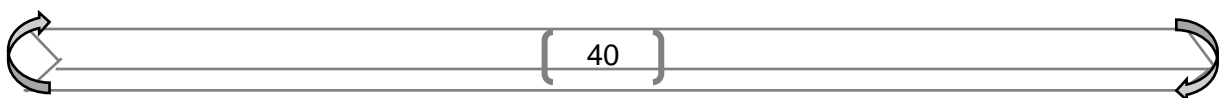
durante la ejecución. Después, se crea un cursor llamado "myCursor" que selecciona todos los registros de "movie\_dataset" y los convierte en un formato compatible con JSON. También se crea un handler llamado "CONTINUE HANDLER" que se ejecutará cuando el cursor alcance el final de los registros.

El cursor se abre y se inicia un loop infinito que se repetirá hasta que se alcance el final del cursor o se salga explícitamente del loop. Dentro del loop, se obtiene un registro a la vez y se guardan los valores en las variables correspondientes. También se usa un segundo loop "WHILE" para procesar los datos en formato JSON en la columna "idCrew". Se extraen los valores de "job", "id", "department", y "credit\_id" con la función "JSON\_EXTRACT" y se guardan en las variables correspondientes.

Finalmente, se construye una sentencia SQL dinámica utilizando la función "CONCAT" y se ejecuta con las funciones "PREPARE", "EXECUTE", y "DEALLOCATE PREPARE". Después de procesar todos los registros, el cursor se cierra y el procedimiento termina. Para ejecutar el procedimiento, se puede utilizar la llamada "call TablaCrew();".

## LIMPIEZA COLUMNA CREW

1. Primero, empezamos con análisis de la columna "crew" mediante la lectura del CSV utilizando IntelliJ y el lenguaje de programación Scala. Pudimos encontrar distintos casos en los que existen comillas simples dentro de los valores de la key "name".





2. Analizamos los distintos patrones presentes en la data y encontramos casos en que existe una sola comilla como en el caso de O'Brian, o dos comillas para encerrar un apodo como 'D.J.' y era necesario realizar los replace necesarios para convertir las comillas en \' para que se tomen en cuenta como parte de la cadena name y no del JSON.
3. Al analizar los casos, nos dimos cuenta de que existen casos como e', t' o d' en los cuales estos patrones coinciden con las llaves que tenemos 'name', 'department', 'credit\_id' o 'id'. Es por esto por lo que no se podía reemplazar solamente los valores e', t' o d' como e\', t\' o d\' debido a que ocasionarían errores con los key.
4. En caso de los géneros 0, 1 y 2, tuvimos que transformarlos según su posición y los caracteres que se encontraban a su lado, por ejemplo ': 0,', ': \'0\',' ya que estaba después de: y antes de, y así lograr colocar comillas simples a su alrededor.
5. En el caso de los valores del key "id", fue necesario encerrarlos con comillas simples también, es por eso por lo que las colocamos según los valores alrededor \'id\': ' se convirtió en \'id\': \' para colocar la primera comilla sencilla antes del valor y },' fue reemplazado con '\'),' para colocar la comilla simple al final del valor.
6. En el caso del último valor de "id" reemplazamos '}]' al colocar '\"]]' en su lugar. A diferencia del último reemplazo en el punto 5, el } no está seguido por un , debido a que es el último valor del arreglo. Termina con una llave] así que por eso tomamos este caso en consideración para que todos los últimos valores de key tengan comillas cerradas.

7. Tras realizar todos los REPLACE, realizamos un JSON\_EXTRACT JSON\_EXTRACT (CONVERT(\*SENTENCIA\_DE\_TODOS\_LOS REPLACE\* USING UTF8MB4), '\$[\*]') para extraer todos los valores JSON.

Hemos realizado exitosamente la limpieza de la columna “crew”. El siguiente paso es realizar la carga de estos datos a su respectiva tabla/s.

## Conclusiones

Este proyecto fue llevado a cabo siguiendo las normas de modelado de bases de datos, incluyendo la normalización, el diseño conceptual y lógico, entre otros. Y aunque el trabajo era individual, gracias al grupo de la materia de prácticum y algunas colaboraciones, fuimos capaces de cumplir con todos los objetivos planteados al comienzo del desarrollo. Durante el proceso, tuvimos que enfrentar varios desafíos, pero gracias a las pautas de nuestros tutores y a nuestra habilidad para resolver problemas, logramos satisfacer todos los requerimientos.

Hemos demostrado un gran avance en nuestras habilidades de modelado y población de una base de datos, así como en el aprendizaje de nuevas técnicas de programación aplicables a situaciones reales. Este proyecto nos acerca un paso más a ser profesionales confiables en proyectos futuros.

Aprendimos la importancia de tratar los datos con cuidado antes de cualquier procesamiento para evitar problemas de consistencia y errores. También entendemos la importancia de las herramientas para interactuar con la base de datos. En general, podemos decir que todo lo aprendido durante este ciclo ha sido crucial para el éxito de este proyecto integrador, que representó un desafío para nosotros como estudiantes ya que fue nuestra primera experiencia trabajando con una cantidad significativa de datos.