UNIVERSIDAD DE LA REPÚBLICA

FACULTAD DE INGENIERÍA

BASES DE DATOS NO RELACIONALES

Migración automática de bases de datos MySQL a MongoDB

Alumnos:

Mateo Nogueira - Gastón Morales

ÍNDICE

I. Introducción

Las bases de datos relacionales se han convertido en la primera opción para el almacenamiento de información en bases de datos utilizadas principalmente para registros financieros, datos de personal y salarios desde su concepción aproximadamente en 1980. Estas se basan en el modelo relacional definido por un esquema, en donde una tabla representa la relación entre tuplas (también conocidas como filas).

En la práctica, hay situaciones en las que almacenar datos en forma de tabla es inconveniente, dado que existen otro tipo de relaciones entre registros o existe la necesidad de acceder rápidamente a los datos. Para solucionar estos problemas surge un nuevo tipo de bases de datos, conocidas como bases de datos no relacionales (abreviadas como NoSQL o Not only SQL).

Algunas ventajas que ofrecen las bases de datos no relacionales frente a las relacionales es que tienen buen rendimiento frente a problemas de big data (cuando los datos crecen exponencialmente), tienen mejor rendimiento a la hora de almacenar o recuperar un objeto junto con todos los datos relevantes y no padecen el desajuste entre las bases de datos relacionales y la programación orientada a objetos. En consecuencia, el poder migrar de manera organizada y automatizada una base de datos relacional a una no relacional se ha vuelto un tema de interés para los que deseen experimentar con este tipo de bases de datos.

El artículo [9] presenta un algoritmo totalmente automatizado para realizar una migración desde MySQL a MongoDB. Asimismo, los autores utilizan como base de datos para probar su algoritmo una base con aproximadamente 15 tablas con 100 registros y relaciones entre ellas mostrando buenos resultados

II. Objetivos

El objetivo de este trabajo es realizar una implementación del algoritmo propuesto en [9] para migrar bases de datos MySQL a MongoDB. Utilizando los conjuntos de datos de prueba de MySQL sakila [8] y employee data [7], se presentan pruebas de desempeño y se muestra que la herramienta es capaz de migrar correctamente los dos conjuntos de datos.

Otro objetivo que se tiene es investigar si el algoritmo tiene limitaciones, presentando un estudio del mismo, mostrando sus ventajas y desventajas. También se hace una comparación con otras herramientas de migración existentes, mostrando sus principales diferencias.

Se plantea también analizar si la herramienta presenta alguna limitación al utilizarse con bases de datos más grandes. Para esto, se muestra un estudio con la base de datos de prueba de MySQL airportdb [6]. La misma está compuesta por 14 tablas, con 55 millones de registros y un tamaño de 2GB.

Por otro lado, se tiene como objetivo estudiar los patrones de diseño de MongoDB [2]. Analizando la viabilidad de que algun patrón pueda implementarse al momento de realizar la migración.

III. Conjuntos de datos a utilizar

En esta sección se van a describir brevemente los conjuntos de datos que van a utilizarse para probar la herramienta.

III-A. Employees

Uno de los conjuntos con los que se va a trabajar es la base de datos Employees, perteneciente al conjunto de bases de datos de ejemplos de MySQL [7]. Esta base contiene 6 tablas distribuidas en aproximadamente 4 millones de registros, ocupa aproximadamente 167MB. Contiene información sobre aproximadamente 300.000 empleados de una empresa junto con información relacionada a sus salarios y el departamento en el que trabajan. En la figura 1 se puede ver una imagen del esquema de la base datos y en la tabla I la cantidad de registros por tabla.

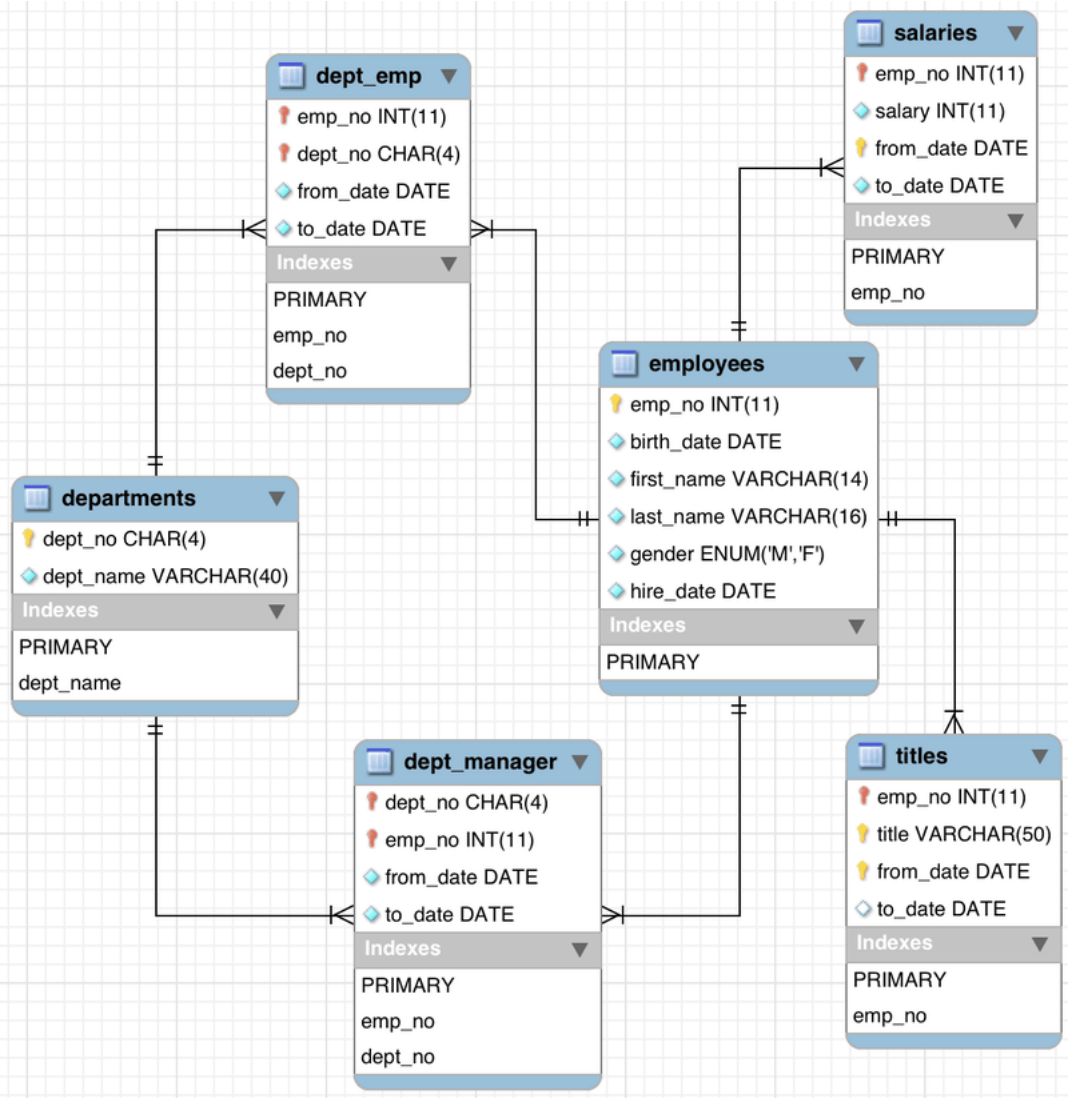

Figura 1. Esquema base de datos Employees

Tabla	# Filas
employees	300024
departments	9
dept_manager	24
dept_emp	331603
titles	443308
salaries	2844047

Tabla I
CANTIDAD DE FILAS ESQUEMA EMPLOYEES

III-B. Sakila

El segundo conjunto de datos a utilizar es también una base de datos de prueba de MySQL llamada sakila [8]. Esta base de datos contiene información sobre alquileres de películas. Contiene tiendas, clientes, películas, actores, inventario de películas para cada tienda, información sobre alquileres y pagos. Está conformada por 16 tablas. En la figura 2 se puede encontrar el esquema de esta base de datos y en la tabla II se puede ver la cantidad de filas en cada tabla.

En total, hay aproximadamente 47.275 filas, muchas menos que los millones de registros que contiene Employees. Sin

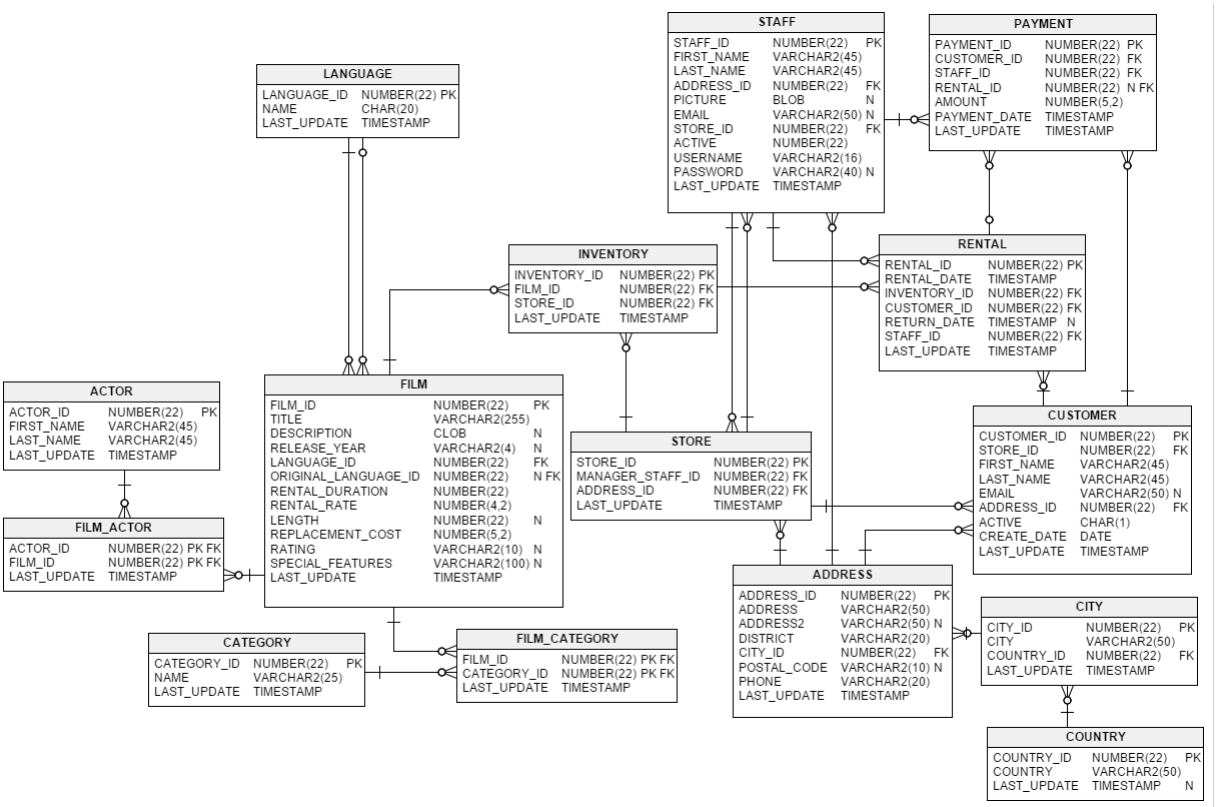

Figura 2. Esquema base de datos Sakila

Tabla	# Filas
actor	200
address	603
category	16
city	600
country	109
customer	599
film	1000
film_actor	5462
film_category	1000
film_text	1000
inventory	4581
language	6
payment	16049
rental	16044
staff	2
store	2

Tabla II
CANTIDAD DE FILAS ESQUEMA SAKILA

embargo, las tablas tienen muchas más relaciones por lo que resulta interesante para probar el algoritmo.

III-C. Airportdb

El último conjunto de datos que desea utilizar para probar es Airportdb [6]. Esta base de datos contiene información sobre aeropuertos, pasajeros, vuelos, aerolíneas y empleados. Es la más grande de las tres dado que son 2 GB de datos y contiene casi 56 millones de registros.

En la tabla III se puede ver la cantidad de filas para cada tabla. Contiene 14 tablas (dos menos que sakila) y la mayoría de los registros pertenecen a la tabla *booking* o *weatherdata*. Además, hay dos tablas que se encuentran vacías. En la figura 3 se encuentra una imagen del esquema.

Tabla	# Filas
booking	50831531
flight	416429
flight_log	0
airport	9939
airport_reachable	0
airport_geo	9854
airline	113
flightschedule	9851
airplane	5583
airplane_type	342
employee	1000
passenger	36346
passengerdetails	37785
weatherdata	4626432

Tabla III
CANTIDAD DE FILAS ESQUEMA AIRPORTDB

En la tabla IV se puede encontrar un resumen de los tres conjuntos de datos a utilizar, que incluye la cantidad de filas y tablas por base de datos.

Base de datos	# Filas	# Tablas
employees	4 millones	6
sakila	16	47.275
airportdb	14	56 millones

Tabla IV
RESUMEN CONJUNTOS DE DATOS A UTILIZAR.

IV. ALGORITMO DE MIGRACIÓN

IV-A. MongoDB

Uno de los motores de bases de datos NoSQL más populares es MongoDB. El mismo es un motor de bases de datos orientado a documentos. Los conceptos más importantes son el de documento y colección. Un documento es un conjunto de parejas clave valor. Los documentos se guardan en una colección. Es decir, el documento es el equivalente a una fila y la colección a una tabla. Sin embargo, los documentos no tienen un esquema fijo y no es necesario que los documentos de una misma colección estén relacionados o compartan esquema. Además, dentro de un documento se pueden embeber otros documentos o guardar un arreglo de documentos, lo que permite expresar relaciones de una manera diferente a SQL. Esto hace que muchos joins en SQL no sean necesarios en Mongo y que un objeto JSON o XML pueda ser guardado en Mongo sin necesidad de utilizar un ORM (Object relational mapper).

No obstante, los tipos de datos que maneja Mongo no son los mismos que en JSON. Mongo utiliza BSON (Binary JSON), un formato similar a JSON que lo extiende para soportar más tipos de datos. Por ejemplo, se agregan los tipos de datos decimal, binario, fecha, timestamp y GeoJSON (permite guardar ubicaciones geográficas). Asimismo, los tipos de datos de Mongo son diferentes a los de MySQL y no siempre vamos a tener un tipo de datos totalmente equivalente en el otro motor.

IV-B. 'Automatic mapping of MySQL databases to NoSQL MongoDB'

Existen varias formas de realizar una migración desde una base de datos relacional a MongoDB. La más trivial consiste en crear una colección por tabla y copiar cada fila de una tabla a su correspondiente colección creando un nuevo documento.

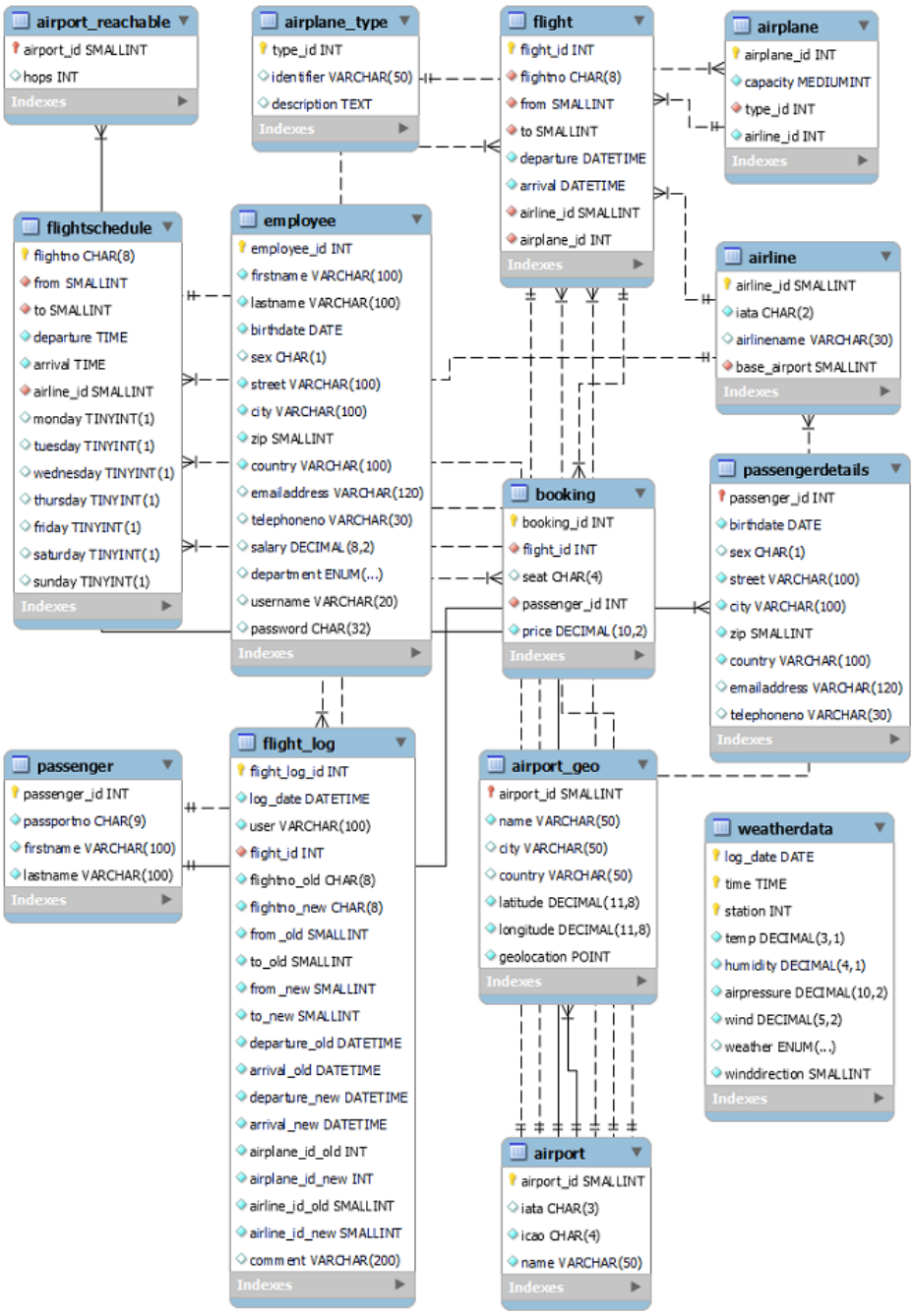

Figura 3. Esquema base de datos Airportdb

Sin embargo, esta forma no aprovecha todas las funcionalidades brindadas por MongoDB. Por ejemplo, los ya mencionados documentos embebidos.

Los autores del artículo [9], que se busca implementar, tuvieron esto en cuenta al desarrollar su algoritmo. En particular, para cada tipo de relación posible en SQL (1:1, 1:M, M:N) proponen una manera para representarla en MongoDB.

Empezando por las relaciones 1:1, estas pueden modelarse de dos maneras. Utilizando dos colecciones diferentes y almacenando el identificador del documento asociado en alguno de los dos lados de la relación o usando un documento embebido (conocido como el patrón embedding [5]). Para este caso, los autores proponen usar un documento embebido, dado que al recuperar el documento principal también se consigue el documento embebido sin realizar una búsqueda adicional. Por ejemplo,

un estudiante tiene una sola dirección y una dirección pertenece a un solo estudiante. Entonces, la dirección se embebe en el documento del estudiante. De esta manera, al obtener el estudiante no es necesario buscar su dirección en otra colección.

En el caso de las relaciones 1:M, se pueden modelar de tres maneras diferentes. Se pueden embeber los documentos utilizando un arreglo, utilizar el método linking (que consiste en agregar una referencia en cualquiera de los dos lados de la relación, similar a una clave foránea) embebiendo una lista de identificadores o utilizar el patrón de MongoDB bucket [3] que es útil para algunos tipos de datos como series temporales. Los autores sugieren utilizar nuevamente el patrón embedding.

Finalmente, las relaciones N:M, que en SQL se representan mediante una llamada tabla de join donde cada fila contiene las claves primarias de los elementos involucrados en la relación, pueden ser modeladas nuevamente de varias maneras. Los autores del artículo mencionan dos. La primera, conocida como Two Way Embedding, consiste en crear en cada lado de la relación un arreglo con las claves foráneas de los elementos asociados. La otra manera, llamada One Way Embedding consiste en solamente agregar las referencias en un lado de la relación.

Para obtener los datos necesarios para comprender las relaciones de manera automatizada, se utiliza el conjunto de tablas INFORMATION_SCHEMA, que contiene información relacionada a los tipos de datos, nombres de columnas, claves primarias y foráneas entre otros datos.

Una vez se tiene esta información, en base a la cantidad de claves foráneas o referencias que tiene una tabla, se propone como migrar la tabla.

1. Si la tabla no es referida por otra tabla, se crea una colección nueva.

2. Si la tabla no tiene claves foráneas, pero es referida por otra tabla, se crea una nueva colección.

3. Si la tabla contiene una clave foránea y es referida por otra tabla, va a crearse una nueva colección utilizando el método link similar a una clave foránea.

4. Si la tabla tiene una clave foránea pero no es referenciada por otra tabla, estamos ante una relación 1:M o 1:1. Se utiliza one way embedding dentro de la otra tabla de la relación.

5. Si la tabla tiene dos claves foráneas y no es referenciada por otra tabla, se utiliza two way embedding.

6. Si la tabla tiene tres o más claves foráneas, se está frente a una relación N:M ternaria o cuaternaria. Se utiliza la técnica linking en este caso.

En la siguiente sección se explicará la implementación del algoritmo presentado.

V. Implementación

El algoritmo fue implementado en Python. El código puede encontrarse en el Gitlab de la facultad en el siguiente enlace: https://gitlab.fing.edu.uy/mateo.nogueira/sql-mapping-nosql. Al realizar la ejecución, primero se genera un listado de claves foráneas y relaciones para cada tabla. Esto es utilizado para poder clasificar cada tabla dentro de las categorías del algoritmo. Luego, según la clasificación se procede a copiar los datos creando nuevas colecciones o embebiendo documentos.

Para conseguir el listado de tablas se utiliza la siguiente consulta:

```
SELECT
    table_name
FROM
    information_schema.tables
WHERE
    table_schema = "{schema_name}" AND TABLE_TYPE="BASE TABLE";
```

Se busca que TABLE_TYPE sea BASE TABLE para excluir vistas.

Luego, para cada tabla se pueden obtener sus claves foráneas y relaciones utilizando la siguiente consulta:

```
SELECT
    CONSTRAINT_NAME,
    REFERENCED_TABLE_NAME
FROM
    information_schema.REFERENTIAL_CONSTRAINTS
WHERE
    CONSTRAINT_SCHEMA ="{schema_name}" AND TABLE_NAME = "{table}";
```

Finalmente, antes de obtener los datos utilizando un ***SELECT * FROM ...*** se necesita obtener las columnas de las claves foráneas, ya que el nombre que la consulta anterior nos informa es el de la restricción que suele ser diferente al de la columna.

```
SELECT
    COLUMN_NAME,
    REFERENCED_COLUMN_NAME,
    REFERENCED_TABLE_NAME
FROM
    INFORMATION_SCHEMA.KEY_COLUMN_USAGE
WHERE
    TABLE_SCHEMA="{schema_name}" AND
    TABLE_NAME="{table_name}" AND
    CONSTRAINT_NAME <> "PRIMARY" AND
    REFERENCED_COLUMN_NAME IS NOT NULL AND
    REFERENCED_TABLE_NAME IS NOT NULL;
```

En el resto de la sección, se exponen los resultados obtenidos al migrar cada uno de los conjuntos de datos utilizados.

V-A. Conjunto de datos Employees

La primera base de datos que se migró utilizando la implementación fue Employees. Antes de mostrar los resultados consideramos pertinente explicar dentro de que punto de la clasificación del algoritmo se encuentra cada tabla.

La tabla *departments* no tiene claves foráneas, pero es referenciada por *dept_emp* y por *dept_manager* por lo tanto, se encuentra dentro del punto 2 del algoritmo y se representa por una colección nueva. La tabla *employees* se encuentra en una situación similar, ya que es referenciada por *salaries*, *dept_emp*, *dept_manager* y *titles*.

Las tablas *salaries* y *titles* se encuentran bajo el punto 4 del algoritmo. Ambas tienen una única clave (número de empleado) foránea y no son relacionadas por otra tabla. Se utiliza entonces, one way embedding a la colección de empleados.

Finalmente, las dos tablas restantes *dept_emp* y *dept_manager* se clasifican dentro del punto 5 del algoritmo. Se está ante une relación N:M y se utiliza en este caso, two way embedding. Cabe destacar que el algoritmo no explicita dónde guardar información asociada a la relación (es decir, datos que pertenezcan a la tabla de JOIN pero que no sean claves foráneas). En ambas relaciones se tienen los atributos *from_date* y *to_date* que van a ser guardados en dos lugares, resultando en datos duplicados.

Se observa que no hay tablas que clasifiquen dentro de los puntos 1, 3 y 6 del algoritmo en este caso.

V-A1. Resultados: El primer problema observado en el algoritmo es el alto tiempo de ejecución que tiene. En total, para migrar la base de datos employees el algoritmo demoró 18 horas. Sin embargo, no es ninguna sorpresa ya que es necesario leer toda la base de datos para poder escribirla en Mongo. Además, el uso de two way embeddings añade tiempo extra, ya que cada registro de la tabla JOIN leído debe ser guardado en dos colecciones de Mongo.

En particular, nuestra implementación maneja estas relaciones de la siguiente manera. Por ejemplo, para la relación entre employees y departments se seleccionan todos los *emp_no* diferentes de la tabla dept_emp. Luego, para cada empleado se consultan los registros de dept_emp y se guardan en un arreglo en el documento correspondiente a ese employee en Mongo.

Luego, se hace lo mismo, pero con la tabla departments. En total, esto significa 300024 + 9 consultas SQL (cantidad de registros en employees sumado a la cantidad de registros en departments). La alternativa es recorrer cada registro de dept_emp guardado cada registro en su correspondiente documento en employees y en departments al mismo tiempo. Esto hubiera requerido 331603 consultas SQL.

Si bien en este ejemplo la primera opción tiene menos consultas, en el caso general va a depender de la cantidad de elementos relacionados. Si en la tabla de join hay más registros que en las dos tablas de la relación sumadas, entonces conviene elegir la primera opción. Una posible mejora para el algoritmo es tener esto en cuenta, alcanza con utilizar consultas para contar la cantidad de registros.

En la figura 4 se pueden ver las colecciones resultantes de realizar la migración. Se paso de tener 6 tablas a tener solamente dos colecciones. Las tablas titles y salaries se encuentran exclusivamente embebidas en la colección employees y las tablas dept_emp y dept_manager se encuentran embebidas en ambas colecciones.

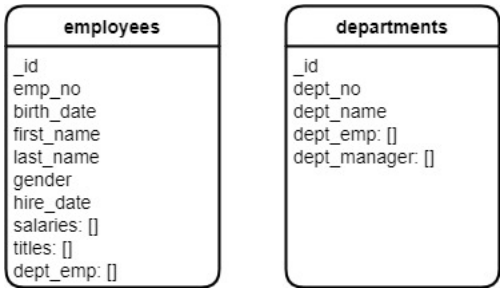

Figura 4. Resultado migración employees

Se tuvo un problema con el tipo de datos DATE de MySQL. Este tipo de datos representa una fecha en formato año/mes/día. En Mongo, el tipo de datos de fecha solo representa fechas en formato UTC por lo que debe tener también hora y minutos. La solución utilizada fue usar el formato fecha de Mongo agregando la hora 0 con 0 minutos y 0 segundos.

V-B. Conjunto de datos Sakila

Una vez que el algoritmo funcionaba con la base de datos employees, se procede a migrar Sakila. Analizando manualmente como va a migrarse cada tabla se encuentra que hay algunas inconsistencias y casos no manejados por los autores del algoritmo.

Dentro de la primera categoría, se encuentra solamente la tabla *film_text*. Esta tabla es una copia exacta de la tabla *film* pero que solamente contiene **film_id, title** y **description**. Se mantiene actualizada mediante triggers.

En la segunda categoría se encuentran las tablas *actor, category, country* y *language*. *actor* es referida por la tabla *film_actor*, *category* es referida por *film_category*, *country* es referida por *city* y, finalmente, *language* es referida por *film*.

La tercera categoría contiene *address* y *city*. La primera contiene una clave foránea a *city* y es referida por *staff* mientras que la segunda contiene una clave foránea a *country* y es referida por *address*.

En este caso, no tenemos tablas dentro de la cuarta categoría. Esto nos indica que no va a utilizarse one way embedding en esta migración.

Dentro de la quinta categoría que representa relaciones N:M se encuentran las tablas *film_actor* (representando que una película puede tener varios actores y que un actor actúa en varias películas) y *film_category* (representando que una película puede tener varias categorías). En este caso, se va a utilizar two way embedding.

Finalmente, en la sexta categoría, que representa asociaciones entre 3 o más entidades se encuentran las tablas *payment* y *rental*. La primera contiene claves foráneas hacia *staff*, *rental* y *customer* mientras que la segunda contiene claves foráneas hacia *inventory*, *customer* y *staff*.

Se observa que entre los dos conjuntos de datos se tienen tablas dentro de las 6 categorías del algoritmo.

No obstante, hay tablas en Sakila que quedan por afuera de las seis categorías. Estas son *customer*, *store*, *inventory*, *film* y *staff*.

La tabla *customer* tiene dos referencias y dos claves foráneas. Las referencias son de las tablas *payment* y *rental*. Las claves foráneas son hacia las tablas *address* y *store*. Si se observan las categorías de la clasificación de tablas, no hay ninguna a la que esta tabla pertenezca. Dado que conocemos el dominio del problema, sabemos que no tiene sentido embeber la tabla customers dentro de rentals, ya que esta representa los alquileres de películas y por lo tanto requiere que el customer exista antes de crear un rental. Algo similar sucede con la tabla *payment* ya que para crear un pago primero tiene que existir su correspondiente rental y por lo tanto su correspondiente customer. De la misma manera, no tiene sentido embeber la tabla *customer* dentro de *address*. Si tiene sentido embeber esta tabla en *store* ya que cada cliente va a alquilar de una sola tienda.

La tabla *store* tiene 3 referencias y 2 claves foráneas. Las referencias son de las tablas *customer*, *inventory* y *staff*. Las claves foráneas son hacia las tablas *address* y *staff*. Nuevamente es necesario utilizar conocimiento sobre el dominio del problema para evaluar la mejor manera de migrar esta tabla. Debido a que la tienda es una entidad que no depende de ninguna de sus asociaciones para existir (un customer debe pertenecer a una tienda al ser creado, el inventory está asociado a una tienda, el staff pertenece a una tienda y la dirección está asociada a la tienda), esta tabla debería ser migrada como una colección nueva.

En cuanto a la tabla *inventory*, la misma tiene una referencia de la tabla *rental* y dos claves foráneas: una hacia *film* y otra hacia *store*. Dado que la tabla *inventory* representa los discos físicos que una tienda posee, una forma de migrar esta tabla es embebiéndola dentro de la colección *store*.

Con la tabla *film* ocurre algo similar a la tabla *store*, también tiene 3 referencias y 2 claves foráneas. Las referencias son de las tablas de join *film_category* y *film_actor* y de la tabla *inventory*. Las foreign keys son las dos hacia la misma tabla: *language*. No tiene sentido embeber *film* dentro de alguna de las tablas de join ya que representan una relación. Si es posible embeber *film* dentro de *language*, pero no es la mejor idea en esta realidad. Lo mejor es crear una nueva colección.

Finalmente, la tabla *staff* tiene también 3 referencias y 2 claves foráneas. Lo mejor para esta situación es embeber al staff de una tienda dentro del documento que representa esa tienda.

Para todos estos casos se decide optar por crear una nueva colección para cada una de estas tablas ya que implementar lógica específica para esas tablas haría que la herramienta deje de ser general y también podría traer problemas al migrar otras tablas.

En base a lo visto anteriormente, se puede ver que el caso donde el algoritmo falla es cuando se tiene 1 o 2 claves foráneas y más de 2 referencias. Además, se evidencia otra desventaja del algoritmo, no utiliza información específica del dominio del problema. Si se utilizara esta información se podría utilizar mejor la técnica de embedding o two way embedding. Por ejemplo, para la tabla *address* se va a crear una nueva colección. Sin embargo, esta tabla podría embeberse dentro los documentos que tengan una dirección asociada ya que cada dirección debería pertenecer a una sola entidad.

V-B1. Resultados: Fue necesario completar la implementación para poder migrar las categorías 1, 4 y 6 ya que en el conjunto de datos anterior no había tablas que pertenecieran a esas categorías.

En esta ocasión, la ejecución fue bastante más rápida ya que la base de datos no contiene tantos registros. Además, las tablas que se migraron con two way embedding tenían como máximo 1000 registros. En total, el algoritmo demoró solamente 9 segundos en ejecutarse.

El resultado cuenta con 14 colecciones y la base de datos tenía originalmente 16 tablas, la diferencia se deba a que a las tablas *film_category* y *film_actor* se les aplicó two way embedding.

Además, al migrar este conjunto de datos se presentó otro problema que sufre este y todos los algoritmos de migración. No siempre un tipo de datos de MySQL tiene su equivalente en Mongo o se puede guardar directamente sin convertir. En particular, este conjunto tuvo problemas con el tipo de datos geometry que es usando en la tabla *address* para representar la localización, el tipo de datos decimal que se utiliza en la tabla *payment* para el monto de los pagos y el tipo de datos set que se utiliza para las características especiales de las películas.

En MySQL, el tipo de datos geometry permite guardar figuras geométricas. Por ejemplo, un punto o un polígono. En este caso, se utiliza para guardar las coordenadas de una dirección. Mongo no tiene un equivalente para figuras geométricas disponible. No obstante, soporta coordenadas utilizando el formato abierto GeoJSON. Lamentablemente, no es posible conocer si un tipo geometry de MySQL representa una coordenada o un punto cualquiera en el espacio sin mirar el dominio del problema por lo que se decidió migrar este campo como un arreglo con dos entradas.

El tipo de datos decimal de MySQL permite almacenar números exactos definiendo cuantos valores luego de la coma se desea tener. Es por esto por lo que se suele utilizar para valores monetarios, como en el caso de la tabla *payment*. Para este tipo de datos, Mongo recomienda multiplicar las cantidades por un múltiplo de 10 para sacar la parte decimal (por ejemplo, en lugar de guardar 10,50 guardar 1050) o utilizar el tipo decimal de BSON [4]. Se decidió utilizar la segunda opción.

Finalmente, para el tipo de datos **set** (conjunto) de MySQL no existe un equivalente en Mongo por lo que se optó por utilizar una lista. En el caso de agregar datos nuevos a la base va a ser necesario controlar manualmente que no se estén agregando elementos duplicados a la lista.

V-C. Conjunto de datos airportdb

Finalmente, se ejecuta el algoritmo para el último conjunto de datos. Debido a la gran cantidad de filas, se decidió limitar a un máximo de 1000 filas por sentencia de SELECT al realizar consultas relacionadas a los datos a migrar. Si el algoritmo funciona para esta versión reducida de la base de datos, la única diferencia con utilizar la base entera es el tiempo de ejecución.

En esta ocasión, hay tablas en todas las categorías de la clasificación del algoritmo y no queda ninguna por afuera.

Las tablas *employee* y *weatherdata* no tienen referencias ni claves foráneas por lo que pertenecen a la primera categoría y se migran a una nueva colección.

En la segunda categoría, y, por lo tanto, también van a migrarse a una nueva colección, se encuentran las tablas *airplane_type*, *airport* y *passenger*. La primera es refereida por *airplane*. *airport* es referida por *flight, airport_geo, airline, airport_reachable* y *flightschedule*. Finalmente, la tabla *passenger* es referida por *passengerdetails* y *booking*.

En la tercera categoría se encuentran dos tablas, **airplane** y **airline**. La primera es referida por *flight* y tiene una clave foránea hacia *airplane_type*. En el caso de *airline*, es referida por *flight* y por *flightschedule* y tiene una clave foránea hacia *airport*.

Las tablas *airport_geo*, *airport_reachable*, *passengerdetails* y *flight_log* tienen una clave foránea y no tienen referencias por lo que va a utilizarse one way embedding en las colecciones *airport*, *airport*, *passenger* y *flight* respectivamente.

Para la tabla *booking* se va a utilizar two way embedding con las colecciones *passenger* y *flight*.

Finalmente, en la última categoría se tienen las tablas *flight* y *flightschedule* que van a ser migradas a su propia colección.

En esta ocasión, no hay tablas por afuera de la clasificación.

V-C1. Resultados: En esta ocasión, se tuvo un problema con el tipo de datos de MySQL TIME. Este tipo representa la hora en formato hora:minutos:segundos. Al no haber un equivalente en Mongo se tomó la decisión de convertirlo a string.

Al finalizar la migración se obtienen 9 colecciones. Dentro de las que no se migraron como una nueva colección, *flight_log* y *airport_reachable* no se migraron dado que estaban vacías. Las tablas *airport_geo* y *passengerdetails* fueron migradas utilizando one way embedding y la tabla *booking* se encuentra embebida en *passenger* y en *flight*.

VI. Desventajas

En esta sección se planea desarrollar en las desventajas o inconvenientes detectados en el algoritmo.

El primer problema que detectamos es el de los tipos de datos. No todos los tipos de datos de MySQL tienen un equivalente directo en Mongo, esto implica que la realidad debe modelarse de manera diferente en ambos tipos de bases de datos.

Al utilizar two way embedding, hay información que va a estar duplicada. No solo al eliminar una relación (o agregar) es necesario hacerlo en ambas colecciones, sino que además la información en la tabla de join va a estar duplicada. Por ejemplo, en la base employees, la tabla *dept_emp* contiene información asociada a la fecha de inicio y fin del empleado en ese departamento y esa información va a estar duplicada. En caso de cambiar, también es necesario hacerlo en los dos lugares.

Además, tal vez no sea necesario utilizar two way embedding para alguno de esos casos. Esto va a depender de las consultas que se quieran realizar a la base de datos y solamente una persona es capaz de tener esto en cuenta a la hora de migrar. Por ejemplo, en sakila, la tabla *film_actor* representa que un actor trabajó en una película y se migra utilizando two way embedding en *actor* y en *film*. Si dada una película nunca va a consultarse que actores participaron en ella no es necesario embeber la tabla dentro de *films*.

Otra restricción que no se tiene en cuenta es que los documentos tienen un límite de 16MB de tamaño, por lo que si se tiene que embeber una tabla muy grande tampoco va a ser posible.

En caso de tener relaciones 1:1, el algoritmo tampoco es capaz de detectarlo sin ayuda de una persona que conozca la realidad. Estos casos son detectados por el algoritmo como 1:M y se migra utilizando one way embedding. Se podría intentar detectar relaciones 1:1 contando la cantidad de entidades relacionadas, pero incluso que haya una sola asociación no implica que no pueda haber dos o más en el futuro. Lo mejor para este tipo de relaciones sería utilizar embedding sin necesidad de utilizar un arreglo.

Además, dependiendo de donde se encuentre la clave foránea es como va a realizarse el embedding. Por ejemplo, si se supone una realidad con 2 tablas, una de estudiantes con un identificador y otra de direcciones, también con su propio identificador, que contenga la dirección de cada estudiante, dependiendo de si estudiantes tiene el identificador de la tabla de direcciones como clave foránea es que estudiantes va a embeberse en la tabla direcciones o viceversa. Nuevamente, solo el conocimiento de la realidad va a poder discernir cual es la mejor opción.

VII. Comparación con otros enfoques

La interfaz gráfica de Mongo, Studio 3T, contiene su propia herramienta para realizar migraciones [10]. Esta herramienta permite conectarse a una base de datos SQL y una Mongo y migrar tanto de SQL a Mongo como viceversa. Permite controlar que atributo va a seleccionarse como identificador.

En el caso de SQL a Mongo, permite seleccionar los datos a migrar de varias maneras. La más sencilla es eligiendo tablas y migrándolas como una nueva colección. Además, permite cambiar el nombre de las columnas o los tipos de datos y también borrar columnas. Además, es posible utilizar one way embedding. Studio 3T permite elegir otra tabla de la base de datos y hacer un JOIN por cualquier atributo (no necesariamente debe ser una clave foránea), agregando esa nueva tabla como un arreglo al documento (también permite cambiar nombres a las columnas, eliminar y cambiar los tipos de datos). Es posible volver a realizar esto nuevamente para el recién agregado arreglo, lo que permite alcanzar varios niveles de embedding.

Otra forma de seleccionar los datos es utilizando una consulta SQL y generando una nueva colección como resultado. Esto brinda una gran flexibilidad ya que permitiría, por ejemplo, implementar patrones de diseño como tree fácilmente.

La gran desventaja de esta herramienta es que es necesario armar manualmente la estructura y mapeo de las tablas para realizar la migración. Sin embargo, presenta una alternativa muy potente ya que da la libertad de utilizar consultas SQL para seleccionar datos.

Otra forma de migrar bases de datos relacionales a no relacionales es la presentada en el paper [1]. Los autores implementan una herramienta de migración automática, pero en la cual el usuario tiene que tomar decisiones de diseño para que se pueda completar la migración.

La herramienta tiene como entrada una representación del modelo NoSQL al cuál se quiere migrar. Dicha representación son un conjunto de grafos dirigidos acíclicos (DAG, por sus siglas en inglés). Cada DAG se define como $G = (V, E)$, donde el conjunto de vértices V está relacionado con las entidades de la base de datos relacional y el conjunto de aristas E con las relaciones entre las entidades. Cada DAG se puede visualizar como un árbol, donde el vértice raíz es la entidad NoSQL a la que se quiere llegar.

A partir del conjunto de DAG se generan los comandos y las funciones definidas por el usuario (UDF, de sus siglas en inglés), con la capacidad de leer los datos de la base de datos relacional, transformarlos y conservarlos en formato JSON.

Esta herramienta permite producir diferentes bases de datos no relacionales para una misma base de datos relacional al ser decisión del usuario el cómo definir los distintos DAG. Esto da más flexibilidad para poder adecuar la migración al uso específico que se le dará a la base de datos no relacional.

Por otra parte, la tarea de definir el esquema de la base de datos no relacional no es una tarea sencilla a la hora de migrar una base de datos. Esto hace que el algoritmo propuesto tenga cierta complejidad, dado que es lo que se necesita como entrada del mismo.

VIII. Patrones de diseño de MongoDB

Dependiendo el uso que se le vaya a dar a una base de datos no relacional puede requerir un diseño u otro. MongoDB presenta patrones de diseño genéricos con ejemplos de casos de uso que ayudan al usuario a mejorar la arquitectura de la base de datos no relacional y ajustarla lo mejor posible a su realidad. [2].

En esta sección presentamos sugerencias para implementar algunos de estos patrones de diseño, que pueden informarse al usuario por si desea aplicarlos a su problema. Estos patrones son algunos de los que pueden ser detectados por la estructura de la base.

VIII-A. The Bucket Pattern

Este patrón se puede utilizar cuando se tienen datos que se pueden agrupar en ventanas de tiempo. Por ejemplo, si se tiene un sensor que genera medidas cada un minuto, estas medidas se pueden agrupar en ventanas de una hora. En un enfoque relacional, se tiene una tabla con todas las medidas. Esto podría no ser lo mejor en Mongo, ya que podría resultar en un índice muy grande además de que al tener las medidas agrupadas cada cierto tiempo facilita el cálculo de medidas agregadas, que incluso se pueden guardar ya calculadas.

Por ejemplo, considerando nuevamente datos de un sensor, la tabla relacional contendría un identificador del sensor, un timestamp y una o varias medidas del sensor. Al agruparlo mediante este patrón, la colección resultante tendría un documento con el identificador del sensor, el tiempo de comienzo de la ventana de tiempo y el tiempo de fin. Luego, se guardan las medidas y su timestamp en un arreglo embebido. Además, se pueden guardar medidas agregadas como la suma o el promedio para cada ventana en el documento.

Este patrón se puede detectar en SQL y al realizar la migración se podría elegir una ventana de tiempo acorde a la realidad del problema.

Si en SQL se tuviera una tabla que tome medidas para un sensor cada cierto tiempo, lo esperado es que la clave primaria de la tabla este formada por un identificador del sensor y un timestamp. Además, el identificador del sensor debe ser clave foránea de una tabla que contenga información sobre los sensores. Entonces, utilizando esta información junto con que la columna timestamp es una fecha se puede identificar una posible implementación de este patrón.

VIII-B. Extended Reference

Este patrón puede ser utilizado cuando tenemos dos colecciones de datos separadas, en donde una hace referencia a la otra y donde se hagan repetidas consultas entre ellas que necesiten realizar JOINs. Por ejemplo, una aplicación de comercio, en donde se tengan colecciones para los pedidos, el cliente y el inventario. A la hora de la performance de la base de datos, esto puede llegar a dar problemas si tenemos muchas consultas entre ellas, dado que las operaciones JOIN son costosas computacionalmente.

Lo que propone este patrón es duplicar parte de la información más utilizada en las consultas de una colección en la otra. De esta manera, evitamos hacer operaciones tipo JOIN entre las colecciones en las consultas más recurrentes.

Sin embargo, hay que tener en cuenta que al actualizar una colección es necesario también actualizar la otra.

Luego de la migración se podría identificar las colecciones que no fueron embebidas en, pero hacen referencia a otra colección. Se podría sugerir al usuario que se podría aplicar este patrón con dichas colecciones. En este caso, el usuario deberá analizar si en su uso de la base se hacen muchas operaciones JOIN sobre las colecciones que sugiere aplicarse el patrón.

VIII-C. The Subset Pattern

Como resultado de realizar la migración, puede ser que se obtenga una colección muy grande en algunos casos. Por ejemplo, si en SQL se tenía una tabla con una colección de artículos a la venta junto con una tabla de reseñas y al migrarlo se utiliza embedding de las reseñas en cada artículo.

Este patrón de diseño busca reducir el uso de memoria RAM al consultar la base de datos. Dada una colección que es grande y que tiene datos que se utilizan poco, propone mover esos datos poco utilizados a otra colección.

Para aplicarlo se buscarían las tablas que excedan cierto umbral de tamaño o tablas que contengan muchas columnas (lo que indicaría una gran cantidad de datos), para sugerirle al usuario que las divida, eligiendo cuales columnas son las que se acceden más infrecuentemente. Esto a su vez permitiría mejorar los tiempos de acceso de la base de datos. También se le podría consultar al usuario al realizar la migración si desea utilizar la técnica de embedding. Al conocer el dominio, el usuario puede decidir si utilizar embedding resultaría en una colección muy grande.

IX. Conclusiones

Se realizó una implementación de un algoritmo capaz de migrar una base de datos MySQL a MongoDB de manera totalmente automatizada en base a un artículo. Se logra aprovechar algunas de las características de Mongo como los documentos embebidos. Además, se descubrieron casos no contemplados por el algoritmo y se tuvo que proponer una forma de migrar esas tablas.

Debido al alto tiempo de ejecución para bases de datos grandes, consideramos que este algoritmo tiene utilidad limitada. Si es un sistema en producción, resultaría muy difícil el poder apagar, realizar la migración y levantar el sistema sin impactar a los usuarios. Una alternativa puede ser empezar a utilizar MongoDB y realizar la migración de a poco.

La diferencia entre los tipos de datos de Mongo y las bases de datos relacionales no son un problema sencillo de resolver. La representación de los datos es algo muy importante al modelar la realidad y si no hay un tipo de datos equivalente la decisión de cual usar debería ser decidido siempre por una persona, tomando la decisión dependiendo de la realidad u otras limitaciones como puede ser memoria.

El algoritmo implementado utiliza solamente las relaciones entre las tablas mediante el estudio de las claves foráneas para armar la estructura en Mongo. Consideramos que esto puede no ser la mejor opción. Por ejemplo, en sakila, la tabla *inventory* se migra como una nueva colección. No obstante, tal vez la mejor opción sea embeberla dentro de la tabla *store*, y la tabla *rental* podría embeberse dentro de *inventory*, teniendo dos niveles de anidación. Estas decisiones deberían hacerse en base a las consultas que se van a realizar.

Por lo tanto, un algoritmo de migración totalmente automático para migrar de bases de datos relacionales a Mongo no es algo que consideremos útil si se tiene en cuenta todo lo mencionado anteriormente. Sin embargo, puede representar un buen punto de partida se quiere pasar de utilizar SQL a Mongo.

Por último, se realizó un estudio sobre la detección de algunos patrones de diseño de MongoDB. Los patrones son específicos para el dominio de los datos, por lo que en algunos casos puede ser de interés aplicarlos y en otros no. Agregarlos de manera automática a la migración podría dar lugar a resultados no deseados, por lo que el usuario debe ser consultado.

X. Trabajo futuro

Como trabajo a futuro se plantea buscar mejorar el rendimiento del algoritmo para que pueda ser utilizado con bases de datos grandes, dado que actualmente con el tiempo de ejecución de la migración esto no es posible. Es muy posible que deba realizarse un paginado ya que no va a ser posible guardar en memoria una tabla entera.

Por otro lado, queda como trabajo a futuro realizar la implementación para detectar los patrones de diseño de MongoDB y realizarle sugerencias al usuario. Esto permitiría que el usuario pueda decidir si aplicar algún patrón de los detectados por el algoritmo mejora su diseño en la realidad que lo esté aplicando.

No se investigó si los triggers de Mongo son capaces de ejecutar acciones similares a los triggers de las bases de datos relacionales. De manera similar, se debería investigar si las vistas de Mongo cumplen una función similar a las de SQL. El artículo con el algoritmo implementado no menciona nada sobre estas dos funcionalidades.

También se puede intentar extender la herramienta para soportar otros motores de bases de datos relacionales como SQL Server o PostgreSQL.

REFERENCIAS

[1] Kuszera, Evandro Miguel, Leticia M. Peres, and Marcos Didonet Del Fabro. *Toward RDB to NoSQL: transforming data with metamorfose framework.* Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. 2019.

[2] Mongo DB. *Building with Patterns: A Summary*. https://www.mongodb.com/blog/post/building-with-patterns-a-summary/. Online; accedido: 14-mayo-2022.

[3] MongoDB. *Building with Patterns: The Bucket Pattern*. https://www.mongodb.com/blog/post/building-with-patterns-the-bucket-pattern. Online; accedido: 06-junio-2022.

[4] MongoDB. *Model Monetary Data*. https://www.mongodb.com/docs/manual/tutorial/model-monetary-data/. Online; accedido: 06-junio-2022.

[5] MongoDB. *Model One-to-One Relationships with Embedded Documents*. https://www.mongodb.com/docs/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/. Online; accedido: 06-junio-2022.

[6] MySQL. *Airportdb introduction*. https://dev.mysql.com/doc/airportdb/en/airportdb-introduction.html/. Online; accedido: 14-mayo-2022.

[7] MySQL. *Employees Sample Database*. https://dev.mysql.com/doc/employee/en/. Online; accedido: 06-junio-2022.

[8] MySQL. *The Sakila example database*. https://github.com/jOOQ/sakila. Online; accedido: 06-junio-2022.

[9] Stanescu, Liana, Marius Brezovan, and Dumitru Dan Burdescu. *Automatic mapping of MySQL databases to NoSQL MongoDB.* 2016 Federated Conference on Computer Science and Information Systems (FedCSIS). IEEE, 2016.

[10] Studio 3T. *SQL to MongoDB Migration*. https://studio3t.com/knowledge-base/articles/sql-to-mongodb-migration/. Online; accedido: 14-mayo-2022.