



Arquitectura de Datos

Práctica Obligatoria

MongoDB

Curso 2023/24

Grupo 80

Realizado por:

José David Rico Dias [100441800], 100441800@alumnos.uc3m.es

Arianna Potente Vázquez [100432091], 100432091@alumnos.uc3m.es

Ernesto Gracia Cancho [100432026], 100432026@alumnos.uc3m.es

ÍNDICE

1.- Introducción.....	2
2.- Modelo UML y Agregados.....	2
2.1.- Modelo de información UML.....	2
2.2.- Perímetro de los agregados.....	15
3.- Pipelines MongoDB.....	38
3.1.- Limpieza de datos.....	38
3.2.- Migración de datos (pipelines de agregación) y anomalías.....	40
4.- Cluster.....	53
5.- Conclusión.....	63

1.- Introducción

Esta práctica tiene como objetivo llevar a cabo el diseño e implementación de los datos históricos musicales de temas, intérpretes y concierto, que se nos han proporcionado en el enunciado; con el fin de reestructurarlos de tal manera que cumplan los casos de uso y las exigencias descritas en el enunciado.

Como es lógico, tanto la creación de la base de datos como la estructuración de las colecciones y documentos dentro de los anteriores, se llevará a cabo a través de MongoDB Compass.

En cuanto a la estructuración del documento presente, seguiremos la guía que se proporciona en el enunciado; siendo el orden el siguiente. En primer lugar, explicaremos el modelo de agregado, donde detallaremos tanto el modelo UML que hemos realizado como el perímetro de los agregados en sí. En segundo lugar, describiremos el diseño y la construcción de los pipelines en MongoDB que nos han permitido llevar a cabo la migración. Por último, describiremos el diseño de los clusters de fragmentación.

Cabe resaltar que todos los puntos anteriores se detallarán con la mayor precisión y claridad posible para poder así mostrar de primera mano todas nuestras decisiones a lo largo de la práctica.

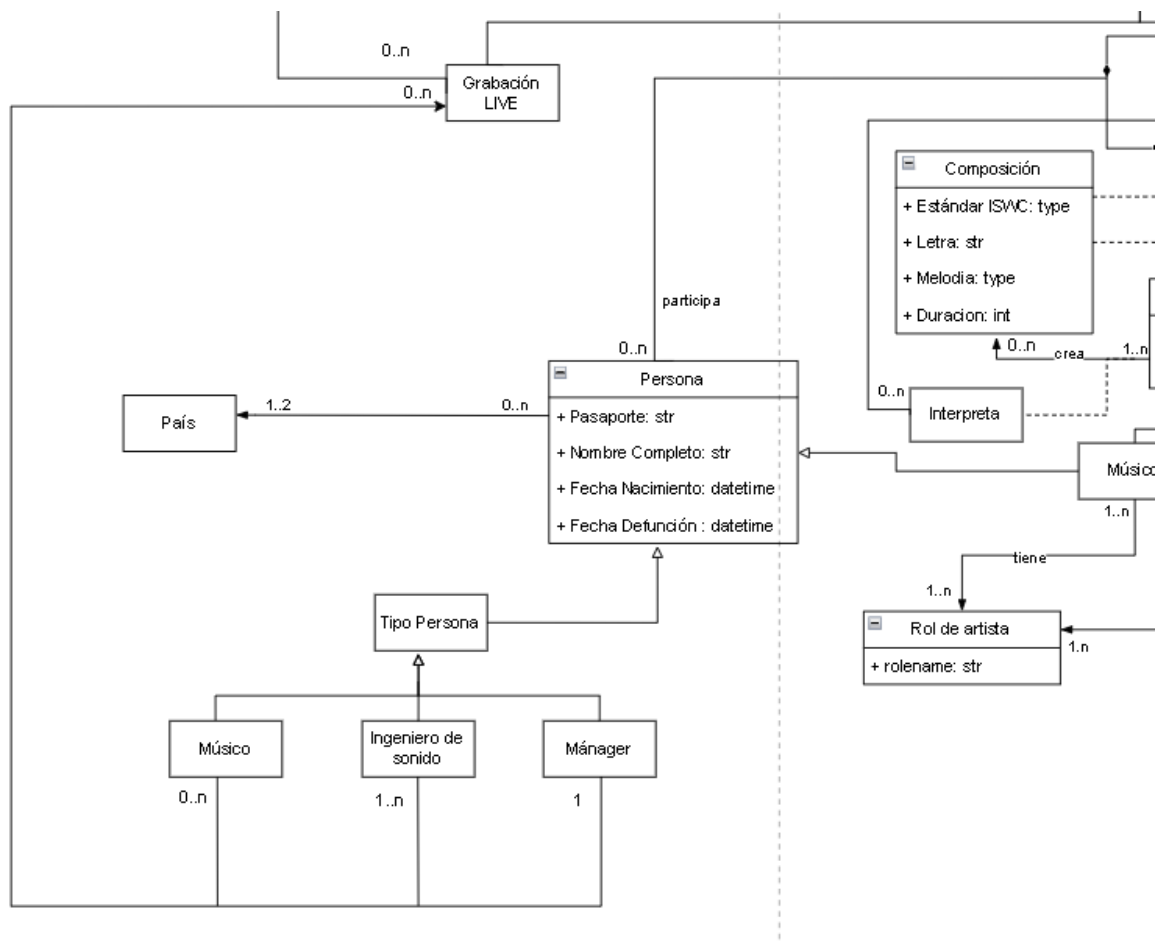
2.- Modelo UML y Agregados

Como hemos mencionado previamente en este apartado, vamos a describir el modelos de información UML y los perímetros de los distintos agregados.

2.1.- Modelo de información UML

En este apartado, mostraremos en primer lugar el diseño completo del diagrama de clases de UML. Posteriormente, explicaremos cada una de las clases que hemos creado por separado.

Por tanto, el diagrama final de UML es el siguiente:



Si nos fijamos en la clase “Persona”, tenemos dos objetos diferentes.

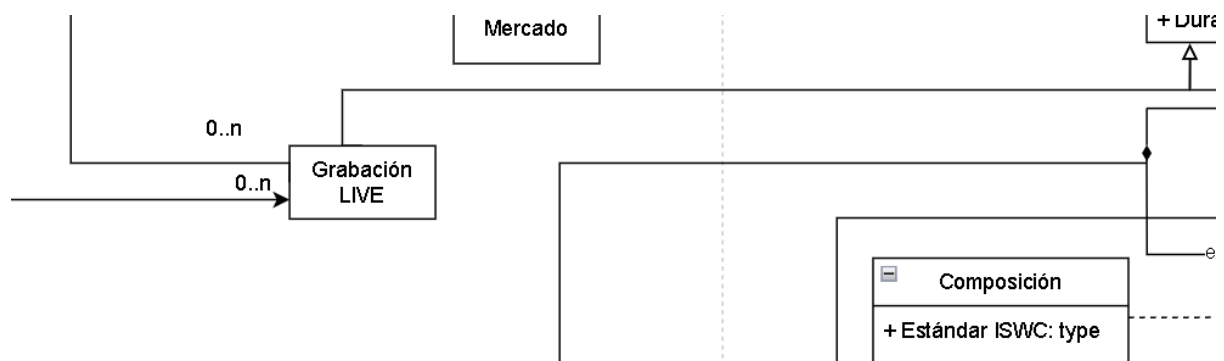
1. País: este objeto hace referencia al país de nacimiento de la persona siendo posible que una persona tenga como mínimo un país y como máximo dos, de ahí la relación que hemos establecido. Esta implica que una persona puede pertenecer a uno o dos países (dependiendo del caso) y el/los países pertenecen a una persona (que es la que tratamos).
2. Tipo persona: hemos establecido este objeto con relación de herencia con persona. Esto quiere decir que una persona es un tipo de persona que a su vez puede ser un músico, un manager o un ingeniero de sonido. Estos tres tipos de personas también son herencia del objeto tipo de persona. Hemos decidido que el diseño sea de esta forma para tener un objeto específico que contenga qué tipo de persona es. Básicamente, ha sido por tenerlo un poco más ordenado (a nuestro parecer).

Si nos fijamos en las relaciones de cada tipo de persona con la clase de “Grabacion_Live”, éstas son de 0..n para el músico, 1..n para el ingeniero de sonido y 1 para el manager. Estas relaciones son de esta forma debido a que en el enunciado se especifica que para hacer una grabación live se necesita si o si un único manager, mínimo un ingeniero de sonido y la posible aparición o no de un músico. Por estas razones, las relaciones con “Grabación_Live” son de la forma especificada en el dibujo.

Para continuar, la clase “Persona” tiene los siguientes atributos:

- Pasaporte: indica el pasaporte de la persona. Este atributo debe ser un string donde se representa según los CSV el país de procedencia con dos caracteres seguido del código numérico.
- Nombre Completo: hace referencia al nombre y apellidos de “pila” de la persona. Este atributo debe ser un string como es lógico.
- Fecha Nacimiento: fecha de nacimiento de la persona. Este atributo deberá ser de tipo datetime, debido a que los datos son pasados en formato fecha DD/MM/AA (día mes año)
- Fecha de defunción: fecha de fallecimiento de la persona. Este atributo deberá ser de tipo datetime, debido a que los datos son pasados en formato fecha DD/MM/AA (día mes año)

Clase Grabación LIVE

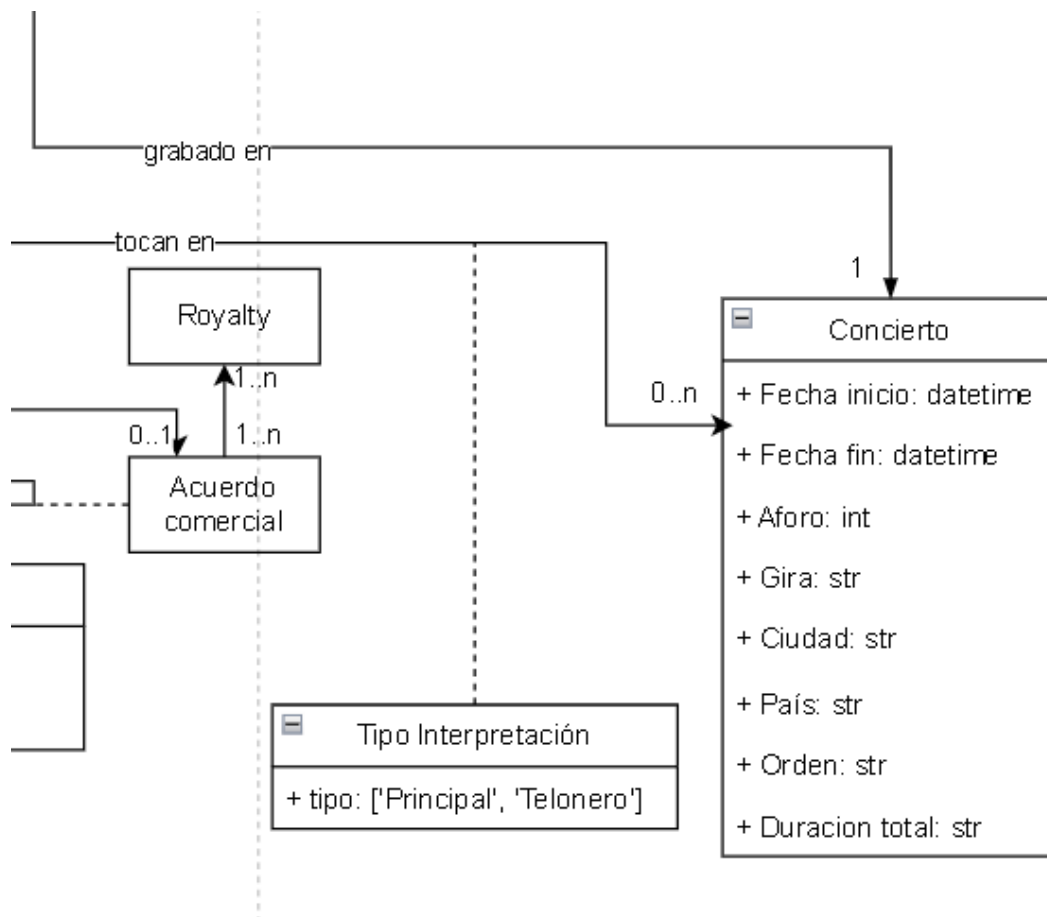


Esta clase se podría en realidad tratar como objeto; sin embargo, en los diseños se va a representar como clase pues en su momento vimos que sería más conveniente. Esta clase no goza de atributos y únicamente como hemos mencionado, muestra relaciones con otras clases. A continuación, vamos a explicar las relaciones:

- Tipo persona: La primera de ellas es con “Tipo persona” explicado anteriormente. Esta relación implica que los tipos de persona pueden hacer 0..n grabaciones live mientras que para hacer una grabación live, se debe tener obligatoriamente un manager (relacion de 1), mínimo un ingeniero de sonido (relación 1..n); y además, puede haber o no músicos en esas grabaciones live.
- Grabación: La siguiente relación es con la clase “Grabación”. Con esta clase mantiene una relación de herencia donde obviamente, representamos que una grabación live es un tipo de grabación.

- Concierto: La última relación que posee es con la clase “Concierto”; ya que en el enunciado se nos proporciona información de que se pueden llevar a cabo grabaciones en vivo de los conciertos. Por tanto, la relación que existe es la siguiente: una grabación live en específico se graba en 1 concierto y de un cierto se pueden grabar de ninguna a infinitas grabaciones siendo esta relación 0..n.

Clase Concierto

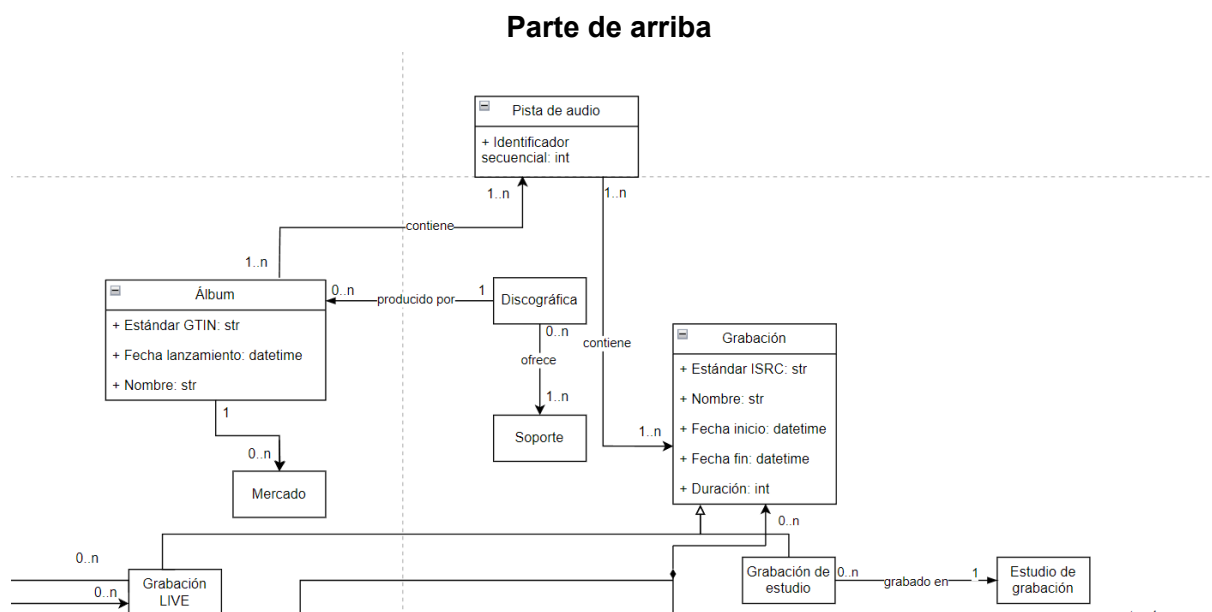


En cuanto a relaciones con otras clases, podemos destacar las siguientes:

- Artista: esta relación que mantiene con conciertos es la de que 1..n artistas tocan en 0..n conciertos (porque un artista puede no haber dado un concierto en su vida). Además, en esta relación, debemos destacar la relación de “Tipo Interpretación” donde los artistas que tocan en un concierto pueden ser o el artista principal o un telonero. Esta relación era importante para satisfacer los requisitos descritos en la práctica.
- Grabación Live: esta relación es la que hemos explicado previamente donde se pueden llevar a cabo de 0..n grabaciones live mientras que solo se pueden llevar a cabo grabaciones live de 1 concierto que está teniendo lugar.

Como se puede observar, la clase “Concierto”, tiene bastantes atributos en referencia a los casos de uso y a los enunciados. Estos son:

- Fecha inicio: este atributo determina la fecha de comienzo de un concierto. Es un atributo de tipo datetime, debido a que los datos son pasados en formato fecha DD/MM/AA (día mes año).
- Fecha fin: este atributo determina la fecha de fin de un concierto. Este atributo existe debido a que un concierto puede empezar a elevadas horas de la noche y terminar de madrugada el día siguiente. Es un atributo de tipo datetime, debido a que los datos son pasados en formato fecha DD/MM/AA (día mes año).
- Aforo: indica el número de asistentes al concierto. Deberá ser un string indicando el número de espectadores físicos del concierto.
- Gira: indica si el concierto es perteneciente a una gira del cantante. El atributo será un string donde se indique el nombre de la gira a la que pertenece.
- Ciudad: indica la ciudad en la que ha tenido lugar el concierto. De nuevo, será un dato tipo string.
- País: indica el país en el que tiene lugar el concierto. El atributo se verá representado como string.
- Orden: va a ser un número que indica el orden en el que se ha tocado en un concierto una canción. Por ejemplo, si el orden es 1, la canción fue la primera en tocar en el concierto. El atributo es de tipo string.
- Duración Total: indica la duración total del concierto. El atributo es de tipo string.



Como se puede apreciar en la imagen superior la parte de arriba del UML consta de 3 clases principales que se relacionan entre sí y a su vez con objetos.

Álbum

- Pista de audio: Se relaciona con la clase “Pista de audio” en una relación 1..n. Esto quiere decir que uno o varios álbumes pueden contener una o varias pistas de audio, ya que como se menciona en el enunciado un álbum al final se compone por una agrupación de pistas de audio que a su vez se compone de una agrupación de grabaciones.
- Mercado: Se relaciona con el objeto “Mercado” en una relación 1 a 0..n. Esto lo hemos decidido primeramente porque los álbumes una vez realizados se pueden sacar al mercado (que hemos decidido que no tenga atributos ya que el enunciado no especifica nada). A su vez, el álbum puede existir pero esto no requiere que se tenga que lanzar al mercado pueden haber solo realizado un álbum que todavía no se quiera lanzar, es por ello que si o si tiene que haber 1 álbum pero 0 o muchos mercados a lo que se lance.
- Discográfica: El objeto “Discográfica” se relaciona con la clase “Álbum” en una relación 1 a 0..n. Esto significa que un álbum es producido por 1 discográfica y las discográficas producen 0 o más álbumes.
- Soporte: El objeto “Discográfica” se relaciona con el objeto “Soporte” en una relación 0..n a 1..n. Es decir que álbum que es producido por 0..n discográficas pero si se hace al menos existe un soporte o varios.

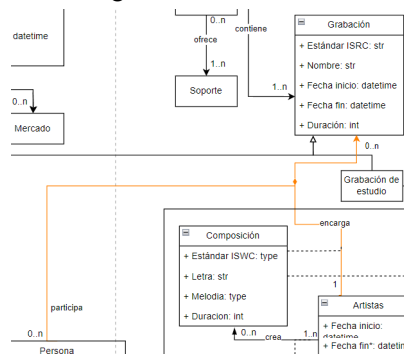
Pista de audio

- Grabación: se relaciona la clase “Pista de audio” con la clase “Grabación” en una relación 1..n, ya que una pista de audio es un conjunto de grabaciones y puede haber varias pistas de audio
- Álbum: Se relaciona con la clase “Álbum” en una relación 1..n. Esto quiere decir que uno o varios álbumes pueden contener una o varias pistas de audio, ya que como se menciona en el enunciado un álbum al final se compone por una agrupación de pistas de audio.

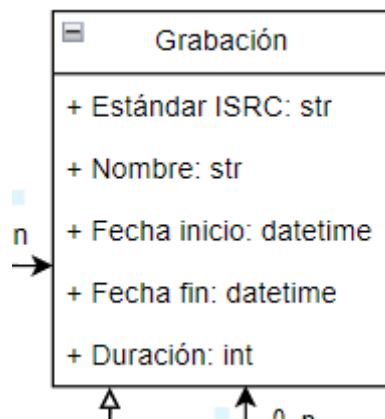
Grabación

- Pista de audio: se relaciona la clase “Pista de audio” con la clase “Grabación” en una relación 1..n, ya que una pista de audio es un conjunto de grabaciones y puede haber varias pistas de audio.
- Grabación LIVE y Grabación de estudio: estos dos objetos se relacionan con grabación en una relación de herencia, es decir todos los atributos de esta última los heredan los objetos de grabación LIVE y grabación estudio. A su vez la grabación de estudio, según consta en el enunciado está grabado si o si en un estudio de grabación. Es decir, si existe alguna grabación de estudio (0..n) solo puede haber sido grabada en un estudio (1).

- Persona y Artistas: la clase “Grabación” está compuesta por cero o más instancias de la clase “Persona” y una instancia de la clase “Artistas”. De esta manera, al ser una composición, nos aseguramos que las varias personas que pueden participar en la grabación y el único artista que la solicita (y a nombre de quién queda registrada la grabación) estén unidas a la misma grabación.



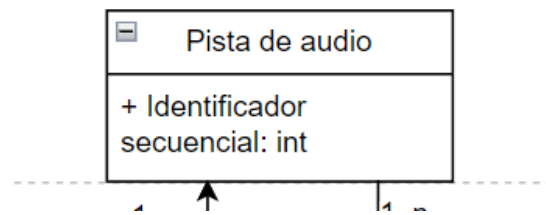
Clase Grabación



La clase “Grabación” tiene los siguientes atributos:

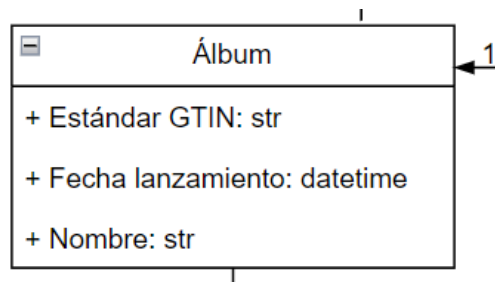
- Estándar ISRC: que corresponde al código universal que las grabaciones deben de tener. Como el estándar es alfanumérico, hemos decidido que sea de tipo string por conveniencia.
- Nombre: corresponde al nombre de la canción o tema. Se ha puesto en grabación en vez de pistas de audio para simplificar luego el esquema, ya que como pistas de audio es una agrupación de grabaciones obtendremos esta información mediante referencias de pistas de audio a grabación. A su vez es un string porque son nombres de canciones.
- Fecha inicio y fecha fin: corresponde a la fecha de inicio de una grabación tal como indica el enunciado y por tanto debe ser un datetime y de esa manera se puede saber si una grabación está “activa” o se ha terminado.
- Duración: la duración total de la canción en minutos, por eso el tipo es integer.

Clase Pista de audio



La clase "Pista de audio" tiene el atributo de Identificador secuencial que corresponde al identificador de la pista de audio que a su vez tiene el requisito de ser secuencial según el enunciado, es por ello que es un integer.

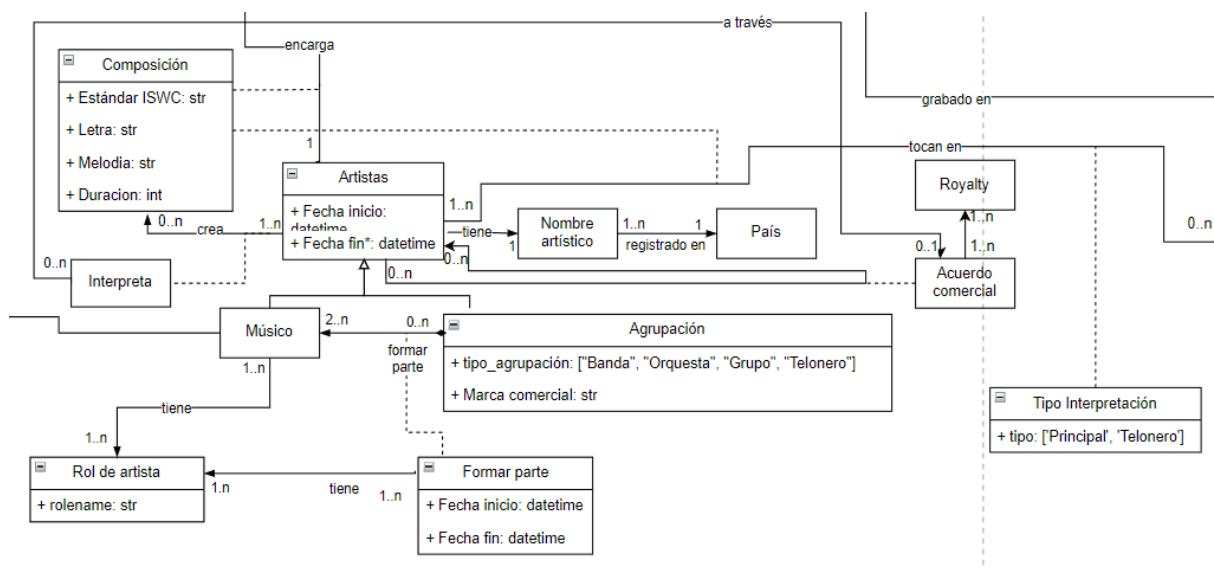
Clase Álbum



La clase "Álbum" tiene los siguientes atributos:

- Estándar GTIN: corresponde con el estándar que deben de seguir los álbumes y por los cuales se identifican, que es una cadena de 8 caracteres de números pero para simplificar lo vamos a tratar con un string.
- Fecha de lanzamiento: corresponde a la fecha en la que se lanza el álbum creado y por es un datetime.
- Nombre: corresponde al nombre del álbum y por tanto es un string.

Parte central



Como se puede apreciar en la imagen superior la parte de arriba del UML consta de 6 clases principales que se relacionan entre sí y a su vez con objetos.

Composición

- Artistas: la clase “Artistas” se relaciona con la clase “Composición” en una relación 1..n a 0..n. Es decir, que los artistas pueden o no crear composiciones pero si se crean como mínimo deben de haber sido creadas por un artista.
- Relación discontinua que sale de línea encarga: la relación de un artista con las grabaciones, es que encarga grabar una composición, con lo que se trata de una relación con atributos.
- Relación discontinua que sale de línea tocan en: Dentro de la relación de un artista con un concierto, la relación se encuentra con el atributo composición, ya que el artista participa en el concierto a través de una composición que interpreta.

Artistas

- Composición: la relación que mantiene la clase artista con composición es la siguiente. Los artistas crean composiciones de tal manera que como mínimo 1..n artistas pueden crear 0..n composiciones. Lo hemos determinado de esta manera porque una artista no tiene porque haber creado ninguna composición para ser denominada artista.
- Nombre artístico y País: esta relación implica que un artista tiene relacionado un nombre artístico. Además, dicho nombre artístico puede estar registrado en un país en específico de tal manera que no haya dos nombres artísticos en un mismo país. Por tanto, la relación resultante entre nombre artístico y país quedaría de la siguiente manera: un nombre artístico está registrado en un país y 1 país puede contener diferentes nombres artísticos.
- Músico: este objeto mantiene una relación de herencia con la clase de artistas. Esto lo que indica es que un músico es un artista. A su vez, el objeto músico está relacionado tanto con la clase “persona” como con la clase “Rol de artista”. En cuanto a la primera relación, de nuevo indicamos en el UML que tienen una relación de herencia; donde indicamos que un músico también es una persona a parte de ser un artista. Con respecto a la segunda relación (con Rol de artista), lo que representamos es que el músico tiene distintos roles. Por roles nos referimos a que puede ser por ejemplo guitarrista, bajista o batería por ejemplo. Por ello, la relación establece que 1..n músicos pueden tener 1..n roles; o lo que es lo mismo, cada músico puede tener asociado de 1 a n roles y viceversa. Esto tiene sentido puesto que puede haber casos donde un solo músico sea capaz de tocar tres instrumentos a la vez.

Por último, debemos mencionar una última relación que mantiene el músico con la clase “Agrupación”. Esta relación establece que un músico puede formar parte en 0..n agrupaciones (indicando que un músico no tiene porqué pertenecer a un grupo,

o puede pertenecer a varios grupos); y que una agrupación debe estar compuesta como mínimo por 2 músicos, pudiendo llegar hasta n músicos como en el caso de una orquesta.

- Concierto: hemos indicado también la relación de la clase artista con la clase concierto. En ella establecemos que 1..n artistas tocan en 0..n conciertos. Esto es así porque un artista puede no haber tocado en un concierto a lo largo de su trayectoria.
- Interpreta: un artista se relaciona con un acuerdo comercial a través de la interpretación que da a una composición.
- Royalty: que se puede generar con un acuerdo con comercia
- Acuerdo comercial: de un artista a otro, ya que varios artistas pueden establecer acuerdos comerciales con otros.

Agrupación

- Músico: la clase “Agrupación” está compuesta por dos o más instancias del objeto “Músico”. De esta manera, al ser una composición, nos aseguramos que los grupos como mínimo tengan dos integrantes músicos.
- Artistas: la clase “Agrupación” tiene una relación de herencia con la clase “Artistas”, es decir que la primera hereda los atributos de la segunda, además de tener los suyos propios.
- Tipo Interpretación: Cuando un artista actúa en un concierto, el tipo de interpretación puede ser como artista principal o como telonero

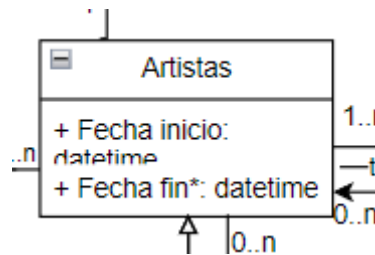
Rol de artista

- Músico: el objeto “músico” se relaciona con la clase “Rol de artista” en una relación 1..n. Es decir, que uno o varios músicos pueden tener uno o varios roles.
- Formar parte: la clase “Formar parte” se relaciona con “Rol de artista” en una relación 1..n. Es decir, que un músico que “forme parte” de una agrupación va a contener una fecha de inicio y fin de su participación en la agrupación. A su vez, esas fechas pueden ser varias si se tienen muchos roles, luego una fecha de inicio y fin para cada rol que tenga el artista en la agrupación.

Formar parte

- Rol de artista: esta relación es la previamente explicada en la clase rol de artista.
- Tipo interpretación clase con línea discontinua: un artista de tipo músico se relaciona con un artista de tipo banda a través de forma_parte, que contiene los atributos de fecha de inicio y fecha de fin de la relación.

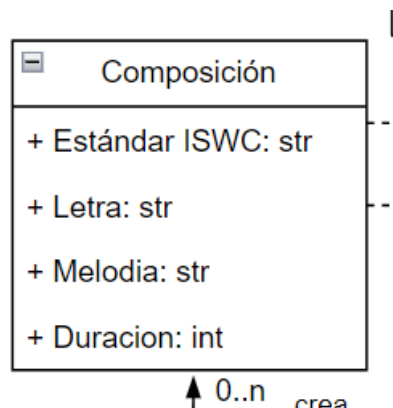
Clase Artistas



La clase “Artistas” se compone de los siguientes atributos:

- Fecha inicio: este atributo representa la fecha en la que comenzó una persona a ser un artista. Como es lógico, el atributo deberá ser representado en formato datetime, donde especificamos el día, mes y año (DD/MM/AA) de comienzo.
- Fecha fin: este atributo se corresponde con la fecha en la que una persona deja de ser un artista. De nuevo, este atributo debe ser representado en formato datetime (DD/MM/AA), indicando la fecha de fin de un artista.

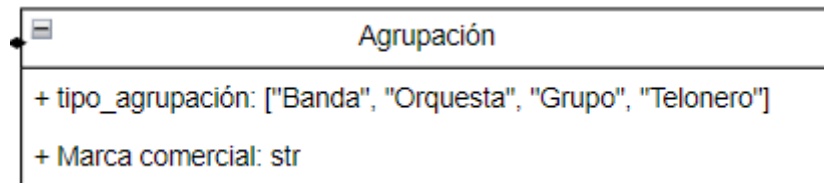
Clase Composición



La clase “Composición” se compone de los siguientes atributos:

- Estándar ISWC: es el código universal por el que las composiciones deben de estar identificadas. Esto es un código alfanumérico y lo pondremos como un string por conveniencia.
- Letra: se refiere a la letra de la canción y por tanto debe de ser un string.
- Melodía: se refiere a la melodía de la canción y por tanto debe de ser un string.
- Duración: se refiere a la duración de la canción en minutos y por ello debe de ser un integer.

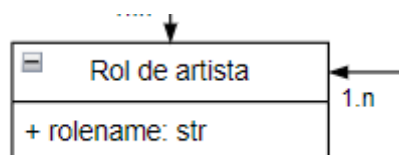
Clase Agrupación



La clase “Agrupación” se compone de los siguientes atributos:

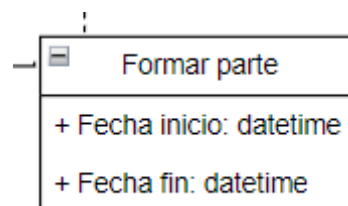
- tipo agrupación: se refiere a que clase de agrupación es la que se forma estableciendo aquellas que hemos decidido que se aceptan como: Banda, Orquesta, Grupo y Telonero, siendo esta última una variante de las 3 primeras.
- Marca comercial: se refiere a la marca comercial que se crea una vez se establece la agrupación y por tanto debe de ser un string.

Clase Rol artista



La clase “Rol artista” se compone del atributo rolename, que va a representar en forma de string el nombre del rol asociado al músico. Por ejemplo, “guitarrista”.

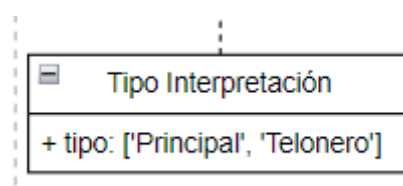
Clase Formar parte



La clase “Formar parte” se compone de los siguientes atributos:

- Fecha inicio: este atributo tiene como objetivo representar la fecha de comienzo de una agrupación. El atributo debe ser representado en formato “datetime”, de la siguiente forma DD/MM/AA.
- Fecha fin: este atributo tiene como objetivo representar la fecha de fin de una agrupación de músicos. El atributo debe ser representado en formato “datetime”, de la siguiente forma DD/MM/AA.

Clase Tipo interpretación



La clase “Artistas” se compone del atributo tipo que hace referencia al tipo de interpretación que el artista da en un concierto si es el “principal” o por el contrario si es el “telonero” que toca al principio del concierto probando la logística del mismo.

2.2.- Perímetro de los agregados

Para asegurarnos de que la nueva arquitectura sigue nuestros criterios y a su vez tanto los casos de uso especificados como el enunciado en general donde se detalla cómo se quiere realizar y que es lo que debe contener, hemos realizado unos esquemas de validación. Nuevamente para realizar esto, nos hemos ceñido a lo que se solicita en el enunciado sin pensar en los numerosos casos que puede existir en el amplio mundo de la música. A continuación detallaremos cuales son los campos que hemos decidido implementar en los esquemas y la justificación de los mismos.

Conviene subrayar que hemos decidido realizar los esquemas de validación directamente en vez de establecer los agregados y sus perímetros y posteriormente explicar los esquemas primeramente por eficiencia. Segundo, porque nos parecía redundante explicar como un “croquis” y luego la ampliación dos veces y se iba a entender mejor explicándolo de una. Esto no quiere decir que nosotros a la hora de crearlos no hayamos pensado primero en los campos y en sus limitaciones y luego rellenarlos con los datos más específicos. Lo mencionado anteriormente es una decisión solo para el contenido de la memoria.

Antes de comentar cada esquema de validación, queremos resaltar los siguientes aspectos comunes en cada uno de ellos:

- **bsonType: 'object'** : este campo se repite en cada esquema. Lo que indica es básicamente que el objeto especificado puede contener varios campos. Donde cada uno de ellos, puede tener o no diferentes tipos de datos. Es básicamente definir la estructura que puede tener un documento de una colección en específico.
- **title**: básicamente es un campo que es meramente informativo y por tanto a nivel de la práctica no es muy relevante. Sin embargo, lo hemos incluido para que la información de cada esquema quede clara.
- **required**: son los diferentes campos que deben existir para que sea válido el esquema según nuestros criterios y entendimiento del enunciado. Además, debemos mencionar que aquellos campos que no se encuentren en “required”, pero si en el esquema, significa que los que no están contenidos pueden ser valores nulos. Esto es así, porque hemos considerado que su existencia en cada colección no es vital para un documento consistente y coherente con lo que representa.
- **properties**: este campo tiene como objetivo especificar cada uno de los campos que pueden existir dentro de la colección de MongoDB. En cada uno de ellos, hemos especificado el tipo de cada valor asociado al campo, y una descripción de cada campo.

Además, hemos pasado todos los nombres en inglés por conveniencia y estandarización. Asimismo, muchos lenguajes y plataformas siguen convenciones basadas en inglés, promoviendo consistencia y legibilidad en el código. Esta práctica contribuye a un desarrollo más eficiente y mantenible a lo largo del tiempo. Por otro lado, para elegir el tipo de dato que ponemos a cada campo hemos elegido el BSON type. El uso de BSON es comúnmente asociado con bases de datos NoSQL, especialmente MongoDB, que utiliza BSON para almacenar documentos de manera eficiente. Por ello, hemos seguido los códigos asociados que se pueden encontrar en el manual de MongoDB.

A continuación, vamos a detallar cada uno de las decisiones que hemos tomado en cada esquema:

- Esquema de Validación Concerts

En primer lugar, debemos mencionar que los campos obligatorios son los siguientes a existir en cada documento de la colección “concerts”: “start_date”, “capacity”, “country”, “city”, “artists”. Es importante tener estos seis campos pues nos va a permitir en primer lugar satisfacer sin problemas los casos de uso. Además, hay algunos campos como por ejemplo “artists” que es indispensable tenerlo pues si o si tiene que haber artistas tocando en un concierto porque sin ellos, no habría concierto.

Para continuar, vamos a explicar cada uno de los campos por separado de forma detallada:

- **start_date**: representa el día de comienzo del concierto, que debe ser un dato de tipo timestamp mostrando DD/MM/AA.
- **end_date**: representa el día de fin de un concierto, que debe ser un dato de tipo timestamp mostrando DD/MM/AA. Este campo no es requerido en los documentos de la colección “concerts”.
- **capacity**: representa el aforo que ha tenido un determinado concierto. Hemos definido que el dato deberá ser de tipo int. Además, hemos establecido un rango de personas entre 0 (ninguna persona acude al concierto) y 100 millones de personas (por si hubiese algún caso en los CSV donde se mostrase un aforo grande poder contenerlo).
- **country**: representa el país en el que tiene lugar el concierto. Deberá ser un dato de tipo string.
- **city**: representa el nombre de la ciudad en la que tiene lugar el concierto. Deberá ser un dato de tipo string.
- **tour**: representa el nombre del tour en el que se encuentra el concierto. Solo si este se encuentra de gira. Posteriormente en fases más avanzadas del proyecto se verá como el valor de este campo en vacío implica que no pertenece a un tour.
- **order**: indica el orden en el que se ha tocado una canción dentro de un concierto. El dato es de tipo int. No hemos incluido esta información como required, debido a que no nos parece un dato importante con respecto al concierto en sí.
- **total_duration**: indica la duración total del concierto. El dato será de tipo int. De nuevo, no hemos incluido este campo como required porque no nos parece un dato lo suficientemente importante sobre el concierto en relación a los casos de uso.
- **artists**: este campo va a ser un array de artistas, donde cada uno de ellos se va a encontrar representado en este array en función de su ObjectId. Este campo lo

hemos incluido puesto que debemos satisfacer la relación que establecimos en el UML donde un concierto es tocado por 1 a n artistas. De esta forma, también hemos especificado que debe haber como mínimo un artista que toque en el concierto.

- **live_records**: este campo es un array de live_records, donde cada array va a estar representado por su ObjectId correspondiente. Además, hemos indicado que el array puede ser vacío puesto que en la relación del UML hemos definido que se pueden realizar de 0 a n grabaciones live. Por ello, debemos mostrar el caso en el que no haya grabaciones del concierto.

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'Concert Object Validation',
    required: [
      'start_date',
      'capacity',
      'country',
      'city',
      'artists'
    ],
    properties: {
      start_date: {
        bsonType: 'timestamp',
        description: '\start_date\' must be a timestamp and is
required'
      },
      end_date: {
        bsonType: 'timestamp',
        description: '\end_date\' must be a timestamp'
      },
      capacity: {
        bsonType: 'int',
        minimum: 0,
        maximum: 100000000,
        description: '\capacity\' must be an integer and is
required'
      },
      country: {
        bsonType: 'string',
```

```

        description: '\country\' must be an string and is
required'
    },
    city: {
        bsonType: 'string',
        description: '\city\' must be an string and is required'
    },
    tour: {
        bsonType: 'string',
        description: '\tour\' must be an string'
    },
    order: {
        bsonType: 'int',
        description: '\order\' must be an int'
    },
    total_duration: {
        bsonType: 'int',
        description: '\tota_duration\' must be an int'
    },
    artists: {
        bsonType: 'array',
        minItems: 1,
        items: {bsonType: 'objectId'},
        description: '\artists\' must be an array and is
required'
    },
    live_records: {
        bsonType: 'array',
        minItems: 1,
        items: {
            bsonType: 'objectId'
        },
        description: '\live_record\' must be an array'
    }
}
}
}
}

```

Esquema de Validación artists

En primer lugar, este esquema tiene dos campos que son requeridos, siendo estos: “start_date” y “artistic_name”.

A continuación, vamos a detallar cada uno de los campos del esquema de una forma más detallada:

- **start_date**: hace referencia a la fecha en la que el artista comenzó, valga la redundancia, a ser artista. Obviamente, el tipo de dato deberá ser del tipo timestamp siguiendo el formato DD/MM/AA.
- **end_date**: hace referencia a la fecha en la que el artista finaliza su recorrido, y deja de ser artista. De nuevo, el tipo de dato que debe contener este campo es del timestamp siguiendo también el formato DD/MM/AA.
- **artistic_name**: este campo hace alusión a la relación de la clase artista con el objeto nombre artístico o “artistic_name”. En esta relación, se especifica que un artista tiene un nombre artístico que a su vez, este nombre artístico, puede estar registrado en 1 país; sin embargo, un país puede contener de 1..n nombres artísticos si este existe. Por tanto, este primer objeto encastrado, nos va a permitir detallar dos relaciones que ya establecimos en el UML.

De esta forma establecemos como objeto a este campo, indicando que va a contener diversos campos; entre los que encontramos 2 que son de carácter obligatorios: “name” y “country”.

El primero de ellos, será un string donde se muestre el nombre que se encuentra asociado a un nombre artístico. El segundo, hace referencia al país que está asociado al nombre artístico; y el tipo de dato será un string. De esta forma, cada nombre artístico tendrá asociado un país y un nombre.

- **musician**: en este campo vamos a encastrar otro objeto definido como “rolenames” que viene a ser directamente los roles. En este campo representamos la relación que existe entre el músico y los roles siendo la relación la siguiente. 1.. n músicos, pueden tener 1..n roles y 1..n roles pueden tener 1..n músicos.

Hemos decidido que vamos a representar los roles por medio de un array de strings, donde como mínimo el array va a estar formado por un string; es decir, por un rol. Además, este rol, va a cumplir un patrón que es que cada rol (el string), se representa en letras mayúsculas '^[A-Z]+\$'.

Por último, hemos decidido añadir un campo adicional donde mostramos la relación de herencia del campo “musician” con respecto a la clase “person”, a través del objectId de la clase “persons”.

- **members**: este campo tiene como objetivo describir la clase agrupación descrita en el UML.

En primer lugar, está formada por “type_grouping”, que hace referencia al tipo de agrupación que se puede tener en la música como por ejemplo una banda o una orquesta. Esta deberá ser un string.

En segundo lugar, tenemos el campo denominado como “comercial_brand”, donde se debe contener un string que muestra la marca comercial de la agrupación.

Por último, tendremos un tercer campo denominado como “members” donde estableceremos la relación de los miembros con la clase “artists” en sí. De esta forma podemos asociar a cada miembro todos los datos que hemos ido definiendo en los campos anteriores; pero de forma específica para los miembros de un grupo. Para ello, hemos decidido que la mejor manera de representarlo sería a través de un array de objetos donde cada objeto del array estaría formado por 4 campos siendo estos:

1. **artistId**: referencia del objeto a la clase “artists”. El dato deberá ser del tipo objectId.
 2. **start_date**: fecha de inicio del miembro en el grupo. El dato deberá ser de tipo datetime.
 3. **end_date**: fecha de fin del miembro en el grupo. El dato deberá ser de tipo datetime.
 4. **roles**: donde mostramos los distintos roles musicales que puede tener el miembro. Se representará por medio de un array de strings en caso de que un miembro tenga más de un rol.
- **concerts**: representa la relación entre la clase “artists” y la clase “concerts”. En ella mostramos la siguiente relación descrita en el UML. 1..n artistas pueden tocar en 0..n conciertos. Para ello, lo representaremos a través de un array, donde cada posición estará conformada por dos campos: el “concertId” referenciando el concierto por medio del “objectId”, y el rol del artista.
 - **compositions**: es una referencia a la clase compositions a través de un array de objectId. Lo hemos puesto en forma de array debido a que un artista puede haber realizado o ninguna composición hasta infinitas composiciones (0..n composiciones).

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'Artists Object Validation',
    required: [
      'start_date',
      'artistic_name'
    ],
    properties: {
```

```

    start_date: {
      bsonType: 'timestamp',
      description: '\start_date\' must be a timestamp and is
required'
    },
    end_date: {
      bsonType: 'timestamp',
      description: '\end_date\' must be a timestamp'
    },
    artistic_name: {
      bsonType: 'object',
      description: '\artistic_name\' must be an object and is
required',
      required: [
        'name',
        'country'
      ],
      properties: {
        name: {
          bsonType: 'string',
          description: '\name\' must be an string and is
required'
        },
        country: {
          bsonType: 'string',
          description: '\country\' must be an string and is
required'
        }
      }
    },
    musician: {
      bsonType: 'object',
      description: 'musician must be an object',
      required: [
        'roles'
      ],
      properties: {
        personID: {
          bsonType: 'objectId'
        },
        roles: {

```

```

        bsonType: 'array',
        description: '\roles\' must be an array and is
required',
        minItems: 1,
        items: {
            bsonType: 'string',
            pattern: '^[A-Z]+$'
        }
    },
    },
    members: {
        type_grouping: {
            bsonType: "string"
        },
        commercial_brand: {
            bsonType: "string"
        },
        members: {
            bsonType: "array",
            minItems: 2,

            items: {
                bsonType: "object",
                properties: {
                    artistId: {bsonType: "objectId"},
                    start_date: {bsonType: "Datetime"},
                    end_date: {bsonType: "Datetime"},
                    roles: {bsonType: "array", items:
{bsonType: "string"}}
                }
            }
        }
    },
    concerts: {
        bsonType: 'array',
        items: {
            bsonType: 'object',
            properties: {
                concertId: {
                    bsonType: 'objectId'

```

```

    },
  },
},
compositions: {
  bsonType: 'array',
  items: {
    bsonType: 'objectId'
  }
}
}
}
}
}
}

```

Esquema de Validación compositions

En este esquema hemos decidido poner en “required”, los campos de “ISWC” ya que esto es lo único que se exige en el enunciado en cuanto a composiciones y “created_by” ya que como al igual que en la grabaciones debemos de saber el artista que ha creado la composición. El resto de campos son información adicional sobre las composiciones y por tanto interesantes pero no requeridas.

En cuanto al resto de campos los iremos explicando uno a uno:

- **ISWC**: hace referencia al código universal por el que las composiciones deben de estar identificadas. Este código debe de empezar siempre con la letra “T” seguido de un número de nueve dígitos único. Por último se añade un dígito al final a modo de chequeo que se calcula mediante la fórmula de Luhn. Por ello, hemos decidido que el campo tenga el regex `'^T-\d{9}-\d$'` mediante el atributo de pattern.
- **lyrics**: hace referencia a la letra de la canción y por tanto debe de ser un string.
- **melody**: hace referencia a la melodía de la canción y por ello tiene que ser un dato binario debido a que los artistas pueden tratar con varias herramientas para crear la melodía y subirla en diferentes formatos. Al poner que sea binario estamos teniendo flexibilidad en aceptar todos estos formatos y no restringir nada.
- **created_by**: hace referencia a los artistas que han creado la composición. Esto es un array (de las artistas) que debe contener ítems del tipo objectId (es decir el Id de cada artista) de esta manera estamos haciendo referencias a ellos como si fuera un puntero.

- **interpreted_by**: hace referencia al escenario en el que se esté interpretando versiones de temas de otros músicos con los que han llegado a un acuerdo comercial o canciones suyas propiamente. Por ello, es un array compuesto por el id del artista “artistId” al que pertenece la canción (haciendo referencias a ellos como si fuera un puntero), el id del acuerdo comercial “commercial_deals_Id” al que se llega con el artista dueño de la canción (haciendo referencias a ellos como si fuera un puntero) y los royalties “royalties”, es decir, el pago a terceros por los derechos de uso de los temas que han compuesto y que ha sido utilizados o versionados por los artistas; por ello este último es un int representando el pago en dinero. Tanto el acuerdo comercial como los royalties no son obligatorios solo se ponen en caso de haberlos, si fuese que un artista interpreta su propio tema estos campos serían nulos pero el “artistId” no.

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'Artists Object Validation',
    required: [
      'ISWC',
      'created_by'
    ],
    properties: {
      ISWC: {
        bsonType: 'string'
        pattern: '^T-\d{9}-\d$'
      },
      lyrics: {
        bsonType: 'string'
      },
      melody: {
        bsonType: 'binData'
      },
      created_by: {
        bsonType: 'array',
        items: {
          bsonType: 'objectId'
        }
      },
      interpreted_by: {
        bsonType: 'array',
        items: {
          bsonType: 'object',

```

```
        properties: {
            artistsId: {
                bsonType: 'objectId'
            },
            commercial_deals_Id: {
                bsonType: 'objectId'
            },
            royalties: {
                bsonType: 'int'
            }
        }
    }
}
}
```

Esquema de Validación album

En este esquema hemos decidido poner todos los campos en “required”, ya que el nombre, la fecha de lanzamiento y el estándar GTIN son obligatorios por el caso de uso. Además, como el álbum es una agrupación de pistas de audio tiene que tener el campo también para hacer una referencia (estilo como un puntero) y “producers” también debe ser necesario ya que si se crea un álbum debe de haber una discográfica de por medio y tener por lo menos un formato. Sin embargo, “markets” no es requerido, ya que como se ha mencionado anteriormente el álbum puede haber sido creado pero no lanzado al mercado aún, y por ello puede ser nulo.

En cuanto al resto de campos los iremos explicando uno a uno:

- **st GTIN:** hace referencia al estándar que deben de seguir los álbumes. Aquí destacar que hemos impuesto que siga un patrón específico mediante el regex `'^[0-8]{8}$'`. Esta decisión está basada en la elección del estándar GTIN-8 ya que se utiliza para productos donde el empaque es pequeño. Sabiendo que estamos tratando con álbumes independientemente de su formato este no es nunca grande. El estándar GTIN-8 está formado por 8 dígitos y estos van del 0 al 8 ya que el 9 está reservado únicamente para el uso del estándar GTIN-14.
- **release_date:** hace referencia a la fecha de lanzamiento del álbum y por ello es un timestamp.
- **name:** hace referencia al nombre del álbum y por ello es un string.

- **markets:** hace referencia al mercado o mercados a los que se lanza el álbum. Hemos decidido que sea un array que contenga todos los mercados del álbum que tenga como mínimo un mercado (minItems:1) ya que como el campo no es requerido puede ser nulo, pero si se pone, mínimamente va a haber un mercado en donde el álbum ha sido lanzado. A su vez, hemos establecido que los ítems que estén dentro del array sean strings que sigan el regex `'^[A-Z]{2}$'` especificado en el patrón. Esto último quiere decir que los países (mercados) van a tener la estructura de dos letras, por ejemplo España = ES.
- **audio_tracks:** hace referencia a las pistas de audio contenidas en el álbum. Esto es un array (de las pistas de audio del álbum) que debe contener ítems del tipo objectId (es decir el Id de cada pista de audio) de esta manera estamos haciendo referencias a ellas como si fuera un puntero.
- **producers:** es un array de objetos que debe contener el nombre de la discográfica (name) y los formatos del álbum (formats) y como mínimo debe de haber un conjunto de estos (discográfica + formato). El nombre de la discográfica debe ser solo uno por tanto es un string. Sin embargo los formatos son un array ya que el álbum se puede haber sacado en uno (minItems:1) o varios formatos. A su vez, mediante "enum" hemos forzado a que los formatos sean solo CD, vinilo y electrónico y que se tengan que escribir de esa manera específica ya que el enunciado nos daba libertad en este ámbito.

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'Album Object Validation',
    required: [
      'st_GTIN',
      'release_date',
      'name',
      'audio_tracks',
      'producers'
    ],
    properties: {
      st_GTIN: {
        bsonType: 'string',
        description: '\st_GTIN\ must be a string and is
required',
        pattern: '^[0-8]{8}$'
      },
      release_date: {
        bsonType: 'timestamp',
        description: '\release_date\ must be a timestamp and is
required'
      },
      name: {
```

```

    bsonType: 'string',
    description: '\name\' must be a string and is required'
  },
  markets: {
    bsonType: 'array',
    minItems: 1,
    items: {
      bsonType: 'string',
      pattern: '^[A-Z]{2}$'
    },
    description: '\markets\' must be an array'
  },
  audio_tracks: {
    bsonType: 'array',
    minItems: 1,
    items: {
      bsonType: 'objectId'
    },
    description: '\audio_tracks\' must be an array and is
required'
  },
  producers: {
    bsonType: 'array',
    minItems: 1,
    items: {
      bsonType: 'object'
    },
    required: [
      'name',
      'formats'
    ],
    properties: {
      name: {
        bsonType: 'string'
      },
      formats: {
        bsonType: 'array',
        minItems: 1,
        'enum': [
          'CD',
          'Vinyl',
          'Electronic'
        ]
      }
    }
  }
}

```

```
}  
}
```

Esquema de Validación recordings

En este esquema hemos decidido poner todos los campos en “required”, ya que según los casos de uso se debe ver los artistas que han participado en una grabación (“ordered_by: participants”), se debe establecer el código ISRC, la fecha de grabación y cualquier dato relevante (“name”, “duration”, “artists” y “audio_tracks”); “Para cada grabación en la que ha participado el artista, el sistema debe listar los álbumes en los que está incluida” esto se encuentra en la referencia a “audio_tracks” ya que este último a su vez luego en su esquema tiene una referencia a álbum. En el caso de uso “ Se localizarán todos los intérpretes que tengan grabaciones activas” se puede ver que los artistas se encuentran en “artists” y como tenemos “start_date” y “end_date” requeridos si end_date no tiene fecha todavía es que la grabación sigue activa. Para el caso de uso “Para cada uno de ellos se podrá acceder a los temas que están grabando, quienes son los autores y con qué formación de músicos se está realizando la grabación” se puede ver en “ordered_by” donde se refleja la relación de composición mencionada anteriormente entre grabaciones, artistas y personas.

Por último para este caso de uso “ Para los artistas que no estén grabando nada nuevo en este momento, se localizará si existen grabaciones antiguas que no se hayan incluido en ningún álbum y que puedan constituir un álbum de “rarezas”, o los temas que más se han interpretado en sus conciertos para plantear un nuevo álbum en vivo” podemos ver que si el artista (en el campo “artists”) tiene todas las grabaciones con end_date, es decir que todas están finalizadas está libre. A su vez, podemos ver mediante las referencias con punteros de grabación \longleftrightarrow pista de audio \longleftrightarrow album cuales de las grabaciones están en los álbumes y cuáles no, y por tanto estarían disponibles para el álbum de rarezas o álbum en vivo.

La única excepción, es el atributo de “studio_recording” que puede ser nulo ya que puede haber sido una grabación de concierto live y por tanto no se necesita de un estudio de grabación.

En cuanto al resto de campos los iremos explicando uno a uno:

- **st ISRC:** hace referencia al código de identificación universal que deben de poseer todas las grabaciones. Este código de 12 caracteres, debe de seguir el siguiente formato:
 - Los dos primeros dígitos son el código del país según ISO 3166-1-Alpha-2.
 - Los tres siguientes son un código de registrante alfanumérico de tres caracteres.
 - Los dos siguientes son los últimos dos dígitos del año de asignación del ISRC.
 - Los cinco últimos son una identificación de la grabación, única en el año de asignación.

Todo esto lo hemos resumido en el siguiente regex `'^[A-Z]{2}-[A-Z0-9]{3}-d{2}-d{5}$'` que se encuentra en el campo de pattern para que cumpla con el estándar.

- **start_date**: hace referencia a la fecha en la que se inicia la grabación y por ello es un timestamp.
- **end_date**: hace referencia a la fecha en la que se finaliza la grabación y por ello es un timestamp.
- **name**: hace referencia al nombre de la grabación y por ello es un string.
- **duration**: hace referencia a la duración de la grabación y por ello es un integer.
- **audio_tracks**: hace referencia a las pistas de audio que contienen la grabación. Esto es un array (de las pistas de audio del álbum) que debe contener ítems del tipo objectId (es decir el Id de cada pista de audio) de esta manera estamos haciendo referencias a ellas como si fuera un puntero.
- **artists**: hace referencia a los artistas que están grabando. Esto es un array (de los artistas) que debe contener ítems del tipo objectId (es decir el Id de cada artista) de esta manera estamos haciendo referencias a ellos como si fuera un puntero.
- **studio_recording**: hace referencia al estudio de grabación en el que se ha grabado la grabación. Por ello tiene que ser un string, haciendo mención al nombre del estudio.
- **ordered_by**: hace referencia a composición mencionada anteriormente entre grabaciones, artistas y personas. Es decir va a contener el id del artista (objectId) que solicitó la grabación (a nombre artístico de quien queda guardada esa grabación), una referencia también mediante objectId a la composición de la grabación y un arrays de las personas que han participado en la grabación. Este último debe ser un array que como mínimo tenga una persona involucrada en la grabación y que sea el id de la persona (objectId).

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'Recordings Object Validation',
    required: [
      'st_ISRC',
      'start_date',
      'end_date',
      'name',
```

```

        'duration',
        'audio_tracks',
        'artists',
        'ordered_by'
    ],
    properties: {
        st_ISRC: {
            bsonType: 'string',
            pattern: '^[A-Z]{2}-[A-Z0-9]{3}-d{2}-d{5}$',
            description: '\st_ISRC\ must be a string and is
required'
        },
        start_date: {
            bsonType: 'timestamp',
            description: '\start_date\ must be a timestamp and is
required'
        },
        end_date: {
            bsonType: 'timestamp',
            description: '\end_date\ must be a timestamp and is
required'
        },
        name: {
            bsonType: 'string',
            description: '\name\ must be a string and is required'
        },
        duration: {
            bsonType: 'int',
            description: '\duration\ must be an integer and is
required'
        },
        audio_tracks: {
            bsonType: 'array',
            minItems: 1,
            items: {
                bsonType: 'objectId'
            },
            description: '\audio_tracks\ must be an array and is
required'
        },
        artists: {

```


arquitectura y el identificador secuencial que el enunciado indica que deben de tener las pistas de audio.

En cuanto al resto de campos los iremos explicando uno a uno:

- **id**: hace referencia al identificador de la pista de audio que a su vez tiene el requisito de ser secuencial según el enunciado, es por ello que es un integer.
- **recordings**: hace referencia a las grabaciones contenidas en la pista de audio. Esto es un array (de las grabaciones) que debe contener ítems del tipo objectId (es decir el Id de grabación) de esta manera estamos haciendo referencias a ellas como si fuera un puntero.
- **album**: hace referencia al álbum/es al que pertenecen las pistas de audio. Esto es un array (de los álbumes) que debe contener ítems del tipo objectId (es decir el Id de cada álbum) de esta manera estamos haciendo referencias a ellos como si fuera un puntero.

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'Audio tracks Object Validation',
    required: [
      'id',
      'recordings',
      'album'
    ],
    properties: {
      id: {
        bsonType: 'int',
        description: '\'id\' must be an array and is required'
      },
      recordings: {
        bsonType: 'array',
        minItems: 1,
        items: {
          bsonType: 'objectId'
        },
        description: '\'recordings\' must be an array and is
required'
      },
      album: {
        bsonType: 'array',
        minItems: 1,
```

```

    items: {
      bsonType: 'objectId'
    },
    description: '\album\' must be an array and is required'
  }
}
}
}

```

Esquema de Validación person

En este esquema los campos obligatorios son: “passport”, “complete_name”, “birth_date”, “date_of_death” y “type_person”. De nuevo, consideramos que estos son los campos esenciales para describir a una persona.

Para continuar, vamos a explicar cada uno de los campos por separado de forma detallada:

- **passport:** representa el pasaporte de la persona, por ello tiene que ser un dato de tipo string.
- **complete_name:** representa el nombre y el/los apellido de la persona a la que se refiere. De nuevo, este debe ser un dato de tipo string.
- **birth_date:** representa la fecha de nacimiento de la persona; por ello, el tipo de dato es un timestamp representando en formato DD/MM/AA.
- **date_of_death:** representa la fecha de fallecimiento de una persona. Hemos determinado que debido a que una persona puede estar viva, el valor del campo deberá ser un timestamp con formato DD/MM/AA si la persona ha fallecido; o un valor vacío en caso contrario.
- **type_person:** representa el tipo de persona que puede ser una persona siendo posible tres tipos. Una persona puede ser un manager, un ingeniero de sonido o un músico. Además, lo hemos establecido de tal manera que una misma persona pueda tener tres roles. Por ejemplo, una persona puede ser manager, ingeniero de sonido y músico simultáneamente; o ingeniero de sonido y manager y el resto de combinaciones posible.

Satisfaciendo el enunciado de la práctica, hemos determinado que el campo de “sound_engineer” y “manager” debe ser required puesto que la existencia de al menos un ingeniero de sonido y exclusivamente un manager será necesaria para llevar a cabo una grabación live. Por tanto, como la existencia del “musician” no es relevante para hacer una grabación live, no hemos puesto este campo como required porque tanto si está como si no está nos da un poco igual en este caso.

Además, para una más fácil representación de las facetas que tiene cada persona, hemos determinado que los datos serán de tipo booleano; indicando con True que la persona es por ejemplo manager y con False que no lo es. Hemos decidido que sea de esta manera pues será, a nuestro parecer, más fácil de identificar su papel.

Debemos mencionar en este campo encastrado, llevar a cabo otro encastramiento con “live_records”. Sin embargo, como preferimos la facilidad de lectura en nuestro sistema, hemos decidido que “live_records” será una clase independiente. Con ello, conectaremos la clase de “persons” a la de “live_records” por medio de referencias.

Por último, hemos creado un campo dentro de “type_person” denominado “live_records” que será un campo que contenga un array con todas las grabaciones live que ha realizado dicha persona. Como es lógico cada grabación live estará referenciada a la clase de “live_records” por medio del ObjectId. No hemos hecho este campo required porque puede existir el caso en el que una persona sea músico, técnico de sonido o manager que nunca haya realizado una grabación live. Por esta razón, no tiene porque existir este campo de forma obligatoria dentro de “type_person”. Por ello, si una persona no tiene una grabación live asociada, dicho campo será un array vacío. En caso contrario, será un array de uno o más ObjectId referenciados a la clase “live_records”.

- **country:** muestra el/los países a los que pertenece esa persona. Por tanto, como una persona puede tener hasta dos nacionalidades, deberá ser representado en el documento de cada colección con un array que contenga como mínimo un país y como máximo dos países. Cada país al que pertenece deberá ser un dato de tipo string.
- **recordings:** este campo lo vamos a emplear para indicar la relación entre la clase “person” y la clase “recordings”. Para ello, vamos a realizarlo a través de un array de ObjectId de la clase “recordings”. De esta forma, vamos a poder determinar la siguiente relación que es que 1..n personas pueden participar en 0..n grabaciones; por ello, el mínimo de Items de este array es 0 puesto que una persona no tiene porque tener una grabación asociada. En caso de no tener grabaciones, el array será vacío; en caso contrario, estará formado por las grabaciones asociadas a esta persona.

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: 'Detailed description of the Person ',
    required: [
      'passport',
      'complete_name',
```

```

        'birth_date',
        'date_of_death',
        'type_person'
    ],
    properties: {
        passport: {
            bsonType: 'string',
            pattern: '^[A-Z]{2}>>\d+$',
            description: '\''passport\' must be a string and is
required'

        },
        complete_name: {
            bsonType: 'string',
            description: '\''complete_name\' must be a string and is
required'

        },
        birth_date: {
            bsonType: 'timestamp',
            description: '\''birth_date\' must be a timestamp and is
required'

        },
        date_of_death: {
            bsonType: 'timestamp',
            description: '\''date_of_death\' must be a timestamp if the
field exists and is required'

        },
        type_person: {
            bsonType: 'object',
            title: 'Type of person able to create a Live Record',
            required: [
                'manager',
                'sound_engineer'
            ],
            properties: {
                manager: {
                    bsonType: 'bool'
                    description: '\''manager\' must be a boolean and is
required'

                },
                sound_engineer: {

```


Esquema de Validación live_records

Esta clase tiene como objetivo como hemos mencionado previamente evitar un doble encastramiento de la clase anterior para así garantizar lecturas más rápidas. Debido a ello, este esquema está formado por los siguientes campos “recordings”, “concerts”, “person”. Los tres campos son imprescindibles para poder determinar de forma correcta la información correcta correspondiente a las grabaciones live.

Por todo lo anterior, los campos del esquema descritos de una forma más detallada son los siguientes:

- **recordings:** este campo tiene como objetivo referenciar la clase “live_records” a la clase “recordings”. De esta forma, conseguimos representar la representación del UML donde indicamos una relación de herencia de grabación con respecto a la grabación live. La forma de referenciar, va a ser a través del ObjectId de la clase de grabación.
- **concerts:** este campo tiene como objetivo fijar la relación que hemos establecido entre la clase de “live_records” y la clase de “concerts” establecida en el UML. En esta relación se muestra que 0 a n grabaciones live son grabadas en 1 concierto y 1 concierto tiene 0..n grabaciones live. Por ello, lo que debemos mostrar en el esquema es que cada grabación live debe estar asociada a un concierto; o en este caso a un ObjectId de la clase concierto. Esto lo representamos haciendo que el número mínimo de ObjectId sea 1.
- **person:** este campo tiene como objetivo relacionar los valores de la clase “persons” con la clase “live_records”. En este último campo del esquema, establecemos la siguiente relación expresada en el diagrama de clases de UML. La relación consiste en que 1 manager, de 1..n ingenieros de sonido y de 0..n músicos graban de 0..n grabaciones live; y 0..n grabaciones live son grabadas por 1 manager, de 1..n ingenieros de sonido y de 0..n músicos. Por tanto, si existe una grabación live tiene que tener asociado como mínimo de forma indispensable 1 manager y 1 ingeniero. Esto lo representamos en el esquema a través de un array de objectId donde cada uno de ellos representa a un participante de la grabación live. Además, se tiene que cumplir que como mínimo haya 2 objectId en el array, siendo uno para el manager y otro para el ingeniero de sonido.

```
{
  $jsonSchema: {
    bsonType: 'object',
    title: ' Collection focused in live records ',
    required: [
      'recordings',
      'concerts',
      'person'
    ],
  },
}
```

```

properties: {
  recordings: {
    bsonType: 'objectId',
    description: '\recordings\' must be an objectId and is
required'
  },
  concerts: {
    bsonType: 'objectId',
    description: '\concerts\' must be an objectId and is
required'
  },
  person: {
    bsonType: 'array',
    minItems: 1,
    items: {
      bsonType: 'objectId'
    },
    description: '\person\' must be an array and is required'
  }
}
}
}

```

3.- Pipelines MongoDB

Antes de proceder a explicar el proceso de migración de los datos argumentando los pipelines de agregación creados y utilizados para el propósito mencionado anteriormente así como las anomalías encontradas durante el proceso y como se han resuelto, explicaremos la limpieza de datos que hemos realizado previa a la migración.

3.1.- Limpieza de datos

Antes de realizar la carga de los datos en bruto de los backups lógicos en las tres colecciones solicitadas por el enunciado, hemos realizado un proceso exhaustivo de limpieza de los datos antiguos proporcionados en la práctica. Para ello, como en el enunciado no se especifica ninguna herramienta especial para realizar este proceso hemos optado por hacer la limpieza en Python mediante el uso de la librería “pandas”. La justificación de esta decisión radica en que el uso de la herramienta mencionada anteriormente nos permite automatizar el proceso de limpieza y corregir los 25 csv en una sola ejecución, además de que al cliente al que le estamos proporcionando el nuevo sistema de música le va a ser indiferente con que se haga el trabajo mientras se haga y a ser posible en el menor tiempo permitido.

Primeramente, previo a escribir el código en Python por el que vamos a realizar la limpieza, hemos decidido mirar exhaustivamente a mano mediante Excel los datos con los que estábamos tratando mirando las columnas y las filas con el fin de tener un conocimiento más profundo sobre ellos y ver cuales son los errores que podríamos identificar para posteriormente arreglarlos. Se podría decir que de todo el tiempo que nos llevó el proceso de limpieza de los datos este paso fue el que más tiempo nos consumió.

Una vez entendido los datos e identificados los problemas que encontramos en los mismos procedimos a automatizar la limpieza de los datos (como se pueden ver en el fichero adjuntado como “conciertos.ipynb”, “interpretes.ipynb”, “temas.ipynb”). Los cambios más relevantes que hemos hecho son los citados a continuación:

- **Cambio 1:** hemos observado que debido a la inserción de los datos en los csv, había columnas que se encontraban desplazadas a la derecha; es decir, no se encontraba en la columna a la que verdaderamente pertenecen. Un ejemplo concreto sería el caso de “\Tanzania”. En este caso, no tenía una coma al final de la palabra, lo que provocó en ese csv que las columnas se desplazarán a la derecha destruyendo la estructura de los datos. Este fallo, lo solucionamos uniendo los strings que se quedaban correctos en la columna de lugar y lo unimos a “Tanzania” quitándole la barra invertida. De esta forma conseguimos corregir el desacople de columnas.
- **Cambio 2:** también nos hemos dado cuenta que algunos países conformados por más de una palabra estaban en el formato siguiente: “Republica_Dominicana”. Como no nos gustaba que algunos países tuvieran barra baja y otros no, nos hemos decantado por eliminar “_” de aquellos países que lo tuviesen.
- **Cambio 3:** en los csv, nos hemos dado cuenta también de que los símbolos “especiales” como por ejemplo tildes españolas, francesas o incluso símbolos nórdicos, han sido cambiados por “?”. Un ejemplo sería el siguiente: “cami?n” en vez de camión. Pensando inocentemente que las palabras del primer csv se iban a repetir en gran medida en el resto, fuimos sustituyendo en cada palabra que vimos interrogación por la palabra correcta sin el símbolo especial; es decir, pusimos “camion” en el ejemplo anterior.

Debido a que una vez que implementamos el mismo código para todos los csv, vimos que en el resto de csv existían muchas palabras que nos faltaban porque eran nuevas. Por ello, decidimos que en el resto de csv, eliminar el símbolo de interrogación “?” y unir lo que quedase de palabra en caso de quedar. Por ejemplo, siguiendo con el ejemplo del camión, la palabra se quedará en el csv como “camin”. Por tanto, para que quede claro, cambiamos uno a uno en el primer csv y luego en el resto de csv optamos por eliminar el símbolo de “?”.

Cabe resaltar que estos cambios los hemos realizado en todos los csv.

- **Cambio 4:** en el csv de intérpretes, nos dimos cuenta que la columna de rol, venía un poco mezclada con palabras en inglés (ej: percussion), en español (ej: percusion) o incluso en mayúsculas (ej: SOLIST). Para tener los datos ordenados, decidimos

que íbamos a cambiar todos los datos a minúsculas y en español; y así lo hicimos. Por ello, en los dos ejemplos anteriores, convertimos las palabras en “percussion” y “solist”, respectivamente.

3.2.- Migración de datos (pipelines de agregación) y anomalías

Para la migración de datos usamos los pipelines de agregación que se incluyen en MongoDB. A continuación vemos lo que realiza cada parte del pipeline:

```
const fs = require('fs');
fs.writeFileSync("./mongo.log", "") logToFile = function(data) {
  fs.appendFileSync("./mongo.log", data + '\n',
  function(err) {
    if (err) {
      return console.log(err);
    }

    console.log("Logfile saved");
  });
}

logToFile("-----STARTING NEW MIGRATION AT " + Date().toString() + "-----" + "\n\n")

use('Prueba') logToFile("Using Prueba as database")

db.persons.drop() db.createCollection('persons')

db.artists.drop() db.createCollection('artists') db.getCollection('artists').createIndex({
  "artistic_name.name": 1,
  "artistic_name.country": 1,
},
{
  unique: true
});
db.getCollection('artists').createIndex({
  "artistic_name.name": 1,
},
{
  unique: true
});

db.compositions.drop() db.createCollection('compositions')
db.getCollection('compositions').createIndex({
  "name": 1,
  "titulo": 1
},
{
  unique: true
});

db.concerts.drop() db.createCollection('concerts')
```

```

db.getCollection('concerts').createIndex({
  "city": 1,
  "country": 1,
  "duration": 1,
  "start_date": 1
},
{
  unique: true
});

logToFile("Finished dropping the collections: persons, artists, concerts ✓")

```

En esta primera fase escribimos una función de log personalizada para llevar la cuenta del tiempo de ejecución; además de eliminar las colecciones antiguas y crear nuevos índices.

```

// Solistas que no matchean (11 documentos)
db.getCollection('interpretes').updateMany({
  $expr: {
    $and: [{
      $eq: ["$rol", "Solista"]
    },
    {
      $ne: ["$interprete_o_banda", "$miembro"]
    }
  ]
},
{
  $set: {
    interprete_o_banda: "$miembro"
  }
});

```

Limpiamos aquellas traducciones incorrectas de la colección de intérpretes.

```

logToFile("Started inserting from temas to persons.") start = Date.now()
db.temas.aggregate([
  $group: {
    _id: "$pasaporte_autor",
    complete_name: {
      $first: "$autor"
    }
  }
},
{
  $set: {
    country: {

```

```

        $substrBytes: ["$_id", 0, 2]
    }
}
},
{
    $merge: {
        into: 'persons',
        whenMatched: 'merge',
        whenNotMatched: 'insert'
    }
}
});
logToFile("Finished inserting from temas to persons. Took " + (Date.now() - start) / 1000 +
" seconds. ✓")

```

Comenzamos con el primer pipeline a persons. El primer paso se trata de agrupar por pasaporte, que será el id de la colección. Cogemos el primer nombre encontrado. Terminamos uniéndolo a la colección, *mergeando* las colecciones con `_id` coincidente.

```

logToFile("Started inserting from interpretes to persons.") start = Date.now()
db.interpretes.aggregate([
    $group: {
        _id: "$pasaporte",
        complete_name: {
            $first: "$miembro"
        },
        birth_date: {
            $first: "$fecha_nacimiento"
        }
    }
},
{
    $set: {
        country: {
            $substrBytes: ["$_id", 0, 2]
        },
        birth_date: {
            $dateFromString: {
                dateString: "$birth_date",
                format: "%d/%m/%Y",
                onError: "$$REMOVE",
                onNull: "$$REMOVE"
            }
        }
    }
},
{
    $merge: {
        into: 'persons',
        whenMatched: 'merge',
        whenNotMatched: 'insert'
    }
}
]);

```

```
logToFile("Finished inserting from interpretes to persons. Took " + (Date.now() - start) / 1000 + " seconds. ✓")
```

Continuamos la inserción desde los intérpretes a persons. De la misma forma agrupamos por `_id` e insertamos hacia persons. Así encontramos nuevos campos que forman parte de persona (esencialmente el año de nacimiento)

```
logToFile("Started inserting from conciertos to persons.") start = Date.now()
db.conciertos.aggregate([
  $group: {
    _id: "$pasaporte_autor",
    complete_name: {
      $first: "$autor"
    }
  },
  {
    $set: {
      country: {
        $substrBytes: ["$_id", 0, 2]
      }
    }
  },
  {
    $merge: {
      into: 'persons',
      whenMatched: 'merge',
      whenNotMatched: 'insert'
    }
  }
]);
logToFile("Finished inserting from conciertos to persons. Took " + (Date.now() - start) / 1000 + " seconds. ✓")
```

Por último insertamos de conciertos a personas. Aquí no encontramos ninguna persona nueva pero lo considerábamos necesario.

En todas los pipelines hacia persona se ha usado el pasaporte para extraer el país de origen.

Este es un ejemplo de un documento de persons quedaría de la forma:

```
{
  "_id": "CA>>0020224844",
  "complete_name": "Isidro Berrospi",
  "country": "CA",
  "birth_date": {
    "$date": {
      "$numberLong": "-385257600000"
    }
  }
}
```

```

// Add to artists
logToFile("Started inserting from interpretes to artists (only Solists).") start = Date.now()

db.interpretes.aggregate([
  $match: {
    $expr: {
      $eq: ["$interprete_o_banda", "$miembro"]
    }
  },
  {
    $project: {
      start_date: {
        $dateFromString: {
          "dateString": "$incorporacion",
          "format": "%d/%m/%Y",
          "onError": "$$REMOVE",
          "onNull": "$$REMOVE"
        }
      },
      end_date: {
        $dateFromString: {
          "dateString": "$cese",
          "format": "%d/%m/%Y",
          "onError": "$$REMOVE",
          "onNull": "$$REMOVE"
        }
      },
      artistic_name: {
        name: "$interprete_o_banda",
        country: "$nacionalidad_registro"
      },
      musician: {
        personId: "$pasaporte"
      },
      _id: 0
    }
  },
  // Get all roles that a Solist has participated throught their carreer
  {
    $lookup: {
      from: "interpretes",
      localField: "musician.personId",
      foreignField: "pasaporte",
      pipeline: [{
        $project: {
          "_id": 0,
          "rol": 1
        }
      }],
      as: "musician.roles"
    }
  },
  // Transform from array of objects to array of strings

```

```

{
  $set: {
    "musician.roles": {
      $reduce: {
        "input": "$musician.roles",
        "initialValue": [],
        "in": {
          $concatArrays: ["$$value", ["$$this.rol"]]
        }
      }
    }
  }
},
{
  $lookup: {
    from: "temas",
    localField: "musician.personId",
    foreignField: "pasaporte_autor",
    as: "musician.isWriter"
  }
},
{
  $set: {
    "musician.roles": {
      $cond: [{
        $gt: [{
          $size: "$musician.isWriter"
        },
        0]
      },
      {
        $concatArrays: ["$musician.roles", ["Writer"]]
      },
      "$musician.roles"
    ]
  }
},
{
  $unset: ["musician.isWriter"]
},
{
  $merge: {
    into: "artists",
    on: ["artistic_name.name", "artistic_name.country"],
    whenMatched: "merge",
    whenNotMatched: "insert"
  }
}
});
logToFile("Finished inserting from interpretes to artists (only Solists). Took " + (Date.now()
- start) / 1000 + " seconds. ✅")

```

En la segunda parte de la migración, encontramos los pipelines hacia artists. En este primer pipeline insertamos todos los solistas (cuando interprete_o_banda es igual a miembro). Hacemos el merge usando el nombre del artista y el país en el que está registrado.

```
logToFile("Started inserting from temas to artists (only Writers).") start = Date.now()

db.temas.aggregate([
  $set: {
    musician: {
      personId: "$pasaporte_autor",
      roles: ["Writer"]
    },
    artistic_name: {
      name: "$autor",
    }
  }
],
{
  $unset: ["_id", "titulo", "pasaporte_autor", "autor"]
},
{
  $merge: {
    into: "artists",
    on: "artistic_name.name",
    whenMatched: "merge",
    whenNotMatched: "insert"
  }
}
]);
logToFile("Finished inserting from temas to artists (only Writers). Took " + (Date.now() - start) / 1000 + " seconds. ✅")
```

Seguimos con la inserción de temas a artistas. Usamos el nombre del autor en temas para asignar el role de escritor (Write) a aquellos artistas que han escrito una canción.

```
logToFile("Started inserting from interpretes to artists (only Bands).") start = Date.now()

db.interpretes.aggregate([
  $match: {
    $expr: {
      $and: [{
        $ne: ["$interprete_o_banda", "$miembro"]
      },
      {
        $ne: ["$rol", "Solista"]
      }
    ]
  }
},
{
  $group: {
    _id: "$interprete_o_banda",
```

```

nacionalidad_registro: {
  $first: "$nacionalidad_registro"
},
members: {
  $addToSet: {
    pasaporte: "$pasaporte",
    start_date: {
      $cond: [{
        $eq: [{
          $strLenCP: "$incorporacion"
        }],
        10]
      },
      {
        $dateFromString: {
          "dateString": "$incorporacion",
          "format": "%d/%m/%Y"
        }
      },
      {
        $literal: null
      }
    ]
  },
  end_date: {
    $cond: [{
      $eq: [{
        $strLenCP: "$cese"
      }],
      10]
    },
    {
      $dateFromString: {
        "dateString": "$cese",
        "format": "%d/%m/%Y"
      }
    },
    {
      $literal: null
    }
  ]
},
  roles: "$rol"
}
}
},
{
  $set: {
    type_grouping: "Band",
    artistic_name: {
      name: "$_id",
      country: "$nacionalidad_registro"
    }
  }
},

```



```
{
  $unset: ["_id", "nacionalidad_registro"]
},
{
  $merge: {
    into: "artists",
    on: ["artistic_name.name", "artistic_name.country"],
    whenMatched: "merge",
    whenNotMatched: "insert"
  }
}
});

logToFile("Finished inserting from interpretes to artists (only Bands). Took " + (Date.now() -
start) / 1000 + " seconds. ✅")
```

En el último pipeline de agregación hacia artistas, insertamos las bandas. Primeros buscamos aquellos con nombre de intérprete y miembro distinto, agrupamos por nombre de la banda y a través de *lookups* rellenamos las referencias a *artists* para que la banda quede relacionada con los artistas que son o han sido miembros de la banda. Volvemos a insertar por nombre y por país.

```
logToFile("Started inserting from temas to compositions.") start = Date.now()
// temas to compositions
db.temas.aggregate([
  $set: {
    "name": "$autor",
    "createdBy": "$titulo"
  }
},
{
  $unset: ["_id"]
},
{
  $lookup: {
    from: "artists",
    localField: "pasaporte_autor",
    foreignField: "musician.personId",
    pipeline: [{
      $project: {
        "_id": 1
      }
    }],
    as: "createdById"
  }
},
{
  $set: {
    "createdBy": {
      $first: "$createdById._id"
    },
    "pasaporte_titulo": {
```

```

        $concat: ["$pasaporte_autor", "$titulo"]
    }
}
},
{
    $unset: ["createdById", "autor"]
},
{
    $merge: {
        into: "compositions",
        on: ["titulo", "name"],
        whenMatched: "merge",
        whenNotMatched: "insert"
    }
}
});
logToFile("Finished inserting from temas to compositions. Took " + (Date.now() - start) /
1000 + " seconds. ✅")

```

En este pipeline se inserta de temas a composiciones. A través de un *lookup* sacamos a los artistas que han escrito dicho tema (están ya añadidos con el rol de *Writer*). Insertamos por título y nombre del autor. Como nota el campo `pasaporte_titulo` se usa para una optimización posterior, al hacer la operación de *lookup* significativamente más rápida.

```

logToFile("Started inserting from conciertos to concerts.") start = Date.now()

db.conciertos.aggregate([
    $limit: 100000000
},
{
    $lookup: {
        from: "artists",
        localField: "interprete",
        foreignField: "artistic_name.name",
        pipeline: [{
            $project: {
                "_id": 1,
            }
        }],
        as: "result"
    },
},
{
    $set: {
        "interprete": {
            $first: "$result._id"
        },
        "pasaporte_tema": {
            $concat: ["$pasaporte_autor", "$tema"]
        }
    }
},
{
}

```

```

$group: {
  _id: {
    artistId: "$interprete",
    tour: "$gira",
    start_date: "$fecha",
    city: "$lugar",
    country: "$pais",
    duration: "$duration"
  },
  compositions: {
    $push: "$pasaporte_tema"
  },
  duration: {
    $first: "$duracion_total"
  }
}
},
{
  $addFields: {
    type_interpretation: "Principal"
  }
},
{
  $lookup: {
    from: "compositions",
    localField: "compositions",
    foreignField: "pasaporte_titulo",
    pipeline: [{
      $project: {
        "compositionId": "$_id",
        "_id": 0
      }
    }],
    as: "compositions"
  }
},
{
  $group: {
    _id: {
      country: "$_id.country",
      city: "$_id.city",
      duration: "$duration",
      start_date: "$_id.start_date",
    },
    artists_played: {
      $push: {
        artistId: "$_id.artistId",
        tour: "$_id.tour",
        type_interpretation: "$type_interpretation",
        compositions: "$compositions"
      }
    }
  }
},
},

```

```

{
  $set: {
    start_date: {
      $dateFromString: {
        "dateString": "$_id.start_date",
        "format": "%d/%m/%Y",
        "onError": "$_id.start_date",
        "onNull": "$$REMOVE"
      }
    },
    city: "$_id.city",
    country: "$_id.country",
    duration: {
      $toInt: "$_id.duration"
    }
  }
},
{
  {
    $unset: "_id"
  },
  {
    $out: "concerts"
  }
}], {
  allowDiskUse: true,
  maxTimeMS: 50400000 // 12h
});

logToFile("Finished inserting from conciertos to concerts. Took " + (Date.now() - start) /
1000 + " seconds. ✅")

```

En este último pipeline de agregación es sobre el que hemos trabajado más. Primero tuvimos que limitar los documentos a 10 millones, por la inestabilidad de la conexión con el clúster Picasso. Respecto a la operación, primero agrupamos por todos los campos excepto el tema que se tocó. Esto nos da un array sobre el que hacemos un *lookup* contra compositions, para poder sacar su `_id`. Luego sacamos el `_id` de artista y por último agrupamos para sacar un array con los artistas que interpretaron. Por último mencionar las optimizaciones de permitir el uso de disco para optimizaciones intermedias y el máximo de tiempo de operación, dadas las limitaciones del servidor.

```

logToFile("Started cleaning from compositions.") start = Date.now()
db.compositions.aggregate([
  $unset: ["pasaporte_autor", "pasaporte_titulo"]
],
{
  $out: "compositions"
});
logToFile("Finished cleaning from compositions. Took " + (Date.now() - start) / 1000 + "
seconds. ✅")

logToFile("Started creating recordings.") start = Date.now() db.compositions.aggregate([
  $unset: '_id' // Generates new Id

```

```

},
{
  $out: 'recordings'
})
logToFile("Finished creating recordings. Took " + (Date.now() - start) / 1000 + " seconds.
✓")

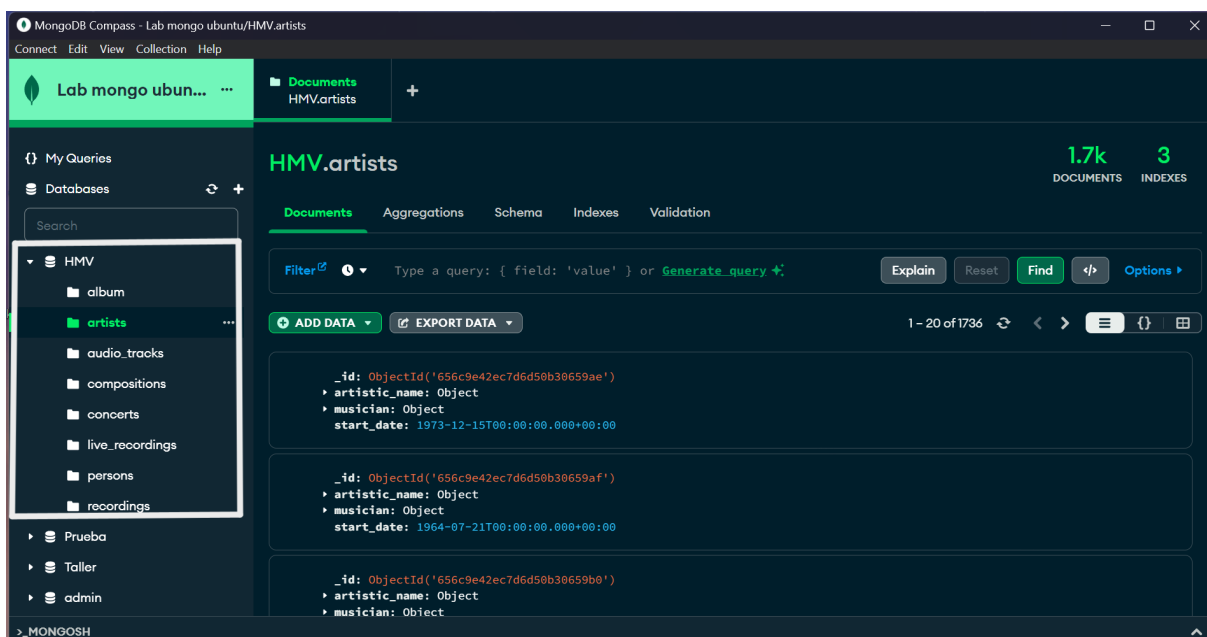
```

Por último suponemos que todas las composiciones han sido grabadas, por lo que copiamos directamente.

Entre los diferentes obstáculos o anomalías que hemos encontrado en la creación de los pipelines, podemos destacar los siguientes:

- En primer lugar, el tamaño de la colección de conciertos es demasiado grande para la máquina.
- En segundo lugar, nos ha resultado bastante problemático aquellos campos nulos o con “\N”.
- Por último, ha sido también problemática la lentitud de los lookups para montar los arrays con referencias.
- Problemas de conexión con la VPN, siempre había un corte de conexión y la operación que se estaba realizando se cancelaba.

NOTA: La base de datos con los datos migrados en las nuevas colecciones, más aquellas colecciones que no tienen datos (porque antes no existían los casos pedidos) están en la base de datos denominada HMV. Las colecciones tienen los esquemas de validación explicados en el apartado anterior.



4.- Cluster

En este apartado, vamos a explicar cómo haríamos un cluster respecto al contexto planteado en la práctica.

Para gestionar los datos vamos a implementar la técnica de partición sabiendo que se centra en dividir la base de datos en fragmentos que son manejados por diferentes nodos lo que distribuye la carga de trabajo entre los nodos y así hay un mejor rendimiento además de la escalabilidad horizontal que ofrece. No obstante, hay que tener cuidado a la hora de escoger el número de particiones ya que hay que mantener la consistencia entre los datos. Además hay que tener en cuenta que es un proceso delicado ya que no es reversible o al menos muy costoso para poder permitírselo, luego solo se aplicaría esta técnica cuando se alcancen los límites de memoria, ocupación en el disco y capacidad computacional que explicaremos más adelante a la hora de decir en cuantos shards hemos decidido fragmentar la base de datos.

A su vez, lo vamos a mezclar con la técnica de replicación una vez se haya producido la partición. La replicación es la copia y almacenamiento de los datos en varios nodos. Esto se lo hemos escogido dado que al estar los datos en muchas ubicaciones si un nodo falla se pueden obtener los datos de las réplicas y por tanto hay mucha tolerancia ante los fallos ya que se pueden aplicar estrategias de recuperación. Sin embargo, hay que tener cuidado al igual que en la partición para que la consistencia se mantenga entre las réplicas y a su vez el almacenamiento del que disponemos ya que este incrementará significativamente al realizar las réplicas.

Conviene remarcar que la distribución de los fragmentos la vamos a realizar de la forma más homogénea posible empleando el procedimiento de “basado en rangos”. Como sabemos de esta manera se divide el dominio del campo en rangos disjuntos y se asigna un rango a cada shard. Esto lo empleamos en vez del procedimiento de “basada en hash” porque usamos el valor de un campo que sabemos que tiene máximo y mínimo.

- **Elección de la clave de particionamiento (Shard Key)**: en este primer apartado debemos establecer una clave de particionamiento o shard key. Como ya sabemos, la shard key es un criterio que debemos especificar nosotros sobre cuál es el shard o fragmento idóneo en el que debemos guardar cada conjunto de información. Hacer una buena elección de esta clave de particionamiento, nos permitirá obtener un mejor rendimiento, eficiencia y escalabilidad en el cluster encontrado.

De esta manera es imprescindible que analicemos cual es la carga de trabajo a la que está sometido el clúster (que se verá más adelante en el apartado de routers de mongo) y determinar cuál es el elemento que queremos optimizar. En nuestro caso queremos optimizar el balance entre la memoria y el disco de los nodos ya que queremos que esté todo bien distribuido pero teniendo datos similares cerca para evitar el rebalanceo.

Hay que tener en cuenta que a la hora de elegir una clave de fragmentación hay tres situaciones habituales: claves ascendentes, distribución aleatoria y basada en la localización. Al principio pensamos que era una buena idea utilizar claves

ascendentes como la fecha del concierto o del artista para las colecciones. Sin embargo, descartamos esta idea rápidamente ya que las nuevas incorporaciones de los datos que se hagan (sobre este año en adelante) irían siempre al mismo shard y no conseguiríamos las necesidades mencionadas anteriormente. Por ello, decidimos optar por la distribución aleatoria que según hemos visto se produce cuando los campos que forman la clave de fragmentación tienen una distribución de probabilidad uniforme. Con esto nos aseguramos de que no vaya siempre al mismo shard y estar rebalanceando todo el rato.

Teniendo en cuenta lo anteriormente mencionado, vamos a describir los campos que forman la clave de fragmentación para cada colección (una clave por colección de la nueva arquitectura, siendo las nuevas colecciones las mencionadas en apartados anteriores):

- “concerts”: en esta colección hemos elegido el campo país como shard key ya que al ser el valor de este campo “aleatorio” va a seguir una distribución aleatoria dentro de la colección; lo cual va a ser beneficioso. Además, esto nos permite encontrar todos los conciertos que se han producido en el mismo país; lo que nos va a permitir tener en el mismo shard todas las canciones que se hayan tocado en un concierto concreto en el país. Esto va a ser bueno en términos de organización dentro del sistema puesto que la localidad de los datos va a ser próxima y al emplear el campo país como shard key, va a permitir que se agrupen todos los conciertos que se toquen en un mismo país. De esta manera, quedan los países con nombres más cortos en los primeros shards y los más largos en los otros. Así, conseguimos una distribución más o menos aleatoria y al estar todo en el mismo país la búsqueda es más eficiente.
- “artists”: en esta colección, vamos a emplear el nombre artístico del artista como shard key (es un campo dentro del objeto, se puede acceder como : artistic_name.name). Esto lo hemos decidido de esta manera porque es un campo que es único y representativo del artista. Además, hemos elegido también como shard key el nombre artístico, porque también tiene cierta aleatoriedad y por tanto, los documentos dentro de la colección podrán alcanzar de una mejor manera una distribución aleatoria.
- “compositions”: en esta colección, decidimos que el campo “autor” sea shard key porque hemos establecido que un autor genera una composición. De esta manera, este campo nos permite tener aleatoriedad (porque el nombre del autor raramente será el mismo). Además, nos permite tener composiciones del mismo autor “cerca”. Esto significa que la información relacionada se encuentra dentro del mismo shard, lo que facilita el orden al almacenar documentos y gestionarlos.

- “persons”: hemos decidido que la shard key de esta colección va a ser el campo “passport”. De nuevo, este campo es único para cada persona y le va a dar aleatoriedad a los datos, lo que permitirá alcanzar una distribución aleatoria de los mismos. De la misma manera que en shard keys previos, hemos escogido un campo que sea único de la persona en este caso para que permita que las personas del mismo país se encuentren en el mismo shard.
- “recordings”: en esta colección hemos decidido que el mejor shard key va a ser el nombre de la grabación (“name”). De nuevo el valor de este campo tiene cierta aleatoriedad (es complicado que dos grabaciones se llamen igual). Además, nos va a permitir de nuevo tener “cerca” aquellas canciones que tengan asociado el mismo nombre de grabación.
- “audio_tracks”: esta colección tendrá como shard key el identificador secuencial de la pista de audio. La idea es que vamos a hashear con md5 este identificador con el fin de equi distribuirlo entre los campos.
- “album”: el shard key de esta colección va a ser el campo “producers”. Hemos escogido este campo porque solo puede haber una discográfica asociada a un álbum. De nuevo, el nombre de una discográfica es lo suficientemente aleatorio y nos permite tener los álbumes de una misma discográfica en un mismo shard.
- “live_records”: en esta colección, hemos decidido que la shard key sea el id del concierto; ya que como mencionamos un live_recording está asociado a un único concierto. Además, nos permite tener todas los live_recordings de un mismo concierto en el mismo shard.
- **Creación de índices con la shard key**: es indispensable la creación de índices en función de las shard keys que hemos mencionado previamente puesto que nos van a permitir buscar y acceder a los datos de una manera más eficiente (en lo referente a operaciones de lectura y escritura).

Conviene mencionar que crearemos índices en todos los shards. Además hemos decidido realizar los siguientes índices para mejorar la eficiencia de las operaciones:

- Concerts: crearemos una key compuesta en la colección de concerts que está formada por los campos de “país” y el “artistId”. De esta manera podemos ver que artistas han tocado en un país determinado.

- Artists: buscaremos por nombre artístico (“artistic_name”) y por tanto utilizamos un índice simple ya que estamos empleando un solo campo.
- Compositions: en este caso hemos pensado en un índice compuesto ya que estaría formado por el nombre de la composición (“name”) y la persona que lo ha compuesto, “created_by”.
- Persons: para esta colección también sería un índice compuesto para los campos de pasaporte, “passport” y nombre de la persona, “complete_name”.
- Audio_tracks: en este caso sería un índice simple ya que solo estamos empleando un único campo, que es la misma clave con la que hemos creado el shard.
- Live_recording: también emplearíamos un índice simple, que sería la misma clave que el id.
- Recordings: en este caso hemos pensado en un índice compuesto ya que estaría formado por el nombre de la grabación (“name”) y los artistas que están grabando , “artists”.
- **Determinación del número de enrutadores mongos (instancias de MongoDB).**

Este tercer punto es bastante importante puesto que escoger un número correcto de enrutadores nos va a permitir eliminar los siguientes problemas:

1. Si el número de enrutadores no es el adecuado; es decir, hemos determinado menos de los que deberíamos, se puede producir el problema de cuello de botella con las consultas. Este problema implicaría que el rendimiento del sistema empeora. Además, lo que queremos es evitar a toda costa el problema anterior; puesto que lo que queremos buscar es lo contrario, la eficiencia del sistema mediante un número adecuado de instancias que enruten correctamente todas las consultas.
2. Relacionado con el anterior, problemas de escalabilidad (que afectan después al rendimiento del sistema), en función del tráfico de consultas o la topología del cluster según vaya creciendo el sistema. Este problema se resuelve con una buena elección del número de enrutadores y haciendo un buen uso de la escalabilidad horizontal.
3. Un mal cálculo en el número de enrutadores puede derivar en un posterior mal equilibrio o balanceo de la carga del sistema.

Debido a los anteriores problemas mencionados y muchos otros, consideramos que este paso es crucial al realizar la fragmentación.

Por tanto, para evitar los problemas descritos, hemos considerado que el número de enrutadores correctos es de 7.

Hemos decidido situar estratégicamente los enrutadores en cada continente poniendo como máximo 2 en función de la población; es decir, si un continente tiene una elevada población o un potencial de uso elevado, situaremos dos enrutadores en ese continente para evitar problemas de cuello de botella entre otros. Además, esto se ha decidido así porque agregar más enrutadores puede ser beneficioso para distribuir la carga y mejorar el rendimiento, especialmente en entornos con un gran número de solicitudes concurrentes. y también hay más de uno para garantizar alta disponibilidad.

Por ello, los enrutadores serán los siguientes:

- | | |
|-------------------|----------------------|
| - Norteamérica: 1 | - Asia Oriental: 1 |
| - Sudamérica: 1 | - Asia Occidental: 1 |
| - Europa: 1 | - Oceanía: 1 |
| - África: 1 | |

Por ello, como se puede observar, situaremos 2 enrutadores tanto en América como en Asia y 1 enrutador en el resto de continentes. Además, hemos decidido que no situaremos enrutadores para la Antártica, porque no creemos que tengamos usuarios en esa zona.

- **Determinar la cantidad de shards**: en este apartado, vamos a determinar el número de particiones o shards que vamos a realizar en nuestro sistema. Debido a la importancia de este punto, nos hemos basado en cuatro parámetros para definir la cantidad de shards correcta para el sistema. Estos parámetros son:

A partir de tres shards la mejora comienza a compensar el overhead

1. **Volumen de datos y proyección de crecimiento**: observando nuestra base de datos, hemos determinado que nuestro volumen actual es de 6 GB. De la misma manera, con respecto a la proyección de crecimiento, hemos hecho un estudio tanto de la colección conciertos como la de artistas para que se pueda observar en un periodo anual el posible crecimiento de estas colecciones. Además, lo hemos realizado sobre concerts porque es una de las más pesadas (en cuanto a tamaño) en nuestro caso, y es la que mayor proyección o cambios puede llegar a experimentar.

Concerts: en primer lugar, hemos determinado que un documento de media en esta colección pesa 1.12 kB. En segundo lugar, hemos buscado el número de conciertos “importantes” que se hacen de manera anual en todo el mundo (unos 20.000 conciertos). De esta manera, la agregación de estos conciertos a la colección supondrá un aumento del tamaño de la colección “concerts” en 22 MB.

Artists: para el cálculo de la proyección de esta colección, hemos asumido que el crecimiento de los artistas va a crecer en proporción al crecimiento de la población mundial de forma anual que actualmente se encuentra en el 0.8%. Además, hemos buscado el número total de artistas que hay en el mundo actualmente (25 millones de artistas). De esta forma, hemos calculado el incremento del 0.8%. Nuestro razonamiento es que eventualmente la colección de artistas contendrá a la mayoría de los artistas. Cada documento de la colección de la colección artistas tiene un tamaño de media 170 bytes. Suponiendo que todos los artistas mundiales actuales están en nuestra base de datos, pesará la colección de artistas un total de 4GB. Aplicando a esto el crecimiento del 0.8% anual en equivalencia con el incremento de la población mundial actual, se incrementa el tamaño de la colección en 32 MB al año. La operación que hemos llevado a cabo para obtener estos valores ha sido: $(170\text{bytes} * 25 \text{ Millones de documentos}) / (1024^2) = 4\text{GB}$; y luego: $4 \text{ GB} * 1.008 = 4.032$. Concluyendo en un incremento de 32 MB.

2. Carga de trabajo y patrones de acceso: en este apartado entendemos carga de trabajo como las diferentes consultas tanto de lectura como de escritura que debe soportar el sistema en un periodo de tiempo en específico. Es por ello, que con respecto a este parámetro nuestra carga de trabajo se encuentra muy orientada hacia la lectura en toda la base de datos, especialmente en la colección de artistas y conciertos. Con respecto a los patrones de acceso, lo asociamos a la forma en que los datos son utilizados en términos de manipulación y acceso a los mismos. En lo referente a esta parte, consideramos que los patrones de acceso típicos en nuestro sistema será la consulta de una colección filtrada con un lookup para enriquecer los datos de salida. Por ejemplo para obtener información extra de la colección “artists”, se podría añadir la fecha de nacimiento de los miembros.
3. Capacidad de Hardware: en este campo debemos incluir las capacidades del sistema en cuanto a “hierro” en lo referente a CPU, memoria o capacidad de almacenamiento. Hemos podido observar que las características actuales de nuestro sistema son escasas (evidencia de ello ha sido la dificultad y los cuelgues durante la migración), En el futuro, las capacidades pueden llegar a ser las siguientes:
 - a. Máquinas multinúcleo (Xeon 64 cores) de propósito general con al menos 128GB de RAM, 512GB SSD en RAID 4. Buscamos escalabilidad horizontal.
 - b. Máquinas replicadas en cada continente para aumentar la disponibilidad.

4. Rendimiento y Latencia: por último, relacionado con todo lo anterior, debemos saber cual es el número óptimo de shards que necesitamos para garantizar el mejor rendimiento del sistema. De la misma manera, conocer la latencia del sistema también será crucial para la selección de este número óptimo. Observamos que dado que posiblemente miles de usuarios van a acceder a los datos de conciertos simultáneamente. Consideramos que para una experiencia de usuario fluida como máximo puede tardar 100 milisegundos en dar la respuesta a la query de conciertos activos de una ciudad.

Considerando todo lo anterior en conjunto, consideramos que el número ideal de shards es 7. Aplicando una regla empírica, este número se debe a que necesitamos al menos tres veces el tamaño de nuestro mayor índice. En nuestro caso, el mayor índice es el de conciertos que tiene un tamaño de 900MB. Por ello, cada shard deberá tener 900 MB. Por tanto, obtendremos el número total de shards que necesitamos dividiendo el tamaño total de todos los documentos (6GB) entre el tamaño de cada shard (900MB). Realizando esta operación, obtenemos un resultado de 6.7 shards, que redondeando, lo dejamos en 7. Por último, nos gustaría resaltar que este cálculo, como hemos mencionado previamente, lo hemos realizado a ojo.

- **Replicación y número de réplicas**: en este apartado vamos a mencionar dos cuestiones. La primera de ellas es la estrategia de replicación que hemos empleado para nuestro sistema teniendo en cuenta tanto la ubicación geográfica de los nodos que se replican para determinar el número de nodos. La segunda hace referencia al nivel de preocupación de las escrituras (write concerns) y las operaciones de lectura (read concerns).

Por ello, analizaremos cada cuestión por separado:

- Ubicación geográfica: este factor es muy importante pues nos va a permitir en un futuro, en caso de desastre, poder recuperar los datos que se hubiesen perdido.

Hay que tener en cuenta que el número mínimo de nodos para conformar un conjunto de réplicas es de tres, ya que en caso de fallo en un nodo primario, el proceso de elección se activa entre los nodos restantes para encontrar el sustituto. Si solo fueran dos nodos no habría mayoría de votación y el nodo primario quedaría inactivo debido a la falta de selección de uno nuevo.

Teniendo en cuenta lo anteriormente mencionado, debido a que tenemos 5 continentes, en cuanto a los aspectos geográficos, en cada continente deberíamos tener:

- 6 nodos regulares (que son los que tienen los datos y son secundarios y primarios), en caso de que pueda haber más de un fallo.

- 1 nodo árbitro (para que se permita el proceso de selección y se escoja un nodo primario en caso de fallo), ya que participa en la votación pero no puede ser un nodo primario.
- 1 nodo retrasado (ya que está por detrás de los demás nodos, para los desastres), tampoco puede ser nodo primario y no participa en la votación, este solo esta para aspectos relacionados con los desastres.



Por ello, la suma total de estos nodos serían de 8 por continente (cada conjunto de réplicas), siendo en total 40 nodos.

- Write concerns: para el aspecto de los procesos de escrituras y reconocimiento de las mismas, hay que tener en cuenta que en la configuración por defecto van siempre dirigidas al nodo primario. Esto es que solo cuando el nodo primario reconoce los inserts y updates se da por exitosa la operación. Sin embargo, esto se puede configurar según unos parámetros para que vaya de acuerdo a nuestras preferencias. De entre todos los parámetros (0, 1, majority, y n) hemos pensado que el que más se adecua a nuestras necesidades es majority. Este parámetro establece que las operaciones de escritura se devuelven exitosas si la mayoría de los nodos del conjunto reconocen las operaciones, a diferencia del anterior que era solo el nodo primario. De esta manera nos aseguramos más fielmente que las operaciones de escritura hayan sido exitosas de verdad, y que si justo en el momento de una operación de escritura el nodo primario haya muerto y no se haya descubierto todavía o se esté en el proceso de votación esa operación se realice de manera exitosa independientemente de la situación.
- Read concerns: en cuanto a las lecturas y las preferencias de las mismas, por defecto mongoDB envía todas las operaciones de lectura al nodo primario para proporcionar una consistencia entre los datos escritos y leídos. No obstante, como en la escritura esto se puede modificar pudiendo seleccionarse entre los parámetros de primary, primaryPreferred, secondary, secondaryPreferred y nearest. Según nuestras necesidades hemos seleccionado el primaryPreferred que la diferencia con el de por defecto es que en caso de que el nodo primario no exista o no esté disponible en el momento las operaciones de lectura van a los nodos secundarios. De esta manera garantizamos más disponibilidad.
- **Balanceo de carga automático**: un buen balanceo de carga nos va a permitir de nuevo mejorar el rendimiento del sistema y evita los problemas de cuello de botella. El balanceo como sabemos es una función autónoma, que nos permite distribuir los chunks hacia los distintos shards que tenemos.

Es importante mencionar que aunque este balanceo sea autónomo, podemos configurarlo de tal manera que podamos adecuar su función a nuestras necesidades en el sistema. Algunos ejemplos de dicha configuración son:

- **Horarios:** el balanceo se deberá hacer teniendo en cuenta las horas valle de cada continente para poder así establecer un horario donde esta acción no afecte a los usuarios. Para determinar este horario, vamos a ver las horas valles en promedio de cada continente:
 - **Africa:** 10pm - 6 am
 - **America:** 11pm - 7 am
 - **Europa:** 12 pm - 8 am
 - **Asia:** 12 pm - 8 am
 - **Oceania:** 11pm - 7 am

Por tanto, viendo estos horarios, asumimos que estas son las horas donde habrá menos tráfico dentro del sistema y que por tanto, a nosotros nos vendrá bien para hacer el balanceo. Como se puede ver, el balanceo de datos, nos vendrá bien establecerlo entre las 5 y las 6 de la mañana en la zona UTC. En Beijing a las 13 es la hora del almuerzo, lo que pensamos que dará menos actividad para la zona asiática.

 Los Angeles, CA, USA <small>PST (UTC -8)</small>	dom, 3 de dic de 2023	21:00 😊 ⋮
 Madrid, Spain <small>CET (UTC +1) 9 hour(s) ahead</small>	lun, 4 de dic de 2023	6:00 😞 ⋮
 Beijing, China <small>CST (UTC +8) 16 hour(s) ahead</small>	lun, 4 de dic de 2023	13:00 😊 ⋮

- **Tamaño de cada chunk:** Para poder tratar esta cuestión, nos hemos fijado en el tamaño máximo de un documento que es da como resultado en nuestro caso 16MB. Con esto en mente se va ampliando según el tamaño máximo de un documento.

En cuanto a los aspectos de mantenimiento vamos a enumerar que implementaríamos y como:

- **Estrategia de backup:** para el respaldo y recuperación de los datos en caso de que haya una pérdida de los mismo hemos decidido que ya está implementada al realizar replicación. Ya que si falla el nodo primario, el nodo secundario (el resto de nodos) actúan como nodos de respaldo pudiendo responder a operaciones de lectura. Además al aplicar la replicación sabemos que el nodo primario es el único punto de fallo de la arquitectura.

Por otro lado, si nos fijamos en cuanto a la tolerancia ante fallo de se podría aplicar mediante el algoritmo de RAFT partiendo del uso de la replicación. Sabiendo que los nodos pueden tener uno de los tres roles (líder, candidato y seguidor), el algoritmo

muy resumidamente se basa en que tras vencer el tiempo del temporizador de inactividad del nodo líder, se le da por muerto. El nodo que detecta la muerte del líder (candidato) se presenta a sí mismo como líder en el proceso de votación y la marca de ciclo term se incrementa, aquellos nodos con un id de ciclo menor que este se convierten automáticamente en seguidores. Luego se puede decir que la tolerancia a fallos en el líder se logra mediante la elección continua y el uso de temporizadores para mantener la consistencia de los datos entre réplicas.

- **Seguridad:** Para que cumpla con los aspectos legales del GDPR y en sí la seguridad de los datos capando la lectura de los datos proponemos el uso del atributo "\$redact" en un pipeline que restringe la salida de documentos enteros o del contenido de documentos basándose en la información almacenada en los propios documentos. Pudiendo decidir entre las variables de:
 - **\$\$descend:** que devuelve los campos en el nivel del documento excluyendo aquellos que están embebidos.
 - **\$\$prune:** excluye todos los campos en el nivel del documento o documento embebido. Esto se aplica incluso si el campo excluido contiene documentos embebidos que pueden tener diferentes niveles de acceso.
 - **\$\$keep:** devuelve o mantiene todos los campos en el nivel del documento. Esto se aplica incluso si el campo incluido contiene documentos embebidos que pueden tener diferentes niveles de acceso.

Se podría utilizar por ejemplo para una visualización sencilla de lo que queremos implementar (que no la implementación verdadera sobre la base de datos) sería excluir todos los campos de un nivel determinado. Por ejemplo, que el pasaporte que es información delicada se pueda usar para las operaciones internas necesarias (como queries) pero que no se permita su visualización. Supongamos que el campo de pasaporte se encuentra en el nivel 2, pues quedaría de la siguiente manera (muy simplificado):

```
db.persons.aggregate (
  [
    { $redact: {
      $cond: {
        if: { $eq: [ "$level", 2 ] },
        then: "$$PRUNE",
        else: "$$DESCEND"
      }
    }
  ]
);
```

5.- Conclusión

Este trabajo nos ha servido para tener una mayor comprensión y conocimiento en la gestión de bases de datos noSQL, aplicando los conocimientos teóricos adquiridos a lo largo de la asignatura y los nuevos descubiertos para poder crear la mejor arquitectura con respecto al enunciado propuesto. De esta manera hemos visto de primera mano como MongoDB es una herramienta bastante interesante y fuera de nuestro “espacio de confort” que resulta bastante potente para abordar los obstáculos que pueden acarrear un gran volumen de datos y estructuras de datos dinámicas.

Asimismo, nos ha sorprendido, sobretodo en comparación a lo que estábamos acostumbrados a tratar que es el mundo SQL, la flexibilidad que tiene MongoDB en lo referente a los esquemas y su enfoque en la escalabilidad horizontal ya que aunque esto se nos mencionó en clase hasta que no se prueba de primera mano no se aprecia la capacidad que tiene para escalar de manera eficiente en entornos distribuidos. Esto último es lo que creemos que hace MongoDB tan atractivo hoy en día a los programadores y las aplicaciones modernas y ágiles con las que estamos en contacto en la actualidad.

Conforme íbamos avanzando en la práctica hemos aplicado conceptos clave del modelado de datos en MongoDB como la optimización de consultas mediante índices (empleadas en los pipelines) y el uso de la herramienta de MongoDB Compass para una mejor visualización de nuestro progreso en la práctica. A su vez, estamos orgullosos de haber descubierto la extensión de Visual Studio de MongoDB, lo que nos ha permitido avanzar más holgadamente en los pipelines con la infinidad de facilidades que ofrece la extensión. Un ejemplo de esto, es el mongo playground que nos ha permitido refinar los pipelines ya que así podíamos probar consultas, insertar datos y explorar funcionalidades de MongoDB sin necesidad de probarlo en el entorno local.

Conviene subrayar que nos ha dado un poco de rabia el no poder realizar la migración de datos completa ya que por problemas fuera de nuestro control (tras varios intentos en diferentes días no hemos podido realizar la migración de conciertos en su totalidad por los fallos de conexión de la VPN debido al pobre funcionamiento de GlobalProtect).

En síntesis, se puede decir que hemos fortalecido nuestras habilidades técnicas específicas en el manejo de MongoDB y en los conocimientos requeridos para ello. Se puede decir con firmeza que estas competencias son fundamentales en nuestro desarrollo como profesionales del sector en especial en la gestión de datos moderna.

6.- Bibliografía

Aggregation Stages — MongoDB Manual. (s. f.).

<https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/>

BSON Types — MongoDB Manual. (s. f.).

<https://www.mongodb.com/docs/manual/reference/bson-types/>

Create an aggregation pipeline — MongoDB Compass. (s. f.).

<https://www.mongodb.com/docs/compass/current/create-agg-pipeline/>

Concertful - find concerts near you! (2024, 1 enero). Concertful.

https://concertful.com/?from=2024-01-01&category=&order=event_date

Cuvit, A. P. (2021, 12 marzo). *Lenguaje UML. La importancia de modelar* (página 2).

Monografias.com.

<https://www.monografias.com/trabajos82/lenguaje-uml-importancia-modelar/lenguaje-uml-importancia-modelar2>

Davis, A. (2021, 4 octubre). *What is an ISWC? | Exploration.* Exploration.

<https://exploration.io/what-is-an-iswc/#>

Done, P. (2023). *Practical MongoDB Aggregations: The Official Guide to Developing Optimal Aggregation Pipelines with MongoDB 7.0.*

Import and export Data — MongoDB Compass. (s. f.).

<https://www.mongodb.com/docs/compass/current/import-export/>

\$JsonSchema — MongoDB Manual. (s. f.).

<https://www.mongodb.com/docs/v5.3/reference/operator/query/jsonSchema/#std-label-jsonSchema-keywords>

Karlsson, J. (2022, 31 mayo). *MongoDB Schema Design Best Practices | MongoDB.*

<https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/>

Matt. (2023, 7 junio). *Optimice la migración de datos en MongoDB: técnicas de fragmentación para velocidad y escalabilidad.* HackerNoon.

<https://hackernoon.com/es/optimizar-migracion-de-datos-en-mongodb-resharding-tecnicas-para-velocidad-y-escalabilidad>

Mongo Playground. (s. f.). <https://mongoplayground.net/>

MongoDB. (s. f.). *MongoDB Aggregation Pipeline.*

<https://www.mongodb.com/basics/aggregation-pipeline>

\$Redact (Aggregation) — MongoDB Manual. (s. f.).

<https://www.mongodb.com/docs/manual/reference/operator/aggregation/redact/#:~:text=%24redact%20returns%20the%20fields%20at.access%20for%20these%20embedded%20documents>.

Replicación y particionado (sharding) en MongoDB. - IABD. (s. f.).

<https://aitor-medrano.github.io/iabd2223/sa/06replicacion.html#conjunto-de-replicas>

Rootstack. (s. f.). *Replicación en MongoDB.* Rootstack.

<https://rootstack.com/es/blog/replicacion-en-mongodb>

Shekhawat, S. (2019, 14 junio). *How to Automate database Migrations in MongoDB*.

freeCodeCamp.org.

<https://www.freecodecamp.org/news/how-to-automate-database-migrations-in-mongodb-d6b68efe084e/>

Studio 3T. (2023, 27 febrero). *MongoDB Aggregation: Tutorial with examples and exercises* | Studio 3T.

<https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/#mongodb-lookup>

Specify validation with query operators — MongoDB Manual. (s. f.).

<https://www.mongodb.com/docs/manual/core/schema-validation/specify-query-expression-rules/#std-label-schema-validation-query-expression>

Specify JSON Schema Validation — MongoDB Manual. (s. f.).

<https://www.mongodb.com/docs/manual/core/schema-validation/specify-json-schema/>

Topologías con particiones y réplicas de particiones | NCACHE Docs. (s. f.).

<https://www.alachisoft.com/es/resources/docs/ncache/admin-guide/partitioned-topologies.html#replication-strategies>

Venditto, N. (2022, 15 julio). *Bases de datos distribuidas: Sharding*. DEV Community.

<https://dev.to/anfibiacreativa/bases-de-datos-distribuidas-sharding-4a39>

Write Scripts — MongoDB Shell. (s. f.).

<https://www.mongodb.com/docs/mongodb-shell/write-scripts/>

Wikipedia contributors. (2023, 28 septiembre). *International Standard Recording Code*. Wikipedia.

https://en.wikipedia.org/wiki/International_Standard_Recording_Code#:~:text=

[Format,-ISRC%20Code%20Example&text=ISRC%20codes%20are%20always%2012,make%20them%20easier%20to%20read.](#)