

INFORME BASE DE DATOS PERSONAL

ESTUDIANTE:
IVAN BECERRA

DOCENTE:
BRAYAN ARCOS

INSTITUTO TECNOLOGICO DEL PUTUMAYO
BASE DE DATOS Y ALMACENAMIENTO MASIVO
OCTAVO SEMESTRE
17/10/2024

Tabla de contenido

Resumen Ejecutivo	3
Introducción	4
Contexto y Motivación	4
Alcance del Informe	4
Objetivos	4
Metodología.....	5
Herramientas Utilizadas	5
Procedimientos	5
1. Métodos y Pasos Seguidos para Llevar a Cabo el Análisis o el Trabajo.....	5
Desarrollo del Informe	7
Descripción de la Base de Datos Personal:	7
1. Esquema de la Base de Datos y análisis mediante Studio 3T.....	7
Diseño de la Base de Datos Personal:.....	8
1. Modelo de Datos: Normalización y cardinalidad	8
2. Consideraciones de Diseño:.....	8
base de datos	10
Análisis de esquemas	12
Consultas plataforma streaming:	14
Métodos de captura	18
1.Creación y explicación del procedimiento de las bases de datos apartir de los archivos JSON proporcionados	18
Consultas.....	21
1.Consultas y Explicación Realizadas	21
Análisis y Discusión	50
Conclusiones	51
Recomendaciones	52
Referencias.....	52
Enlace de GitHub.....	52

Resumen Ejecutivo

Este informe documenta el diseño y la implementación de una base de datos MongoDB para una plataforma de streaming. El objetivo principal fue desarrollar una base eficiente que maneje la información de usuarios, contenidos (películas y series), géneros y listas de reproducción. Se utilizaron relaciones uno a uno, uno a muchos y muchos a muchos para optimizar la estructura de la base de datos, asegurando escalabilidad y un manejo eficiente de la información. El análisis abarcó desde la creación del esquema hasta la aplicación de consultas para gestionar los datos.

Introducción

Contexto y Motivación

En un mundo cada vez más digitalizado, las plataformas de streaming representan una fuente clave de entretenimiento. Con el aumento exponencial del contenido y el número de usuarios, la correcta estructuración y gestión de bases de datos se vuelve crucial para garantizar una experiencia de usuario fluida. Este informe se realiza para analizar el diseño de una base de datos en MongoDB que permita manejar de manera eficiente películas, usuarios, géneros y listas de reproducción, abordando tanto la estructura como las relaciones entre los datos.

Alcance del Informe

Este informe cubre la estructura de una base de datos MongoDB, su diseño y las relaciones entre colecciones. Se analiza la implementación de consultas y los procedimientos seguidos para capturar y organizar los datos. Además, se aborda la optimización de las relaciones de uno a uno, uno a muchos y muchos a muchos, junto con el uso de colecciones intermedias para mantener la integridad y escalabilidad de los datos.

Objetivos

- Diseñar una base de datos eficiente para una plataforma de streaming.
- Implementar relaciones entre colecciones usando las mejores prácticas de MongoDB.
- Describir y justificar el uso de colecciones intermedias en relaciones muchos a muchos.
- Realizar consultas para la captura y organización de los datos en la plataforma.

Metodología

Herramientas Utilizadas

- **MongoDB:** Sistema de gestión de bases de datos NoSQL utilizado para el almacenamiento y consulta de datos.
- **Studio 3T:** Herramienta utilizada para visualizar y gestionar la base de datos de forma gráfica, facilitando el análisis del esquema.
- **JSON:** Formato de intercambio de datos utilizado para representar los documentos dentro de las colecciones.

Procedimientos

1. Métodos y Pasos Seguidos para Llevar a Cabo el Análisis o el Trabajo

1.1 Creación de la Base de Datos Personal

1. Definición del Proyecto:

- Se diseñaron varias colecciones para representar diferentes entidades en una plataforma de streaming. Estas colecciones incluían users, contents (para películas y series), genres, subscriptions, playlists, y las relaciones de muchos a muchas necesarias para la estructura de la plataforma (contentGenres y playlistContents).

2. Estructura Inicial:

- Se diseñó la estructura de la base de datos definiendo las colecciones necesarias:
 - **users:** Para almacenar información de los usuarios.
 - **subscriptions:** Para definir los tipos de suscripciones disponibles.
 - **contents:** Para almacenar detalles sobre películas y series.
 - **genres:** Para clasificar el contenido.
 - **playlists:** Para permitir a los usuarios organizar su contenido favorito.

3. Configuración del Entorno:

- Se utilizó **MongoDB** como sistema de gestión de bases de datos NoSQL, con **Studio 3T** para facilitar la creación y gestión de las colecciones.

4. Inserción de Datos:

- Se diseñaron scripts para insertar datos iniciales en las colecciones, asegurando que se cubrieran diferentes casos de uso (ej. diferentes tipos de suscripciones, géneros y contenidos).

1.2. Realización del taller:

Para consolidar los conocimientos y habilidades adquiridos durante el desarrollo del proyecto, se llevó a cabo un taller práctico en el que se utilizaron los datos de la base de datos personal para realizar consultas y operaciones sobre MongoDB.

1. **Simulación del entorno real:**
Se recreó un entorno de streaming utilizando la base de datos diseñada. Esto permitió a los participantes del taller explorar cómo se gestionan los datos en un sistema de esta naturaleza y realizar operaciones como la búsqueda de contenidos por género, la creación de nuevas listas de reproducción y la actualización de suscripciones.
2. **Consultas y análisis de datos:**
Se ejecutaron diversas consultas para analizar la información en las colecciones. Esto incluyó:
 - Obtener la lista de todos los contenidos activos.
 - Buscar películas o series específicas por su nombre.
 - Consultar las listas de reproducción de un usuario determinado y los contenidos asociados a ellas.
 - Obtener la relación entre géneros y contenidos mediante agregaciones.

1.2.1. Métodos de captura:

Para llevar a cabo la creación y análisis de la base de datos, se utilizaron los siguientes métodos de captura:

1. **Carga de datos mediante JSON:**
Los datos de los usuarios, contenidos, géneros, suscripciones y listas de reproducción fueron capturados en formato JSON y cargados en MongoDB utilizando comandos como `insertOne()` y `insertMany()`. Esto permitió la creación de colecciones robustas y bien estructuradas que reflejan un entorno realista de plataforma de streaming.
2. **Captura de datos mediante Studio 3T:**
La herramienta **Studio 3T** permitió capturar los esquemas de las colecciones para su análisis y validación. Esta herramienta facilitó la visualización de los datos y la estructura de la base de datos, permitiendo una comprensión clara de las relaciones entre colecciones y cómo se gestionan los datos en MongoDB.
3. **Comandos de MongoDB para la manipulación de datos:**
Para las actualizaciones y consultas dentro de las colecciones, se utilizaron comandos como `updateOne()`, `upsert`, y operaciones de agregación como `$lookup` para capturar la relación entre contenidos y géneros o entre listas de reproducción y sus contenidos.

Desarrollo del Informe

Descripción de la Base de Datos Personal:

1. Esquema de la Base de Datos y análisis mediante Studio 3T

La base de datos fue diseñada para una plataforma de streaming que almacena y gestiona información relacionada con usuarios, películas, series, géneros, suscripciones y listas de reproducción. El análisis del esquema de la base de datos se realizó utilizando **Studio 3T**, una herramienta que facilita la visualización de las colecciones y sus relaciones.

Esquema General: La base de datos consta de las siguientes colecciones principales:

- **Users:** Almacena la información de los usuarios, como su correo electrónico, contraseñas, y detalles personales.
- **Contents:** Contiene información sobre películas y series, incluyendo título, descripción, duración y si está activa.
- **Genres:** Almacena los géneros asociados a los contenidos (por ejemplo, acción, drama, comedia, etc.).
- **Subscriptions:** Define los tipos de suscripciones disponibles (Básica y Premium).
- **Playlists:** Almacena listas de reproducción personalizadas de cada usuario, las cuales contienen una serie de películas o series.
- **Relaciones de muchos a muchos:** Se creó una colección `contentGenres` para gestionar la relación entre contenidos y géneros (muchos a muchos), y otra colección `playlistContents` para gestionar la relación entre las listas de reproducción y los contenidos.

Análisis del Esquema usando Studio 3T: Mediante el uso de **Schema** en Studio 3T, se pudo identificar la estructura de los documentos en cada colección, analizar su cardinalidad y validar que las relaciones de uno a muchos y de muchos a muchos fueron implementadas correctamente. Esto permitió verificar que el esquema era eficiente y adecuado para consultas futuras.

Diseño de la Base de Datos Personal:

1. Modelo de Datos: Normalización y cardinalidad

El diseño de la base de datos sigue principios de normalización, manteniendo la coherencia y reduciendo la redundancia de datos. Las decisiones clave de diseño se tomaron en función de la **cardinalidad** de las relaciones entre entidades.

- **Normalización:** Se decidió separar ciertas entidades como Genres y Subscriptions en colecciones independientes, en lugar de embeberlas, para evitar la duplicación de datos. Esto permite reutilizar la información de géneros y suscripciones en múltiples documentos y facilita la actualización y el mantenimiento de los mismos.
- **Cardinalidad:**
 - **1 a 1:** La relación entre un **usuario** y su **perfil personal** (almacenado como un subdocumento en users) es una relación de uno a uno, ya que cada usuario tiene un perfil único.
 - **1 a muchos:** La relación entre un **usuario** y sus **listas de reproducción** es de uno a muchos. Un usuario puede tener varias listas de reproducción, pero cada lista pertenece solo a un usuario.
 - **Muchos a muchos:** Las relaciones de **géneros** y **contenidos** así como de **listas de reproducción** y **contenidos** son de muchos a muchos. Un contenido puede tener varios géneros, y un género puede estar asociado a múltiples contenidos. Del mismo modo, un contenido puede estar presente en varias listas de reproducción, y una lista puede contener varios contenidos.

2. Consideraciones de Diseño:

Al momento de definir la estructura y el diseño de la base de datos, se tuvieron en cuenta las siguientes consideraciones clave:

- **Relaciones entre colecciones:**
 - **Usuarios y Suscripciones:** En la colección users, se almacena un campo subscription que indica el tipo de suscripción (Básica o Premium). Esta relación se gestiona mediante una referencia al _id de la colección subscriptions.
 - **Contenidos y Géneros:** En lugar de embeber los géneros directamente en los documentos de contents, se optó por crear una colección intermedia contentGenres para gestionar la relación de muchos a muchos entre los contenidos y los géneros. Esto facilita la adición de nuevos géneros sin modificar los documentos de los contenidos.
 - **Playlists y Contenidos:** De manera similar, las listas de reproducción (playlists) y los contenidos (contents) están relacionados mediante la colección intermedia playlistContents. Esto permite que un contenido esté en múltiples listas sin duplicar información.
- **Definición de campos comunes:**
 - Los campos comunes entre los documentos, como _id, fueron cuidadosamente manejados para mantener la coherencia y facilitar las consultas. Por ejemplo, en la colección contents, cada película o serie tiene

un campo `isActive` para señalar si está disponible en la plataforma, lo que es un campo común que facilita la gestión del catálogo.

- Se estandarizó el uso de campos de fecha y URL en diferentes documentos, como el uso de `createdAt` y `updatedAt` en `users`, `contents`, y `playlists` para mantener un historial preciso de modificaciones.

Embeber o no hacerlo:

En las relaciones uno a uno, se decidió **embeber** los documentos. Por ejemplo, la información personal del usuario (nombre, apellidos, dirección, teléfonos) está embebida dentro del documento de `users`. Esta decisión se tomó para mantener la integridad y rapidez en las consultas de datos relacionados directamente con un usuario.

Por otro lado, para las relaciones de muchos a muchos, **no se embebieron** los documentos, sino que se utilizaron colecciones intermedias (`contentGenres` y `playlistContents`), lo que mejora la escalabilidad y flexibilidad del diseño de la base de datos.

base de datos

desarrollamos la base de datos y cada una de las colecciones con mongo compass

Create Database



Database Name

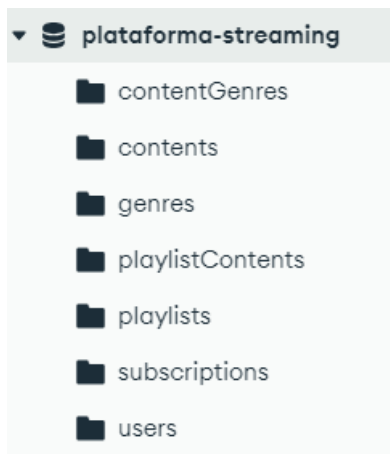
Collection Name

☐ Time-Series

Time-series collections efficiently store sequences of measurements over a period of time. [Learn More](#)

➤ **Additional preferences** (e.g. Custom collation, Capped, Clustered collections)

Colecciones realizadas



Agregamos los datos a cada una de las colecciones

```
use plataforma-streaming;
db.users.insertMany([
  {
    "_id": 1,
    "email": "brayanarcosk@gmail.com",
    "password": "123456",
    "people": {
      "_id": 1,
      "names": "Brayan",
      "firstLastNames": "Arcos",
      "secondLastNames": "Burbano",
      "direction": "Barrio Centro",
      "phones": [314567894, 8909877],
      "photo": "https://example.com/photo1.jpg"
    }
  },
  {
    "_id": 2,
    "email": "andreamaria123@gmail.com",
    "password": "abcdef",
    "people": {
      "_id": 2,
      "names": "Andrea",
      "firstLastNames": "Maria",
      "secondLastNames": "Sanchez",
      "direction": "Avenida Libertad",
      "phones": [320567839, 99871234],
      "photo": "https://example.com/photo2.jpg"
    }
  }
])

use plataforma-streaming;
db.movies.insertMany([
  { "_id": 1, "title": "Inception", "director": "Christopher Nolan", "year": 2010, "genre": "Action" },
  { "_id": 2, "title": "The Matrix", "director": "The Wachowskis", "year": 1999, "genre": "Action" },
  { "_id": 3, "title": "Breaking Bad", "director": "Vince Gilligan", "year": 2013, "genre": "Drama" },
  { "_id": 4, "title": "Stranger Things", "director": "The Dufferin Brothers", "year": 2016, "genre": "Drama" },
  { "_id": 5, "title": "The Crown", "director": "Peter Jackson", "year": 2016, "genre": "Drama" },
  { "_id": 6, "title": "Interstellar", "director": "Christopher Nolan", "year": 2013, "genre": "Action" },
  { "_id": 7, "title": "Parasite", "director": "Bong Joon-ho", "year": 2019, "genre": "Drama" },
  { "_id": 8, "title": "Avengers: Endgame", "director": "Anthony Russo", "year": 2019, "genre": "Action" },
  { "_id": 9, "title": "The Mandalorian", "director": "Jon Favreau", "year": 2019, "genre": "Action" },
  { "_id": 10, "title": "Black Mirror", "director": "Charlie Brooker", "year": 2011, "genre": "Drama" },
  { "_id": 11, "title": "Friends", "director": "David Crane", "year": 1994, "genre": "Comedy" },
  { "_id": 12, "title": "The Office", "director": "Greg Kinnear", "year": 2005, "genre": "Comedy" },
  { "_id": 13, "title": "The Witcher", "director": "Tommy Amiel", "year": 2019, "genre": "Action" }
])
```

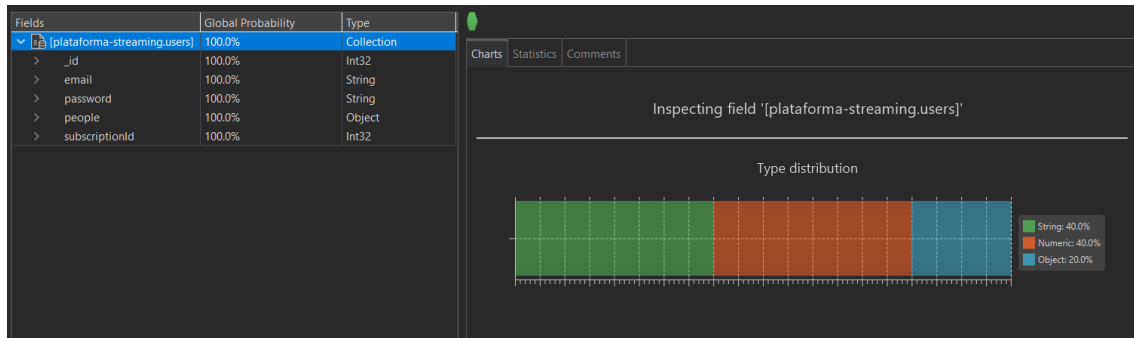
```
use plataforma-streaming;
db.genres.insertMany([
  { "_id": 1, "name": "Action" },
  { "_id": 2, "name": "Drama" },
  { "_id": 3, "name": "Science Fiction" },
  { "_id": 4, "name": "Comedy" }
])
```

```
use plataforma-streaming;
db.subscriptions.insertMany([
  { "_id": 1, "name": "Basic", "price": 9.99, "resolution": "HD" },
  { "_id": 2, "name": "Premium", "price": 14.99, "resolution": "4K" }
])
```

Análisis de esquemas

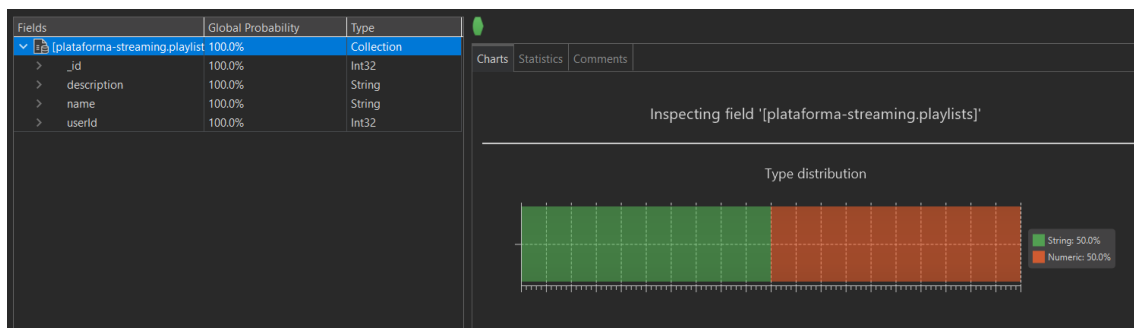
Colección users

La combinación de tipos numéricos y cadenas permite una estructura clara para almacenar la información personal y de acceso de los usuarios.



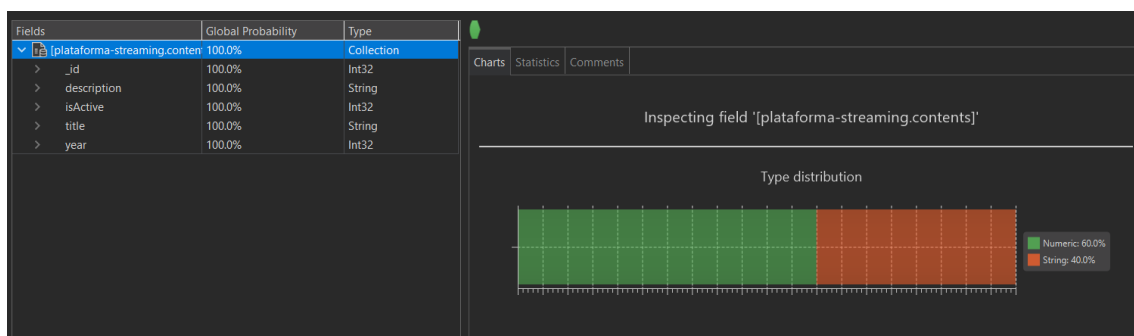
Colección playlists

Esta estructura es eficiente para manejar relaciones entre usuarios y sus listas de reproducción, utilizando principalmente referencias numéricas.



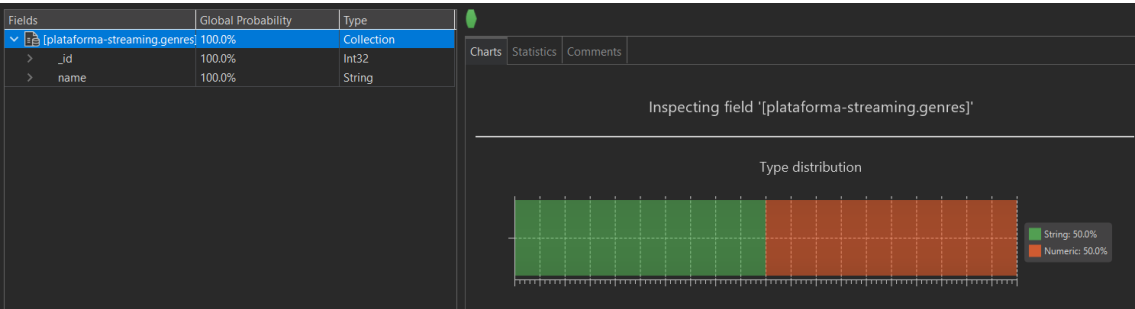
Colección contents

Este esquema es coherente para una colección de contenidos donde los campos clave incluyen un identificador numérico único, descripciones en formato de texto, y un campo para el año en formato numérico. Esto muestra una estructura organizada y eficiente para gestionar los datos de los contenidos multimedia.



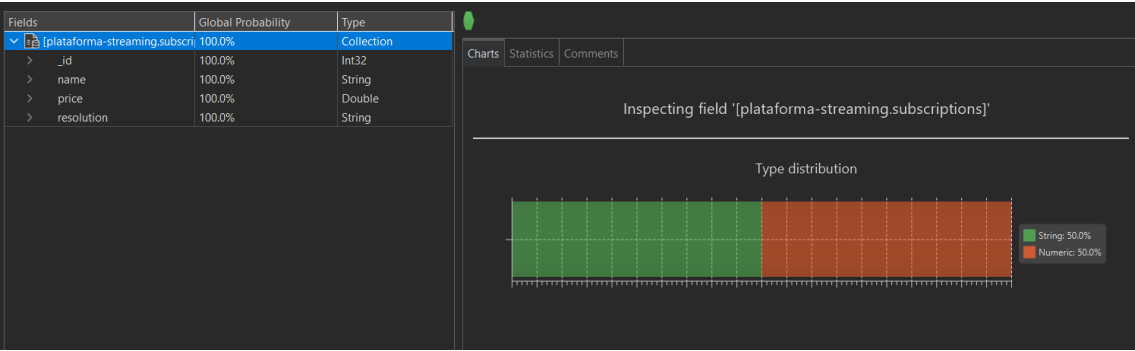
Colección genres

La colección genres está diseñada para almacenar los géneros de las películas o series, con un identificador numérico y un nombre de género textual. Esta estructura simple permite gestionar fácilmente las relaciones muchos a muchos con los contenidos, ya que un género puede asociarse a múltiples películas o series y viceversa.



Colección subscriptions

Esta colección contiene la información básica sobre los tipos de suscripciones que ofrece la plataforma. La inclusión de un precio asociado a cada tipo de suscripción facilita las operaciones relacionadas con la facturación y la gestión de los planes disponibles para los usuarios.



Consultas plataforma streaming:

1. Uso de updateOne() para relaciones uno a uno

La función updateOne() es útil cuando se necesita actualizar un solo documento que cumple con una condición específica. A continuación, se muestra un ejemplo que actualiza el correo electrónico de un usuario en la colección users, donde la relación entre el usuario y su perfil es uno a uno.

```
use plataforma-streaming;
db.users.updateOne(
  { "_id": 1 }, // Coincidencia puntual por el ID del usuario
  { $set: { "email": "nuevo_correo@example.com" } } // Actualización del campo de email
)
```

Explicación:

- La consulta busca al usuario con _id: 1 en la colección users.
- Luego, se actualiza el campo email con un nuevo valor utilizando \$set.

```
▼{
  "_id" : NumberInt(1),
  "email" : "cabrita@example.com",
  "password" : "123456",
  ▼  "people" : {
    "_id" : NumberInt(1),
    "names" : "Brayan",
    "firstLastNames" : "Arcos",
    "secondLastNames" : "Burbano",
    "direction" : "Barrio Centro",
    ▼  "phones" : [
      NumberInt(314567894),
      NumberInt(8909877)
    ],
    "photo" : "https://example.com/photo1.jpg"
  },
  "subscriptionId" : NumberInt(1)
}
```

2. Uso de upsert con setOnInsert

El uso de upsert: true es útil cuando se necesita insertar un documento si no existe, o actualizarlo si ya está presente. Además, con setOnInsert, podemos definir campos específicos que solo se insertarán si el documento es nuevo.

```
use plataforma-streaming;
db.users.updateOne(
  { "_id": 5 }, // Coincidencia por ID, si el usuario no existe, se creará
  {
    $set: { "email": "ejemplo_upsert@example.com", "subscription": "premium" }, // Campos que siempre se actualizarán
    $setOnInsert: { "createdAt": new Date() } // Campo que se agregará solo si el documento es nuevo
  },
  { upsert: true } // Insertar si no existe
)
```

Explicación:

- Si no existe un documento con `_id: 5`, se insertará un nuevo documento.
- En caso de inserción, además de actualizar el correo y la suscripción, se añadirá un campo `createdAt` con la fecha de creación.
- Si el documento ya existía, solo se actualizarán los campos `email` y `subscription`.

```
{
  "_id" : NumberInt(5),
  "createdAt" : ISODate("2024-10-17T15:38:44.197+0000"),
  "email" : "ejemplo_upsert@example.com",
  "subscription" : "premium"
}
```

3. Uso de \$each para actualizar arrays

El operador \$each se utiliza para agregar múltiples elementos a un array en una sola operación. En este ejemplo, se agrega un nuevo contenido a una lista de reproducción existente en la colección playlists.

```
use plataforma-streaming;
db.playlists.updateOne(
  { "_id": 2 }, // Coincidencia puntual por el ID de la playlist
  { $addToSet: { "contents": { $each: [ 10, 12 ] } } } // Agregar múltiples contenidos como enteros
)
```

Explicación:

- Se busca la lista de reproducción con _id: 2.
- Se añaden los IDs de los contenidos content_10 y content_12 al array contents en la lista de reproducción, evitando duplicados gracias al uso de \$addToSet.

```
▼{
  "_id" : NumberInt(1),
  "userId" : NumberInt(1),
  "name" : "Brayan's Favorites",
  "description" : "All-time favorite movies and s
}
▼{
  "_id" : NumberInt(2),
  "userId" : NumberInt(2),
  "name" : "Andrea's Watchlist",
  "description" : "Movies and series to watch",
  ▼
  "contents" : [
    NumberInt(10),
    NumberInt(12)
  ]
}
```


4. Obtener todas las playlists y sus contenidos

Esta consulta encuentra todas las playlists y usa \$lookup para obtener los contenidos relacionados.

```
use plataforma-streaming;
db.playlists.aggregate([
  {
    $lookup: {
      from: "contents",
      localField: "contents",
      foreignField: "_id",
      as: "contentDetails"
    }
  },
  {
    $project: {
      name: 1,
      contentDetails: {
        title: 1,
        description: 1
      }
    }
  }
])
```

Explicación:

- La consulta realiza un \$lookup para combinar datos de la colección playlists con contents.
- Relaciona el campo contents de playlists con el campo _id de contents.
- Utiliza \$project para devolver el nombre de la playlist junto con el título y la descripción de los contenidos asociados.

```
{
  "_id" : NumberInt(3),
  "name" : "John's Collection",
  "contentDetails" : [
  ]
}
{
  "_id" : NumberInt(4),
  "name" : "Maria's Binge List",
  "contentDetails" : [
    {
      "title" : "Friends",
      "description" : "Six friends navigate life and relationships"
    },
    {
      "title" : "Avengers: Endgame",
      "description" : "The Avengers unite to undo Thanos' actions"
    },
    {
      "title" : "Parasite",

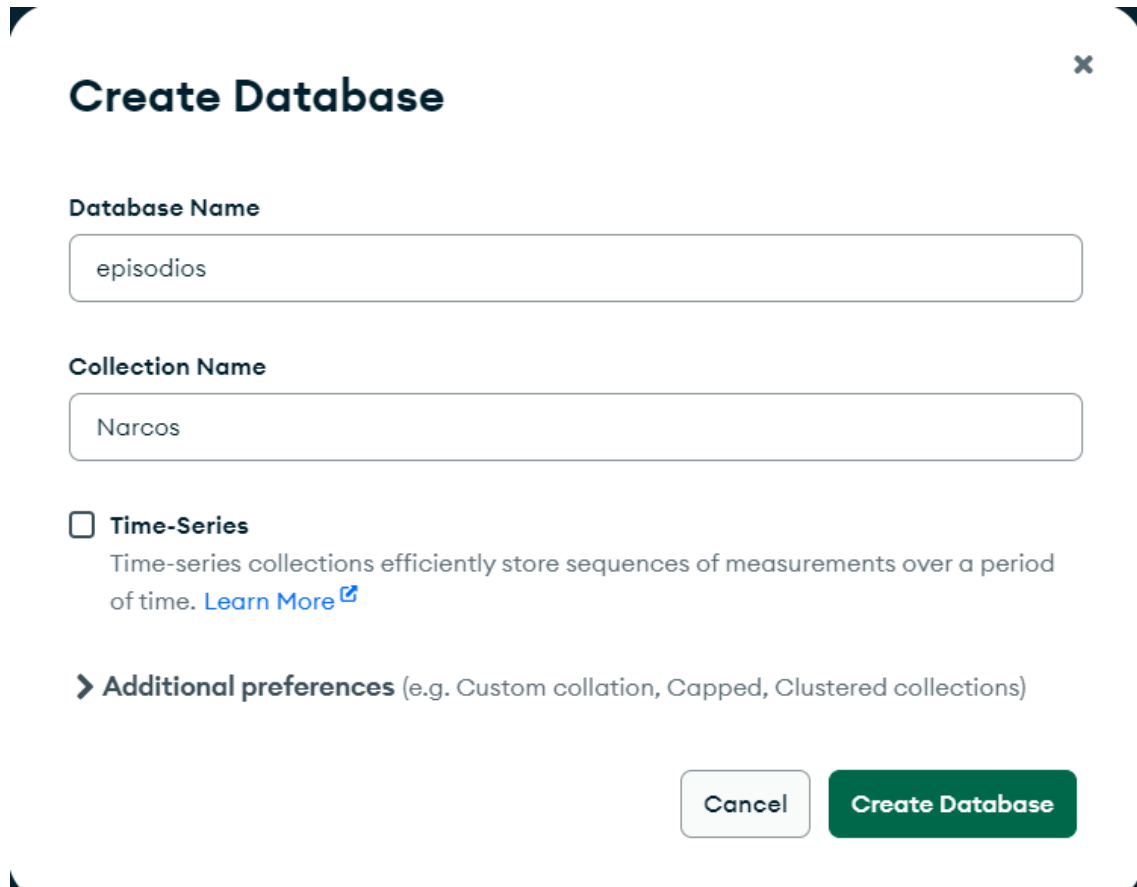
```

Métodos de captura

1.Creación y explicación del procedimiento de las bases de datos apartir de los archivos JSON proporcionados

1) Carga los datos del fichero “**ejercicio_00.json**” en la base de datos llamada **Episodios**, sobre la colección **Narcos**.

Primero creamos la base episodios y la colección narcos



Create Database

Database Name

episodios

Collection Name

Narcos

☐ Time-Series

Time-series collections efficiently store sequences of measurements over a period of time. [Learn More](#)

➤ Additional preferences (e.g. Custom collation, Capped, Clustered collections)

Cancel Create Database

Después importamos el fichero y así tendremos cargados los datos

+ ADD DATA **EXPORT I**

```
_id: ObjectId('67086
id : 203469
url : "https://www.t
name : "Descenso"
season : 1
number : 1
type : "regular"
```

2) Visualiza el contenido del fichero “ **ejercicio_01.json** ” con la herramienta online <https://jsongrid.com/json-viewer>. Observa la estructura del documento y recupera solo la lista de episodios que contienen dicho fichero.

Con la lista de episodios crear otro fichero JSON llamado “episodios_westworld.json”.

Usando mongo cargar dichos ficheros en la colección “westwold” dentro de la base de datos episodios

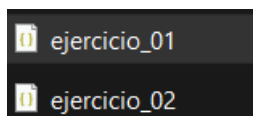
Luego repetir el mismo proceso con los demás ficheros

“ejercicio-02.json” en la colección RickiLake.

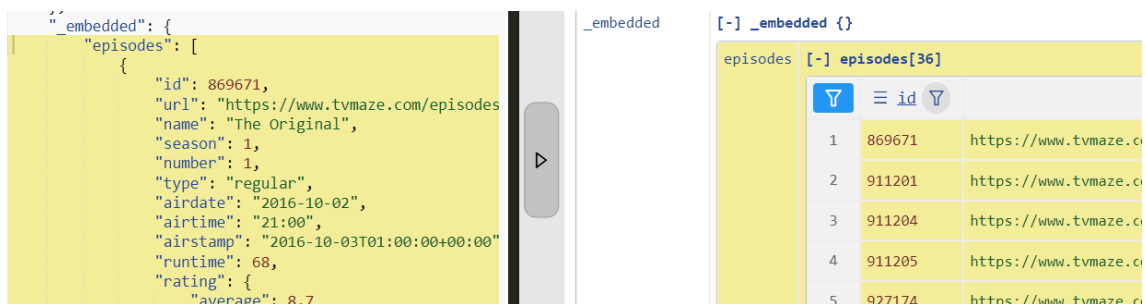
“ejercicio-03.json” en la colección Homeland.

“ejercicio-grads.json” en la base de datos School sobre la colección grades.

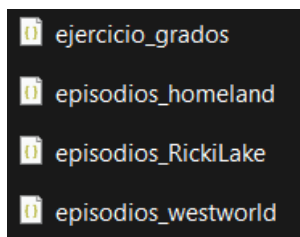
Primero visualizamos los datos de los ficheros



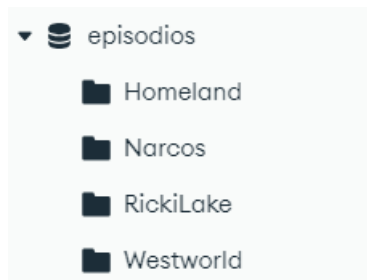
Usando la herramienta jsongrid podremos acceder a los datos y extraerlos



Así crearemos los nuevos archivos json



E importaremos los datos a cada una de las colecciones creadas



Visualizar todas las bases de datos existentes: show dbs

```
31 biblioteca      224.00 KiB
32 config          108.00 KiB
33 ecommerce       624.00 KiB
34 episodios       192.00 KiB
35 local           80.00 KiB
36 school          64.00 KiB
37 test            40.00 KiB
38
```













Consultas

1.Consultas y Explicación Realizadas

TRABAJA CON READ

Localiza la base de datos School, ejecuta las siguientes consultas y describe que hace cada una

db.grades.find({ student_id: 2 })

_id	student_id	class_id	scores
 50b59cd75bed7...	 2	 25	 [5 elements]
 50b59cd75bed7...	 2	 27	 [4 elements]
 50b59cd75bed7...	 2	 24	 [4 elements]









































































La consulta `db.grades.find({student_id: 2})` en MongoDB busca todos los documentos dentro de la colección `grades` que tengan un campo `student_id` con el valor 2.

Desglose:

`db.grades` se refiere a la colección llamada **grades** en la base de datos actual.

`. find({ student_id: 2 })` es una consulta que busca documentos donde el campo `student_id` sea igual a 2.

db.grades.find({ "scores.0.score": { \$lte: 10 } })

_id	student_id	class_id	scores
 id 50b59cd75bed7...	 0	 24	 [4 elements]
 id 50b59cd75bed7...	 2	 25	 [5 elements]
 id 50b59cd75bed7...	 3	 13	 [6 elements]
 id 50b59cd75bed7...	 3	 16	 [3 elements]
 id 50b59cd75bed7...	 4	 5	 [5 elements]
 id 50b59cd75bed7...	 4	 12	 [4 elements]
 id 50b59cd75bed7...	 6	 22	 [4 elements]
 id 50b59cd75bed7...	 8	 29	 [3 elements]
 id 50b59cd75bed7...	 11	 25	 [5 elements]
 id 50b59cd75bed7...	 12	 13	 [6 elements]
 id 50b59cd75bed7...	 15	 6	 [6 elements]
 id 50b59cd75bed7...	 20	 12	 [4 elements]
 id 50b59cd75bed7...	 23	 27	 [6 elements]
 id 50b59cd75bed7...	 23	 24	 [6 elements]
 id 50b59cd75bed7...	 26	 6	 [4 elements]
 id 50b59cd75bed7...	 29	 12	 [6 elements]
 id 50b59cd75bed7...	 30	 25	 [5 elements]
 id 50b59cd75bed7...	 30	 2	 [3 elements]

La consulta **db.grades.find({ "scores.0.score": { \$lte: 10 } })** en MongoDB busca en la colección **grades** todos los documentos cuyo primer elemento en el array **scores** tenga un valor en el campo **score** menor o igual a 10.

Desglose de la consulta:

db.grades: Se refiere a la colección **grades** en la base de datos actual.

find(): Busca documentos que coincidan con los criterios especificados.

"scores.0.score": Esto se refiere al campo **score** del **primer elemento (índice 0)** dentro del array **scores**. MongoDB usa índices numéricos para acceder a los elementos en arrays.

\$lte: 10: Este operador significa "menor o igual que 10". Así que está buscando documentos donde el primer elemento del array **scores** tiene un valor de **score** que es menor o igual a 10.

db.grades.find({ "scores.4.score": { \$lte: 10 } })

_id	student_id	class_id	scores
50b59cd75bed7...	0	5	[6 elements]
50b59cd75bed7...	13	22	[6 elements]
50b59cd75bed7...	15	21	[5 elements]
50b59cd75bed7...	15	22	[5 elements]
50b59cd75bed7...	17	19	[5 elements]
50b59cd75bed7...	19	22	[5 elements]
50b59cd75bed7...	24	22	[6 elements]
50b59cd75bed7...	30	0	[6 elements]
50b59cd75bed7...	41	0	[5 elements]
50b59cd75bed7...	42	18	[6 elements]

La consulta `db.grades.find({ "scores.4.score": { $lte: 10 } })` en MongoDB busca todos los documentos en la colección **grades** donde el **quinto elemento** (índice 4) del array **scores** tenga un campo **score** con un valor menor o igual a 10.





















Desglose:

scores.4.score: Hace referencia al campo score del **quinto elemento** en el array scores.

Recuerda que los arrays en MongoDB están indexados desde 0, por lo que el índice 4 corresponde al quinto elemento.

\$lte: 10: El operador \$lte significa "menor o igual que", por lo que está buscando aquellos documentos donde el quinto elemento del array scores tiene un campo score que es menor o igual a 10.

db.grades.find({ "scores.5.score": { \$lte: 10 } })

_id	student_id	class_id	scores
 50b59cd75bed7...	 6	 8	 [6 elements]
 50b59cd75bed7...	 12	 4	 [6 elements]
 50b59cd75bed7...	 35	 18	 [6 elements]
 50b59cd75bed7...	 36	 23	 [6 elements]
 50b59cd75bed7...	 41	 18	 [6 elements]

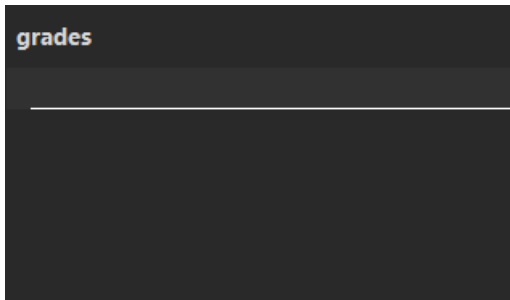
La consulta `db.grades.find({ "scores.5.score": { $lte: 10 } })` busca todos los documentos en la colección **grades** donde el **sexto elemento** (índice 5) del array **scores** tenga un campo **score** con un valor menor o igual a 10.

Desglose:

scores.5.score: Esto accede al campo score del **sexto elemento** en el array scores. Como MongoDB indexa los arrays desde 0, el índice 5 se refiere al sexto elemento.

\$lte: 10: El operador \$lte significa "menor o igual que", por lo que la consulta busca documentos donde el sexto elemento del array scores tenga un valor de score de 10 o menos.

db.grades.find({ "scores.6.score": { \$lte: 10 } })



La consulta `db.grades.find({ "scores.6.score": { $lte: 10 } })` busca todos los documentos en la colección **grades** donde el **séptimo elemento** (índice 6) del array **scores** tenga un campo **score** con un valor menor o igual a 10.

















Desglose:

scores.6.score: Hace referencia al campo **score** del **séptimo elemento** en el array **scores**.

Como MongoDB usa índices basados en cero, el índice 6 representa el séptimo elemento del array.

\$lte: 10: Es el operador de comparación "menor o igual que". Busca los documentos donde el valor del campo **score** en el séptimo elemento sea 10 o menos.

db.grades.find({ "scores.0.score": { \$gte: 60, \$lte: 61 } })

_id	student_id	class_id	scores
 id 50b59cd75bed7...	 0	 27	 [3 elements]
 id 50b59cd75bed7...	 4	 8	 [5 elements]
 id 50b59cd75bed7...	 25	 0	 [4 elements]
 id 50b59cd75bed7...	 30	 29	 [4 elements]

La consulta `db.grades.find({ "scores.0.score": { $gte: 60, $lte: 61 } })` busca todos los documentos en la colección **grades** donde el **primer elemento** (índice 0) del array **scores** tenga un campo **score** que esté en el rango de 60 a 61, inclusive.

Desglose:

scores.0.score: Hace referencia al campo score del **primer elemento** en el array scores.

















\$gte: 60: El operador \$gte significa "mayor o igual que", por lo que está buscando un valor que sea al menos 60.

\$lte: 61: El operador \$lte significa "menor o igual que", buscando un valor que no sea mayor a 61.

Combinación de los operadores:

Al combinar ambos operadores (\$gte y \$lte), la consulta efectivamente busca documentos donde el score del primer elemento esté entre 60 y 61, inclusive.

db.grades.find({ "scores.0.score": { \$gte: 60, \$lte: 61 } }).sort({ student_id: 1 })

_id	student_id	class_id	scores
 id 50b59cd75bed7...	 0	 27	 [3 elements]
 id 50b59cd75bed7...	 4	 8	 [5 elements]
 id 50b59cd75bed7...	 25	 0	 [4 elements]
 id 50b59cd75bed7...	 30	 29	 [4 elements]

La consulta `db.grades.find({ "scores.0.score": { $gte: 60, $lte: 61 } }).sort({ student_id: 1 })` hace lo siguiente:

Busca en la colección **grades** aquellos documentos donde el **primer elemento** (índice 0) del array **scores** tenga un campo **score** que esté entre **60 y 61**, inclusive.

scores.0.score: Se refiere al **score** del **primer elemento** en el array **scores**.

\$gte: 60 y **\$lte: 61**: Filtra los documentos cuyo valor de **score** esté entre 60 y 61, ambos incluidos.

Ordena los resultados por el campo **student_id** en **orden ascendente**:

{ student_id: 1 }: Ordena los documentos de menor a mayor valor de **student_id** (ascendente). Si quieres un orden descendente, usarías **{ student_id: -1 }**.

db.grades.find({ student_id: 2, class_id: 24 })

_id	student_id	class_id	scores
id 50b59cd75bed7...	2	24	[4 elements]

La consulta `db.grades.find({ student_id: 2, class_id: 24 })` busca en la colección **grades** todos los documentos que cumplan **dos condiciones simultáneamente**:

student_id sea igual a **2**.

class_id sea igual a **24**.

Funcionamiento:

student_id: 2: Filtra los documentos donde el valor de `student_id` es exactamente 2.

class_id: 24: Filtra los documentos donde el valor de `class_id` es exactamente 24.

MongoDB interpreta estas condiciones como un **"AND" implícito**, lo que significa que ambos criterios deben cumplirse al mismo tiempo para que el documento sea devuelto.

```
db.grades.find( { class_id: 20, $and: [ {"scores.0.score": { $gte: 15 } }, {
"scores.0.score": { $lte: 30 } } ] } )
```

_id	student_id	class_id	scores
id 50b59cd75bed76f46522c45e	46	20	[5 elements]

```
{
  "_id" : ObjectId("50b59cd75bed76f46522c45e"),
  "student_id" : NumberInt(46),
  "class_id" : NumberInt(20),
  "scores" : [
    {
      "type" : "exam",
      "score" : 27.79390777259294
    },
    {
      "type" : "quiz",
      "score" : 29.18509374453628
    },
    {

```

Funcionamiento:

class_id: 20: Filtra los documentos donde el campo class_id tiene un valor de 20.

\$and: [...]: Utiliza el operador lógico \$and para combinar dos condiciones.

scores.0.score: { \$gte: 15 }: Esta condición especifica que el campo score del primer elemento (índice 0) en el array scores debe ser **mayor o igual** a 15.

scores.0.score: { \$lte: 30 }: Esta condición especifica que el campo score del primer elemento en el array scores debe ser **menor o igual** a 30.

El operador \$and asegura que ambas condiciones relacionadas con el campo score sean verdaderas simultáneamente, es decir, que el valor de score esté en el rango entre **15 y 30**.

db.grades.find({ scores: { \$elemMatch: { type: 'quiz', score: { \$gte: 50 } } } })

_id	student_id	class_id	scores
id 50b59cd75bed7...	46	20	[5 elements]

```
▼ {
  "_id" : ObjectId("50b59cd75bed76f46522c45e"),
  "student_id" : NumberInt(46),
  "class_id" : NumberInt(20),
  "scores" : [
    ▼ {
      "type" : "exam",
      "score" : 27.79390777259294
    },
    ▼ {
      "type" : "quiz",
      "score" : 29.18509374453628
    },
  ],
}
```

Busca en la colección grades aquellos documentos donde el array scores tenga **al menos un elemento** que cumpla las siguientes condiciones:

El campo type sea 'quiz'.

El campo score sea **mayor o igual a 50** (\$gte: 50).





































Funcionamiento:

scores: { \$elemMatch: ... }: El operador \$elemMatch se utiliza para encontrar **elementos dentro de un array** que cumplan **todas las condiciones** especificadas. En este caso, busca un elemento en el array scores que tenga:

Un campo type con el valor 'quiz'.

Un campo score con un valor **mayor o igual a 50**.

db.grades.find({ scores: { \$elemMatch: { type: 'exam', score: { \$gte: 50 } } } })

_id	student_id	class_id	scores
 id 50b59cd75bed7...	 0	 2	 [5 elements]
 id 50b59cd75bed7...	 0	 5	 [6 elements]
 id 50b59cd75bed7...	 0	 16	 [5 elements]
 id 50b59cd75bed7...	 0	 6	 [6 elements]
 id 50b59cd75bed7...	 0	 27	 [3 elements]
 id 50b59cd75bed7...	 0	 11	 [4 elements]
 id 50b59cd75bed7...	 1	 18	 [3 elements]
 id 50b59cd75bed7...	 1	 16	 [5 elements]
 id 50b59cd75bed7...	 1	 13	 [4 elements]

```
▼ {
  "_id" : ObjectId("50b59cd75bed76f46522c34e")
  "student_id" : NumberInt(0),
  "class_id" : NumberInt(2),
  "scores" : [
    ▼ {
      "type" : "exam",
      "score" : 57.92947112575566
    },
    ▼ {
      "type" : "exam",
      "score" : 57.92947112575566
    },
    ▼ {
      "type" : "exam",
      "score" : 57.92947112575566
    },
    ▼ {
      "type" : "exam",
      "score" : 57.92947112575566
    },
    ▼ {
      "type" : "exam",
      "score" : 57.92947112575566
    }
  ]
}
```

Busca en la colección **grades** aquellos documentos donde el array **scores** tenga al menos un elemento que cumpla las siguientes condiciones:

El campo type debe ser 'exam'.

El campo score debe ser mayor o igual a 50 (\$gte: 50).

Funcionamiento:

scores: { \$elemMatch: ... }: El operador \$elemMatch se utiliza para encontrar **un único elemento** dentro del array scores que cumpla **todas las condiciones** especificadas:

El campo type del elemento debe ser **'exam'**.

El campo score debe ser **mayor o igual a 50**.

3. Busca la colección Narcos, ejecuta las siguientes consultas y describe qué hace cada una de ellas:

`db.Narcos.find({ runtime: { $gte: 55 } }, { _id:0, name:1, season:1, number:1 })`

```
"airstamp" : "2015-08-28T12:00:00Z"
"runtime"  : NumberInt(57),
"rating"   : {
```

name	season	number
Descenso	1	1
There Will Be a F...	1	5
The Good, the B...	2	4
Deutschland 93	2	7
Exit El Patrón	2	8
Nuestra Finca	2	9
The Kingpin Stra...	3	1
Follow the Money	3	3
MRO	3	5
Best Laid Plans	3	6

Busca en la colección Narcos todos los documentos donde el campo **runtime** sea **mayor o igual a 55** (`$gte: 55`).

Devuelve únicamente los campos:

name: El nombre del episodio.

season: La temporada a la que pertenece el episodio.

number: El número del episodio dentro de la temporada.

Además, **excluye el campo _id** (al establecer `_id: 0`), para que no sea parte de los resultados.


```
db.Narcos.find( { runtime: { $gte: 15 } }, { _id:0, season:1, number:1 } ).sort( {  
season:1, number:-1 } )
```

season	number
1	9
1	8
1	7
1	6
1	5
1	4
1	3
1	2
1	1
2	10
2	9

Busca en la colección **Narcos** todos los documentos donde el campo **runtime** sea mayor o igual a 15 (\$gte: 15).

Devuelve únicamente los campos:

season: La temporada del episodio.

number: El número del episodio dentro de la temporada.

No incluye el campo **_id** (ya que se ha establecido _id: 0).

Ordena los resultados con el operador .sort():

Ordena en forma **ascendente** por el campo season (season: 1), es decir, de la temporada más baja a la más alta.

Dentro de cada temporada, los episodios se ordenan en forma **descendente** por el campo number (number: -1), es decir, del número de episodio más alto al más bajo.

```
db.Narcos.find( { season: { $type: 'number' } } )
```

_id	id	url	name	season	number	type
id 6708699f94da5b...	203469	https://www.tv...	Descenso	1	1	regular
id 6708699f94da5b...	208978	https://www.tv...	The Sword of Si...	1	2	regular
id 6708699f94da5b...	208979	https://www.tv...	The Men of Alw...	1	3	regular
id 6708699f94da5b...	208980	https://www.tv...	The Palace in Fl...	1	4	regular
id 6708699f94da5b...	208981	https://www.tv...	There Will Be a ...	1	5	regular

Busca en la colección **Narcos** todos los documentos donde el campo **season** sea de tipo **número**. Esto se logra utilizando el operador `$type`, que permite especificar el tipo de datos que debe tener el campo.

En este caso, `{ $type: 'number' }` filtra los documentos donde el campo **season** sea numérico.

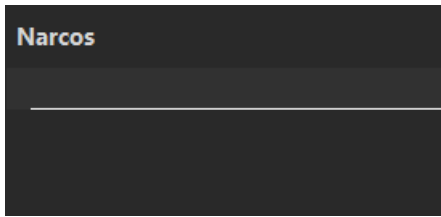
```
db.Narcos.find( { rating: { $exists: 1 } } )
```

runtime	rating	image
57	{ 1 fields }	{ 2
47	{ 1 fields }	{ 2
47	{ 1 fields }	{ 2
44	{ 1 fields }	{ 2
55	{ 1 fields }	{ 2
50	{ 1 fields }	{ 2
51	{ 1 fields }	{ 2
51	{ 1 fields }	{ 2
51	{ 1 fields }	{ 2

Busca en la colección **Narcos** todos los documentos donde el campo **rating** existe. El operador `$exists` permite verificar si un campo está presente en los documentos, ya sea que tenga un valor o no.

`{ $exists: 1 }` (o `{ $exists: true }`) indica que solo se devuelvan los documentos que contienen el campo **rating**, sin importar su valor.

```
db.Narcos.find( { rating: { $exists: 1 }, rating: { $type: "string" } } )
```



Busca en la colección Narcos:

Donde el campo rating existe ({ rating: { \$exists: 1 } }).

Y el campo rating es de tipo string ({ rating: { \$type: "string" } }).

En este caso no muestra nada ya que el campo rating no es de tipo string

4. Prueba las siguientes consultas:

db.grades.distinct("student_id")

[Index]	
0	123 0.0
1	123 1.0
2	123 2.0
3	123 3.0
4	123 4.0
5	123 5.0
6	123 6.0

se utiliza para obtener un conjunto de valores únicos para el campo **student_id** en la colección **grades**. Aquí tienes un desglose de lo que hace esta consulta:

Descripción de la Consulta

distinct("student_id"): Esta función devuelve un arreglo que contiene todos los valores únicos del campo **student_id** de los documentos en la colección **grades**.

db.grades.countDocuments()

```
already on db school  
280
```

se utiliza para contar el número total de documentos en la colección **grades**. Aquí hay un desglose de lo que hace esta consulta:

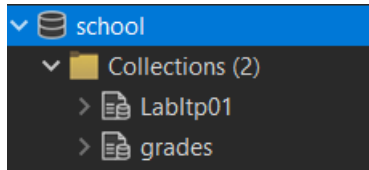
Descripción de la Consulta

countDocuments(): Este método devuelve el número total de documentos que coinciden con el criterio de consulta. Si no se proporciona ningún filtro, contará todos los documentos en la colección.

Trabaja con CREATE

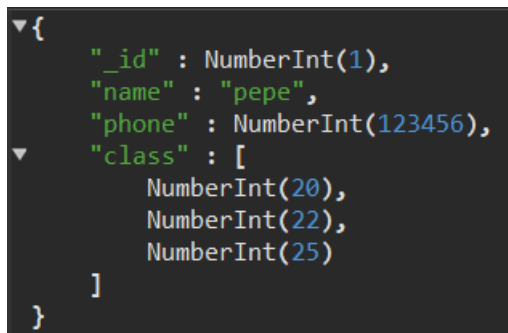
En la BD School crea la colección LabItp:

db.createCollection("LabItp01")



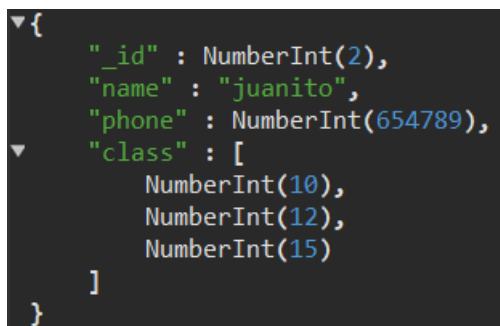
db.LabItp01.insert({ _id:1, name: "pepe", phone: 123456, class: [20, 22, 25] })

Este método se utiliza para insertar un nuevo documento en la colección.



















db.LabItp01.insertOne({_id:2, name: "juanito", phone: 654789, class: [10, 12, 15] })

Este método se utiliza para insertar un único documento en la colección. Si el documento ya existe (es decir, tiene el mismo _id), se generará un error.



















```
db.LabItp01.insertMany( [ { _id:3, name: "carlito", phone: 639852, class: [ 11, 10] }, { _id:4, name: "camilito", phone: 741258, class: [ 15] }, { _id:5, name: "anita", phone: 852741, class: [ 10] }, { _id:5, name: "joselito", phone: 1254896, class: [ 55, 458, 236, 20, 22, 10, 15] } ] )
```

Este método se utiliza para insertar múltiples documentos a la vez en la colección.

 3	 carlito	 639852	 [2 elements]
 4	 camilito	 741258	 [1 elements]
 5	 anita	 852741	 [1 elements]
 6	 joselito	 1254896	 [7 elements]

```
db.LabItp01.find( { class: 10 } )
```

_id	name	phone	class
 2	 juanito	 654789	 [3 elements]
 3	 carlito	 639852	 [2 elements]
 5	 anita	 852741	 [1 elements]
 6	 joselito	 1254896	 [7 elements]

se utiliza para buscar documentos en la colección **LabItp01** que tengan el campo **class** igual a **10**. Aquí tienes un desglose de la consulta:

Descripción de la Consulta

db.LabItp01: Esto se refiere a la colección llamada **LabItp01** en la base de datos en la que estás trabajando (en este caso, parece que estás usando la base de datos **school**).

find({ class: 10 }): El método **find** se utiliza para buscar documentos que coincidan con el criterio especificado en el objeto. En este caso, busca documentos donde el campo **class** tenga un valor de **10**.

db.LabItp02.insertOne({ name: "carolita" })

```
▼{
  "_id" : ObjectId("670dd230aa1c4394fd072f8a"),
  "name" : "carolita"
}
```

**db.LabItp02.insertOne({ name: "carolita", information: { classroom: "room_01",
locker: 12 }, age: 25 })**

```
▼{
  "_id" : ObjectId("670dd262aa1c4394fd072f8b"),
  "name" : "carolita",
  "information" : {
    "classroom" : "room_01",
    "locker" : NumberInt(12)
  },
  "age" : NumberInt(25)
}
```

db.LabItp02.find()

```
▼{
  "_id" : ObjectId("670dd230aa1c4394fd072f8a"),
  "name" : "carolita"
}
▼{
  "_id" : ObjectId("670dd262aa1c4394fd072f8b"),
  "name" : "carolita",
  "information" : {
    "classroom" : "room_01",
    "locker" : NumberInt(12)
  },
  "age" : NumberInt(25)
}
```

se utiliza para recuperar todos los documentos de la colección **LabItp02** en la base de datos actual. Aquí tienes un desglose de lo que hace esta consulta:

Descripción de la Consulta

db.LabItp02: Esto se refiere a la colección llamada **LabItp02** en la base de datos que estás utilizando (en este caso, parece que estás trabajando en la base de datos **school**).

find(): Este método, sin ningún argumento, devuelve todos los documentos de la colección.

Trabaja con UPDATE

En la colección LabItp01 realiza las siguientes actualizaciones

```
db.LabItp01 .updateOne( { _id: 7 }, { $set: { virtues: ['cheerful', 'funny', 'comprehensive', 'sociable', 'respectful'] } } )
```

_id	name	phone	class
1	pepe	123456	[3 elements]
2	juanito	654789	[3 elements]
3	carlito	639852	[2 elements]
4	camilito	741258	[1 elements]
5	anita	852741	[1 elements]
6	joselito	1254896	[7 elements]

La consulta que has proporcionado intenta actualizar un documento en la colección LabItp01 donde el campo _id es igual a 7, añadiendo o modificando el campo virtues con los valores del array ['cheerful', 'funny', 'comprehensive', 'sociable', 'respectful'].

Si no existe un documento con _id: 7, esta operación no afectará ningún documento.

Explicación:

{ _id: 7 }: Busca el documento con _id igual a 7.

\$set: { virtues: [...] }: Añade o actualiza el campo virtues con el array de virtudes proporcionado.


```
db.LabItp01 .updateOne( { _id: 7 }, { $set: { information: { classroom: "room_A",  
locker: 15 }, age: 18 } } )
```

La consulta que has proporcionado actualiza o agrega campos adicionales al documento con `_id: 7` en la colección `LabItp01`. Si este documento ya existe, se le añadirán los campos `information` y `age`; si no existe, no habrá ningún cambio (a menos que uses la opción `upsert`).

Explicación:

{ _id: 7 }: Busca el documento con `_id` igual a 7.

\$set: { information: { classroom: "room_A", locker: 15 }, age: 18 }: Añade o actualiza el campo `information` con un subdocumento que contiene `classroom: "room_A"` y `locker: 15`, y además añade o actualiza el campo `age` con el valor 18.

```
db.LabItp01 .updateOne( { _id: 7 }, { $set: { virtues: ['cheerful', 'funny',  
'comprehensive', 'sociable', 'respectful'] }, $currentDate: { lastModified: true } } )
```

La consulta que has proporcionado realiza las siguientes acciones en la colección `LabItp01` sobre el documento con `_id: 7`:

Actualiza el campo `virtues` con el array `['cheerful', 'funny', 'comprehensive', 'sociable', 'respectful']`.

Establece la fecha y hora actual en el campo `lastModified` usando el operador `$currentDate`.

Explicación:

{ _id: 7 }: Busca el documento con `_id` igual a 7.

\$set: { virtues: [...] }: Añade o actualiza el campo `virtues` con los valores proporcionados.

\$currentDate: { lastModified: true }: Establece el valor de `lastModified` como la fecha y hora actual.

```
db.LabItp01 .updateOne( { _id: 7 }, { $set: { information: { classroom: "room_A",  
locker: 15 }, age: 18 }, $currentDate: { lastModified: true } } )
```

La consulta que has proporcionado realiza las siguientes acciones en el documento con `_id: 7` dentro de la colección `LabItp01`:

Actualiza o agrega el campo `information` con el subdocumento `{ classroom: "room_A", locker: 15 }`.

Actualiza o agrega el campo `age` con el valor `18`.

Actualiza el campo `lastModified` con la fecha y hora actuales usando el operador `$currentDate`.

Explicación:

{ _id: 7 }: Busca el documento con `_id` igual a `7`.

\$set: { information: { classroom: "room_A", locker: 15 }, age: 18 }: Añade o actualiza los campos `information` y `age`.

\$currentDate: { lastModified: true }: Añade o actualiza el campo `lastModified` con la fecha y hora actuales.

```
db.LabItp01.updateOne( { _id: 10 }, { $set: { name: "Joan", age: 19, virtues: [],  
information: {} }, $currentDate: { lastModified: true } }, { upsert: true } )
```

```
▼{  
  "_id" : NumberInt(10),  
  "age" : NumberInt(19),  
  "information" : {  
    },  
  "lastModified" : ISODate("2024-10-15T04:09:56.931+0000"),  
  "name" : "Joan",  
  "virtues" : [  
    ]  
}
```

Busca el documento con `_id: 10`.

Si el documento existe, actualiza los campos name, age, virtues e information, y establece el campo lastModified con la fecha y hora actuales.

Si el documento no existe, lo inserta con los campos proporcionados, ya que usaste la opción upsert: true.

Explicación de los operadores:

{ _id: 10 }: Busca un documento con _id igual a 10.

\$set: { name: "Joan", age: 19, virtues: [], information: {} }: Actualiza o agrega los campos name, age, virtues e information con los valores indicados.

name: "Joan": Establece el nombre como "Joan".

age: 19: Establece la edad en 19.

virtues: []: Establece un array vacío en virtues.

information: {}: Establece un objeto vacío en information.

\$currentDate: { lastModified: true }: Actualiza el campo lastModified con la fecha y hora actuales.

upsert: true: Si no encuentra un documento con _id: 10, inserta uno nuevo con los datos proporcionados.

Actualiza los documentos con `_id` 1 – 6 y agrega el campo `virtues` con un array que contenga un único valor, el que decidas de la lista siguiente: ['cheerful', 'funny', 'comprehensive', 'sociable', 'respectful'].

Para actualizar los documentos con `_id` de 1 a 6 y agregar el campo `virtues` con un único valor de la lista proporcionada, usamos la siguiente consulta:

```
db.LabItp01.bulkWrite([
  { updateOne: { filter: { _id: 1 }, update: { $set: { virtues: ['cheerful'] } } } },
  { updateOne: { filter: { _id: 2 }, update: { $set: { virtues: ['funny'] } } } },
  { updateOne: { filter: { _id: 3 }, update: { $set: { virtues: ['comprehensive'] } } } },
  { updateOne: { filter: { _id: 4 }, update: { $set: { virtues: ['sociable'] } } } },
  { updateOne: { filter: { _id: 5 }, update: { $set: { virtues: ['respectful'] } } } },
  { updateOne: { filter: { _id: 6 }, update: { $set: { virtues: ['cheerful'] } } } }
]);
```

Explicación:

bulkWrite: Permite realizar múltiples operaciones de escritura en una sola llamada.

updateOne: Cada operación es de tipo `updateOne`, que especifica el filtro (documento a actualizar) y el cambio a aplicar.

filter: Especifica el criterio para encontrar el documento (en este caso, el `_id`).

update: Contiene el cambio que deseas hacer (agregar o actualizar el campo `virtues`).

```
{
  "_id" : NumberInt(2),
  "name" : "juanito",
  "phone" : NumberInt(654789),
  "class" : [
    NumberInt(10),
    NumberInt(12),
    NumberInt(15)
  ],
  "virtues" : [
    "funny"
  ]
}
{
  "_id" : NumberInt(3),
  "name" : "carlito",
  "phone" : NumberInt(639852),
  "class" : [
    NumberInt(11),
    NumberInt(10)
  ],
  "virtues" : [
    "comprehensive"
  ]
}
```

Actualiza todos los documentos con una única instrucción y agrega el siguiente campo: status: 'A'.

Para actualizar todos los documentos en la colección `LabItp01` y agregar el campo `status` con el valor 'A' en una única instrucción, puedes utilizar el siguiente comando:

```
db.LabItp01.updateMany(  
  {},  
  { $set: { status: 'A' } }  
);
```

Explicación:

updateMany: Esta función se utiliza para actualizar múltiples documentos que cumplen con un criterio.

{}: El primer argumento vacío (`{}`) significa que se seleccionan todos los documentos de la colección.

\$set: { status: 'A' }: Este comando establece el campo `status` con el valor 'A' en todos los documentos seleccionados.

```
  }  
  "virtues" : [  
    "cheerful"  
  ],  
  "status" : "A"  
}  
▼{  
  "_id" : NumberInt(2),  
  "name" : "juanito",  
  "phone" : NumberInt(654789),  
  "class" : [  
    NumberInt(10),  
    NumberInt(12),  
    NumberInt(15)  
  ],  
  "virtues" : [  
    "funny"  
  ],  
  "status" : "A"  
}  
▼{
```

Actualiza los documentos de “pepe” y “camilito” y agrega el siguiente campo:
role: 'student'.

Para actualizar los documentos de "pepe" y "camilito" y agregar el campo `role`:

'student', puedes usar el siguiente comando en MongoDB:

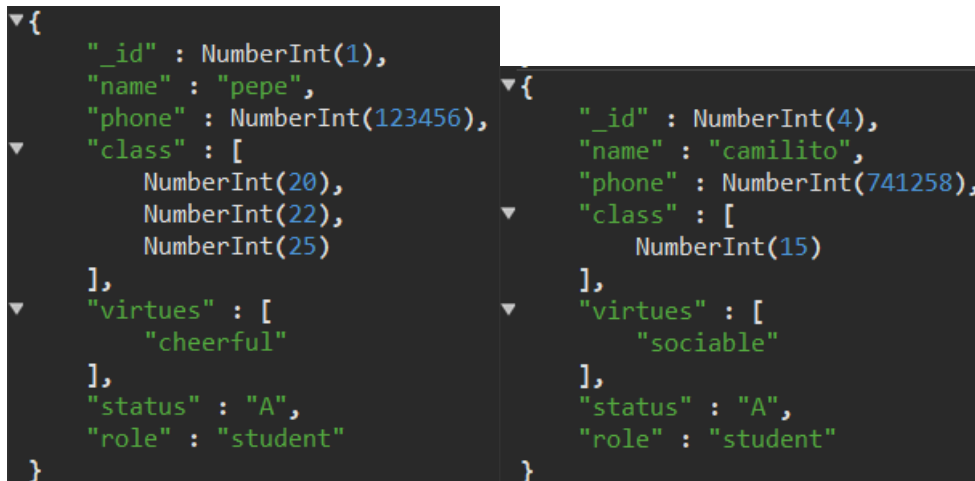
```
db.LabItp01.updateMany(  
  { name: { $in: ["pepe", "camilito"] } },  
  { $set: { role: 'student' } }  
);
```

Explicación:

`{ name: { $in: ["pepe", "camilito"] } }`: Este filtro selecciona los documentos donde el campo `name` sea igual a "pepe" o "camilito".

`$set: { role: 'student' }`: Con esta operación, se agrega o actualiza el campo `role` con el valor 'student'.

Al ejecutar este comando, ambos documentos correspondientes a "pepe" y "camilito" serán actualizados con el campo `role`.



```
{  
  "_id" : NumberInt(1),  
  "name" : "pepe",  
  "phone" : NumberInt(123456),  
  "class" : [  
    NumberInt(20),  
    NumberInt(22),  
    NumberInt(25)  
  ],  
  "virtues" : [  
    "cheerful"  
  ],  
  "status" : "A",  
  "role" : "student"  
}  
  
{  
  "_id" : NumberInt(4),  
  "name" : "camilito",  
  "phone" : NumberInt(741258),  
  "class" : [  
    NumberInt(15)  
  ],  
  "virtues" : [  
    "sociable"  
  ],  
  "status" : "A",  
  "role" : "student"  
}
```

Trabaja con DELETE

En la colección LabItp01 realiza las siguientes actualizaciones:

db.LabItp01.deleteOne({ name: "carlito" })

Explicación:

deleteOne: Este método elimina el primer documento que coincida con el filtro especificado.

{ name: "carlito" }: Este filtro selecciona el documento donde el campo name sea igual a "carlito".

_id	name	phone
1	pepe	12
2	juanito	65
3	carlito	63
4	camilito	74
5	anita	85
6	joeselito	12

_id	name
1	pepe
2	juanito
4	camilito
5	anita
6	joeselito

db.grades.deleteOne({ student_id: 0 })

Explicación:

deleteOne: Este método elimina el primer documento que coincida con el filtro especificado.

{ student_id: 0 }: Este filtro selecciona el documento donde el campo student_id sea igual a 0.

_id	name	p
1	pepe	
2	juanito	
4	camilito	
5	anita	
6	joselito	
10	Joan	

db.grades.deleteMany({ student_id: 0 })

Explicación:

deleteMany: Este método elimina todos los documentos que coincidan con el filtro especificado.

{ student_id: 0 }: Este filtro selecciona los documentos donde el campo student_id sea igual a 0.

_id	name	p
1	pepe	
2	juanito	
4	camilito	
5	anita	
6	joselito	
10	Joan	


```
db.grades.remove( { student_id: 1 }, {justOne: true} )
```

remove: Elimina documentos de una colección.

{ student_id: 1 }: Este filtro selecciona los documentos donde el campo student_id sea igual a 1.

{ justOne: true }: Esta opción indica que solo se debe eliminar un documento, incluso si hay varios que coincidan con el filtro.

_id	name	p
1	pepe	
2	juanito	
4	camilito	
5	anita	
6	joselito	
10	Joan	

```
db.grades.remove( { student_id: 1 } )
```

El comando que estás utilizando, debería eliminar todos los documentos de la colección grades que tengan un student_id igual a 1.

```
db.grades.remove( { } )
```

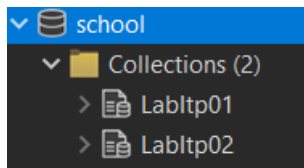
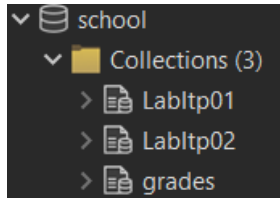
El comando eliminará **todos los documentos** de la colección grades. Básicamente, está indicando que no hay un filtro específico, por lo que MongoDB interpretará que debe eliminar cada documento de la colección.

acknowledged	deletedCount
true	264.0

grades

db.grades.drop()

El comando `db.grades.drop()` elimina **completamente** la colección `grades` de la base de datos, incluyendo todos los documentos almacenados y los índices asociados. Este es un proceso irreversible, por lo que después de ejecutarlo, la colección `grades` ya no existirá en la base de datos.



Análisis y Discusión

Interpretación de Resultados: Los resultados obtenidos de las consultas ejecutadas reflejan una estructura robusta y eficiente de la base de datos. La utilización de `$lookup` en las consultas permitió combinar datos de diferentes colecciones, facilitando la obtención de información relevante, como detalles de contenidos en playlists y géneros de películas. Esto cumple con los objetivos establecidos de demostrar la capacidad de MongoDB para manejar relaciones complejas a través de sus características de agregación.

La capacidad de realizar actualizaciones condicionales mediante `updateOne()` y `upsert` demuestra la flexibilidad de MongoDB para mantener la integridad de los datos, permitiendo la adición y modificación de información sin complicaciones. Además, los resultados obtenidos de las consultas reflejan la efectividad de las relaciones muchos a muchos, evidenciando cómo las playlists pueden contener múltiples contenidos y cómo un contenido puede pertenecer a diferentes playlists.

Conclusiones

El análisis de la base de datos en MongoDB ha demostrado su eficacia en el manejo de datos relacionados y la flexibilidad en la actualización y consulta de información. La implementación de esquemas de relaciones entre colecciones ha permitido una organización más clara y accesible de los datos. Los métodos de agregación utilizados son esenciales para optimizar el rendimiento y la velocidad de las consultas, lo que a su vez contribuye a una mejor experiencia de usuario en la plataforma.

Recomendaciones

1. **Optimización de Consultas:** Considerar el uso de índices en los campos más consultados para mejorar el rendimiento de las búsquedas y agregaciones.
2. **Mantenimiento de Datos:** Implementar procedimientos para la validación y limpieza de datos periódica, asegurando la integridad de la información almacenada.
3. **Documentación de Esquemas:** Mantener una documentación clara y actualizada de los esquemas de las colecciones y sus relaciones, facilitando la comprensión y el uso de la base de datos por parte de nuevos desarrolladores.

Referencias

1. MongoDB Documentation: [MongoDB Official Documentation](#)
2. Studio 3T Documentation: Studio 3T Documentation
3. Ejemplos de consultas y análisis de bases de datos de ejemplo en MongoDB.

Enlace de GitHub

https://github.com/ivanbecerrq/plataforma_streaming.git