**Informe Almacenamiento Masivo Mongo Crud**

Aldeir Muñoz Gaviria

Ingeniería de Sistemas, Instituto Tecnológico del Putumayo

8 semestre

Bases De Datos y Almacenamiento Masivo

Ing. Brayan Arcos

17 de octubre de 2024

**Informe Mongo**

[**Resumen Ejecutivo** 2](#_Toc179992846)

[**Introducción** 2](#_Toc179992847)

[**Contexto y Motivación** 2](#_Toc179992848)

[**Alcance del Informe** 2](#_Toc179992849)

[**Objetivos** 3](#_Toc179992850)

[**Metodología** 3](#_Toc179992851)

[**Herramientas Utilizadas** 3](#_Toc179992852)

[**Procedimientos** 4](#_Toc179992853)

[1. Métodos y pasos seguidos para llevar a cabo el análisis o el trabajo 4](#_Toc179992854)

[**Desarrollo del Informe** 5](#_Toc179992855)

[**Descripción de la Base de Datos Personal** 5](#_Toc179992856)

[**Diseño de Base de Datos Personal** 5](#_Toc179992857)

[**1.** **Modelo de Datos** 5](#_Toc179992858)

[**2.** **Consideraciones de Diseño** 6](#_Toc179992859)

[**Nuevas Funciones de Actualización** 11](#_Toc179992860)

[**Relaciones entre colecciones** 14](#_Toc179992861)

[**Resumen de las relaciones:** 15](#_Toc179992862)

[**CONSULTAS** 16](#_Toc179992863)

[**DIAGRAMAS DE LAS COLECCIONES** 19](#_Toc179992864)

[***Interpretación de Resultados*** 21](#_Toc179992865)

[. **CONSULTAS TALLER CRUD** 22](#_Toc179992866)

[TRABAJA CON READ 25](#_Toc179992867)

[Trabaja con CREATE 41](#_Toc179992868)

[**Trabaja con UPDATE** 44](#_Toc179992869)

[**Trabaja con DELETE** 52](#_Toc179992870)

[**Conclusiones** 55](#_Toc179992871)

[Recomendaciones 55](#_Toc179992872)

[Referencias 56](#_Toc179992873)

# **Resumen Ejecutivo**

Este informe documenta el diseño y desarrollo de una base de datos para un sistema de reservas de vuelos utilizando MongoDB. La base de datos incluye colecciones que permiten la gestión de usuarios, aeropuertos, vuelos y reservas, integrando relaciones uno a uno, uno a muchos, y muchos a muchos. El informe analiza la estructura de los datos, los métodos de captura utilizados y los resultados obtenidos a través de consultas y actualizaciones en MongoDB. Se describen las decisiones de diseño tomadas en cuanto a la normalización y la elección de embebido o referencia de documentos, lo que permite escalabilidad y eficiencia en las consultas.

# **Introducción**

## **Contexto y Motivación**

En la actualidad, los sistemas de gestión de reservas de vuelos requieren soluciones robustas y escalables que puedan manejar grandes volúmenes de datos de manera eficiente. MongoDB es una base de datos NoSQL que ofrece flexibilidad en el diseño y una estructura de documentos adecuada para manejar estos escenarios. El presente informe aborda la creación de una base de datos de reservas de vuelos en MongoDB, explicando las relaciones y estructuras empleadas en su diseño.

## **Alcance del Informe**

Este informe cubre los siguientes aspectos de MongoDB:

* Diseño de la base de datos: estructura de colecciones y relaciones.
* Consultas de actualización de datos: uso de updateOne(), upsert, setOnInsert y otros métodos.
* Análisis de esquemas y normalización.
* Discusión sobre el diseño de las relaciones entre los datos.

# **Objetivos**

Esta actividad te permitirá profundizar en el uso de MongoDB. Realizarás una serie de ejercicios guiados que te ayudarán a comprender mejor las operaciones del CRUD de esta base de datos documental, así como sus particularidades.

A través del desarrollo de la base de datos de reservas de vuelos, se busca:

1. Explorar las capacidades de MongoDB para la creación, lectura, actualización y eliminación de datos (CRUD).
2. Comprender las operaciones clave del CRUD en MongoDB:
   * **CREATE**: Añadir documentos a las colecciones con insertOne() y insertMany().
   * **READ**: Recuperar documentos con find() y findOne().
   * **UPDATE**: Modificar documentos existentes con updateOne() y updateMany().
   * **DELETE**: Eliminar documentos de las colecciones con deleteOne() y deleteMany().
3. Utilizar herramientas como MongoDB Compass o la consola de Mongo (mongosh) para realizar las operaciones necesarias.
4. Visualizar todas las bases de datos existentes usando el comando show dbs para comprender mejor el entorno de trabajo.

# **Metodología**

## **Herramientas Utilizadas**

* **MongoDB**: Base de datos NoSQL utilizada para almacenar y gestionar la información del sistema de reservas de vuelos.
* **Studio 3T**: Herramienta visual para la administración de MongoDB que facilita la consulta, manipulación y análisis de datos.
* **MongoDB Compass**: Herramienta gráfica para la gestión de bases de datos MongoDB.
* **JavaScript (shell MongoDB)**: Para la creación de consultas y scripts de actualización de datos.
* **Jsongrid**: Visualiza el contenido del fichero.

## **Procedimientos**

### 1. Métodos y pasos seguidos para llevar a cabo el análisis o el trabajo

* 1. **Creación base de datos personal**

La base de datos fue diseñada para gestionar la información clave relacionada con reservas de vuelos, incluyendo las siguientes colecciones: usuarios, aeropuertos, vuelos y reservas. Cada una de estas colecciones tiene un esquema claro que permite la relación entre ellas mediante IDs.

* 1. **Realización del taller**

Durante el desarrollo del taller se realizaron las siguientes tareas:

* Definición de colecciones y sus relaciones.
* Implementación de métodos de actualización avanzados (updateOne(), upsert, setOnInsert).
* Creación de datos de prueba para simular escenarios reales de uso.

**1.2.1 Métodos de captura**

Se utilizó Studio 3T y MongoDB Compass para visualizar los esquemas y relaciones, además de realizar consultas de prueba para validar la estructura y coherencia de los datos. Las consultas se realizaron directamente desde la interfaz de Studio 3T y desde la shell de MongoDB para evaluar el rendimiento.

# **Desarrollo del Informe**

## **Descripción de la Base de Datos Personal**

La base de datos diseñada contiene cuatro colecciones principales: usuarios, aeropuertos, vuelos y reservas. A continuación se muestra una breve descripción de cada colección:

1. **Usuarios**: Almacena la información personal de los clientes, como nombre, correo, pasaporte, teléfono y dirección.
2. **Aeropuertos**: Almacena los detalles de los aeropuertos, incluyendo el código IATA, nombre, ciudad, país, capacidad anual y número de terminales.
3. **Vuelos**: Contiene los datos relacionados con los vuelos, como origen, destino, fecha, duración, asientos disponibles, precio, aerolínea y número de vuelo.
4. **Reservas**: Relaciona a los usuarios con los vuelos, almacenando la información de la reserva, incluyendo asientos reservados, estado y método de pago.

## **Diseño de Base de Datos Personal**

1. **Modelo de Datos**

El diseño del modelo de datos sigue los principios de normalización en cuanto a la separación de entidades (colecciones) y el uso de referencias entre ellas. Por ejemplo, los usuarios y vuelos se mantienen en colecciones separadas y se relacionan a través de la colección de reservas, que almacena los IDs de los usuarios y los vuelos.

**Normalización**: Las entidades principales (usuarios, aeropuertos, vuelos) se mantienen en colecciones separadas, mientras que las relaciones (reservas) se manejan con referencias para evitar redundancias.

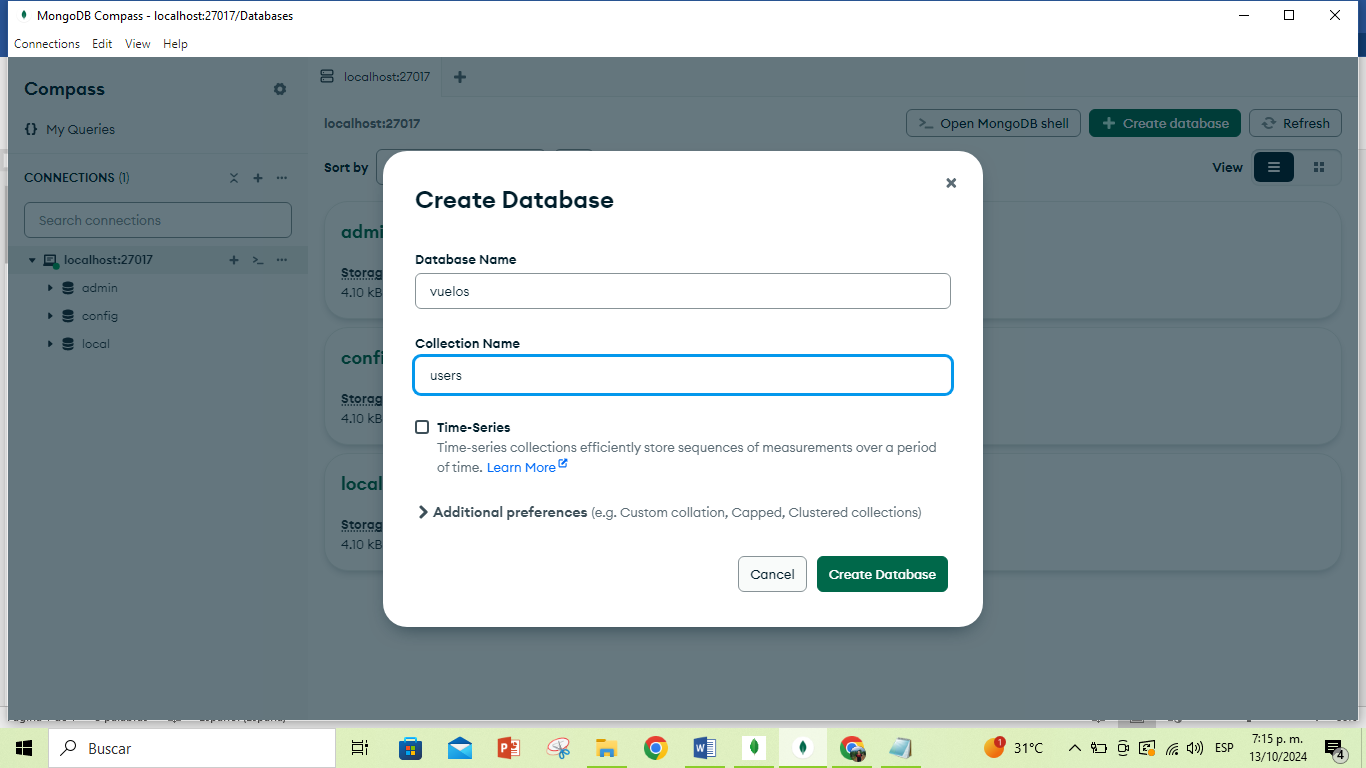
**Cardinalidad**:

* **Relaciones Uno a Uno (1:1)**: Un usuario tiene un pasaporte único.
* **Relaciones Uno a Muchos (1-M)**: Un vuelo puede tener múltiples reservas.
* **Relaciones Muchos a Muchos (M-M)**: Un usuario puede reservar múltiples vuelos, y un vuelo puede tener múltiples usuarios reservados.

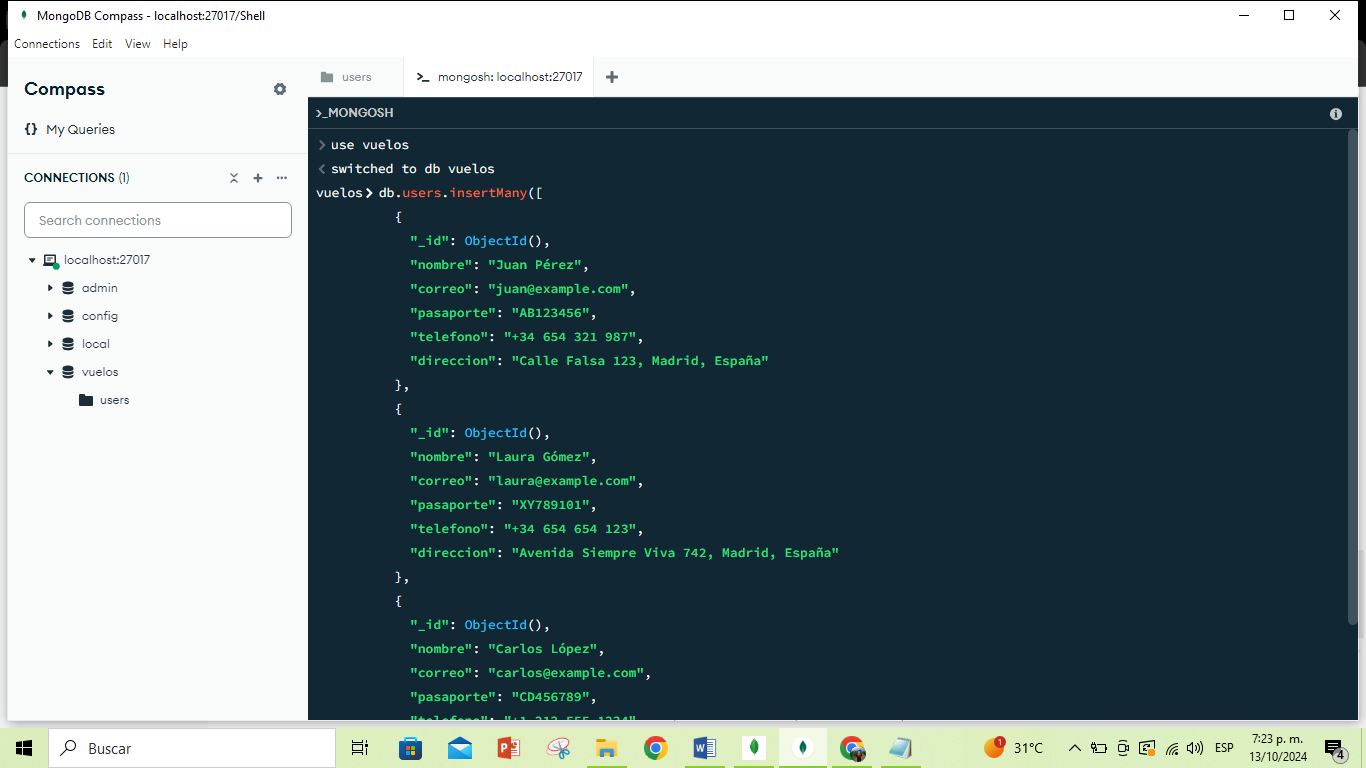
1. **Consideraciones de Diseño**

* **Relaciones entre colecciones**: Las relaciones se establecen utilizando referencias a través de los IDs de los documentos. Esto asegura que los datos se mantengan separados y que puedan ser accedidos de manera eficiente mediante consultas.
* **Definición de campos comunes**: Los campos como usuario\_id y vuelo\_id permiten conectar usuarios y vuelos a través de las reservas, asegurando una relación clara y concisa entre las colecciones.

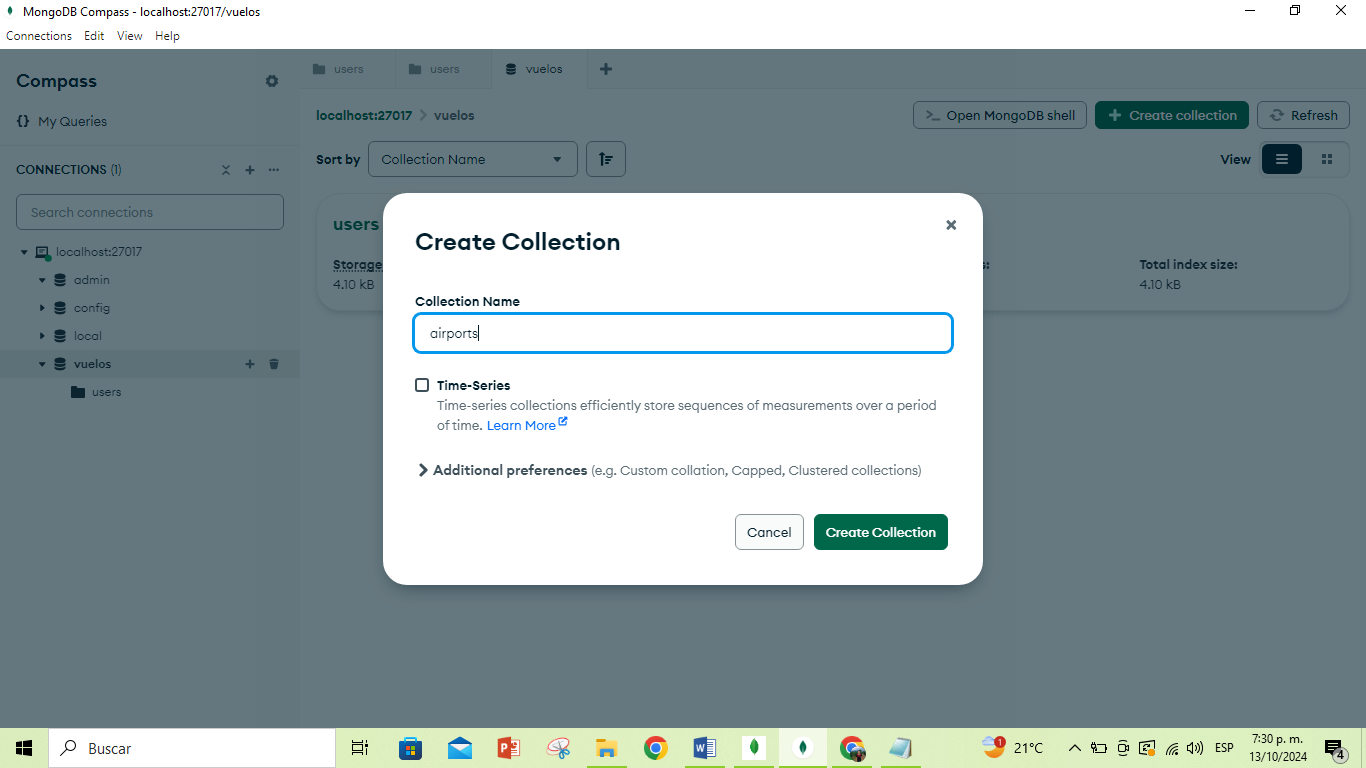
**Creación de base de datos**



**Ceración de la collection users y sus datos**



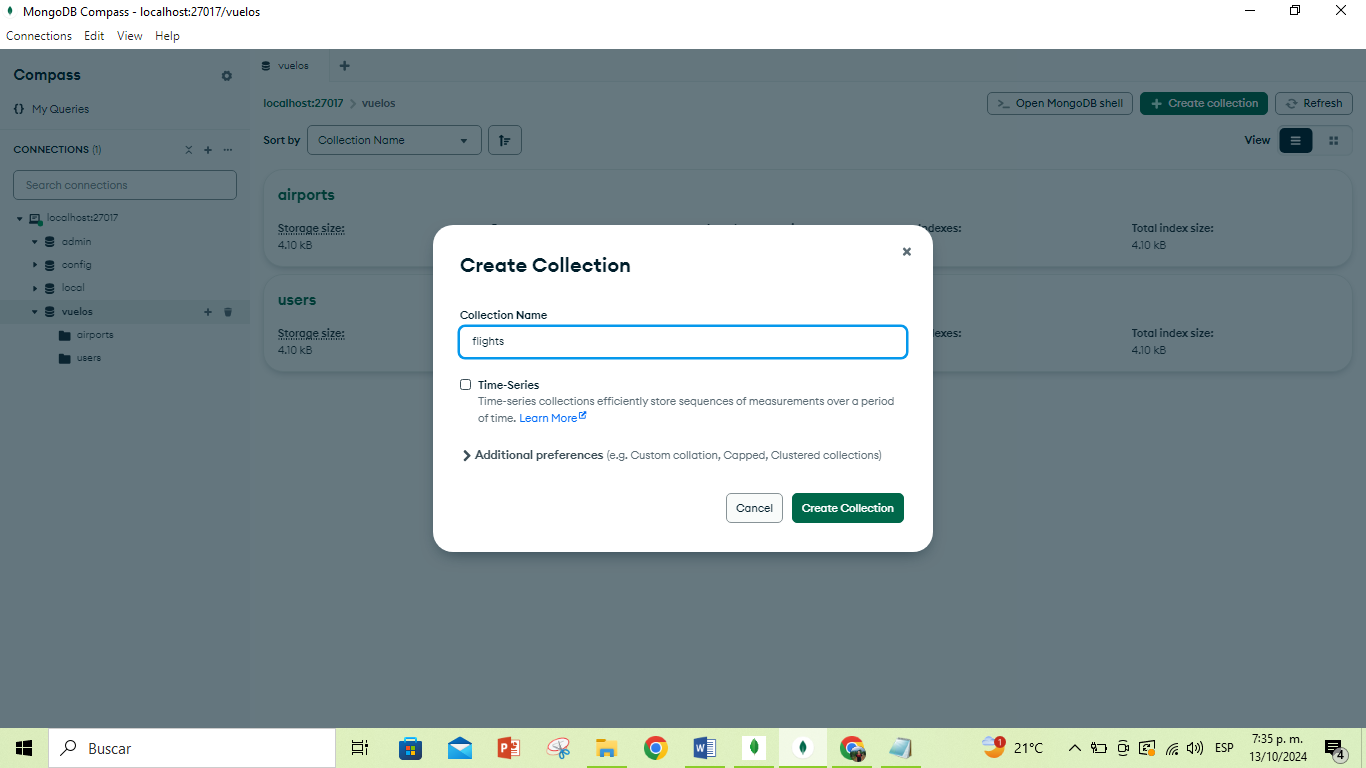
**Creación de la colecction airports**



**Creación y llenado de datos de la collection airports**



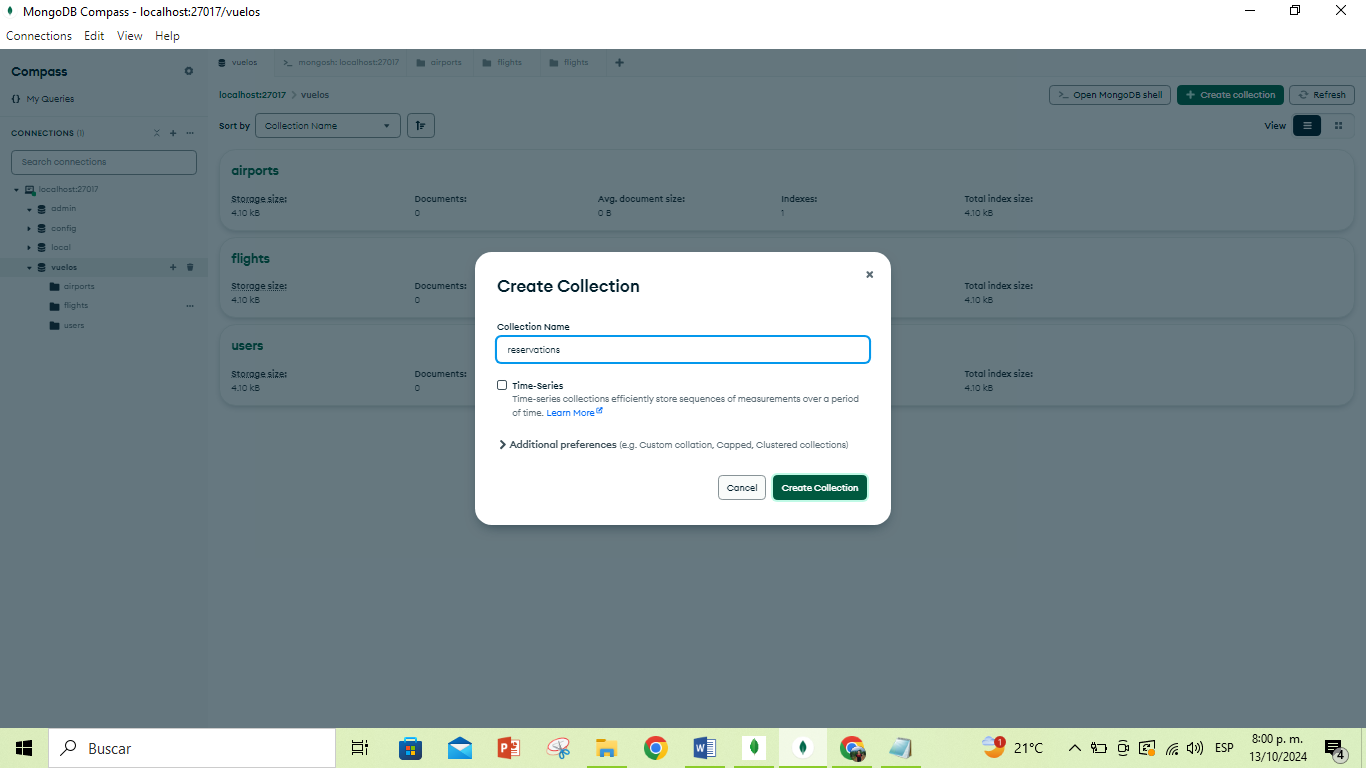
**Creación de la collection fligths**



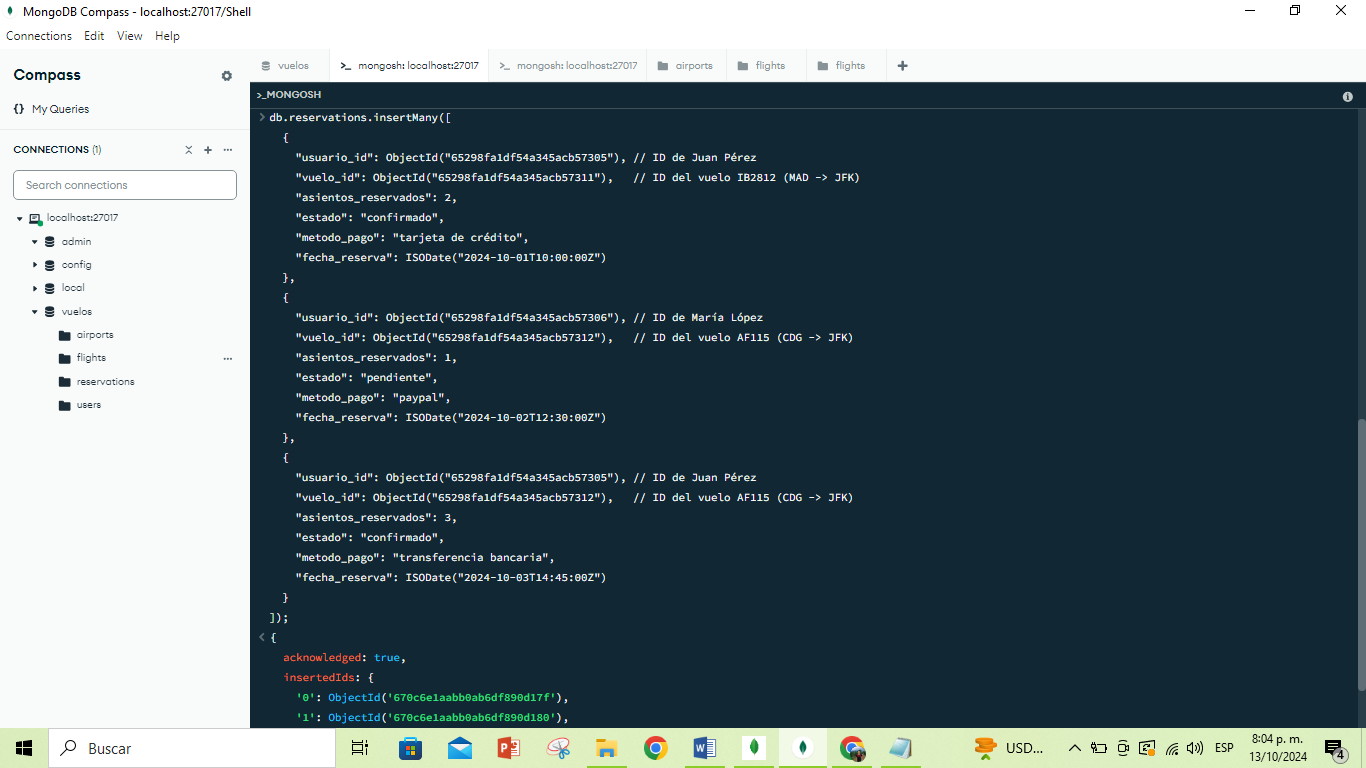
**Creación y llenado de datos de la colección flights**



**Creación de la collection reservations**



**Creación y llenado de datos de la colección reservations**

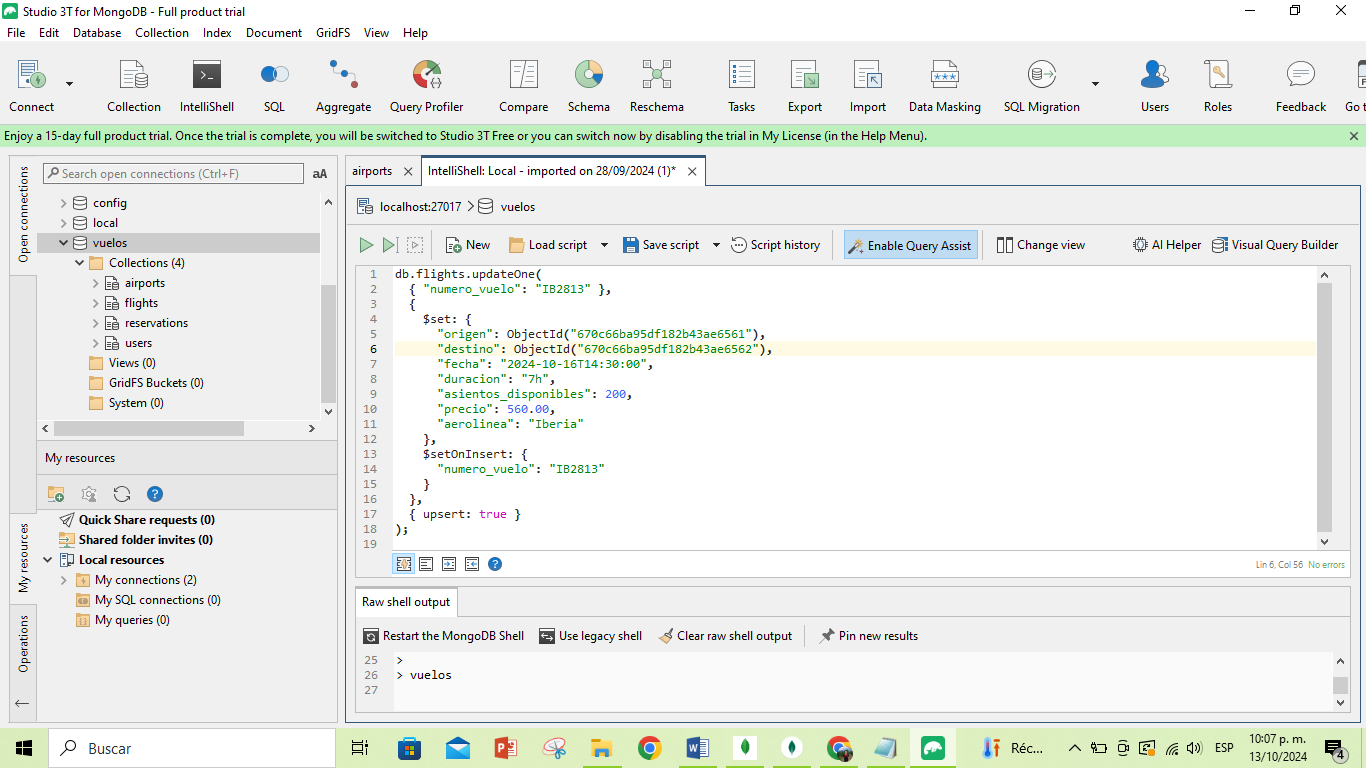


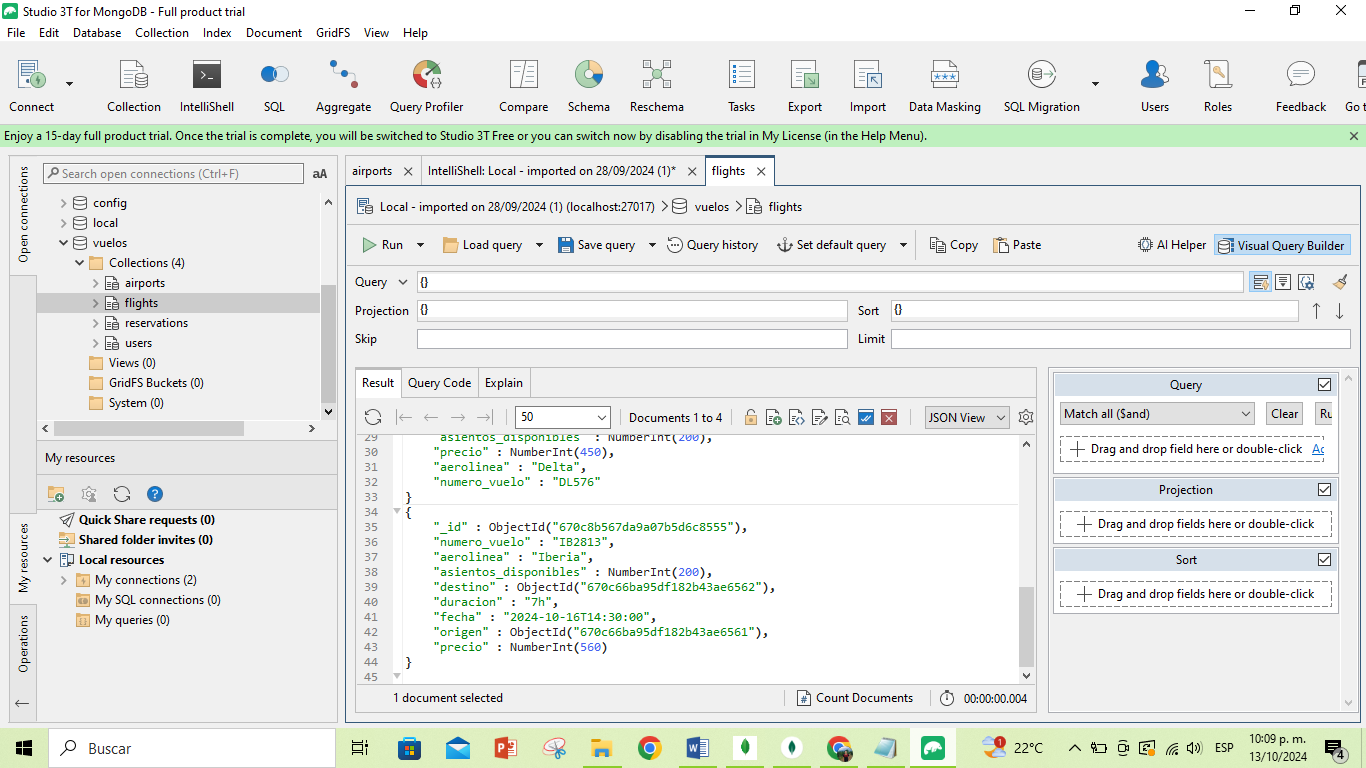
## **Nuevas Funciones de Actualización**

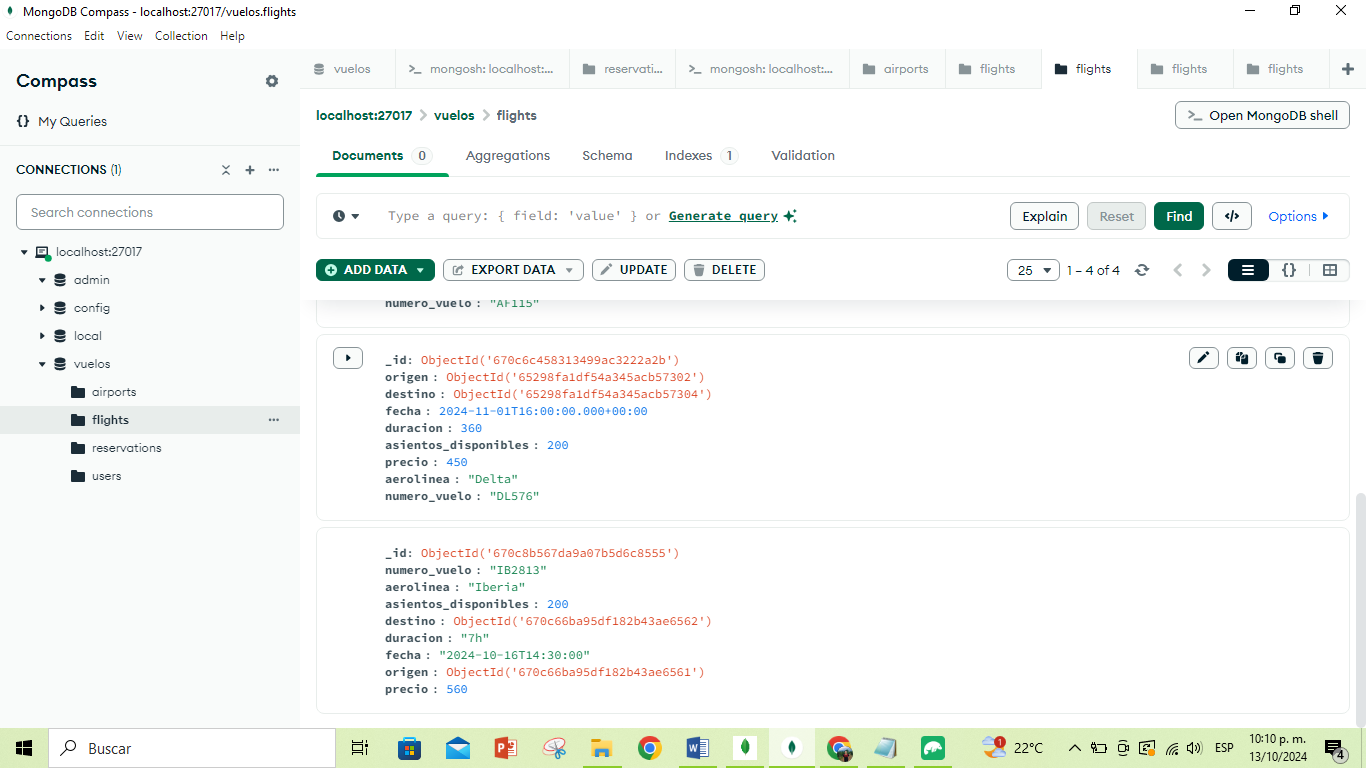
**1. Agregar un nuevo vuelo usando upsert**

Si queremos agregar un nuevo vuelo o actualizar uno existente, podemos hacerlo con **upsert.**

Por ejemplo, aquí estamos insertando un nuevo vuelo

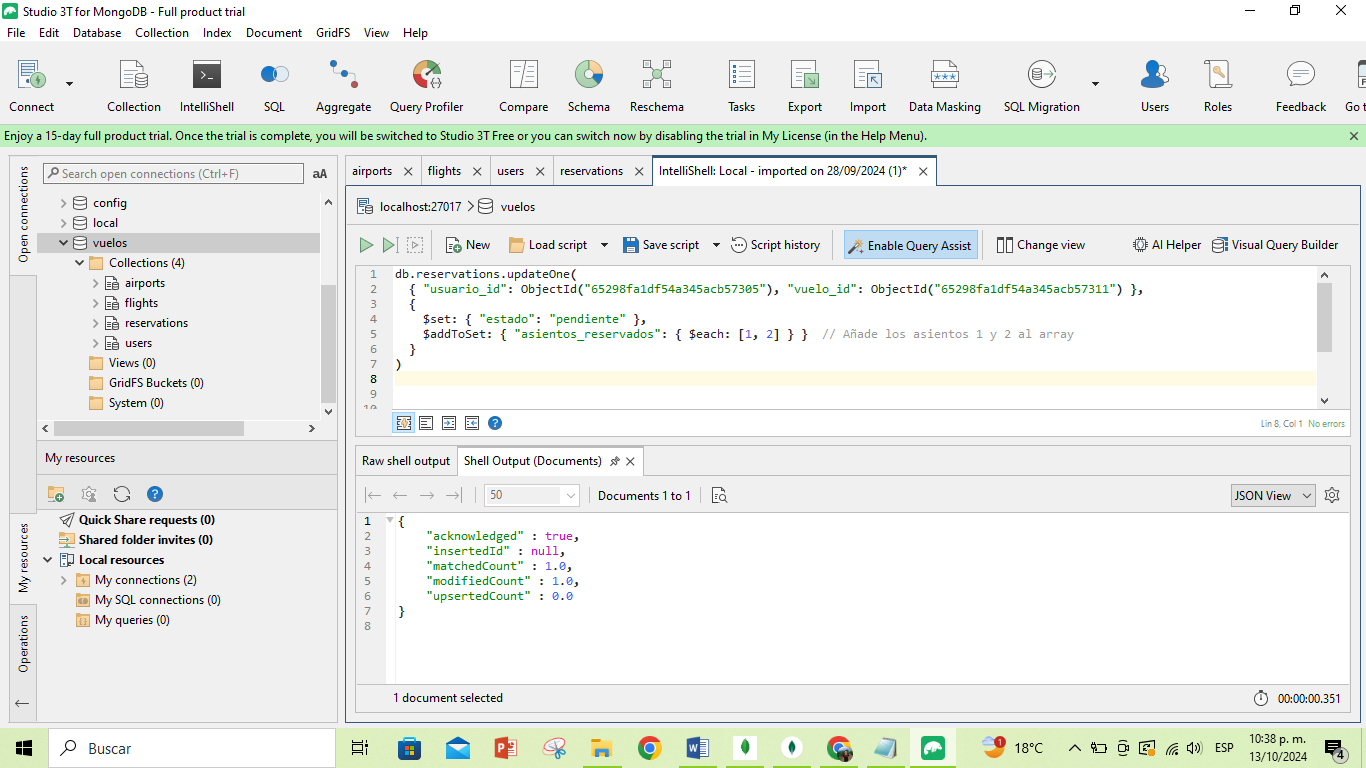


Aquí ya tenemos nuestro nuevo vuelo:

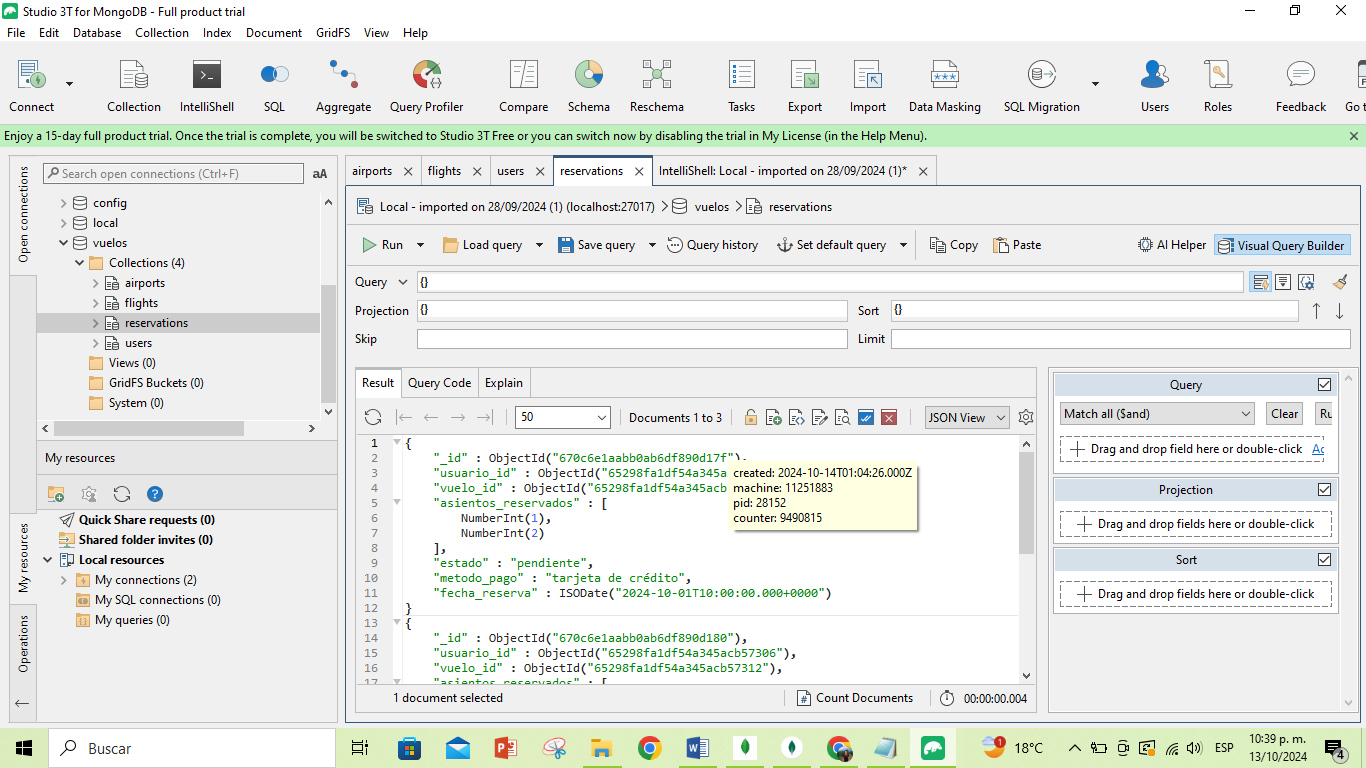


**2. Actualizar las reservas de un usuario usando each**

Si un usuario reserva varios asientos en un vuelo, podemos actualizar la reserva usando $each para agregar múltiples asientos.



Y aquí tenemos nuestra actualización de reservación de vuelo



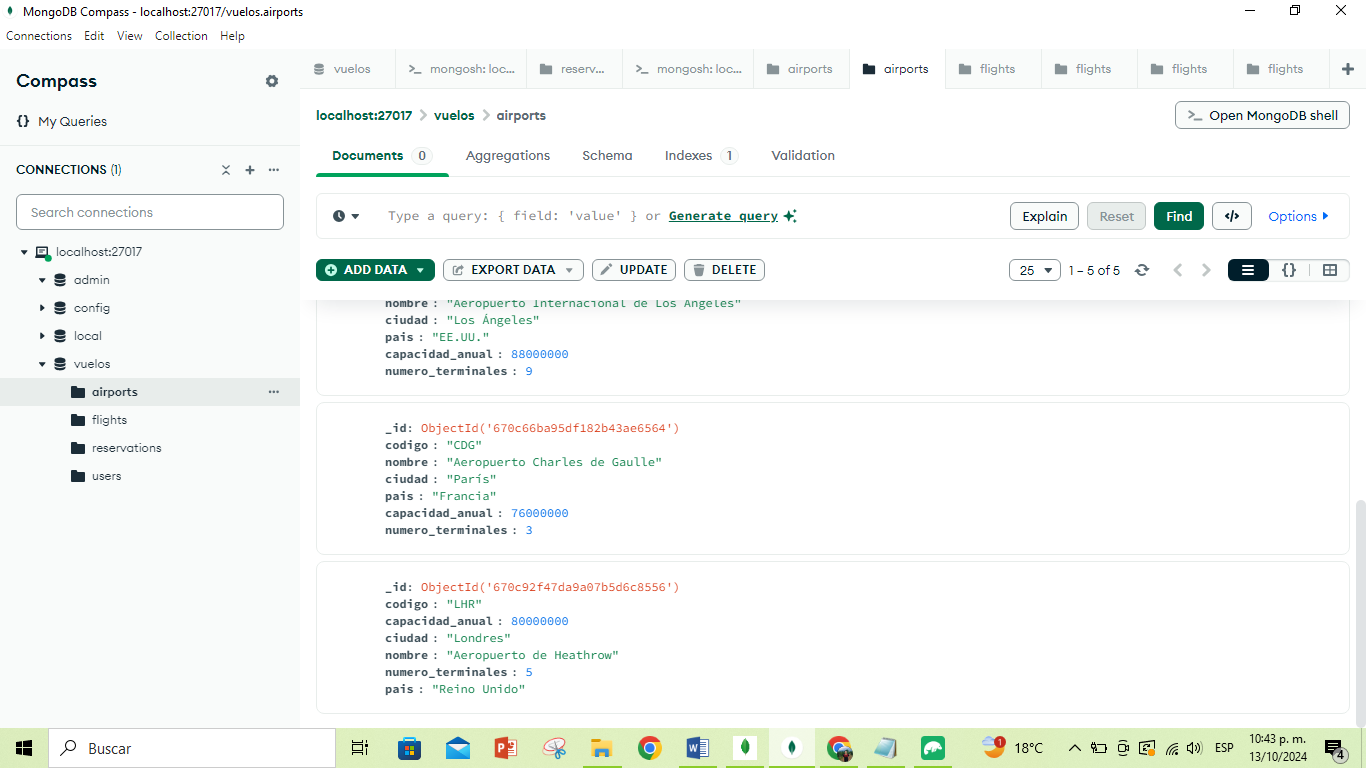
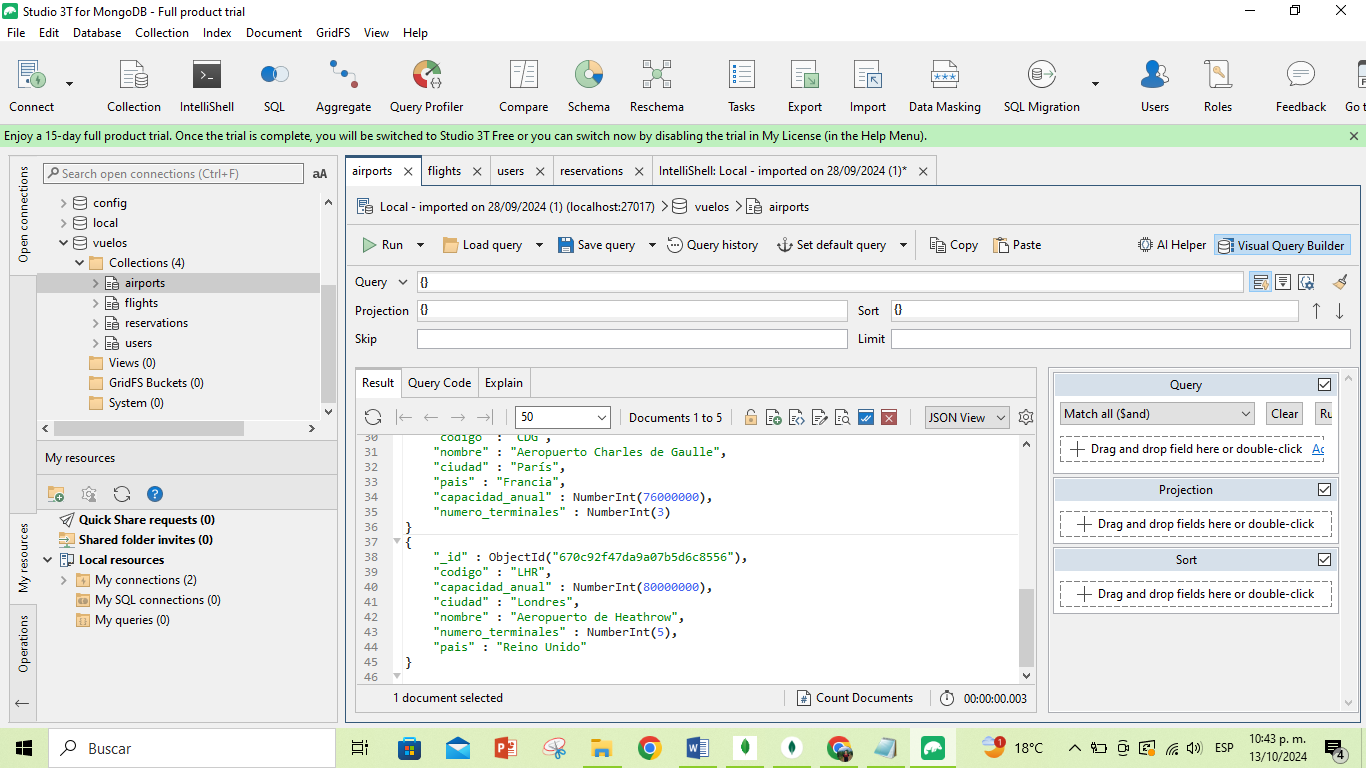
Lo que nos genera aquí es la actualización de vuelos donde dentro de un array podemos obtener muchos vuelos

**3. Actualizar la capacidad de un aeropuerto usando setOnInsert**

Para actualizar la capacidad de un aeropuerto solo si es un nuevo aeropuerto, podemos usar setOnInsert.



Aquí generamos la actualización de un nuevo aeropuerto



Con este diseño ampliado, puedes manejar un sistema más robusto de reservas de vuelos con más datos y funcionalidades realistas. Las relaciones entre colecciones siguen reflejando las necesidades del sistema, y las funciones de actualización permiten mantener la base de datos dinámica y actualizada.

## **Relaciones entre colecciones**

**1. Relación uno a uno (1:1)**

* **Usuario - Pasaporte**: Cada usuario tiene un solo pasaporte, que es un atributo del usuario.
  + **Colección involucrada**: users
  + **Estructura**:
    - Un campo pasaporte está dentro de la colección users.

{

"\_id": ObjectId(),

"nombre": "Juan Pérez",

"pasaporte": "AB123456"

}

**2. Relación uno a muchos (1 - m)**

* **Vuelo - Reservas**: Un vuelo puede tener muchas reservas asociadas. Un usuario puede reservar más de un asiento en el mismo vuelo.
  + **Colecciones involucradas**: flights, reservations
  + **Estructura**:
    - Cada documento en la colección reservations contiene referencias a usuario\_id y vuelo\_id.

{

"\_id": ObjectId(),

"origen": ObjectId("..."), // ID del aeropuerto MAD

"destino": ObjectId("..."), // ID del aeropuerto JFK

"fecha": "2024-10-15T14:30:00"

}

**3. Relación muchos a muchos (M-M)**

* **Usuario - Vuelos (a través de Reservas)**: Un usuario puede reservar múltiples vuelos, y un vuelo puede tener múltiples usuarios.
  + **Colecciones involucradas**: users, flights, reservations
  + **Estructura**:
    - La colección reservations sirve de tabla intermedia (relación M-M) que conecta usuarios y vuelos con los campos usuario\_id y vuelo\_id.

{

"\_id": ObjectId(),

"usuario\_id": ObjectId("..."), // ID de Juan Pérez

"vuelo\_id": ObjectId("..."), // ID del vuelo IB2812

"estado": "confirmado"

}

## **Resumen de las relaciones:**

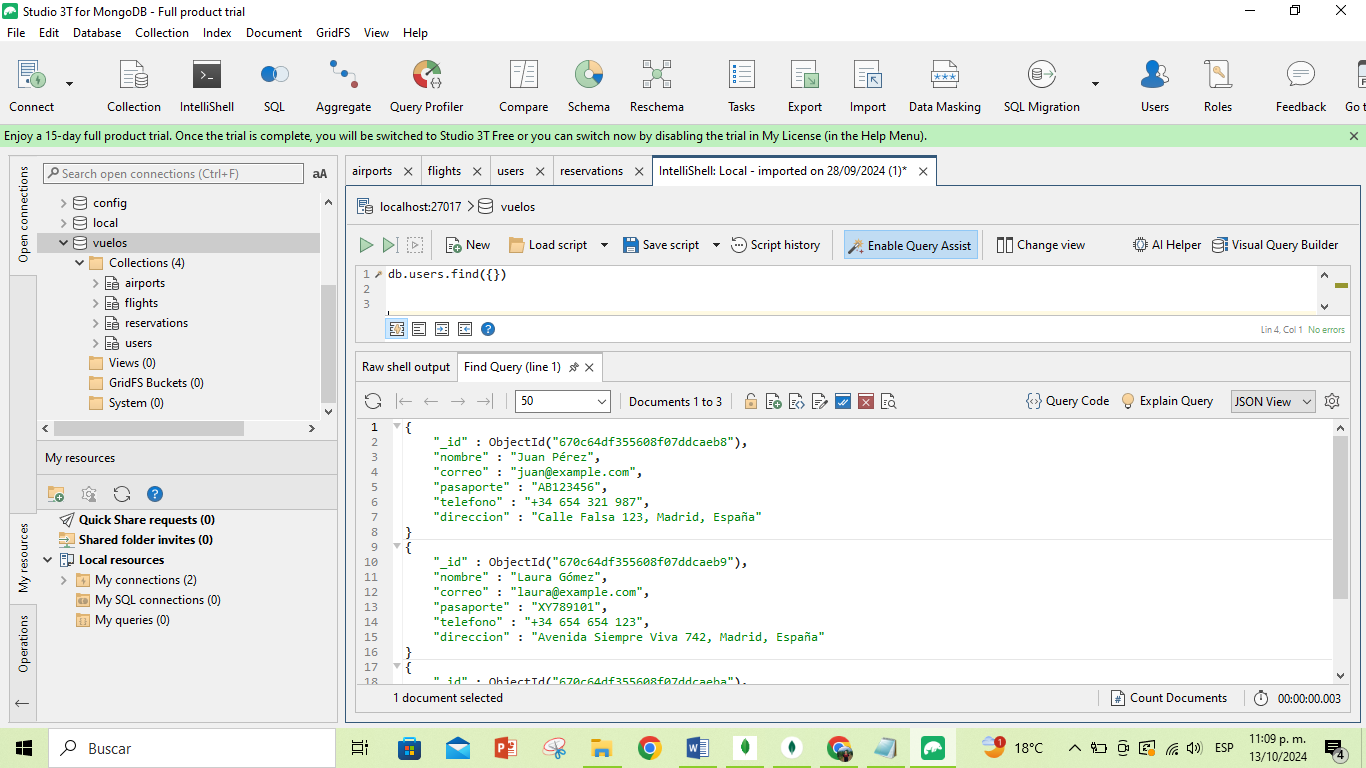
1. **Uno a Uno (1:1)**:
   * Usuario - Pasaporte (Dentro de la colección users)
2. **Uno a Muchos (1-M)**
   * Vuelo - Reservas (Colecciones flights, reservations)
   * Aeropuerto - Vuelos (Colecciones airports, flights)
3. **Muchos a Muchos (M-M)**:
   * Usuario - Vuelos (a través de reservations que conecta users y flights)

Estas relaciones permiten un diseño flexible en MongoDB, permitiendo un fácil escalado y adaptabilidad al sistema de reservas de vuelos.

# **CONSULTAS**

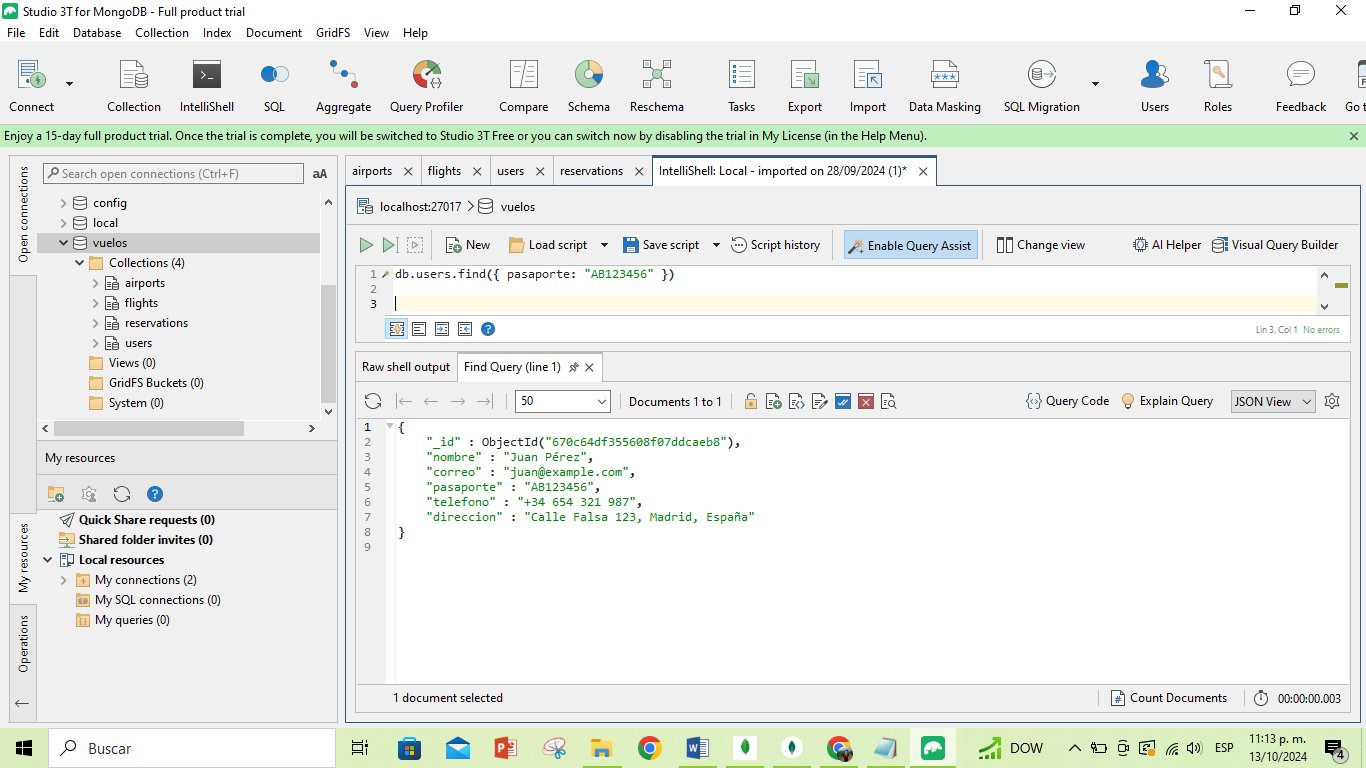
* 1. **Buscar todos los usuarios**

**db.users.find({})**



* 1. **Buscar un usuario por su pasaporte**

**db.users.find({ pasaporte: "AB123456" })**



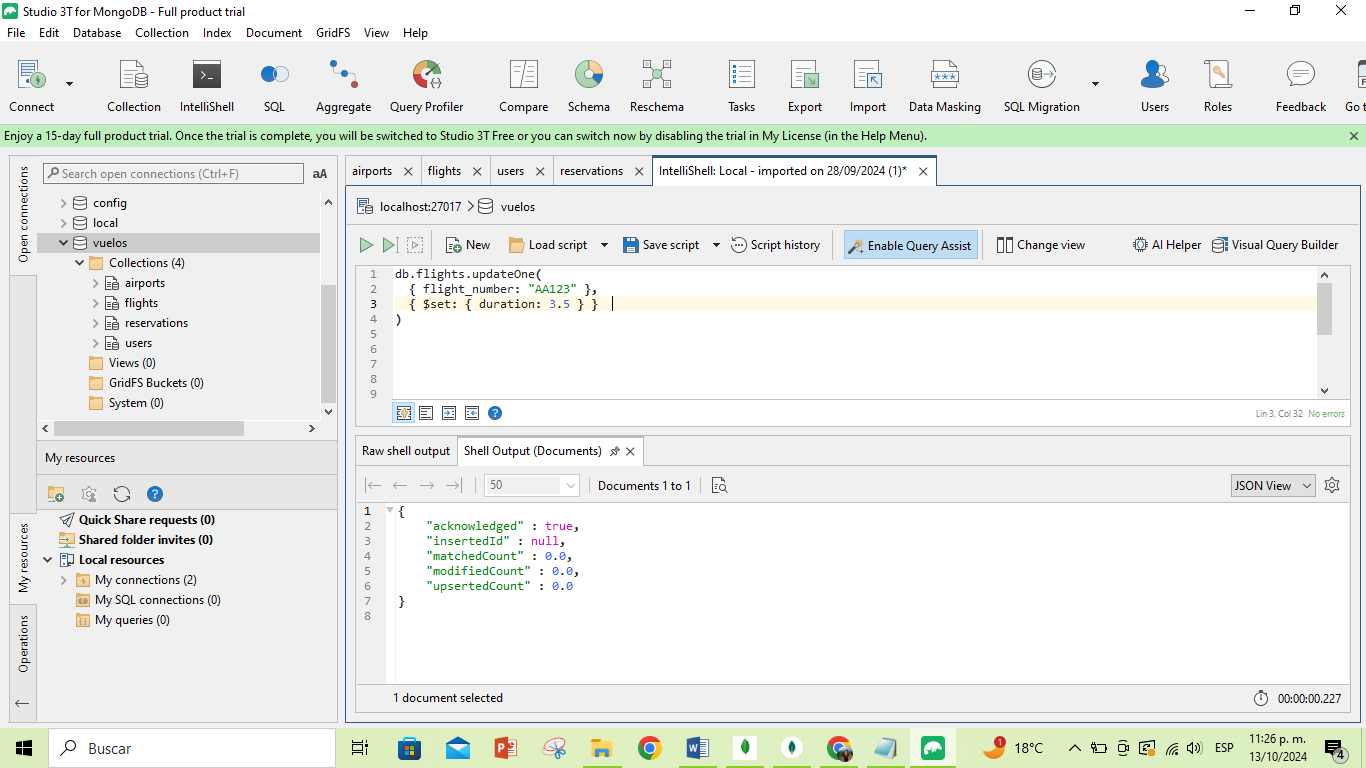
1. Actualizar la duración de un vuelo:

**db.flights.updateOne(**

**{ flight\_number: "AA123" },**

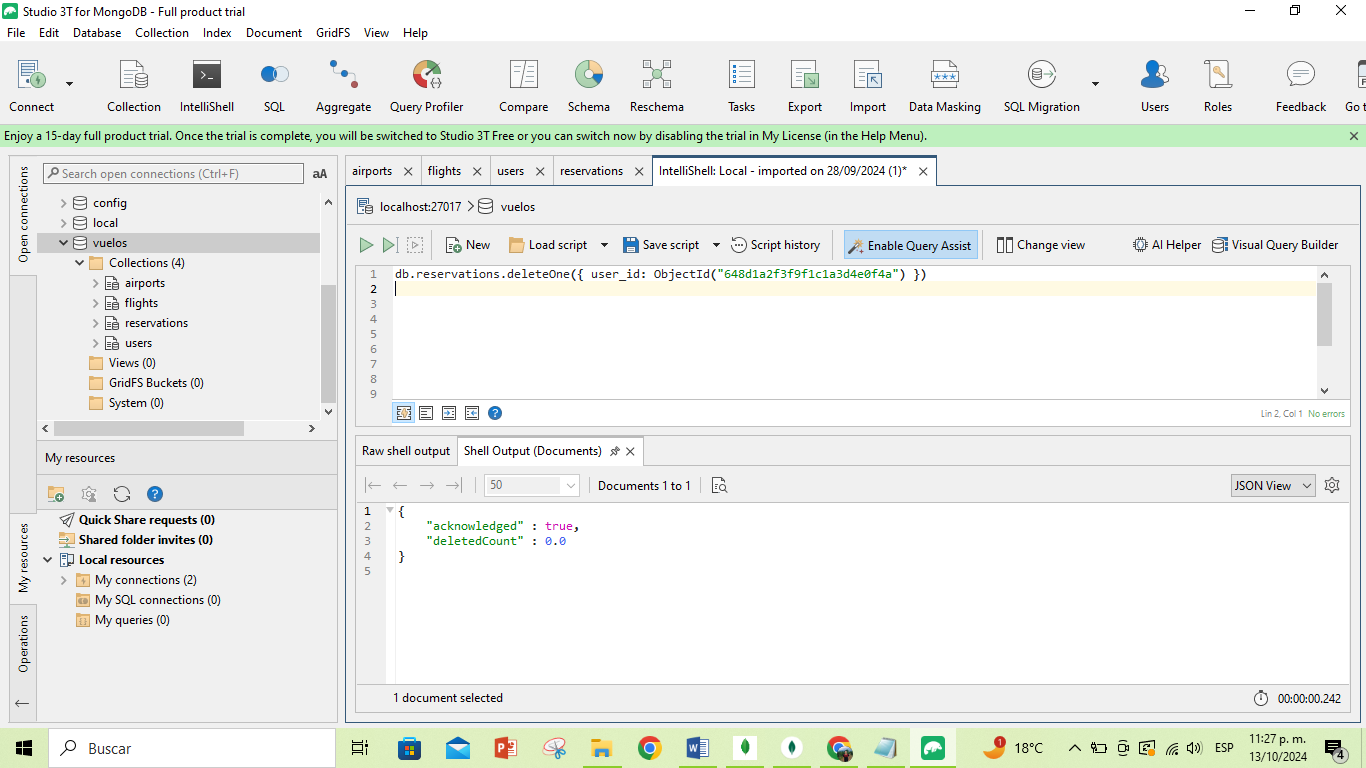
**{ $set: { duration: 3.5 } } // Cambiando la duración a 3.5 horas**

**)**



1. Eliminar una reserva por ID de usuario:

**db.reservations.deleteOne({ user\_id: ObjectId("648d1a2f3f9f1c1a3d4e0f4a") })**



1. Insertar una nueva reserva:

**db.reservations.insertOne({**

**user\_id: ObjectId("648d1a2f3f9f1c1a3d4e0f4a"),**

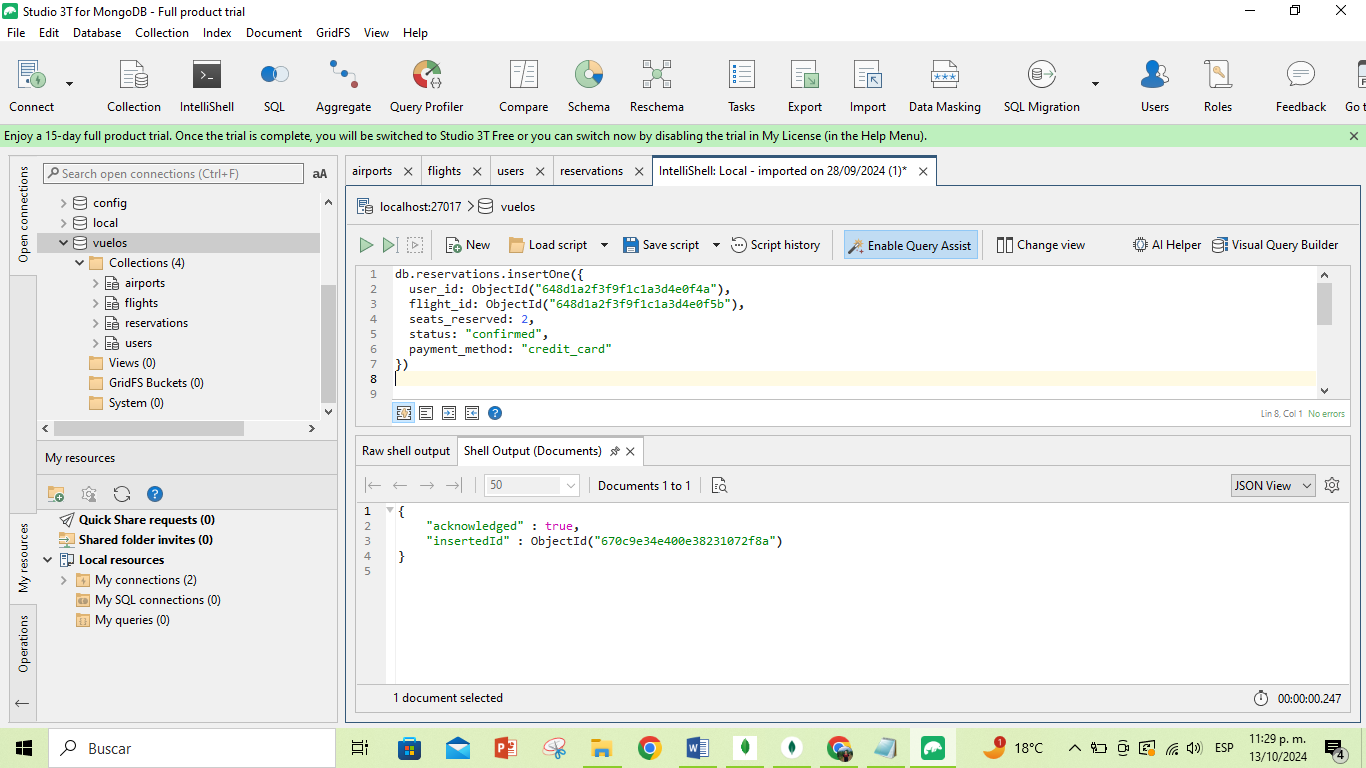
**flight\_id: ObjectId("648d1a2f3f9f1c1a3d4e0f5b"),**

**seats\_reserved: 2,**

**status: "confirmed",**

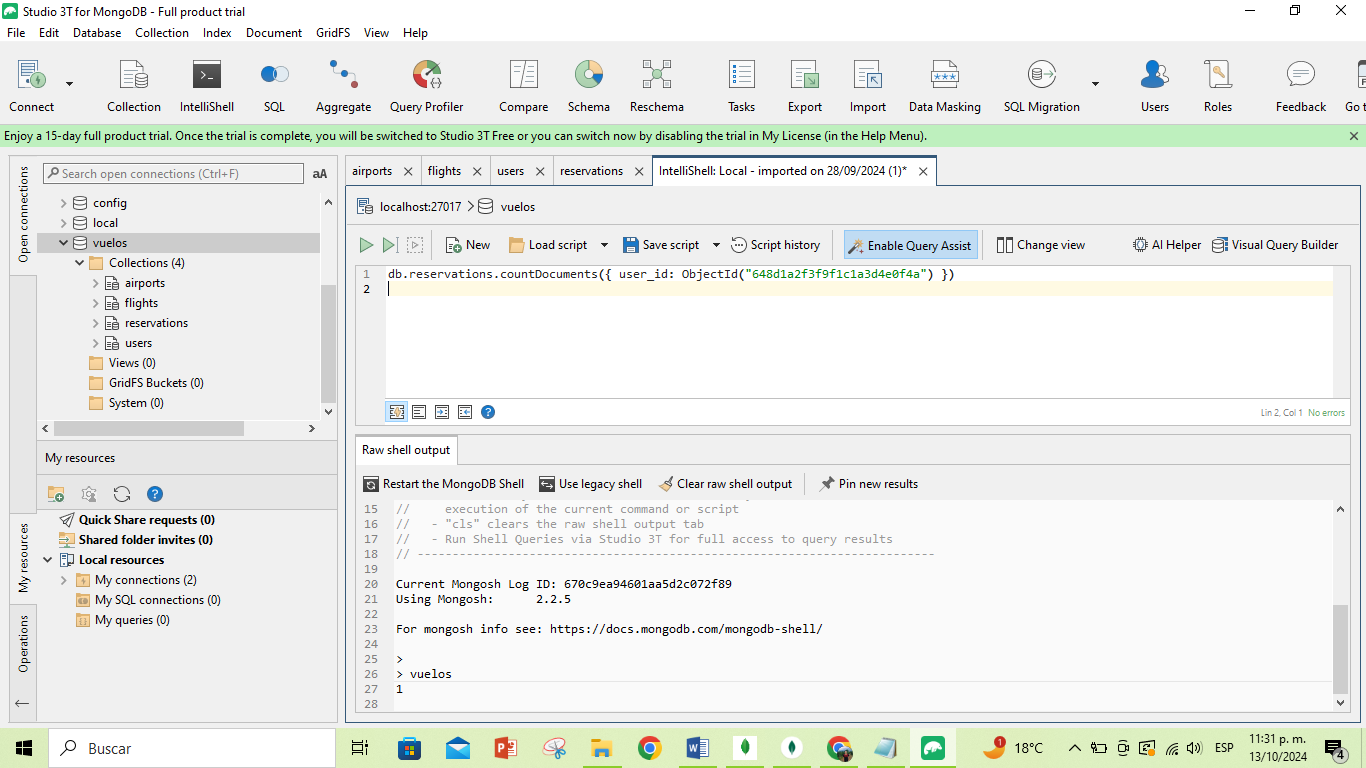
**payment\_method: "credit\_card"**

**})**



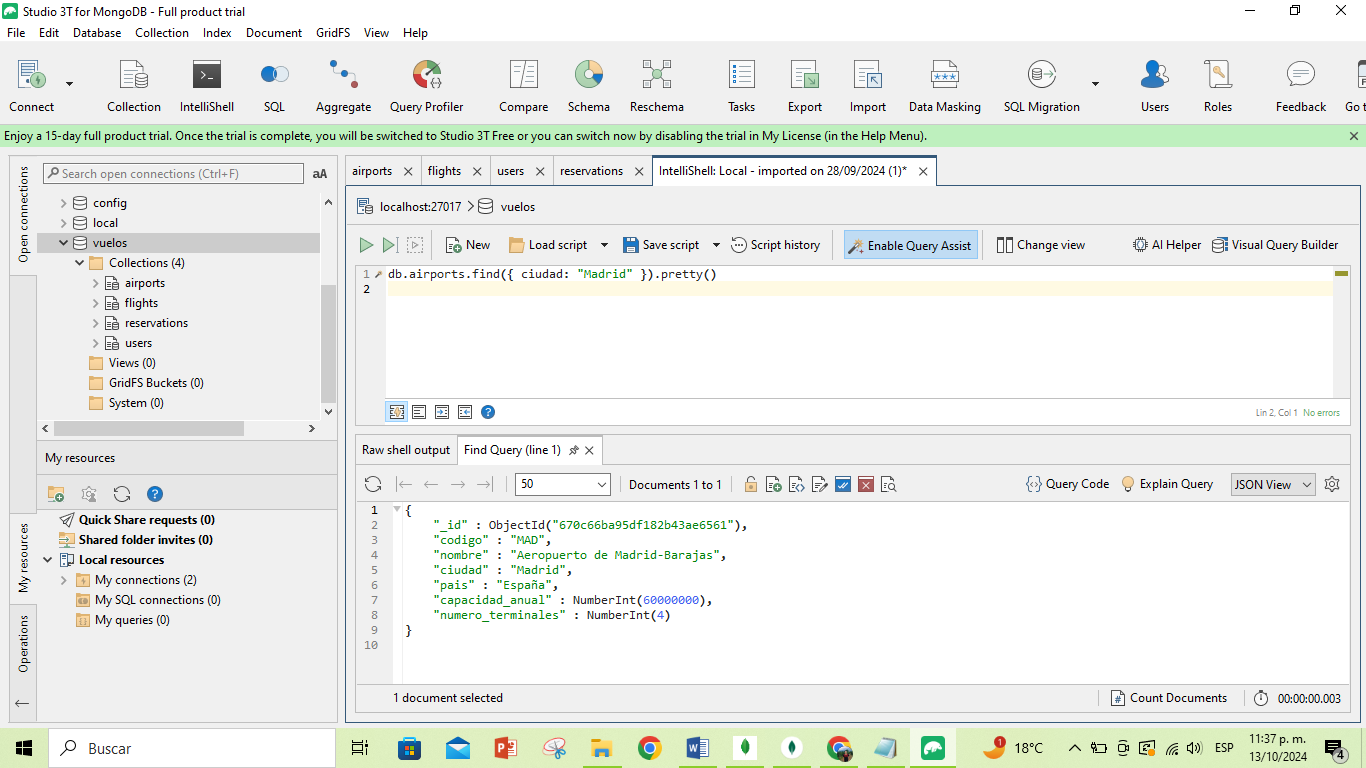
1. Contar cuántas reservas tiene un usuario específico:

**db.reservations.countDocuments({user\_id:ObjectId("648d1a2f3f9f1c1a3d4e0f4a") })**



1. Mostrar todos los aeropuertos en una ciudad específica (ej. paris):

**db.airports.find({ ciudad: "Madrid" }).pretty()**



# **DIAGRAMAS DE LAS COLECCIONES**

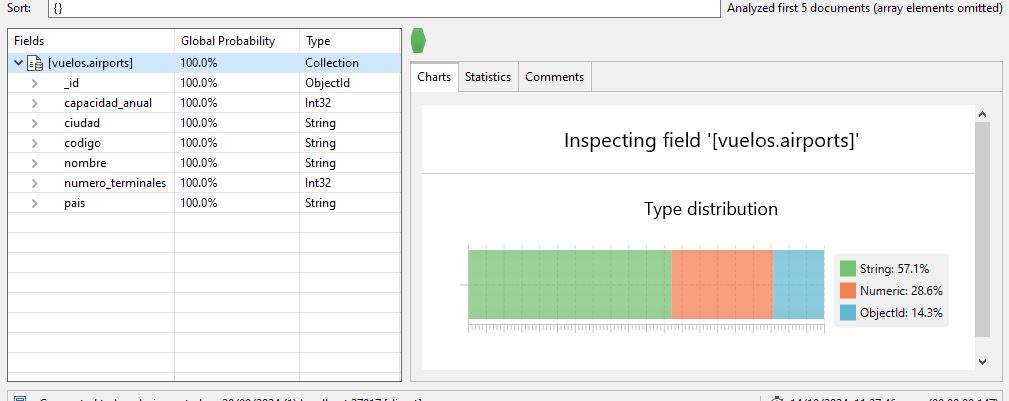
****

Diagrama de la colección airports representa el porcentaje de strings que manejamos en la colección al igual que el porcentaje de datos numéricos y datos tipo objectid.

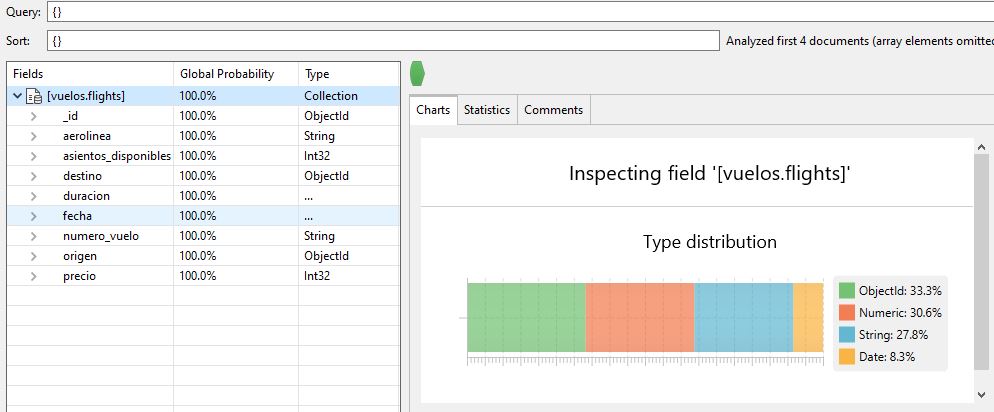
****

Diagrama de la colección flights representa el porcentaje de datos tipo strings que manejamos en la colección al igual que el porcentaje de tipo de datos numéricos, los datos tipo objectid y los tipos de datos de fechas en este caso date.

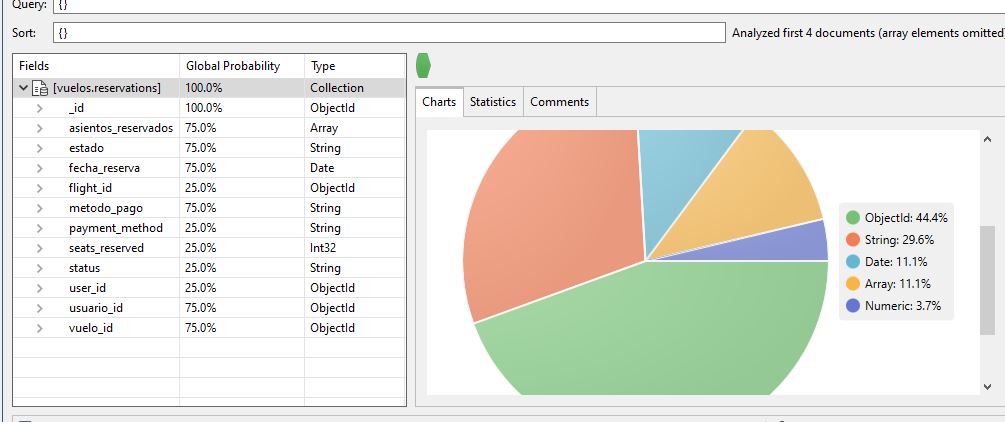
****

Diagrama de la colección reservations representa el porcentaje de datos tipo strings que manejamos en la colección al igual que el porcentaje de tipo de datos numéricos, los datos tipo objectid, los tipos de datos de fechas en este caso date y manejamos un tipo de datos array.

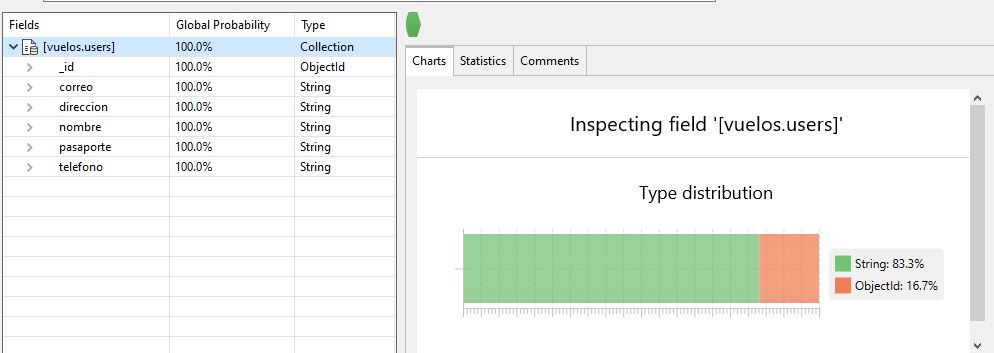
****

Diagrama de la colección users representa el porcentaje de strings que manejamos en la colección al igual que el porcentaje de datos tipo objectid.

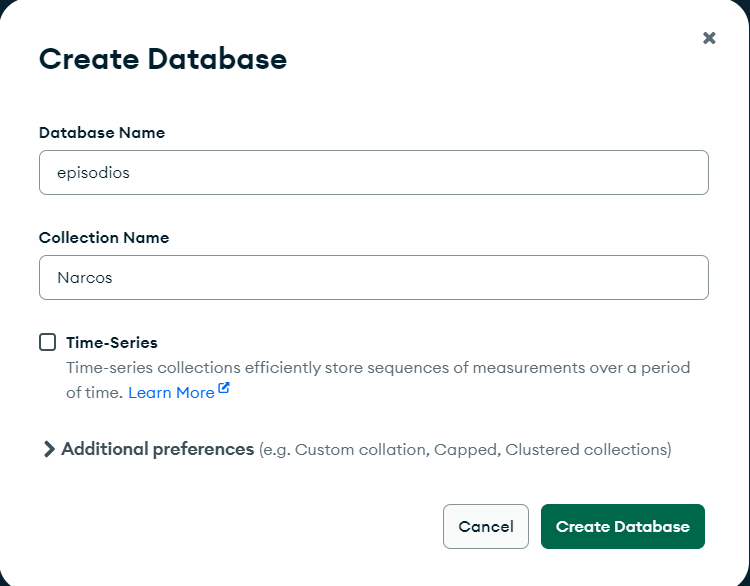
# ***Interpretación de Resultados***

Los resultados obtenidos muestran que el sistema diseñado es capaz de manejar grandes volúmenes de datos, con actualizaciones eficientes a través de métodos como upsert, setOnInsert, y la capacidad de manejar relaciones complejas. Las pruebas de consulta realizadas confirmaron que el esquema está bien estructurado para consultas rápidas y actualizaciones simultáneas.

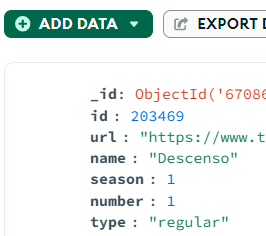
. **CONSULTAS TALLER CRUD**

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



Después importamos el fichero y asi tendremos cargados los datos



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

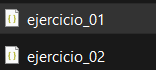
Luego repetir el mismo proceso con los demás ficheros

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

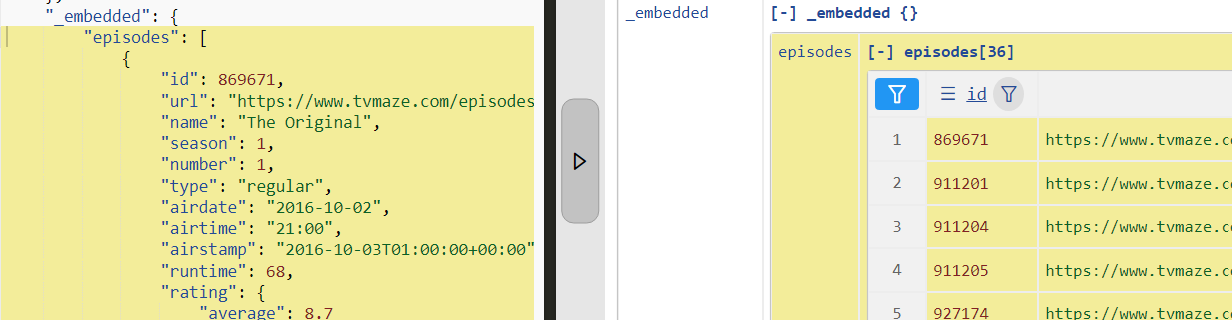
“ejercicio-03.json” en la colección Homeland.

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

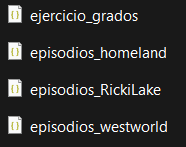
Primero visualizamos los datos de los ficheros



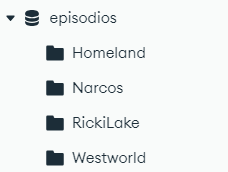
Usando la herramienta jsongrid podremos acceder a los datos y extraerlos



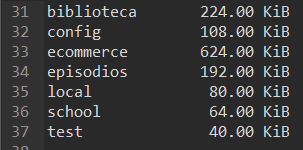
Así crearemos los nuevos archivos json



E importaremos los datos a cada una de las colecciones creadas



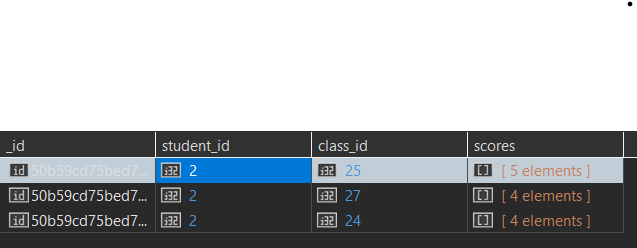
Visualizar todas las bases de datos existentes: show dbs



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 } )**



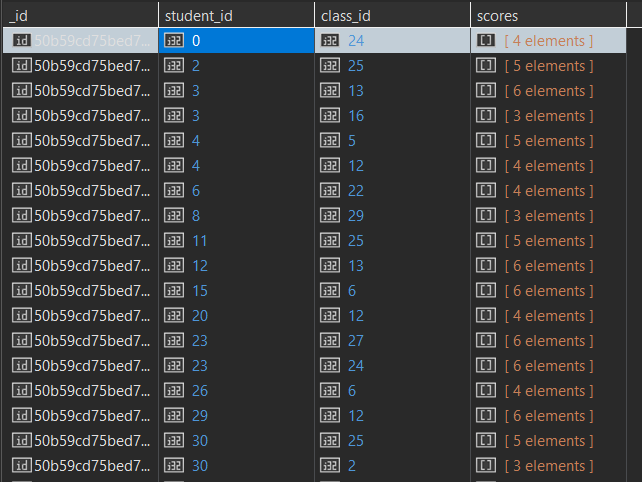
La consulta db.grades.find({student\_id: 2}) en MongoDB busca todos los documentos dentro de la colección grades que tengan un campo student\_id con el valor 2.

**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 } } )**



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:**

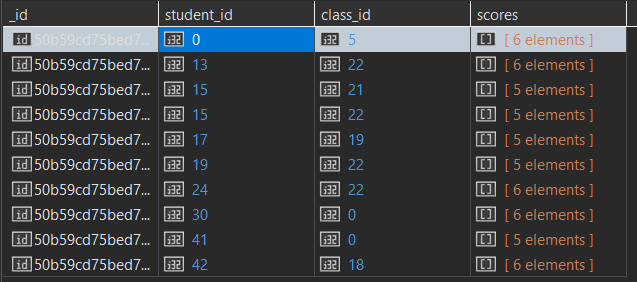
**db.grades**: Se refiere a la colección **grades** en la base de datos actual.

**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 } } )**



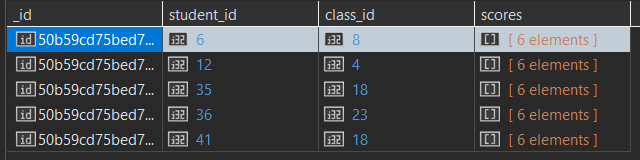
La consulta db.grades.find({ "scores.4.score": { $lte: 10 } }) en MongoDB busca todos los documentos en la colección **grades** donde el **quinto elemento** (índice 4) del array **scores** tenga un campo **score** con un valor menor o igual a 10.

**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 } } )**



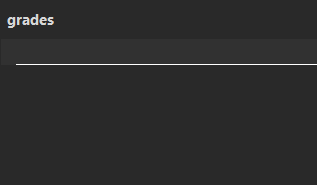
La consulta db.grades.find({ "scores.5.score": { $lte: 10 } }) busca todos los documentos en la colección **grades** donde el **sexto elemento** (índice 5) del array **scores** tenga un campo **score** con un valor menor o igual a 10.

**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 } } )**



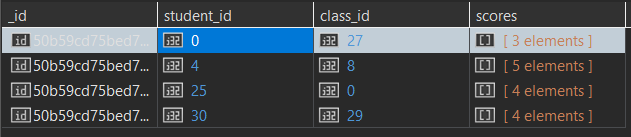
La consulta db.grades.find({ "scores.6.score": { $lte: 10 } }) busca todos los documentos en la colección **grades** donde el **séptimo elemento** (índice 6) del array **scores** tenga un campo **score** con un valor menor o igual a 10.

**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 } } )**



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.

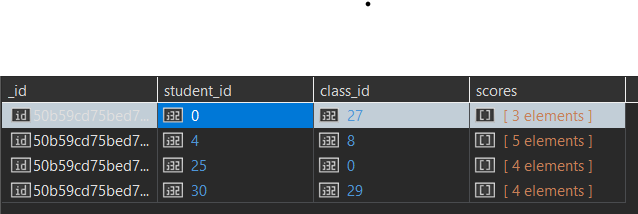
**$gte: 60**: El operador $gte significa "mayor o igual que", por lo que está buscando un valor que sea al menos 60.

**$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} )**

****

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 } )**



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:**

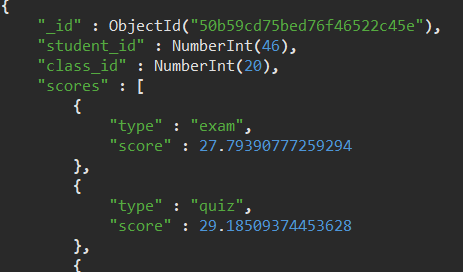
**student\_id: 2**: Filtra los documentos donde el valor de student\_id es exactamente 2.

**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 } } ] } )**





**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.

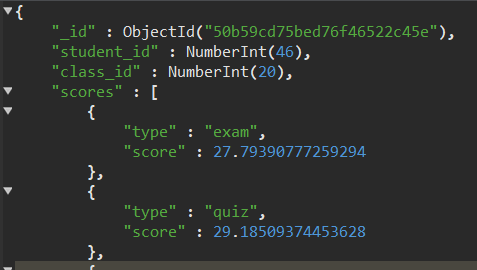
**scores.0.score: { $gte: 15 }**: Esta condición especifica que el campo score del primer elemento (índice 0) en el array scores debe ser **mayor o igual** a 15.

**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 } } } } )**





**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).

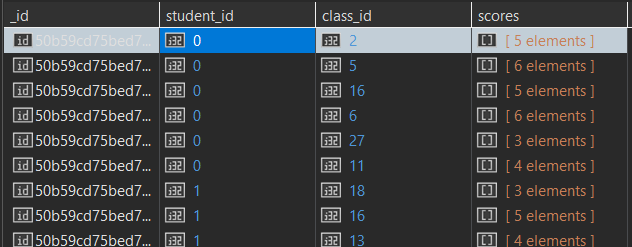
**Funcionamiento:**

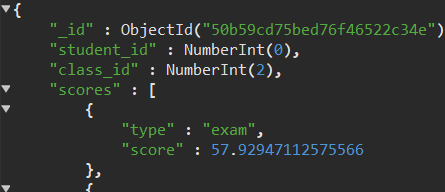
**scores: { $elemMatch: ... }**: El operador $elemMatch se utiliza para encontrar **elementos dentro de un array** que cumplan **todas las condiciones** especificadas. En este caso, busca un elemento en el array scores que tenga:

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 } } } } )**





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:

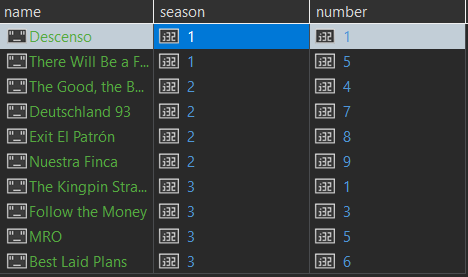
El campo type del elemento debe ser **'exam'**.

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 } )**





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

**Devuelve únicamente los campos**:

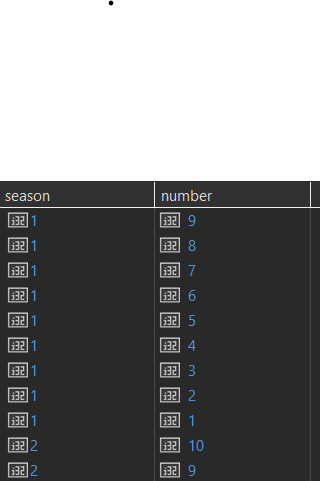
name: El nombre del episodio.

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 } )**



**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.

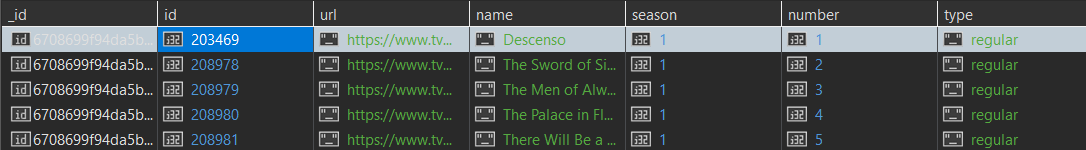
**No incluye el campo \_id** (ya que se ha establecido \_id: 0).

**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.

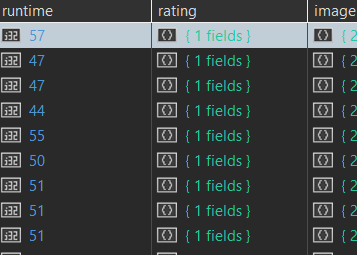
**db.Narcos.find( { season: { $type: 'number' } } )**



**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 } } )**



**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**:

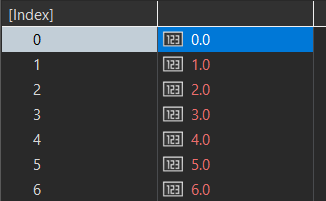
**Donde el campo rating existe** ({ rating: { $exists: 1 } }).

**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")**



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()**



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

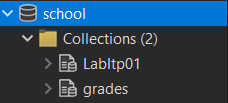
**Descripción de la Consulta**

**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

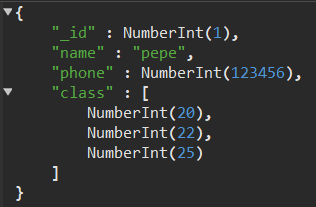
En la BD School crea la colección LabItp:

**db.createCollection("LabItp01")**



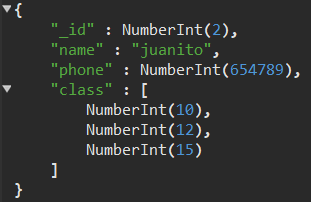
**db.LabItp01.insert( { \_id:1, name: "pepe", phone: 123456, class: [ 20, 22, 25 ] } )**

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.

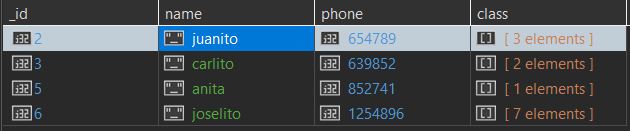


**db.LabItp01.insertMany( [ { \_id:3, name: "carlito", phone: 639852, class: [ 11, 10] }, { \_id:4, name: "camilito", phone: 741258, class: [ 15] }, { \_id:5, name: "anita", phone: 852741, class: [ 10] }, { \_id:5, name: "joselito", phone: 1254896, class: [ 55, 458, 236, 20, 22, 10, 15] } ] )**

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



**db.LabItp01.find( { class: 10 } )**



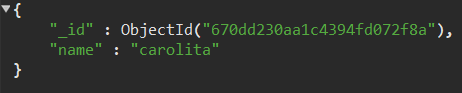
se utiliza para buscar documentos en la colección **LabItp01** que tengan el campo **class** igual a **10**. Aquí tienes un desglose de la consulta:

**Descripción de la Consulta**

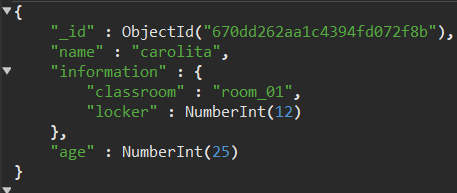
**db.LabItp01**: Esto se refiere a la colección llamada **LabItp01** en la base de datos en la que estás trabajando (en este caso, parece que estás usando la base de datos **school**).

**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**.

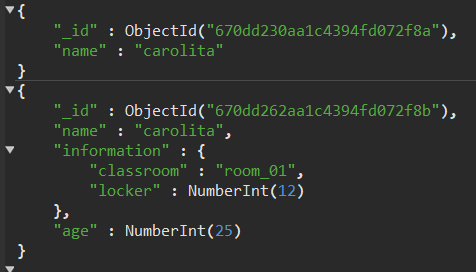
**db.LabItp02.insertOne( { name: "carolita" } )**



**db.LabItp02.insertOne( { name: "carolita", information: { classroom: "room\_01", locker: 12 }, age: 25 } )**



**db.LabItp02.find()**



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**

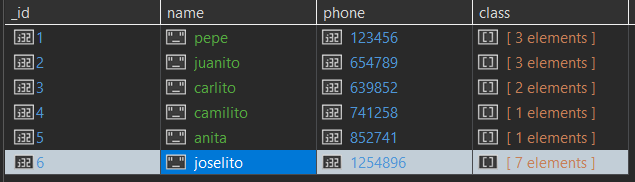
**db.LabItp02**: Esto se refiere a la colección llamada **LabItp02** en la base de datos que estás utilizando (en este caso, parece que estás trabajando en la base de datos **school**).

**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'] } } )**



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.

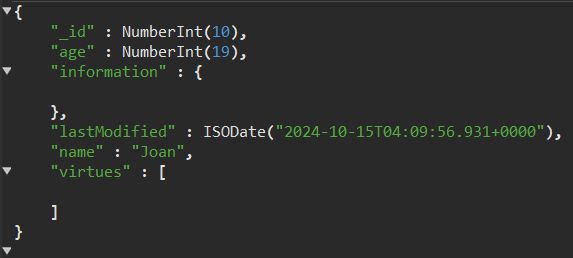
**Explicación:**

**{ \_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 } )**



**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:



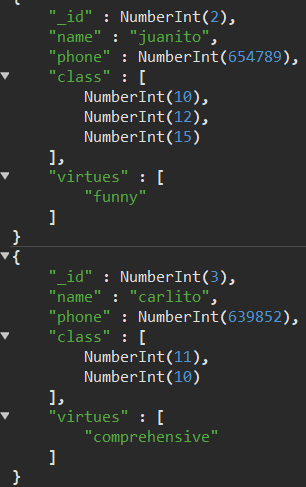
**Explicación:**

**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.

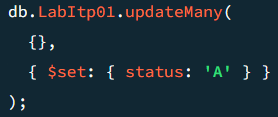
**filter**: Especifica el criterio para encontrar el documento (en este caso, el \_id).

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



**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:

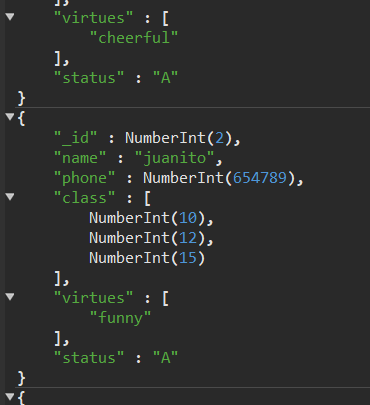


**Explicación:**

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

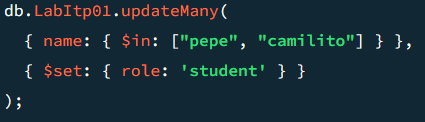
**{}**: El primer argumento vacío ({}) significa que se seleccionan todos los documentos de la colección.

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



**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:

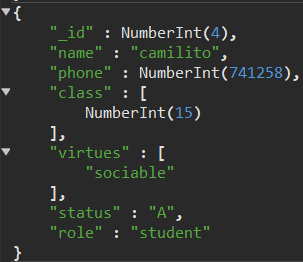
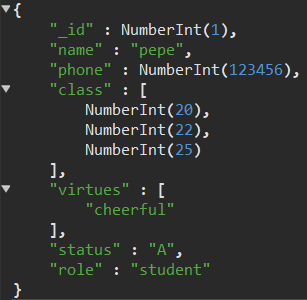


**Explicación:**

**{ 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.



**Trabaja con DELETE**

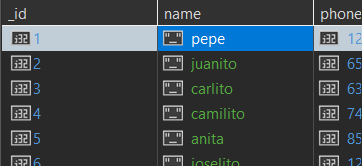
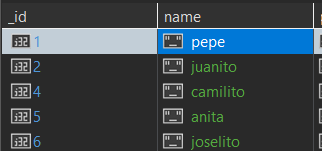
En la colección LabItp01 realiza las siguientes actualizaciones:

**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".

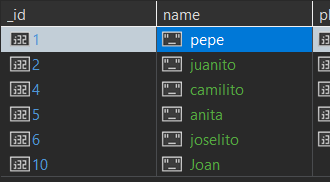
 

**db.grades.deleteOne( { student\_id: 0 } )**

**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.

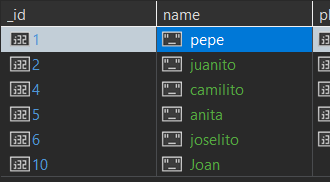


**db.grades.deleteMany( { student\_id: 0 } )**

**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.

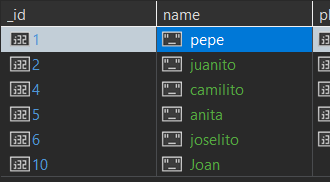


**db.grades.remove( { student\_id: 1 }, {justOne: true} )**

**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.



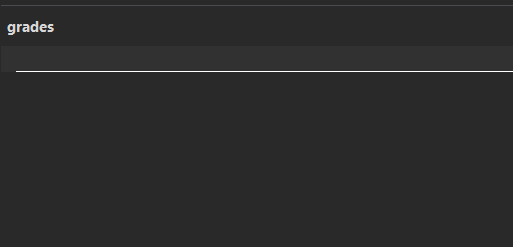
**db.grades.remove( { student\_id: 1 } )**

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( { } )**

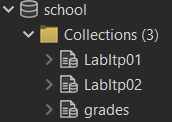
El comando eliminará **todos los documentos** de la colección grades. Básicamente, está indicando que no hay un filtro específico, por lo que MongoDB interpretará que debe eliminar cada documento de la colección.

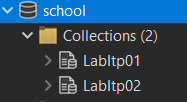




**db.grades.drop()**

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.





# **Conclusiones**

El diseño de la base de datos para el sistema de reservas de vuelos en MongoDB ha demostrado ser eficiente y escalable. Las decisiones de diseño, como el uso de referencias y la normalización de colecciones, han permitido una estructura flexible para manejar la interacción entre usuarios, vuelos, aeropuertos y reservas. Además, el uso de herramientas como Studio 3T y MongoDB Compass facilitó el análisis y la visualización del esquema.

# **Recomendaciones**

* Utilizar índices en las colecciones para mejorar aún más el rendimiento de las consultas.
* Continuar explorando las capacidades de agregación de MongoDB para generar informes más detallados.
* Considerar la replicación y particionamiento de la base de datos para garantizar alta disponibilidad y rendimiento en un entorno de producción a gran escala.

# **Referencias**

* MongoDB Documentation: <https://www.mongodb.com/docs/>
* Studio 3T Documentation: <https://studio3t.com/>
* Curso de MongoDB avanzado: <https://www.example.com>
* GitHub documentación: <https://github.com/gaviriaaldeir/CRUD_MONGO>
* JSONGRID Visualiza el contenido del fichero: <https://jsongrid.com/json-grid>