Bases de datos II

Bases de datos NoSQL - MongoDB

Laurence Thiago, Ignacio Traberg, Ulises Geymonat

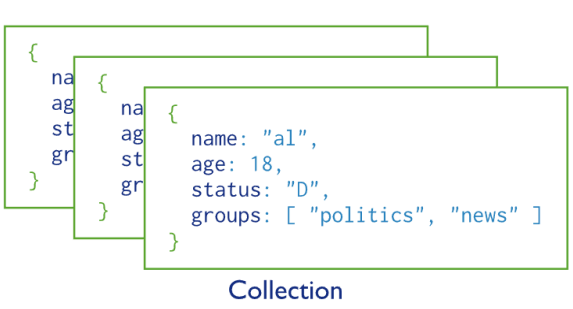
31 de mayo del 2024

Parte 1: Bases de datos NoSQL y Relacionales

1)

MongoDB DBMS NoSQL para bases de datos orientada a documentos, por lo que en base a esta premisa parten las siguiente diferencias y similitudes respecto a los RDBMS:

- Las **Tablas/Relaciones** pasan a ser **Colecciones**, donde estas corresponden a la recopilación de documentos. Por lo que una BD NoSQL almacena una o más colecciones de documentos.
- El almacenamiento de **Filas/Tuplas** pasan a ser **Documentos** denominamos objetos **BSON** (Binary JSON) los cuales se visualizan encerrados entre llaves “{ }”.
- Las **Columnas y Valores de las celdas** pasan a ser las **Claves y Valores** de los objetos BSON respectivamente, donde la Clave representa al atributo del objeto con su correspondiente asignación de valor.

Tablas → Colecciones	Tupla → BSON
	

2)

MongoDB permite la definición de **Claves foráneas** de dos formas:

- ***Las referencias manuales*** → Aquí al igual que las BD relacionales, se almacena una *Clave* denominada *_id<document>* dentro de un documento el cual almacena el valor del

ID de otro documento. Donde al momento de recuperar el 1er documento, la aplicación debe realizar una segunda consulta para recuperar el 2do documento relacionado.

- Estas permiten establecer referencias simples 1 a 1, 1 a N, N a N suficientes en la mayoría de los casos de usos.
  - En relaciones **1 a 1** alcanza con almacenar el valor del ID del documento relacionado en el campo *_id<document>*.
  - En relaciones **1 a N** o **N a N** alcanza con almacenar en un arreglo los valores de los IDs de los documentos relacionados.
- La única limitación es que al no transmitir el nombre de la BD ni de la colección con la cual se está estableciendo la relación, si el documento se relaciona con más de un documento en más de una colección es necesario el uso de *DBRefs*.

1 a 1	1 a N
`` original_id = ObjectId() db.places.insertOne({ "_id": original_id, "name": "Broadway Center", "url": "bc.example.net" }) db.people.insertOne({ "name": "Erin", "places_id": original_id, "url": "bc.example.net/Erin" }) ``` *Referencia manual 1 a 1 entre un documento "People" y su correspondiente documento "Place".*	``` { name: "O'Reilly Media", founded: 1980, location: "CA", books: [123456789, 234567890, ...] } { _id: 123456789, title: "MongoDB: The Definitive Guide", author: ["Kristina Chodorow", "Mike Dirolf"], published_date: ISODate("2010-09-24"), pages: 216, language: "English" } { _id: 234567890, title: "50 Tips and Tricks for MongoDB Developer", author: "Kristina Chodorow", published_date: ISODate("2011-05-06"), pages: 68, language: "English" } ``` *Referencia manual 1 a N entre un documento Autor y sus Libros escritos.*

- ***Las referencias DBRefs*** → Aquí se permite relacionar un documento con otro utilizando el valor del *Campo _id<document>* con el ID del objeto a relacionar, el nombre de la colección donde este documento se encuentra almacenado, el nombre de la BD donde existe esta colección, y opcionalmente la inclusión de otros campos.
    - Esta forma permite establecer relaciones entre documentos más fácilmente con documentos almacenados en múltiples colecciones o BDs.

```
{
  "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
  // .. application fields
  "creator" : {
                  "$ref" : "creators",
                  "$id" : ObjectId("5126bc054aed4daf9e2ab772"),
                  "$db" : "users",
                  "extraField" : "anything"
                }
}
```

Referencia DBRef el cual relaciona un documento con otro documento llamado "Creator" almacenado en al colección "Creators" ($ref), dentro de la BD "Users" ($db), el cual posee el ID "5126bc054aed4daf9e2ab772" ($id), donde además posee un campo extra opcional "extraField".

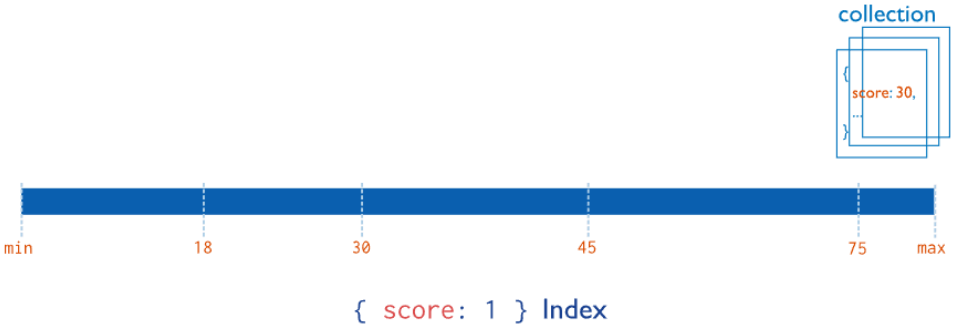
Cabe destacar, que MongoDB no provee de controles por integridad referencial, aspecto que si trae RDBMS. Es decir, al momento de definir una relación mediante DBRefs, MongoDB no valida que el valor insertado en el campo de la relación exista en la colección relacionada, situación que los RDBMS no permiten justamente porque el valor no existe para establecer la relación.

3)

MongoDB provee la definición de índices con el objetivo de mejorar la performance de las consultas, ya que sin la utilización de estos MongoDB debe escanear cada documento de una colección para poder retornar los resultados. Sin embargo la utilización de índices puede afectar el rendimiento de las consultas sobre las colecciones con altas proporciones de lectura/escritura, ya que ante cada inserción se deben actualizar los índices definidos sobre la colección.

MongoDB permite la definición de los siguientes tipos de índices:

- ***Single field index*** → Este tipo de índice permite almacenar la información de un solo campo de una colección. Por defecto, las colecciones poseen un índice sobre el campo *_id*.
    - Es posible indicar el orden del índice respecto de los valores del campo, ya sea ascendente o descendente, sin embargo el orden en este tipo de índice no importa ya que MongoDB puede atravesar el índice en cualquier dirección.
    - Es posible crear un índice de un solo campo en cualquier tipo de campo del documento, ya sea:
        - Un campo de nivel superior del documento.
        - Un documento incrustado.
        - Un campo de un documento incrustado.
    - Este tipo de índice es útil cuando las consultas sobre una colección constantemente se efectúan sobre el mismo campo, por ejemplo:
        - Un departamento de RRHH a menudo busca a sus empleados a través de su ID o Código de empleado.

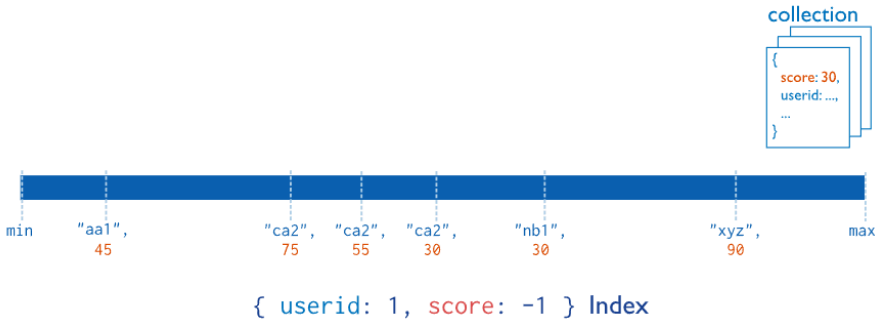


Índice con orden ascendente sobre el campo “score” de una colección de Usuarios

- ***Compound index*** → Este tipo de índice recopila y ordena los datos de dos o más campos de cada documento en una colección, donde los datos se agrupan por el primer campo del índice y luego por sus campos posteriores. Cabe destacar que la limitación de este tipo de índice es de una composición de máximo 32 campos.
    - El orden de los campos indexados afecta la eficacia de búsqueda de un índice compuesto, ya que estos contienen referencias a los documentos según el orden

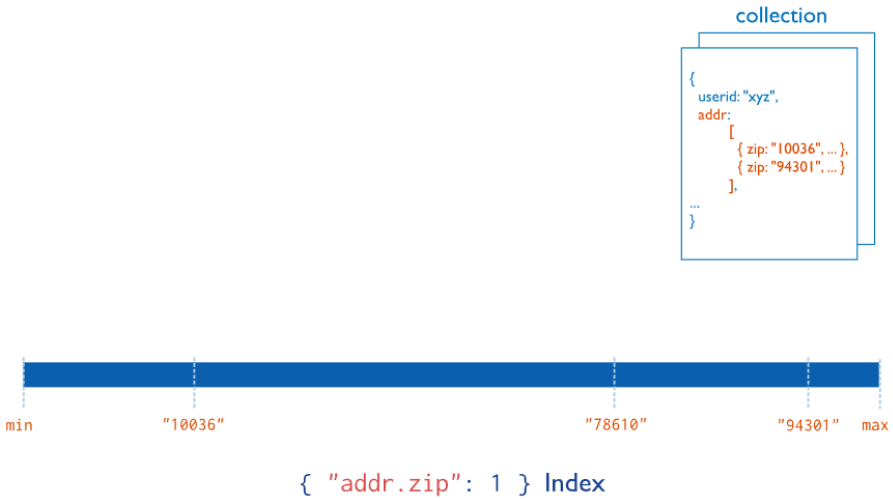
de los campos del índice, por lo que para definir índices compuestos eficiencientes se recomienda seguir la regla ESR (Igualdad, Ordenamiento y Rango).

- Este tipo de índice es útil cuando las consultas sobre una colección se realizan por constantemente varios campos, por ejemplo:
  - El encargado de un comercio busca los artículos de su inventario por nombre y cantidad para determinar aquellos con poco stock.



Índice con orden ascendente para el campo "userId" y descendente para el campo "score"

- ***Multikey index*** → Este tipo de índice permite recopilar y ordenar los datos de campos que contiene valores mediante arreglos, inclusive si el arreglo almacena valores escalares (como strings o números) o documentos embebidos.
  - Además es posible la creación de índices únicos y compuestos sobre campos que almacenan arreglos, donde para el primer tipo los valores del arreglo pueden repetirse para el mismo documento pero más de un documento no puede tener un arreglo con los mismos valores; y para el segundo caso la composición del índice es permitida si a lo sumo un campo es un arreglo.
  - Este tipo de índice es útil cuando se consultan con frecuencia datos almacenados en arreglos, por ejemplo:
    - Los documentos de una colección de *Estudiantes* poseen un arreglo con las calificaciones obtenidas en sus exámenes, donde periódicamente se actualiza una lista con los mejores estudiantes que tienen al menos 5 calificaciones superiores a 90.



Índice con orden ascendente sobre el arreglo “address”

- ***Geospatial index*** → Este tipo de índice permite recopilar y ordenar datos almacenados como objetos GeoJSON o pares de coordenadas, donde los tipos geoespaciales soportados permiten índices 2dsphere que interpretan la geometría de una esfera e índices 2d que interpretan la geometría de una superficie plana.
    - Este tipo de índice es útil cuando se consultan datos geoespaciales, por ejemplo:
        - Se tiene una colección de *Subtes* donde un campo *Ubicaciones* del documento el cual indica las coordenadas de las estaciones de la ciudad, y a con frecuencia se consulta sobre las estaciones cercanas dentro de un área específica de la ciudad.

```
{
   loc: { type: "Point", coordinates: [ -73.97, 40.77 ] },
   name: "Central Park",
   category : "Park"
},
```

```
db.places.createIndex( { loc : "2dsphere" } )
```

Índice 2dsphere para el campo “loc” con las coordenadas de un punto sobre una esfera

- ***Text index*** → Este tipo de índice permite recopilar y ordenar datos sobre campos que almacenan texto.
    - Las colecciones pueden contener un único índice de tipo texto, pero este puede abarcar uno o varios campos.
    - Este tipo de índice es útil cuando se consultas palabras o frases específicas dentro del contenido de una cadena, por ejemplo:
        - Los documentos de la colección de *Ropa* incluyen un campo *Descripción* el cual contiene una cadena que describe al artículo, donde para buscar ropa de sepa se busca sobre este campo todos aquellos documentos que posean la palabra seda dentro de él.

```
db.blog.createIndex(
   {
      "about": "text",
      "keywords": "text"
   }
)
```

Índice sobre los campos "about" y "keywords" sobre los documentos de una colección que representan "blogs"

- ***Hashed index*** → Este tipo de índice recopilan y almacenan hashes de los valores de los campos indexados. Los índices hash permiten la dispersión mediante claves hash, donde se utiliza un índice hash de campo como clave de partición para particionar los datos en todo el cluster disperso.
    - El uso de claves de dispersión hash para dispersar una colección permite una distribución más uniforme de los datos dispersos.
    - No es posible definir índices hash sobre índices multiclaves debido a que la función de hash no soporta más de un campo, como los arreglos.
    - Este tipo de índice es útil si la clave de dispersión aumenta de forma monótona, por ejemplo:
        - Las marcas de tiempo y los valores de ObjectId.

```
db.orders.createIndex( { _id: "hashed" } )
```

Índice hash sobre el campo "_id" de una colección de "pedidos"

- ***Clustered index*** → Este tipo de índice permite especificar el orden en que las colecciones clusterizadas almacenan los datos, es decir, las colecciones creadas con un índice clusterizado se denominan colecciones clusterizadas.
    - Las colecciones clusterizadas o agrupadas almacenan los documentos indexados junto a la especificación del índice en el mismo archivo, donde almacenar ambos aspectos juntos permite un mayor beneficio del almacenamiento y rendimiento a los índices normales.

```
db.createCollection(
   "stocks",
   { clusteredIndex: { "key": { _id: 1 }, "unique": true, "name": "stocks clustered key" } }
)
```

Crea una colección y define sobre la misma un índice único sobre el campo "_id"

4)

Las vistas en MongoDB se basan en las vistas en las RDBMS, donde permite abstraer una consulta compleja de obtención de datos mediante la invocación de la vista como una función la cual se ejecuta sobre la BD y retorna los datos correspondientes.

MongoDB admite la definición de las siguientes dos tipos de vistas:

- ***Standard views*** → Este tipo es un objeto consultable de sólo lectura, cuyo contenido está definido por las agregaciones realizadas sobre otras colecciones o vistas. Las vistas no se almacenan en disco, por lo que su contenido se calcula bajo demanda cuando un cliente la consulta.
    - Un ejemplo de uso podría ser: Una vista que visualice de forma simplificada una compra *(Purchase)* junto a su usuario *(User)*, calificación obtenida *(Review)*, la ruta realizada *(Route)* y los servicios adquiridos *(ItemService)*.

```
db.createView(
   "firstYears",
   "students",
   [ { $match: { year: 1 } } ]
)
```

Vista que obtiene los estudiantes de primer año

- ***On-demand materialized views*** → Este tipo de vista es resultado de un proceso de agregación precalculado que se almacena y lee en disco, utilizado la etapa *$merge* o *$out* para actualizar los datos guardados, es decir, este tipo de vista crea una nueva colección (en la misma u otra BD) con los resultados de la consulta mediante la etapa *$out*, y actualiza la colección ante modificaciones con los nuevos resultados de la consulta mediante la etapa *$merge*.
    - Un ejemplo de uso podría ser: Una vista que visualice de forma breve la información de un proveedor de servicios *(Supplier)*, los servicios que este provee *(Service)* y la cantidad vendida de cada uno *(ItemService)*.

```
updateMonthlySales = function(startDate) {
   db.bakesales.aggregate( [
      { $match: { date: { $gte: startDate } } },
      { $group: { _id: { $dateToString: { format: "%Y-%m", date: "$date" } }, sa
      { $merge: { into: "monthlybakesales", whenMatched: "replace" } }
   ] );
};
```

```
sales_quantity: { $sum: "$quantity"}, sales_amount: { $sum: "$amount" } } },
```

La función updateMonthlySales() define una vista materializada de ventas mensuales de pasteles, donde la función recibe como parámetro la fecha en la cual actualizar las ventas mensuales a partir de dicha fecha

Ambos tipos de vistas retornan los resultados del proceso de agregación, sin embargo la **diferencia entre ellas** son las siguientes:

- Las *Standar views* se calculan cuando se lee la vista y no se almacenan en disco, y las *On-demand materialized views* se almacenan y leen en disco utilizando las etapas *$merge* o *$out* para actualizar los datos guardados.
- Las *On-demand materialized views* proporcionan un mayor rendimiento de lectura que las *Standard views*, ya que las primeras se leen desde el disco en lugar de calcularse

como parte de la consulta, beneficio que aumenta según la complejidad y tamaño de los datos consultados.

5)

MongoDB permite un modelado de datos dentro de la BD de forma muy flexible, permitiendo que los documentos de una misma colección no posean el mismo conjunto de campos, y a su vez los tipos de datos de los campos entre los distintos documentos también puedan ser distintos. Sin embargo, si se desea mantener coherencia en el modelo de datos de una colección es posible crear reglas de validación del esquema sobre sus campos, tipos de datos permitidos y rangos de valores aceptados.

A continuación se mencionan las tres formas de validación que se proveen:

- ***JSON schema validation*** → Permite validar objetos JSON especificando reglas de validación sobre sus campos en un lenguaje legible para humanos. A la hora de crear una colección se pueden definir las reglas documento JSON a almacenar, siendo el título de la validación, los campos que son requeridos de forma obligatoria, las propiedades que deben cumplir dichos campos, como el tipo de dato a almacenar, el rango de valores mínimo y máximo permitido, junto a una descripción que explica la validación definida.

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         title: "Student Object Validation",
         required: [ "address", "major", "name", "year" ],
         properties: {
            name: {
               bsonType: "string",
               description: "'name' must be a string and is required"
            },
            year: {
               bsonType: "int",
               minimum: 2017,
               maximum: 3017,
               description: "'year' must be an integer in [ 2017, 3017 ]
            },
            gpa: {
               bsonType: [ "double" ],
               description: "'gpa' must be a double if the field exists"
            }
         }
      }
   }
} )
```

- ***Validation with query operators*** → Permite especificar validaciones de esquemas JSON mediante reglas de validación dinámicas que comparan los valores de múltiples campos en tiempo de ejecución, utilizando operadores de consulta como *$eq* y *$gt* para comparar campos.

    - Es utilizado cuando el valor de un campo depende del valor de otro campo, y es necesario asegurarse que los valores sean correctos entre sí. Por ejemplo, un pedido que tiene un precio base y un IVA, y la colección de *Pedidos* realiza el seguimiento de los campos *precio, IVA,* y *precioTotalConIVA*, donde al momento de insertar un pedido se valida que el precio y el valor del IVA coincida con el precio total con el IVA incluido.

```
db.createCollection( "orders",
   {
      validator: {
         $expr:
            {
               $eq: [
                  "$totalWithVAT",
                  { $multiply: [ "$total", { $sum:[ 1, "$VAT" ] } ] }
               ]
            }
      }
   }
)
```

- ***Allowed field values*** → Permite especificar al momento de crear un esquema JSON los rangos de valores permitidos para un campo particular, asegurándose que los valores ingresados al campo coincida con el conjunto posibles valores esperado o evitar errores de tipeo humano, donde este conjunto posible se define mediante los tipos enumerativos.

```
db.createCollection("shipping", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         title: "Shipping Country Validation",
         properties: {
            country: {
               enum: [ "France", "United Kingdom", "United States" ],
               description: "Must be either France, United Kingdom, or
            }
         }
      }
   }
} )
```

6)

MongoDB implementa el uso de **transacciones**, pero estas están definidas de forma distinta a las RDBMS, ya que se implementan dos tipos de transacciones dependiendo de su alcance:

- ***Transacciones "simples"*** ➜ Las operaciones en este tipo de transacciones son atómicas pero sí se ejecutan sobre un único documento, ya que este tipo de atomicidad de un solo documento, al poder utilizar campos con arreglos o documentos embebidos para capturar múltiples datos en un único documento, en lugar de normalizar múltiples documentos y colecciones, elimina la necesidad de utilizar transacciones distribuidas en muchos casos de uso prácticos.

- ***Transacciones distribuidas*** ➜ Este tipo de transacción permite la atomicidad en operaciones de lectura/escritura sobre múltiples documentos, colecciones, BDs y fragmentos. Cabe destacar que el uso de este tipo de transacciones genera un mayor costo de rendimiento en comparación a la transacciones sobre un único documento debido al costo que genera la sincronización entre las distintas colecciones, documentos, y BDs, por lo que en muchos casos la utilización de esquemas de datos desnormalizados (con documentos y arreglos integrados en los documentos) permitirá una utilización óptima en el uso de sus datos, al minimizar la necesidad de transacciones distribuidas.

7)

En una BD Relacional, se almacena cada entidad individual en su propia tabla y se relacionan mediante claves foráneas. Si bien MongoDB admite referencias de un documento a otro, e incluso uniones de múltiples documentos, es un error usar una BD orientada a documentos de la misma manera que usa una BD Relacional.

MongoDB permite establecer relaciones mediante los patrones de **Embeber un documento en otro** mediante la incrustación o definición explícita de un documento dentro de otro. Los documentos incrustados dentro de otros permite establecer relaciones eficientes y limpias, especialmente cuando se acceden a datos de forma conjunta con regularidad. En general, al diseñar esquemas de BD mediante MongoDB es preferible definir de forma predeterminada la incrustación de documentos, y utilizar referencias del lado de la aplicación o BD solo cuando valga la pena, ya que cuanto más a menudo la carga de trabajo pueda recuperar en una única consulta un solo documento y este tenga todos los datos que se requieren, más consistente y

performante será el rendimiento de la aplicación. Estos patrones se basan en "*Si los usas juntos, almacenalos juntos*".

A continuación se presentan distintas patrones de incrustación de documentos:

- En **relaciones 1 a 1 - relaciones 1 a N** si es necesario acceder constantemente a un segundo documento relacionado con el primero, es más sencillo incrustarlo dentro de él en lugar de establecer relaciones manuales.
    - Por ejemplo: se define la dirección asociada a un único usuario dentro de él.
    - La **ventaja** es que en lugar de realizar una consulta extra para recuperar la *Direccion* del *Usuario* "Jane", ahora podemos acceder a ella como un subdocumento dentro del documento.

.

Relación manual	Embeber documento
	

```
> db.user.findOne()
{
    _id: 111111,
    email: "email@example.com",
    name: {given: "Jane", family: "Han"},
}

> db.address.find({user_id: 111111})
{
    _id: 121212,
    street: "111 Elm Street",
    city: "Springfield",
    state: "Ohio",
    country: "US",
    zip: "00000"
}
```

```
> db.user.findOne({_id: 111111})

{
    _id: 111111,
    email: "email@example.com",
    name: {given: "Jane", family: "Han"},
    address: {
    street: "111 Elm Street",
    city: "Springfield",
    state: "Ohio",
    country: "US",
    zip: "00000",
    }
}
```

Relación 1 a 1

Relación 1 a N

- ***Patrón de subconjunto embebido*** → Es un caso híbrido el cual es útil cuando se tiene un documento el cual posee una lista o relación 1 a N potencialmente larga de documentos almacenados en una colección separada, pero se desea mantener algunos de estos documentos fácilmente a mano dentro del primer documento. Se basa en “*Si accedes regularmente al subconjunto relacionado, integra ese subconjunto*”.
    - Por ejemplo: se tiene un *Película* la cual puede contener un centenar de *Reviews*, pero se desean mantener a disposición las dos más recientes a la hora de visualizar la *Película*.

<table>
<tr><td>Sin patrón</td><td>

```
> db.movie.findOne()

{
    _id: 333333,
    title: “The Big Lebowski”
}

> db.review.find({movie_id: 333333})

{
    _id: 454545,
    movie_id: 333333,
    stars: 3
    text: “it was OK”
}

{
    _id: 565656,
    movie_id: 333333,
    stars:5,
    text: “the best”
}
...
```

</td></tr>
<tr><td>Con patrón</td><td>

```
> db.movie.findOne({_id: 333333})

{
    _id: 333333,
    title: “The Big Lebowski”,
    recent_reviews: [
    {_id: 454545, stars: 3, text: “it was OK”},
    {_id: 565656, stars: 5, text: “the best”}
    ]
}
```

</td></tr>
</table>

- ***Patrón de referencia extendido*** → Similar al patrón anterior, pero en este caso en lugar de una lista es útil cuando en un documento se hace referencia regularmente a los mismos campos de otro documento relacionado dentro del mismo documento, optimizando el acceso a esa pequeña cantidad de información. Se basa en “*Si accedes a los mismos campos de un documento referenciado, incrusta esos campos* ”.

- *Por ejemplo*: Una *Película* posee una referencia al *Estudio* que la filmó, pero siempre se accede al nombre del mismo.

```
> db.movie.findOne({_id: 444444})

{
    _id: 444444,
    title: "One Flew Over the Cuckoo's Nest",
    studio_id: 999999,
    studio_name: "Fantasy Films"
}
```

Ahora bien, podríamos mencionar como **Ventajas** respecto de estos patrones de referencias en DBMS NoSQL en comparación a las referencias definidas en RDBMS, es que las primeras permite una mayor performance y flexibilidad en la definición de los esquemas de datos y su posterior recuperación adaptándose fácilmente a los casos de uso y funcionalidades de la aplicación.

Sin embargo, como posibles **Desventajas** surgen al igual que los RDBMS pero en mayor medida, la redundancia de información al embeber muchos fragmentos de documentos en otros, y los problemas que podría traer al tener que mantener la consistencia de los datos en operaciones de tipo CRUD.

8)

Casos útiles de mapeo **Por referencia** a una BD NoSQL:

- La lista de *Rutas* que tienen los *GuiasTuristas* y los *Conductores*.
    - Las listas de *Route* para los *TourGuideUser* y *DriverUser*.
- La lista de las *Rutas* que tienen las *Paradas*.
    - La lista de *Route* para las *Stop*.
- La lista de *Servicios* que ofrecen los *Proveedores*.
    - La lista de *Service* del *Supplier*.
- La lista de *Items* vendidos de los *Servicios*.

    - La lista de *ItemService* del *Service,*.

Aquí es más conveniente la utilización de mapeos por referencia debido a la gran escalabilidad de los tamaños de las listas en las relaciones como para incrustarlas en los documentos, ya que en casos muy raros debamos acceder a dichas listas a la hora de manipular el objeto principal.

Casos útiles de mapeo **Como documentos embebidos** a una BD NoSQL:

- El *Usuario*, la *Ruta*, la *Review* y los *Items* adquiridos en una Compra.
    - El *User*, *Route, Review*, y *ItemService* de una *Purchase*.
- Las *Paradas* de una *Ruta*.
    - Las *Stop* de los *Route*.

Esto es útil ya que por cuestiones del dominio constantemente se acceden a dichos datos relacionados a la hora de manipular el objeto principal, y si bien los modificadores *FETCH* permiten obtener o no los objetos dentro de la misma consulta (mediante *EAGER* o *LAZY* respectivamente), no siempre se requiere la obtención de los objetos de las relaciones, al igual que en los casos en donde se embeben partes de los objetos relacionados en lugar del objeto completo, aspecto que en este informe no contemplaremos por cuestiones de mantener lo más fiel posible el modelo de objetos original.

Además, es importante destacar que estos objetos a embeber no escalan en un gran tamaño, por lo que en términos de performance es rentable al no conllevar pérdidas considerables en su manipulación.

Parte 2: Operaciones CRUD básicas.

9)

Crear / usar DB: ***use tours***

Crear Collection: ***db.createCollection("recorridos")***

i)

```
db.recorridos.insertOne({
        "nombre": "City Tour",
        "precio": 200,
        "stops": ["Diagonal Norte", "Avenida de Mayo", "Plaza del Congreso" ],
        "totalKm":5
})
```

```
{
  acknowledged: true,
  insertedId: ObjectId('6657cb9a5da37ec4eae5dbd0')
}
```

ii)

Se incluye automáticamente en la inserción al campo id de tipo ObjectId para la identificación unívoca del documento en la colección como una clave primaria, donde si este campo no se especifica explícitamente durante la inserción mongodb lo genera automáticamente.

```
$ db.recorridos.find().pretty()
o bien
$ db.recorridos.find({ nombre: "City Tour" }).pretty()
```

```
{
  _id: ObjectId('6657cb9a5da37ec4eae5dbd0'),
  nombre: 'City Tour',
  precio: 200,
  stops: [
    'Diagonal Norte',
    'Avenida de Mayo',
    'Plaza del Congreso'
  ],
  totalKm: 5
}
```

10)

Insertar muchos: ***db.recorridos.insertMany(contenido_de_material_adicional_1)***


```
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('6657d0c55da37ec4eae5dbd2'),
    '1': ObjectId('6657d0c55da37ec4eae5dbd3'),
    '2': ObjectId('6657d0c55da37ec4eae5dbd4'),
    '3': ObjectId('6657d0c55da37ec4eae5dbd5'),
    '4': ObjectId('6657d0c55da37ec4eae5dbd6'),
    '5': ObjectId('6657d0c55da37ec4eae5dbd7'),
    '6': ObjectId('6657d0c55da37ec4eae5dbd8'),
    '7': ObjectId('6657d0c55da37ec4eae5dbd9'),
    '8': ObjectId('6657d0c55da37ec4eae5dbda'),
    '9': ObjectId('6657d0c55da37ec4eae5dbdb'),
    '10': ObjectId('6657d0c55da37ec4eae5dbdc'),
    '11': ObjectId('6657d0c55da37ec4eae5dbdd'),
    '12': ObjectId('6657d0c55da37ec4eae5dbde'),
    '13': ObjectId('6657d0c55da37ec4eae5dbdf')
  }
}
```


i)

Actualizar el recorrido “Cultural Odyssey” para que su total de kilómetros sea 12.

```
$ db.recorridos.updateOne(
        { nombre: "Cultural Odyssey" },
        { $set: { totalKm: 12 } }
)
```


```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

ii)

Actualizar el listado de Stops del recorrido con nombre “Delta Tour” para agregar “Tigre”.

```
$ db.recorridos.updateOne(
        { nombre: "Delta Tour" },
        { $addToSet: { stops: "Tigre" } }
)
```

```
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
```

iii)

Aumentar un 10% el precio de todos los recorridos.

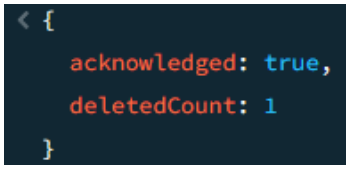
```
$ db.recorridos.updateMany(
        {},
        { $mul: { precio: 1.1 } }
)
```

```
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 15,
    modifiedCount: 15,
    upsertedCount: 0
  }
```

iv)

Eliminar el recorrido con nombre "Temporal Route”.

```
$ db.recorridos.deleteOne(
        { nombre: "Temporal Route" }
)
```



v)

Crear el array de etiquetas (tags) para la ruta “Urban Exploration”. Agregue el elemento “Gastronomía” a dicho arreglo.

```
$ db.recorridos.updateOne(
        { nombre: "Urban Exploration" },
        { $set: { tags: ["Gastronomía"] } }
)
```

11)

i)

Obtenga la ruta con nombre "Museum Tour”

```
$ db.recorridos.find({ nombre: "Museum Tour" })
```

```
{
  _id: ObjectId('6657d0c55da37ec4eae5dbda'),
  nombre: 'Museum Tour',
  precio: 605,
  stops: [
    'Museo Nacional de Bellas Artes',
    'Teatro Colón',
    'Planetario',
    'Bosques de Palermo',
    'San Telmo',
    'La Boca - Caminito',
    'Recoleta',
    'El Monumental (Estadio River Plate)',
    'Av 9 de Julio'
  ],
  totalKm: 13
}
```

ii)

Las rutas con precio superior a $600

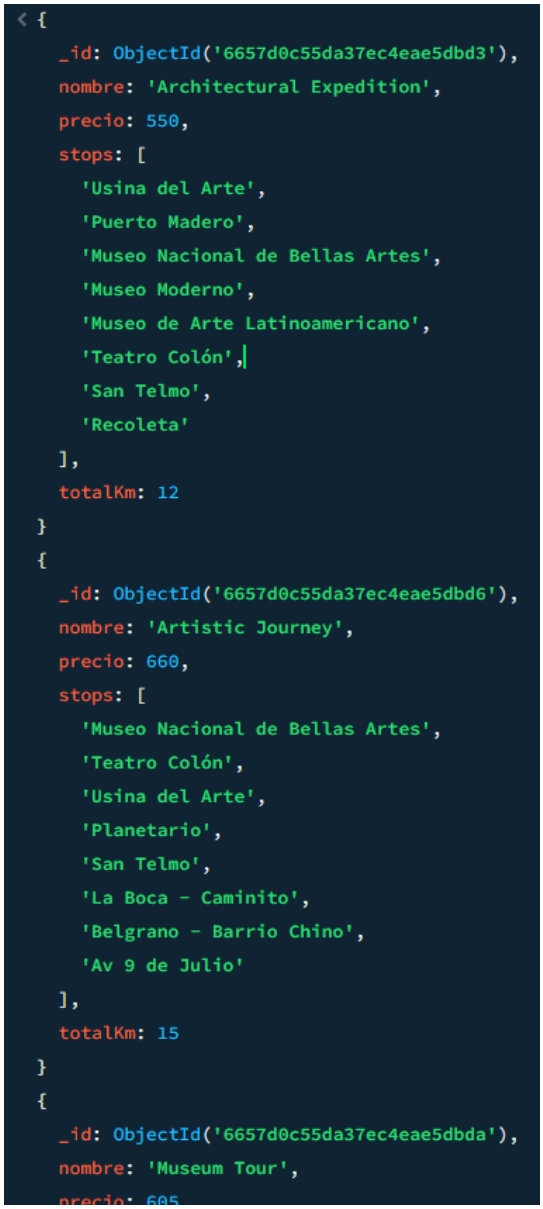
```
$ db.recorridos.find({ precio: { $gt: 600 } })
```

```
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd4'),
  nombre: 'Delta Tour',
  precio: 880.0000000000001,
  stops: [
    'Río de la Plata',
    'Bosques de Palermo',
    'Delta',
    'Tigre'
  ],
  totalKm: 8
}
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd6'),
  nombre: 'Artistic Journey',
  precio: 660,
  stops: [
    'Museo Nacional de Bellas Artes',
    'Teatro Colón',
    'Usina del Arte',
    'Planetario',
    'San Telmo',
    'La Boca - Caminito',
    'Belgrano - Barrio Chino',
    'Av 9 de Julio'
  ],
  totalKm: 15
}
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd8'),
  nombre: 'Gastronomic Delight',
  precio: 770.0000000000001,
  stops: [
    'San Telmo',
    'La Boca - Caminito',
```

iii)

Las rutas con precio superior a $500 y con un total de kilómetros mayor a 10.

```
$ db.recorridos.find({
        precio: { $gt: 500 },
        totalKm: { $gt: 10 }
})
```



iv)

Las rutas que incluyan el stop “San Telmo”.

```
$ db.recorridos.find({ stops: "San Telmo" })
```

```
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd3'),
  nombre: 'Architectural Expedition',
  precio: 550,
  stops: [
    'Usina del Arte',
    'Puerto Madero',
    'Museo Nacional de Bellas Artes',
    'Museo Moderno',
    'Museo de Arte Latinoamericano',
    'Teatro Colón',
    'San Telmo',
    'Recoleta'
  ],
  totalKm: 12
}
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd6'),
  nombre: 'Artistic Journey',
  precio: 660,
  stops: [
    'Museo Nacional de Bellas Artes',
    'Teatro Colón',
    'Usina del Arte',
    'Planetario',
    'San Telmo',
    'La Boca - Caminito',
    'Belgrano - Barrio Chino',
    'Av 9 de Julio'
  ],
  totalKm: 15
}
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd7'),
  nombre: 'Tango Experience',
  precio: 385.0000000000006,
```

v)

Las rutas que incluyan el stop “Recoleta” y no el stop “Plaza Italia”

```
$ db.recorridos.find({
        $and: [
                { stops: "Recoleta" },
                { stops: { $ne: "Plaza Italia" } }
        ]
})
```

vi)

El nombre y el total de km (si es que posee) de las rutas que incluyan el stop “Delta” y tenga un precio menor a 500.

```
$ db.recorridos.find(
        { stops: "Delta", precio: { $lt: 500 } },
        { nombre: 1, totalKm: 1 }
)
```

```
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd5'),
  nombre: 'Nature Escape'
}
```

vii)

Las rutas que incluyan tanto “San Telmo” como “Recoleta” y “Avenida de Mayo” entre sus stops.

```
$ db.recorridos.find({
   stops: { $all: ["San Telmo", "Recoleta", "Avenida de Mayo"] }
})
```

```
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd7'),
  nombre: 'Tango Experience',
  precio: 385.00000000000006,
  stops: [
    'Avenida de Mayo',
    'Plaza del Congreso',
    'Paseo de la Historieta',
    'San Telmo',
    'La Boca - Caminito',
    'Recoleta'
  ],
  totalKm: 10
}
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd9'),
  nombre: 'Urban Exploration',
  precio: 495.00000000000006,
  stops: [
    'Avenida de Mayo',
    'Museo Moderno',
    'Paseo de la Historieta',
    'Usina del Arte',
    'San Telmo',
    'La Boca - Caminito',
    'Belgrano - Barrio Chino',
    'Recoleta',
    'El Monumental (Estadio River Plate)',
    'Costanera Sur',
    'Plaza Italia'
  ],
  totalKm: 14,
  tags: [
    'Gastronomía'
  ]
```

viii)

Solo el nombre de las rutas que dispongan de más de 5 stops.

```
$ db.recorridos.find({
      $expr: { $gt: [{ $size: "$stops" }, 4] }
})
```

```
< {
    _id: ObjectId('6657d0c55da37ec4eae5dbd2'),
    nombre: 'Historical Adventure',
    precio: 330,
    stops: [
      'Avenida de Mayo',
      'Plaza del Congreso',
      'Río de la Plata',
      'Teatro Colón',
      'Av 9 de Julio',
      'Plaza Italia'
    ],
    totalKm: 10
  }
  {
    _id: ObjectId('6657d0c55da37ec4eae5dbd3'),
    nombre: 'Architectural Expedition',
    precio: 550,
    stops: [
      'Usina del Arte',
      'Puerto Madero',
      'Museo Nacional de Bellas Artes',
      'Museo Moderno',
      'Museo de Arte Latinoamericano',
      'Teatro Colón',
      'San Telmo',
      'Recoleta'
    ],
    totalKm: 12
  }
  {
    _id: ObjectId('6657d0c55da37ec4eae5dbd6'),
    nombre: 'Artistic Journey',
```

ix)

Las rutas que no tengan definida el total de sus kilómetros.

```
$ db.recorridos.find({ totalKm: { $exists: false } })
```

```
{
  _id: ObjectId('6657d0c55da37ec4eae5dbd5'),
  nombre: 'Nature Escape',
  precio: 440.00000000000006,
  stops: [
    'Delta',
    'Río de la Plata',
    'Av 9 de Julio',
    'Puerto Madero'
  ]
}
```

x)

Los nombres y el listado de stops de aquellas rutas que incluyen algún museo en sus recorridos.

```
$ db.recorridos.find(
        { stops: /museo/i },
        { nombre: 1, stops: 1 }
)
```

```
< {
    _id: ObjectId('6657d0c55da37ec4eae5dbd3'),
    nombre: 'Architectural Expedition',
    stops: [
      'Usina del Arte',
      'Puerto Madero',
      'Museo Nacional de Bellas Artes',
      'Museo Moderno',
      'Museo de Arte Latinoamericano',
      'Teatro Colón',
      'San Telmo',
      'Recoleta'
    ]
  }
  {
    _id: ObjectId('6657d0c55da37ec4eae5dbd6'),
    nombre: 'Artistic Journey',
    stops: [
      'Museo Nacional de Bellas Artes',
      'Teatro Colón',
      'Usina del Arte',
      'Planetario',
      'San Telmo',
      'La Boca - Caminito',
      'Belgrano - Barrio Chino',
      'Av 9 de Julio'
    ]
  }
  {
    _id: ObjectId('6657d0c55da37ec4eae5dbd9'),
    nombre: 'Urban Exploration',
    stops: [
      'Avenida de Mayo',
      'Museo Moderno'
```

xi)

Obtenga la cantidad de elementos que posee la colección.

```
$ db.recorridos.countDocuments()
```

```
< 14
```

Parte 3: Aggregation Framework

12)

Crear tabla:

```
use tours2
```

Ejecutar “generator1.js”:

```
load('generator1.js');
```

13)

i)

Obtenga una muestra de 5 rutas aleatorias de la colección.

```
db.route.aggregate([
    { $sample: { size: 5 } }
]);
```

```
[
  {
    _id: ObjectId('6656702152bf4d59fccddc48'),
    name: 'route3548',
    price: 441,
    totalKm: 7.11,
    stops: [ 76, 35, 53, 76, 59 ]
  },
  {
    _id: ObjectId('6656702852bf4d59fccdee2b'),
    name: 'route8127',
    price: 373,
    totalKm: 17.36,
    stops: [ 10, 62 ]
  },
  {
    _id: ObjectId('66566fc2afd61d8455ce694d'),
    name: 'route39649',
    price: 747,
    totalKm: 13.31,
    stops: [ 61, 73 ]
  },
  {
    _id: ObjectId('66566f71afd61d8455cdeaff'),
    name: 'route7315',
    price: 804,
    totalKm: 6.26,
    stops: [ 73, 59 ]
  },
  {
    _id: ObjectId('66566f70afd61d8455cde8ce'),
    name: 'route6754',
    price: 411,
    totalKm: 10.14,
    stops: [ 6, 73, 4 ]
  }
]
```

ii)

Extienda la consulta anterior para incluir en el resultado toda la información de cada una de las Stops. Note que puede ligarlas por su código.

```
db.route.aggregate([
    { $sample: { size: 5 } },
    {
        $lookup: {
            from: "stop",
            localField: "stops",
            foreignField: "code",
```

```
            as: "stops_info"
        }
    }
]);
```

```
[
  {
    _id: ObjectId('66566fd9afd61d8455ce73ba'),
    name: 'route2168',
    price: 464,
    totalKm: 9.36,
    stops: [ 35, 89, 92 ],
    stops_info: [
      {
        _id: ObjectId('66566f64afd61d8455cdce19'),
        name: 'Museo del Humor',
        code: 35,
        desciprion: 'Stop number 35'
      },
      {
        _id: ObjectId('66566f64afd61d8455cdce4f'),
        name: 'Basílica Nuestra Señora del Pilar',
        code: 89,
        desciprion: 'Stop number 89'
      },
      {
        _id: ObjectId('66566f64afd61d8455cdce52'),
        name: 'El Monumental (Estadio River Plate)',
        code: 92,
        desciprion: 'Stop number 92'
      },
      {
        _id: ObjectId('66566fd5afd61d8455ce6aef'),
        name: 'Museo del Humor',
        code: 35,
        desciprion: 'Stop number 35'
      },
      {
        _id: ObjectId('66566fd6afd61d8455ce6b25'),
        name: 'Basílica Nuestra Señora del Pilar',
        code: 89,
        desciprion: 'Stop number 89'
      },
      {
        _id: ObjectId('66566fd6afd61d8455ce6b28'),
        name: 'El Monumental (Estadio River Plate)',
        code: 92,
        desciprion: 'Stop number 92'
      },
      {
        _id: ObjectId('6656701c52bf4d59fccdce19'),
        name: 'Museo del Humor',
        code: 35,
        desciprion: 'Stop number 35'
      },
      {
        _id: ObjectId('6656701c52bf4d59fccdce4f'),
        name: 'Basílica Nuestra Señora del Pilar',
        code: 89,
        desciprion: 'Stop number 89'
      },
      {
        _id: ObjectId('6656701c52bf4d59fccdce52'),
        name: 'El Monumental (Estadio River Plate)',
        code: 92,
```

iii)

Obtenga la información de las Routes (incluyendo la de sus Stops) que tengan un precio mayor o igual a 900.

```
db.route.aggregate([
    { $match: { price: { $gte: 900 } } },
    {
        $lookup: {
            from: "stop",
            localField: "stops",
            foreignField: "code",
            as: "stops_info"
        }
    }
]);
```

```
  },
  {
    _id: ObjectId('66566f64afd61d8455cdce6f'),
    name: 'route3',
    price: 967,
    totalKm: 1.4,
    stops: [ 62, 41 ],
    stops_info: [
      {
        _id: ObjectId('66566f64afd61d8455cdce1f'),
        name: 'Circuitos Gastronómicos Peatonales',
        code: 41,
        desciprion: 'Stop number 41'
      },
      {
        _id: ObjectId('66566f64afd61d8455cdce34'),
        name: 'Bosques de Palermo',
        code: 62,
        desciprion: 'Stop number 62'
      },
      {
        _id: ObjectId('66566fd5afd61d8455ce6af5'),
        name: 'Circuitos Gastronómicos Peatonales',
        code: 41,
        desciprion: 'Stop number 41'
      },
      {
        _id: ObjectId('66566fd5afd61d8455ce6b0a'),
        name: 'Bosques de Palermo',
        code: 62,
        desciprion: 'Stop number 62'
      },
      {
        _id: ObjectId('6656701c52bf4d59fccdce1f'),
        name: 'Circuitos Gastronómicos Peatonales',
        code: 41,
        desciprion: 'Stop number 41'
      },
      {
        _id: ObjectId('6656701c52bf4d59fccdce34'),
        name: 'Bosques de Palermo',
        code: 62,
        desciprion: 'Stop number 62'
      }
    ]
  },
```

iv)

Obtenga la información de las Routes que tengan 5 Stops o más.

```
db.route.aggregate([
    {
```

```
        $match: {
            $expr: {
                $gte: [{ $size: "$stops" }, 5]
            }
        }
    },
    ]);
```

```
{
  _id: ObjectId('66566f64afd61d8455cdce81'),
  name: 'route21',
  price: 415,
  totalKm: 2.61,
  stops: [ 112, 52, 38, 72, 48 ]
},
{
  _id: ObjectId('66566f64afd61d8455cdce83'),
  name: 'route23',
  price: 247,
  totalKm: 19.07,
  stops: [ 42, 17, 113, 86, 17 ]
},
{
  _id: ObjectId('66566f64afd61d8455cdce84'),
  name: 'route24',
  price: 840,
  totalKm: 12.17,
  stops: [ 12, 48, 101, 38, 40 ]
},
{
  _id: ObjectId('66566f64afd61d8455cdce86'),
  name: 'route26',
  price: 986,
  totalKm: 19.45,
```

v)

Obtenga la información de las Routes que tengan incluido en su nombre el string “111”.

```
db.route.aggregate([
    { $match: { name: { $regex: "111" } } }

]);
```

```
[
  {
    _id: ObjectId('665670b3e8ba225aaccdcedb'),
    name: 'route111',
    price: 508,
    totalKm: 11.69,
    stops: [ 63, 83, 97 ]
  },
  {
    _id: ObjectId('665670b4e8ba225aaccdd2c2'),
    name: 'route1110',
    price: 777,
    totalKm: 10.16,
    stops: [ 18, 59, 20 ]
  },
  {
    _id: ObjectId('665670b4e8ba225aaccdd2c3'),
    name: 'route1111',
    price: 640,
    totalKm: 8.78,
    stops: [ 79, 83 ]
  },
  {
    _id: ObjectId('665670b4e8ba225aaccdd2c4'),
    name: 'route1112',
    price: 328,
    totalKm: 17.48,
```

vi)

Obtenga solo las Stops de la Route con nombre “Route100”

```
db.route.aggregate([
  { $match: { name: "route100" } },
  {
    $lookup: {
      from: "stop",
      localField: "stops",
      foreignField: "code",
      as: "stops_info"
    }
  },
  {
    $unwind: "$stops_info"
  },
  {
    $replaceRoot: { newRoot: "$stops_info" }
  }
]).pretty();
```

```
[
  {
    _id: ObjectId('66566f64afd61d8455cdce34'),
    name: 'Bosques de Palermo',
    code: 62,
    desciprion: 'Stop number 62'
  },
  {
    _id: ObjectId('66566f64afd61d8455cdce3a'),
    name: 'San Telmo',
    code: 68,
    desciprion: 'Stop number 68'
  },
  {
    _id: ObjectId('66566f64afd61d8455cdce62'),
    name: 'Plaza San Martín',
    code: 108,
    desciprion: 'Stop number 108'
  },
  {
    _id: ObjectId('66566f64afd61d8455cdce64'),
    name: 'Museo de Arte Hispanoamericano Isaac Fernández Blanco',
    code: 110,
    desciprion: 'Stop number 110'
  },
  {
    _id: ObjectId('66566fd5afd61d8455ce6b0a'),
    name: 'Bosques de Palermo',
    code: 62,
    desciprion: 'Stop number 62'
tours2>
  {
    _id: ObjectId('66566fd5afd61d8455ce6b10'),
    name: 'San Telmo',
    code: 68,
    desciprion: 'Stop number 68'
  },
  {
    _id: ObjectId('66566fd6afd61d8455ce6b38'),
    name: 'Plaza San Martín',
    code: 108,
    desciprion: 'Stop number 108'
  },
  {
    _id: ObjectId('66566fd6afd61d8455ce6b3a'),
    name: 'Museo de Arte Hispanoamericano Isaac Fernández Blanco',
    code: 110,
    desciprion: 'Stop number 110'
  },
  {
    _id: ObjectId('6656701c52bf4d59fccdce34'),
    name: 'Bosques de Palermo',
    code: 62,
    desciprion: 'Stop number 62'
  },
  {
    _id: ObjectId('6656701c52bf4d59fccdce3a'),
    name: 'San Telmo',
    code: 68,
    desciprion: 'Stop number 68'
  },
```

vii)

Obtenga la información del Stop que más apariciones tiene en Routes.

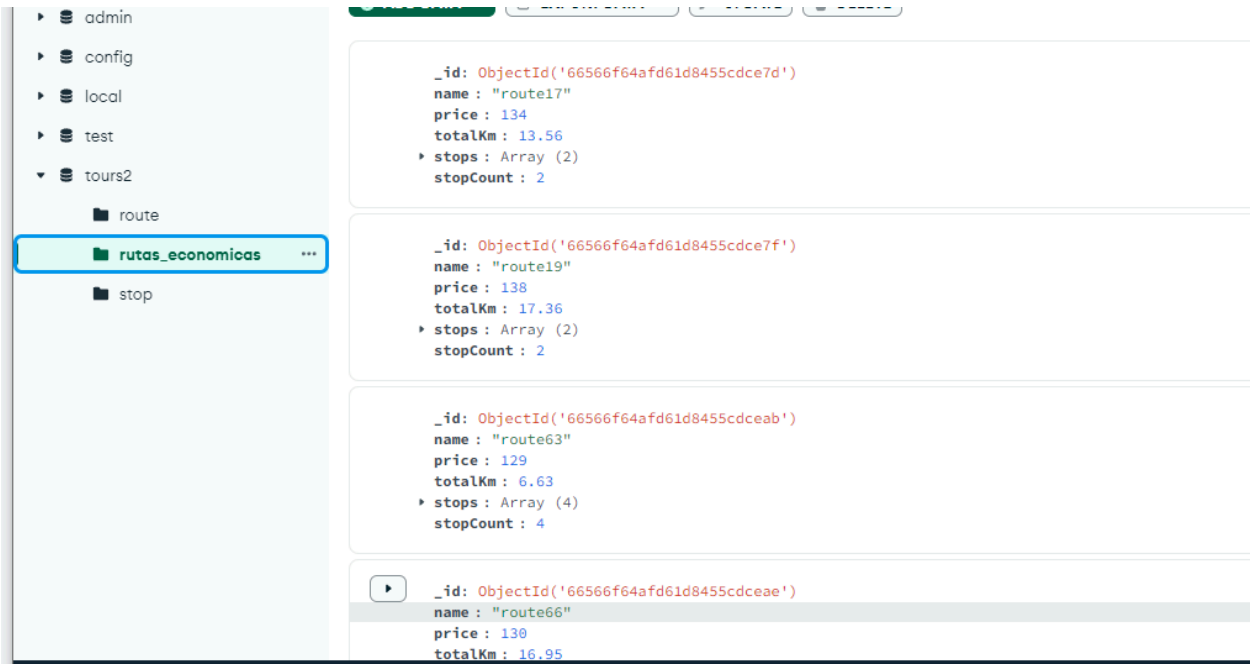
```
db.route.aggregate([
  { $unwind: "$stops" },
  { $group: { _id: "$stops", count: { $sum: 1 } } },
  { $sort: { count: -1 } },
  { $limit: 1 },
  {
    $lookup: {
      from: "stop",
      localField: "_id",
      foreignField: "code",
      as: "stop_info"
    }
  },
  {
    $project: {
      _id: 0,
      stop_info: { $arrayElemAt: ["$stop_info", 0] }
    }
  },
  {
    $project: {
      _id: "$stop_info._id",
      name: "$stop_info.name",
      code: "$stop_info.code",
      descripcion: "$stop_info.descripcion
     }
  }
]).pretty();
```

```
... ]).pretty();
[
  {
    _id: ObjectId('66566f64afd61d8455cdcdf6'),
    name: 'Diagonal Norte',
    code: 0
  }
]
```

viii)

Obtenga las Route que tengan un precio inferior a 150. A ellos agregué una nueva propiedad que especifique la cantidad de Stops que posee la Route. Cree una nueva colección llamada “rutas_economicas” y almacene estos elementos.

```
db.route.aggregate([
  { $match: { price: { $lt: 150 } } },
  {
    $addFields: {
      stopCount: { $size: "$stops" }
    }
  },
  { $out: "rutas_economicas" }
]);
```



ix)

Por cada Stop existente en su colección, calcule el precio promedio de las Routes que la incluyen.

```
db.stop.aggregate([
  {
    $lookup: {
      from: "route",
```

```
        localField: "code",
        foreignField: "stops",
        as: "routes_info"
      }
    },
    { $unwind: "$routes_info" },
    {
      $group: {
        _id: "$code",
        avgPrice: { $avg: "$routes_info.price" }
      }
    },
    { $project: { _id: 0, stopCode: "$_id", avgPrice: 1 } }
]);
```

```
[
  { avgPrice: 559.3061224489796, stopCode: 102 },
  { avgPrice: 569.7432521395655, stopCode: 108 },
  { avgPrice: 549.3493898522801, stopCode: 14 },
  { avgPrice: 545.06466361855, stopCode: 69 },
  { avgPrice: 560.568281938326, stopCode: 22 },
  { avgPrice: 559.1184210526316, stopCode: 74 },
  { avgPrice: 558.8464516129033, stopCode: 11 },
  { avgPrice: 551.4775619557937, stopCode: 117 },
  { avgPrice: 562.5682267633487, stopCode: 19 },
  { avgPrice: 540.9908015768725, stopCode: 32 },
  { avgPrice: 537.8149100257069, stopCode: 43 },
  { avgPrice: 554.7635532331809, stopCode: 98 },
  { avgPrice: 546.5811247575954, stopCode: 77 },
  { avgPrice: 556.4692211055276, stopCode: 118 },
  { avgPrice: 544.5485751295337, stopCode: 26 },
  { avgPrice: 544.6641535298149, stopCode: 50 },
  { avgPrice: 560.2824631860776, stopCode: 79 },
  { avgPrice: 557.4335038363172, stopCode: 111 },
  { avgPrice: 540.688178913738, stopCode: 35 },
  { avgPrice: 552.9093333333333, stopCode: 67 }
]
```

Parte 4: Índices

14)

Creamos la nueva BD:

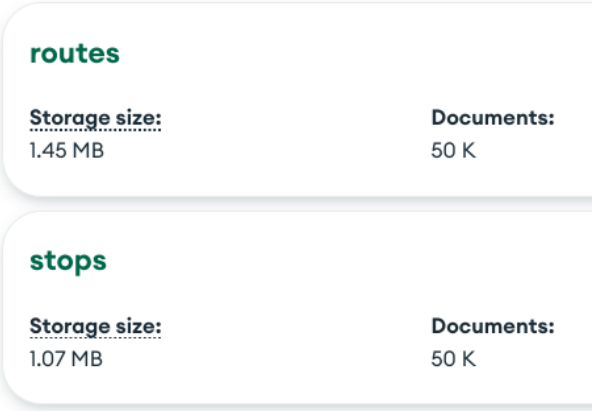
```
$ use tours3
```

Y cargamos el script *generador2.js* mediante el comando *load()*.

```
$ load("$HOME/Desk)
```

Nota: DBCompass no permite ejecutar comandos en su terminal, por lo que para esto se puede ejecutar el código del script literal dentro de ella, o ejecutar *load()* en la terminal Mongosh.

Se generaron dos colecciones, siendo *Route* y *Stop*, con 50000 documentos la primera colección y 119 la segunda colección.



15)

A continuación se listan los índices en la colección *Route*.

```
$ db.route.getIndexes()
```

```
[ { v: 2, key: { _id: 1 }, name: '_id_' } ]
```

Donde su salida lo siguiente:

- **v:** el cual es la versión del índice, siendo la número 2..
- **key:** el cual es el/los campos sobre el cual se registra el índice, siendo en este caso únicamente el campo *_id*.
- **name:** el nombre del índice en cuestión, donde es *"_id_"*.

16)

Se procede a buscar los documentos de la colección *Stops* donde en el nombre se incluya el string "11", y se visualiza el plan de ejecución de la consulta:

```
$ db.stops.find({name: {$regex: "11"}}).explain("executionStats")
```

El resultado de la consulta es el siguiente:

Se visualizan únicamente los primeros dos resultados por simpleza

Las estadísticas de ejecución son las siguientes:

A través del método *explain()* se pueden visualizar información sobre los planes de ejecución de la consulta y sus estadísticas, donde según los datos anteriores se puede determinar la cantidad de documentos que se examinaron siendo 50000 documentos, indicados en el campo ***totalDocsExamined***, para retornar como resultado de la consulta 2291 único documento, indicado por el campo ***nReturned***, donde la consulta llevaría un tiempo de ejecución de 53 milisegundos sin contar la posible utilización de caché, indicado en el campo ***executionTimeMillis***, y la búsqueda se llevaría a cabo sin la utilización de ningún índice, especificado a través del campo ***totalKeysExaminated***.

17)

Se procede a crear un ***Text index*** sobre el campo *name* de la colección *Stop*.

Cabe destacar que el operador de búsqueda ***text*** para índices de texto no es posible utilizarlo, ya que este índice separa por palabras, aspecto que no aplica a este tipo de búsqueda, donde se debe analizar si la palabra contiene un determinado substring “11”.

A continuación se realiza la búsqueda utilizando aggregation framework para aprovechar el uso de hilos y poder optimizar la consulta:

```
$ db.stops.createIndex( { "name": "text" } )
$ db.stops.aggregate([
 {
   $match: {
    name: { $regex: "11" }
   }
 }
]).explain("executionStats")
```

El resultado de la consulta es el siguiente:

```
< {
    _id: ObjectId('664f6ee48341a39140640cc7'),
    name: 'Stop 11',
    code: 11,
    description: 'Description of stop 11'
  }
  {
    _id: ObjectId('664f6ee48341a39140640d2a'),
    name: 'Stop 110',
    code: 110,
    description: 'Description of stop 110'
  }
```

Las estadísticas de ejecución son las siguientes:

```
executionStats: {
  executionSuccess: true,
  nReturned: 2291,
  executionTimeMillis: 29,
  totalKeysExamined: 0,
  totalDocsExamined: 50000,
  executionStages: {
    stage: 'COLLSCAN',
    filter: {
      name: BSONRegExp('11', '')
    },
```

Ahora se puede visualizar que a pesar de no haber utilizado el índice definido, ya que no aplica a este tipo de búsqueda, mejoran los tiempos a 29 milisegundos, a pesar de haber analizado todos los 50000 documentos y retornado la misma cantidad de resultados de 2291 documentos.

18)

Procedemos a declarar la variable en la terminal:

```
$ var polygonCaba = {
 "type":"MultiPolygon",
 "coordinates":[[[
   [-58.46305847167969,-34.53456089748654],
   [-58.49979400634765,-34.54983198845187],
   [-58.532066345214844,-34.614561581608186],
   [-58.528633117675774,-34.6538270014492],
   [-58.48674774169922,-34.68742794931483],
   [-58.479881286621094,-34.68206400648744],
   [-58.46855163574218,-34.65297974261105],
   [-58.465118408203125,-34.64733112904415],
   [-58.4585952758789,-34.63998735602951],
   [-58.45344543457032,-34.63603274732642],
   [-58.447265625,-34.63575026806082],
   [-58.438339233398445,-34.63038297923296],
   [-58.38100433349609,-34.62162507826766],
   [-58.38237762451171,-34.59251960889388],
   [-58.378944396972656,-34.5843230246475],
   [-58.46305847167969,-34.53456089748654]
 ]]]
}
```

Procedemos a realizar la búsqueda de las rutas donde su punto de comienzo se encuentre dentro del área especificada por el polígono de ciudad de buenos aires.

```
$ db.routes.find({startPoint: { $geoWithin: { $geometry: polygonCaba }
}}).explain("executionStats")
```

El resultado de la consulta es el siguiente:

Se visualizan los primeros dos resultados por simpleza

Las estadísticas de ejecución son las siguientes:

Aquí se puede analizar que la consulta demora 81 milisegundos, no se utilizaron índices, y de todos los 50000 documentos analizados se retornan 10884 documentos como resultado.

Procedemos a crear un índice espacial para poder agilizar las búsquedas realizadas sobre puntos en un área.

```
$ db.routes.createIndex({ startPoint: "2dsphere" })
```

Volvemos a ejecutar la consulta anterior para analizar si existe mejora de rendimiento:

```
$ db.routes.find({startPoint: { $geoWithin: { $geometry: polygonCaba }
}}).explain("executionStats")
```

Las estadísticas de ejecución son las siguientes:

Podemos ver que la consulta demoró 44 milisegundos, reduciendo casi en un 50% el tiempo, analizando ahora 13873 documentos.