MongoDB (que proviene de «humongous-enorme ») es la base de datos NoSQL líder del mercado.

Cuándo uno se inicia en MongoDB se puede sentir perdido. No tenemos tablas, no tenemos registros y lo que es más importante, no tenemos SQL. Aun así, MongoDB es una seria candidata para almacenar los datos de nuestras aplicaciones.

MongoDB es una base de datos orientada a documentos. Esto quiere decir que en lugar de guardar los datos en registros, guarda los datos en documentos. Estos documentos son almacenados en BSON, que es una representación binaria de JSON.

Una de las diferencias más importantes con respecto a las bases de datos relacionales, es que **no es necesario seguir un esquema**. Los documentos de una misma colección - concepto similar a una tabla de una base de datos relacional -, pueden tener esquemas diferentes.

Imaginemos que tenemos una colección a la que llamamos Personas. Un documento podría almacenarse de la siguiente manera:

```
{
 Nombre: "Pedro",
 Apellidos: "Martínez Campo",
 Edad: 22,
 Aficiones: ["fútbol","tenis","ciclismo"],
 Amigos: [
  {
   Nombre:"María",
   Edad:22
  },
  {
   Nombre:"Luis",
   Edad:28
  }
 ]
}
```

El documento anterior es un clásico documento JSON. Tiene strings, arrays, subdocumentos y números. En la misma colección podríamos guardar un documento como este:

```
{
 Nombre: "Luis",
 Estudios: "Administración y Dirección de Empresas",
 Amigos:12
}
```

Este documento no sigue el mismo esquema que el primero. Tiene menos campos, algún campo nuevo que no existe en el documento anterior e incluso un campo de distinto tipo.

Esto que es algo impensable en una base de datos relacional, es algo totalmente válido en **MongoDB**.

¿Cómo funciona MongoDB?

MongoDB está escrito en C++, aunque las consultas se hacen pasando objetos JSON como parámetro. Es algo bastante lógico, dado que los propios documentos se almacenan en BSON. Por ejemplo:

```
db.Clientes.find({Nombre:"Pedro"});
```

La consulta anterior buscará todos los clientes cuyo nombre sea Pedro.

¿Dónde se puede utilizar MongoDB?

MongoDB es especialmente útil en entornos que requieran escalabilidad. Con sus opciones de replicación y sharding, que son muy sencillas de configurar, podemos conseguir un sistema que escale horizontalmente sin demasiados problemas.

¿Dónde no se debe usar MongoDB?

En esta base de datos **no existen las transacciones**. Aunque nuestra aplicación puede utilizar alguna técnica para simular las transacciones, **MongoDB** no tiene esta capacidad. Solo garantiza operaciones atómicas a nivel de documento. Si las transacciones son algo indispensable en nuestro desarrollo, deberemos pensar en otro sistema.

Tampoco existen los JOINS. Para consultar datos relacionados en dos o más colecciones, tenemos que hacer más de una consulta. En general, si nuestros datos pueden ser estructurados en tablas, y necesitamos las relaciones, es mejor que optemos por un RDBMS clásico.

Entrar a la consola

Bien ahora ya estamos listos para entrar a la base de datos y comenzar a jugar con ella. Para ello luego de tener el servicio de MongoDB corriendo, ejecutaremos el comando mongo y esto nos llevará a la consola interna de la instancia.

Para ir a la consola de MongoDB en sistemas Windows, si seguiste las instrucciones en el curso anterior, luego de iniciar el servicio, ejecuta el archivo C:\mongodb\bin\mongo.exe

Operaciones Básicas

Creación de registros - .insert()

Las operaciones son como funciones de Javascript, así que llamaremos al objeto base de datos db y crearemos una nueva propiedad o lo que se asemejaría al concepto de tabla con el nombre de autores y le asociaremos su valor correspondiente (un objeto autor), es decir, una **colección** con un **documento** asociado:

```
> db.autores.insert({
  nombre    : 'Jonathan',
  apellido  : 'Wiesel',
  secciones : ['Como lo hago' , 'MongoDB']
});
```

Los **documentos** se definen como los objetos Javascript, u objetos JSON.

Inclusive es posible declarar el documento como un objeto, almacenarlo en una variable y posteriormente insertarlo de la siguiente manera:

```
> var autorDelPost = {
  nombre    : 'Jonathan',
  apellido  : 'Wiesel',
  secciones : ['Como lo hago' , 'MongoDB']
};

> db.autores.insert(autorDelPost);
```

Ahora si ejecutamos el comando show collections podremos ver que se encuentra nuestra nueva colección de autores:

```
autores
...
```

Agreguemos un par de autores más:

```
> db.autores.insert({
  nombre    : 'Oscar',
  apellido  : 'Gonzalez',
  secciones : ['iOS' , 'Objective C' , 'NodeJS' ],
  socialAdmin : true
});
> db.autores.insert({
  nombre    : 'Alberto',
  apellido  : 'Grespan',
  secciones : 'Git',
  genero    : "M"
});
```

Veamos que insertamos nuevos documentos en la colección de autores que tienen otra estructura, en MongoDB esto es completamente posible y es una de sus ventajas.

Búsqueda de registros - .find()

Hagamos un *query* o una busqueda de todos registros en la colección de autores.

```
> db.autores.find();
```

```
{ "_id" : ObjectId("5232344a2ad290346881464a"), "nombre" : "Jonathan", "apellido" : "Wiesel", "secciones" : [ "Como lo hago", "Noticias" ] }
{ "_id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar", "apellido" : "Gonzalez", "secciones" : [ "iOS", "Objective C", "NodeJS" ], "socialAdmin" : true }
{ "_id" : ObjectId("5232383a2ad290346881464c"), "nombre" : "Alberto", "apellido" : "Grespan", "secciones" : "Git", "genero" : "M" }
```

Notemos que la búsqueda nos arroja los objetos resultantes, en este caso los documentos de los 3 autores que insertamos acompañados del identificador único que crea MongoDB, este campo _id se toma además como indice por defecto.

Si lo deseas puedes manualmente especificar el valor del campo _id cuando estas insertando los registros con el comando db.coleccion.insert(); sin embargo ten en cuenta que debes asegurar que este valor sea único, de lo contrario los registros con dicho campo duplicado resultarán en error por clave primaria duplicada.

Una búsqueda como la anterior sería similar en SQL a:

```
SELECT * FROM autores
```

Filtros

Digamos que ahora queremos hacer la búsqueda pero filtrada por los algún parámetro. Para esto sólo debemos pasar el filtro deseado a la función find(), busquemos a los administradores sociales para probar:

```
> db.autores.find({ socialAdmin: true });
```

```
{ "_id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar", "apellido" : "Gonzalez", "secciones" : [ "iOS", "Objective C", "NodeJS" ], "socialAdmin" : true }
```

En SQL sería similar a:

```
SELECT * FROM autores WHERE socialAdmin = true
```

Probemos ahora filtrar por varias condiciones, primero probemos con filtros donde TODOS se deben cumplir:

```
> db.autores.find({ genero: 'M', secciones: 'Git' });
```

```
{ "_id" : ObjectId("5232383a2ad290346881464c"), "nombre" : "Alberto", "apellido" : "Grespan", "secciones" : "Git", "genero" : "M" }
```

Es importante destacar que si el documento resultante hubiese tenido en la propiedad secciones un arreglo en lugar de una cadena de caracteres, si dicho arreglo tuviese el valor Git también cumple la condición.

En SQL sería similar a:

SELECT * FROM autores WHERE genero = 'M' AND secciones = 'Git'

Veamos ahora un ejemplo un poco más avanzado de filtros con condiciones donde queremos que solo ALGUNA de ellas se cumpla:

```
> db.autores.find({
  $or: [
    {socialAdmin : true},
    {genero: 'M'}
  ]
 });
```

```
{ "_id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar", "apellido" : "Gonzalez", "secciones" : [ "iOS", "Objective C", "NodeJS" ], "socialAdmin" : true }
{ "_id" : ObjectId("5232383a2ad290346881464c"), "nombre" : "Alberto", "apellido" : "Grespan", "secciones" : "Git", "genero" : "M" }
```

En este caso estamos filtrando por aquellos autores que son administradores sociales ó aquellos que tengan el campo género con el carácter M.

En SQL sería similar a:

SELECT * FROM autores WHERE socialAdmin = true OR genero = 'M'

Limitar y Ordenar

Si quisiéramos limitar los resultados a un número máximo especificado de registros es tan fácil como agregar .limit(#) al final del comando .find():

```
> db.autores.find().limit(1)
```

```
{ "_id" : ObjectId("5232344a2ad290346881464a"), "nombre" : "Jonathan", "apellido" : "Wiesel", "secciones" : [ "Como lo hago", "MongoDB" ] }
```

En SQL sería similar a:

SELECT * FROM autores LIMIT 1

La misma modalidad sigue la funcionalidad de ordenar los registros por un campo en particular, el cual servirá de argumento a la función .sort():

```
> db.autores.find().sort({apellido : 1})
```

{ "_id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar", "apellido" : "Gonzalez", "secciones" : ["iOS", "Objective C", "NodeJS"], "socialAdmin" : true }
{ "_id" : ObjectId("5232383a2ad290346881464c"), "nombre" : "Alberto", "apellido" : "Grespan", "secciones" : "Git", "genero" : "M" }
{ "_id" : ObjectId("5232344a2ad290346881464a"), "nombre" : "Jonathan", "apellido" : "Wiesel", "secciones" : ["Como lo hago", "MongoDB"] }

El número 1 que acompaña al argumento de ordenamiento es el tipo de orden, 1 para descendiente y -1 para ascendente

En SQL sería similar a:

SELECT * FROM autores ORDER BY apellido DESC

También podemos combinar ambas funciones tan solo llamando una después de otra:

> db.autores.find().sort({apellido : 1}).limit(1)

{ "_id" : ObjectId("523236022ad290346881464b"), "nombre" : "Oscar", "apellido" : "Gonzalez", "secciones" : ["iOS", "Objective C", "NodeJS"], "socialAdmin" : true }

Otros filtros

Existen varios operadores más para filtrar las búsquedas; eso lo dejaremos para más adelante para no sobrecargarte de información tan rápido, pero no te preocupes que no lo pasaremos por alto.

Eliminación de registros - .remove() y .drop()

Si entendiste como buscar registros pues eliminarlos es igual de fácil. Para esto existen 4 posibilidades:

- Eliminar los documentos de una colección que cumplan alguna condición.
- Eliminar todos los documentos de una colección.
- Eliminar la colección completa.

Probemos eliminandome a mí de la colección de autores:

> db.autores.remove({ nombre: 'Jonathan' });

En SQL sería similar a:

DELETE FROM autores WHERE nombre = 'Jonathan'

¿Fácil no?. Eliminemos ahora a los demás autores:

> db.autores.remove();

En SQL sería similar a:

DELETE FROM autores

Ahora que la colección ha quedado vacía deshagamonos de ella:

> db.autores.drop();

En SQL sería similar a:

DROP TABLE autores

Actualizaciones / Updates

Estructura

Para modificar los documentos que ya se encuentran almacenados usaremos el comando .update() el cual tiene una estructura como esta:

```
db.coleccion.update(
   filtro,
   cambio,
   {
      upsert: booleano,
      multi:  booleano
   }
);
```

Aclaremos un poco lo que nos indica la estructura.

filtro - debemos especificar como encontrar el registro que desemos modificar, sería el mismo tipo de filtro que usamos en las búsquedas o **finders**.

cambio - aquí especificamos los cambios que se deben hacer. Sin embargo ten en cuenta que hay 2 tipos de cambios que se pueden hacer:

- Cambiar el documento completo por otro que especifiquemos.
- Modificar nada más los campos especificados.

upsert (opcional, false por defecto) - este parametro nos permite especificar en su estado true que si el filtro no encuentra ningun resutlado entonces el cambio debe ser insertado como un nuevo registro.

multi (opcional, false por defecto) - en caso de que el filtro devuelva más de un resultado, si especificamos este parametro como true, el cambio se realizará a todos los resultados, de lo contrario solo se le hará al primero (al de menor Id).

Actualización sobrescrita (overwrite)

Bien, probemos insertando nuevo autor, el cual modificaremos luego:

```
> db.autores.insert({
   nombre     :  'Ricardo',
   apellido   :  'S'
});
```

Ahora probemos el primer caso, cambiar todo el documento, esto significa que en lugar de cambiar solo los campos que especifiquemos, el documento será sobreescrito con lo que indiquemos:

```
> db.autores.update(
   {nombre: 'Ricardo'},
   {
      nombre: 'Ricardo',
      apellido: 'Sampayo',
      secciones: ['Ruby','Rails'],
      esAmigo: false
   }
);
```

Notemos que como primer parámetro indicamos el filtro, en este caso que el nombre sea Ricardo, luego indicamos el cambio que hariamos, como estamos probando el primer caso indicamos el documento completo que queremos que sustituya al actual. Si hacemos db.autores.find({nombre:'Ricardo'}); podremos ver que en efecto el documento quedó como especificamos:

```
{ "_id" : ObjectId("523c91f2299e6a9984280762"), "nombre" : "Ricardo", "apellido" : "Sampayo", "secciones" : [  "Ruby",  "Rails" ], "esAmigo" : false }
```

Sobreescbirir el documento no cambiará su identificador único _id.

Operadores de Modificación

Ahora probemos cambiar los campos que deseamos, para este caso haremos uso de lo que se denominan como operadores de modificación.

Hablemos un poco sobre algunos de estos operadores antes de verlos en acción:

- $inc - incrementa en una cantidad numerica especificada el valor del campo a en cuestión.
- $rename - renombrar campos del documento.
- $set - permite especificar los campos que van a ser modificados.
- $unset - eliminar campos del documento.

Referentes a arreglos:

- $pop - elimina el primer o último valor de un arreglo.
- $pull - elimina los valores de un arreglo que cumplan con el filtro indicado.
- $pullAll - elimina los valores especificados de un arreglo.
- $push - agrega un elemento a un arreglo.
- $addToSet - agrega elementos a un arreglo solo sí estos no existen ya.
- $each - para ser usado en conjunto con $addToSet o $push para indicar varios elementos a ser agregados al arreglo.

Hagamos una prueba sobre nuestro nuevo documento:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $set: { esAmigo: true , age : 25 }
   }
);
```

En este caso estamos usando el operador $set para 2 propositos a la vez:

- Actualizar el valor de un campo (cambiamos esAmigo de false a true).
- Creamos un campo nuevo (age) asignandole el valor 25.

Supongamos que Ricardo cumplió años en estos días, así que para incrementar el valor de su edad lo podemos hacer así:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $inc: { age : 1 }
   }
);
```

También podemos indicar números negativos para decrementar.

Aquí hablamos español, así que cambiemos ese campo age por lo que le corresponde:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $rename: { 'age' : 'edad' }
   }
);
```

Los que trabajamos en Codehero somos todos amigos así que no es necesario guardar el campo esAmigo:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $unset: { esAmigo : '' }
   }
);
```

El valor que le "asignes" al campo a eliminar no tendrá ningun efecto, pero es necesario escribirlo por motivos de sintaxis. Pasemos ahora a la parte de modificación de arreglos, agreguemosle algunas secciones extra a nuestro autor:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $push: { secciones : 'jQuery' }
   }
);
```

Si se quiere asegurar que el elemento no esté duplicado se usaría $addToSet

Esto agregará al final del arreglo de secciones el elemento jQuery.

Agreguemos algunas secciones más en un solo paso:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $push: { secciones : { $each : ['Haskell','Go','ActionScript'] } }
   }
);
```

Bueno en realidad Ricardo no maneja desde hace un tiempo ActionScript así que eliminemos ese ultimo elemento del arreglo:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $pop: { secciones : 1 }
   }
);
```

Para eliminar el último elemento se coloca 1, para el primero -1.

Ricardo hace tiempo que no nos habla sobre jQuery asi que hasta que no se reivindique quitemoslo de sus secciones:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $pull: { secciones : 'jQuery' }
   }
);
```

Pensandolo bien, Ricardo nunca nos ha hablado de Haskell ni Go tampoco, eliminemoslos también:

```
> db.autores.update(
   { nombre: 'Ricardo' },
   {
      $pullAll: { secciones : ['Haskell','Go'] }
   }
);
```

Comando .save()

Otra manera para actualizar o insertar registros es mediante el uso del comando .save(). Este comando recibe como parámetro únicamente un documento.

Insertar un registro es tal cual como si hicieramos un .insert():

```
> db.autores.save({
   nombre:    'Ramses'
});
```

En cuanto al caso de actualización de registros te estarás preguntando:

¿Si solo recibe un documento, como sabe Mongo que documento debe actualizar?

En estos casos puedes hacer el equivalente a una actualización sobrescrita con tan solo indicar el _id del registro a actualizar como parte del nuevo documento.

```
> db.autores.find({nombre: 'Ramses'});

{ "_id" : ObjectId("5246049e7bc1a417cc91ec8c"), "nombre" : "Ramses" }

> db.autores.save({
   _id:       ObjectId('5246049e7bc1a417cc91ec8c')
   nombre:    'Ramses',
   apellido:  'Velasquez',
   secciones:  ['Laravel', 'PHP']
});
```

En esta caso particular, debido a que Mongo le asigno automáticamente ese ID autogenerado poco amigable, ese mismo es el que pusimos en nuestro documento para que Mongo sepa que debe actualizar ese registro, algunos recomiendan usar _id que no sean establecidos por Mongo sino por la librería del cliente para evitar trabajar con este tipo de identificadores poco amigables.

Modelado de Datos

Tipos de Datos

Al comienzo de la serie explicamos que los documentos de MongoDB son como objetos JSON, para ser especificos son de tipo BSON (JSON Binario), esta estrategia permite la serialización de documentos tipo JSON codificados binariamente. Veamos algunos de los tipos de datos que soporta:

- String - Cadenas de caracteres.
- Integer - Números enteros.
- Double - Números con decimales.
- Boolean - Booleanos verdaderos o falsos.
- Date - Fechas.
- Timestamp - Estampillas de tiempo.
- Null - Valor nulo.
- Array - Arreglos de otros tipos de dato.
- Object - Otros documentos embebidos.
- ObjectID - Identificadores únicos creados por MongoDB al crear documentos sin especificar valores para el campo _id.
- Data Binaria - Punteros a archivos binarios.
- Javascript - código y funciones Javascript.

Patrones de Modelado

Existen 2 patrones principales que nos ayudarán a establecer la estructura que tendrán los documentos para lograr relacionar datos que en una base de datos relacional estarían en diferentes tablas.

Embeber

Este patrón se enfoca en incrustar documentos uno dentro de otro con la finalidad de hacerlo parte del mismo registro y que la relación sea directa.

Si tienes experiencia en bases de datos orientadas o objetos, este patrón seguiría el mismo principio para la implementación de un TDA (Tipo de Dato Abstracto).

Referenciar

Este patrón busca imitar el comportamiento de las claves foráneas para relacionar datos que deben estar en colecciones diferentes.

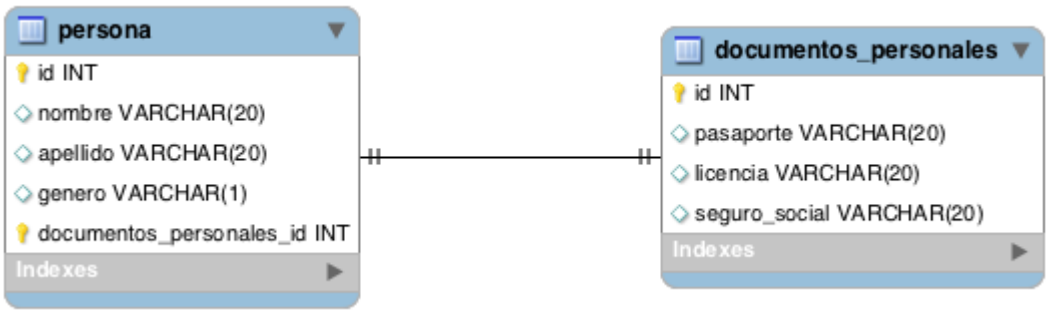
Debes tomar en cuenta cuando estés en el proceso de modelado que si piensas hacer muchas actualizaciones sobre datos de colecciones relacionadas (en especial modificaciones atómicas), trata en lo posible de que aquellos datos que se vayan a modificar se encuentren en el mismo documento.

Modelado de Relaciones

Bien, ha llegado el momento de aprender a transformar las relaciones de las tablas en las bases de datos relacionales. Empecemos con lo más básico.

Relaciones 1-1.

Muchas opiniones concuerdan que las relaciones 1 a 1 deben ser finalmente normalizadas para formar una única tabla; sin embargo existen consideraciones especiales donde es mejor separar los datos en tablas diferentes. Supongamos el caso que tenemos una tabla ***persona*** y otra tabla ***documentos personales***, donde una persona tiene un solo juego de documentos personales y que un juego de documentos personales solo puede pertenecer a una persona.



Si traducimos esto tal cual a lo que sabemos hasta ahora de MongoDB sería algo así:

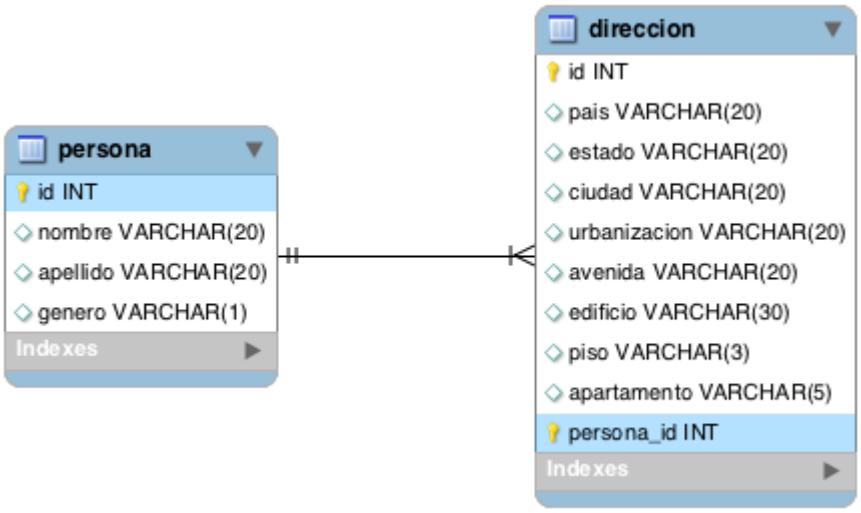
```
Persona = {
   nombre     :  'Jonathan',
   apellido   :  'Wiesel',
   genero     :  'M'
}

DocumentosPersonales = {
   pasaporte      :  'D123456V7',
   licencia       :  '34567651-2342',
   seguro_social  :  'V-543523452'
}
```

Para los casos de relaciones 1-a-1 se utiliza el patrón de **embeber** un documento en otro, por lo que el documento final quedaría así:

```
Persona = {
   nombre     :  'Jonathan',
   apellido   :  'Wiesel',
   genero     :  'M',
   documentos :  {
      pasaporte      :  'D123456V7',
      licencia       :  '34567651-2342',
      seguro_social  :  'V-543523452'
   }
}
```

Relaciones 1-*

Supongamos ahora el caso de una tabla ***persona*** y otra tabla ***dirección***. Donde una persona puede poseer varias direcciones.



Por el momento no nos pondremos creativos al pensar que una dirección puede pertenecer a más de una persona.

Traduciendolo tal cual a MongoDB tendríamos algo así:

```
Persona = {
   nombre     :  'Jonathan',
   apellido   :  'Wiesel',
   genero     :  'M'
}

Direccion1 = {
   pais          :  'Venezuela',
   estado        :  'Distrito Capital'
   ciudad        :  'Caracas'
   urbanizacion  :  'La Florida',
   avenida       :  ...,
   edificio      :  ...,
   piso          :  ...,
   apartamento   :  ...
}

Direccion2 = {
   pais          :  'Estados Unidos',
   estado        :  'Florida'
   ciudad        :  'Miami'
   urbanizacion  :  'Aventura',
   avenida       :  ...,
   edificio      :  ...,
   piso          :  ...,
   apartamento   :  ...
}
```

Ahora para transformar la relación tenemos 2 opciones.

Podemos embeber las direcciones en el documento de la persona al establecer un arreglo de direcciones embebidas:

```
Persona = {
   nombre     :  'Jonathan',
   apellido   :  'Wiesel',
   genero     :  'M',
   direcciones :  [{
      pais          :  'Venezuela',
      estado        :  'Distrito capital'
      ciudad        :  'Caracas',
      urbanizacion  :  'La Florida',
      avenida       :  ...,
      edificio      :  ...,
      piso          :  ...,
```

```
      apartamento    :  ...
   },{
      pais          :  'Estados Unidos',
      estado        :  'Florida'
      ciudad        :  'Miami'
      urbanizacion  :  'Aventura',
      avenida       :  ...,
      edificio      :  ...,
      piso          :  ...,
      apartamento   :  ...
   }]
}
```

ó podemos dejarlo en documentos separados. Para esta segunda opción tenemos 2 enfoques.

Uno sería agregar un campo de referencia a ***dirección*** en ***persona***:

```
Direccion1 = {
   _id           :  1,
   pais          :  'Venezuela',
   estado        :  'Distrito Capital',
   ciudad        :  'Caracas'
   urbanizacion  :  'La Florida',
   avenida       :  ...,
   edificio      :  ...,
   piso          :  ...,
   apartamento   :  ...
}

Direccion2 = {
   _id           :  2,
   pais          :  'Estados Unidos',
   estado        :  'Florida',
   ciudad        :  'Miami'
   urbanizacion  :  'Aventura',
   avenida       :  ...,
   edificio      :  ...,
   piso          :  ...,
   apartamento   :  ...
}

Persona = {
   nombre     :  'Jonathan',
   apellido   :  'Wiesel',
   genero     :  'M',
   direcciones :  [1,2]
}
```

y el otro sería agregar un campo de referencia a ***persona*** en ***dirección***:

```
Direccion1 = {
   _id          :  1,
   pais         :  'Venezuela',
   estado       :  'Distrito Capital',
   ciudad       :  'Caracas'
   urbanizacion   :  'La Florida',
   avenida      :  ...,
   edificio     :  ...,
   piso         :  ...,
   apartamento    :  ...,
   persona_id     :  1
}

Direccion2 = {
   _id          :  2,
   pais         :  'Estados Unidos',
   estado       :  'Florida',
   ciudad       :  'Miami'
   urbanizacion   :  'Aventura',
   avenida      :  ...,
   edificio     :  ...,
   piso         :  ...,
   apartamento    :  ...,
   persona_id     :  1
}

Persona = {
   _id       :  1
   nombre     :  'Jonathan',
   apellido   :  'Wiesel',
   genero     :  'M'
}
```

En lo posible trata de utilizar la opción de embeber si los arreglos no variarán mucho ya que al realizar la búsqueda de la persona obtienes de una vez las direcciones, mientras que al trabajar con referencias tu aplicación debe manejar una lógica múltiples búsquedas para resolver las referencias, lo que sería el equivalente a los *joins*.

En caso de utilizar la segunda opción, ¿Cual de los 2 últimos enfoques utilizar?. En este caso debemos tomar en cuenta que tanto puede crecer la lista de direcciones, en caso que la tendencia sea a crecer mucho, para evitar arreglos mutantes y en constante crecimiento el **segundo** enfoque sería el más apropiado.

Relaciones *-*

Finalmente nos ponemos creativos a decir que, en efecto, varias personas pueden pertenecer a la misma dirección.

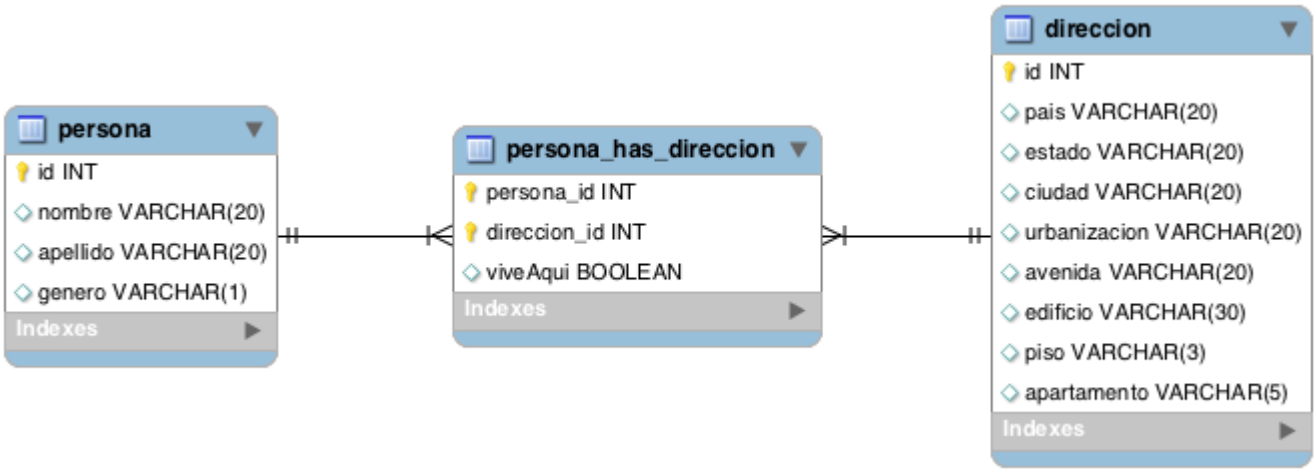
Al aplicar normalización quedaría algo así:

Para modelar este caso es muy similar al de relaciones uno a muchos con referencia por lo que colocaremos en ambos tipos de documento un arreglo de referencias al otro tipo. Agreguemos una persona adicional para demostrar mejor el punto:

```
Direccion1 = {
   _id          :  1,
   pais         :  'Venezuela',
   estado       :  'Distrito Capital',
   ciudad       :  'Caracas'
   urbanizacion   :  'La Florida',
   avenida      :  ...,
   edificio     :  ...,
   piso         :  ...,
   apartamento    :  ...,
   personas     :  [1000]
}

Direccion2 = {
   _id          :  2,
   pais         :  'Estados Unidos',
   estado       :  'Florida',
   ciudad       :  'Miami'
   urbanizacion   :  'Aventura',
   avenida      :  ...,
   edificio     :  ...,
   piso         :  ...,
```

```
   apartamento    :  ...,
   personas       :  [1000,1001]
}

Persona1 = {
   _id       :  1000,
   nombre    :  'Jonathan',
   apellido  :  'Wiesel',
   genero    :  'M',
   direcciones :  [1,2]
}

Persona2 = {
   _id       :  1001,
   nombre    :  'Carlos',
   apellido  :  'Cerqueira',
   genero    :  'M',
   direcciones :  [2]
}
```

Seguro debes estar esperando el caso más complejo de todos, aquellas ocasiones donde la tabla intermedia tiene campos adicionales.



Tomando como base el ejemplo anterior, agregaremos el campo adicional usando el patrón para **embeber** de la siguiente manera:

```
Direccion1 = {
   _id          :  1,
   pais         :  'Venezuela',
   estado       :  'Distrito Capital',
   ciudad       :  'Caracas'
   urbanizacion   :  'La Florida',
   avenida      :  ...,
   edificio     :  ...,
```

```
   piso          :  ...,
   apartamento   :  ...,
   personas      :  [1000]
}

Direccion2 = {
   _id           :  2,
   pais          :  'Estados Unidos',
   estado        :  'Florida',
   ciudad        :  'Miami'
   urbanizacion  :  'Aventura',
   avenida       :  ...,
   edificio      :  ...,
   piso          :  ...,
   apartamento   :  ...,
   personas      :  [1000,1001]
}

Persona1 = {
   _id       :  1000,
   nombre    :  'Jonathan',
   apellido  :  'Wiesel',
   genero    :  'M',
   direcciones :  [{
         direccion_id  :  1,
         viveAqui      :  true
      },{
         direccion_id  :  2,
         viveAqui      :  false
      }]
}

Persona2 = {
   _id       :  1001,
   nombre    :  'Carlos',
   apellido  :  'Cerqueira',
   genero    :  'M',
   direcciones :  [{
         direccion_id  :  2,
         viveAqui      :  true
      }]
}
```

De igual manera se pudiera embeber el campo viveAqui del lado de las direcciones, esto dependerá de como planeas manipular los datos.

Secuencias Auto-incrementadas

Una de las necesidades con la cual nos hemos encontrado en algún punto al tener nuestro esquema de base de datos es la de poseer aquella estructura que permite asignar automáticamente el siguiente valor de la secuencia de un campo particular al insertar un nuevo registro. A esta funcionalidad se le conoce como secuencias auto-incrementadas, algunas bases de datos permiten establecer un campo con esta propiedad con tan solo definir una restricción o *constraint*; sin embargo el esquema de datos de MongoDB no adopta nativamente dicho aspecto pero permite su implementación siguiendo el patrón abajo descrito.

Para entender el comportamiento supongamos el caso que tenemos una colección de autores y deseamos establecer el campo _id como auto-incrementado.

Este patrón se basa en el uso de una colección y función auxiliar que permita llevar y obtener los valores siguientes de la secuencia incremental. Para esto primero crearemos una colección de contadores de la siguiente manera:

```
> var usuariosAutoincrement = {
  _id:    'autoresid',
  secuencia:  0
}

> db.contadores.insert(usuariosAutoincrement)
```

Al especificar el campo secuencia como 0 indicará que la misma comenzará con este número.

Ahora crearemos una función **Javascript** la cual se encargará de buscar el próximo número en la secuencia, veamos de que se trata:

```
> function proximoEnSecuencia(nombre){
  var resultado = db.contadores.findAndModify({
    query:  { _id:  nombre },
    update: { $inc: { secuencia: 1 } },
    new:  true
  });

  return resultado.secuencia;
}
```

Bien, veamos en detalle que hace nuestra función:

- Sobre la colección contadores hacemos una búsqueda y actualización al mismo tiempo (findAndModify).
- El parámetro query nos especifica qué documento de la colección contadores debemos buscar, es decir, aquel con el _id que se especifique como parámetro de la función.
- El parámetro update indica que luego de haber encontrado el documento en cuestión se debe incrementar ($inc) el campo secuencia en 1.

- El parámetro new le indica al método findAndModify que debe arrojar como resultado el nuevo documento en lugar del original, es decir, aquel que ya ha sido actualizado con el incremento.
- El resultado de este método findAndModify es asignado a una variable y debido a que el resultado es un documento, podemos finalmente retornar el campo secuencia de dicho documento, el cual será el próximo número en la secuencia.

Ahora cuando queramos hacer uso de dicha función para que se encargue de asignar automáticamente el siguiente _id para nuestra colección de autores lo haremos de la siguiente manera:

```
> var oscar = {
  _id:    proximoEnSecuencia('autoresid'),
  nombre:   'Oscar',
  edad:   25
};

> var alberto = {
  _id:    proximoEnSecuencia('autoresid'),
  nombre:   'Alberto',
  edad:   'veintiseis'
};

> var jonathan = {
  _id:    proximoEnSecuencia('autoresid'),
  nombre:   'Jonathan',
  apellido: 'Wiesel'
};

> db.autoresAutoIncrement.insert(oscar);
> db.autoresAutoIncrement.insert(alberto);
> db.autoresAutoIncrement.insert(jonathan);
```

Probemos que en efecto nuestra solución ha hecho su trabajo:

```
> db.autoresAutoIncrement.find()
{ "_id" : 1, "nombre" : "Oscar", "edad" : 25 }
{ "_id" : 2, "nombre" : "Alberto", "edad" : "veintiseis" }
{ "_id" : 3, "nombre" : "Jonathan", "apellido": "Wiesel" }

> db.contadores.find()
{ "_id" : "autoresid", "secuencia" : 3 }
```

Notemos que nuestro autores han tomado su respectivo valor de la secuencia mientras que el documento de la secuencia como tal permanece actualizado con el último valor utilizado.

Selectores de búsqueda

Como mencionamos anteriormente el poder que ofrece una base de datos reside en la capacidad que esta tiene para poder ofrecer los datos que necesitamos en un momento especifico según las necesidades que se nos presenten en dicha situación. Ciertamente vimos como filtrar las búsquedas en nuestra segunda entrada del curso; sin embargo en esta entrada veremos algo un poco más avanzado.

Veamos **algunas** de las diferentes maneras de filtrar nuestras búsquedas haciendo uso de diferentes tipos de operadores o selectores de búsquedas. Adicionalmente notaremos que se enfoca a lo mismo que conocemos en SQL.

Comparativos

- $gt - mayor a X valor.
- $gte - mayor o igual a X valor.
- $lt - menor a X valor.
- $lte - menor o igual a X valor.
- $ne - distinto a X valor.
- $in - entre los siguientes [X, Y, ...]
- $nin no está entre los siguientes [X, Y, ...]

Los primeros 4 evidentemente están enfocados a valores numéricos y pueden ser utilizados de las siguiente manera:

```
> db.autoresAutoIncrement.find({ _id : { $gt : 1 } })
{ "_id" : 2, "nombre" : "Alberto", "edad" : "veintiseis" }
{ "_id" : 3, "nombre" : "Jonathan", "apellido": "Wiesel" }
```

En SQL sería algo como SELECT * FROM autoresAutoIncrement WHERE _id > 1

El operador $ne (distino de...) como podrás adivinar puede utilizarse para campos numéricos y no numéricos. Mientras que los últimos 2 operadores se enfocan en la comparación con arreglos de valores:

```
> db.autoresAutoIncrement.find({ nombre : { $in : ['Alberto', 'Ricardo', 'Oscar'] } })
{ "_id" : 1, "nombre" : "Oscar", "edad" : 25 }
{ "_id" : 2, "nombre" : "Alberto", "edad" : "veintiseis" }
```

En SQL sería algo como SELECT * FROM autoresAutoIncrement WHERE nombre in ('Alberto', 'Ricardo', 'Oscar')

Lógicos

- $or
- $and
- $nor
- $not

Estos operadores lógicos nos permiten juntar múltiples condiciones y dependiendo del cumplimiento de alguna de ellas ($or), todas ellas ($and) o ninguna de ellas ($nor) obtendremos lo que deseamos, inclusive si lo que deseamos es completamente lo opuesto ($not) a lo que especificamos como condición de búsqueda.

```
> db.autoresAutoIncrement.find({ $or : [{_id: 1}, {nombre: 'Jonathan'}] })
{ "_id" : 1, "nombre" : "Oscar", "edad" : 25 }
{ "_id" : 3, "nombre" : "Jonathan", "apellido": "Wiesel" }
```

En SQL sería algo como SELECT * FROM autoresAutoIncrement WHERE _id = 1 OR nombre = 'Jonathan'

En el caso del operador $and, si has prestado atención a lo largo de la serie te darás cuenta que MongoDB maneja implícitamente este tipo de operador, si no lo recuerdas puedes visitar el curso de operaciones básicas para refrescar la memoria.

Para el operador $nor seguiríamos la misma notación que el ejemplo anterior con la diferencia que obtendríamos como resultado aquellos registros que **ni** tengan el _id = 1 **ni** el nombre = Jonathan. Por lo que obtendríamos a Alberto únicamente.

Finalmente el operador $not actúa sobre el operador que le siga y como podrás imaginar, devolverá el resultado contrario.

```
> db.autoresAutoIncrement.find({ _id : { $not: { $gt: 2 }} })
{ "_id" : 1, "nombre" : "Oscar", "edad" : 25 }
{ "_id" : 2, "nombre" : "Alberto", "edad" : "veintiseis" }
```

Al principio pensarás:

¿Por qué usar este operador si pude haber utilizado el $lte?

Una de las ventajas que quizás pasaste por alto es que suponiendo el caso donde dicho filtro se hace sobre otro campo distinto al de _id el cual no es obligatorio, usar el operador $lte obtendrá aquellos documentos con el campo mayor o igual al valor indicado; sin embargo al utilizar el operador $not también **obtendremos aquellos documentos que ni siquiera poseen el campo**.

Elementales

- $exists
- $type

Este tipo de operadores elementales permiten hacer comparaciones referentes a las propiedades del campo como tal.

En el caso de $exist, es un operador booleano que permita filtrar la búsqueda tomando en cuenta la existencia de un campo en particular:

```
> db.autoresAutoIncrement.find({ apellido: { $exists: true }})
```

{ "_id" : 3, "nombre" : "Jonathan", "apellido" : "Wiesel" }

Notaremos que hemos filtrado la búsqueda para que arroje únicamente los documentos que poseen el campo apellido.

Para el caso de $type podemos filtrar por la propiedad de tipo de campo y como valor especificaremos el ordinal correspondiente a su tipo de dato BSON basado en lo siguiente:

- 1 - Double
- 2 - String
- 3 - Objeto
- 4 - Arreglo
- 5 - Data binaria
- 6 - Indefinido (deprecado)
- 7 - Id de objeto
- 8 - Booleano
- 9 - Fecha
- 10 - Nulo
- 11 - Expresión regular
- 13 - Javascript
- 14 - Símbolo
- 15 - Javascript con alcance definido
- 16 - Entero de 32bit
- 17 - Estampilla de tiempo
- 18 - Entero de 64bit
- 127 - Llave máxima
- 255 - Llave mínima

```
> db.autoresAutoIncrement.find({ edad: { $type: 1 }})
{ "_id" : 1, "nombre" : "Oscar", "edad" : 25 }

> db.autoresAutoIncrement.find({ edad: { $type: 2 }})
{ "_id" : 2, "nombre" : "Alberto", "edad" : "veintiseis" }
```

Notemos que para el primer caso indicamos el tipo de campo Double en lugar de uno entero, esto se debe a que el único tipo de dato numérico nativo existente en Javascript es de tipo Double y al ser insertado por la consola de MongoDB se torna en este tipo de dato.