DISEÑO Y ESQUEMA EN MONGODB

Índice

Referencias:

Curso básico https://openwebinars.net/accounts/login/?next=/academia/portada/mongodb/
Curso avanzado https://openwebinars.net/academia/aprende/mongodb-2017/
Tutorial https://www.tutorialspoint.com/mongodb/index.htm
Manual https://docs.mongodb.com/manual/

SCHEMALESS (Sin esquema)

- El esquema en MongoDB es flexible (schemaless) y dinámico (diferentes tipos de datos para un campo en distintos documentos), mientras que en las bases de datos convencionales el esquema es rígido y una modificación de una tabla requiere del bloqueo temporal de la misma.
- Los datos en MongoDB son naturales/autodescriptivos (no están tabulados) y por tanto pueden ser procesados más fácilmente por la aplicación y entendidos por el desarrollador.
- En MongoDB las operaciones sobre un documento son atómicas (incluso si la operación modifica múltiples documentos incrustados en un solo documento).
- Cuando una única operación de escritura (por ejemplo updateMany) modifica varios documentos, la modificación de cada documento es atómica, pero la operación completa no lo es.
- Los datos en MongoDB deberían almacenarse en un formato que favoreciera su tratamiento por la aplicación (patrones de acceso): qué datos van a ser solamente leídos, cuáles serán los escritos, etc.

Ejemplo

Curso C0001

```
{
    "id": 1,
    "referencia": "C0001",
    "nombre": "Física",
    "fInicio": "12/01/2019",
    "esGratuito": true
}
```

Asignaturas que se imparten en el curso C0001

```
{
  "id": 2,
  "nombre": "Física elemental",
  "duracion": 5
},
{
  "id": 3,
  "nombre": "Termodinámica",
  "duracion": 4
}
```

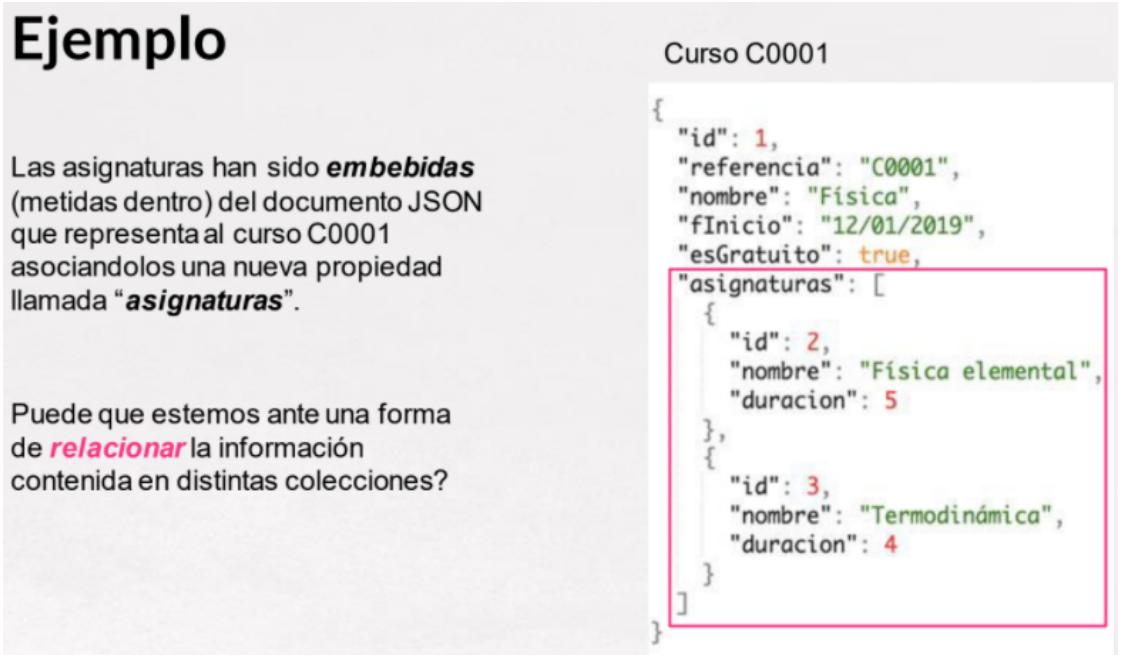
- Por otra parte, la desnormalización es una estrategia utilizada en una base de datos previamente normalizada para aumentar su rendimiento. La idea es agregar datos redundantes donde se estime oportuno para facilitar y mejorar el rendimiento de algunas operaciones de lectura.
- MongoDB proporciona una característica muy potente e inexistente en las bases de datos relacionales que ayuda a la desnormalización: los documentos embebidos o incrustados.
- La lectura y escritura de documentos embebidos es una operación muy rápida y eficiente debido que la información se encuentra almacenada de forma contigua.
- Los documentos incrustados capturan las relaciones entre los datos almacenando datos en una sola estructura de documento. Estos modelos de datos desnormalizados permiten a las aplicaciones recuperar y manipular datos relacionados en una única operación.

- En general, es conveniente utilizar normalización con referencias a otros documentos cuando:
    - La incrustación da lugar a la duplicación de datos, pero no proporciona suficientes ventajas de rendimiento de lectura para compensar las implicaciones de la duplicación.
    - Para representar relaciones complejas de muchos a muchos.
    - Para modelar grandes conjuntos de datos jerárquicos, como los árboles.
- En general, es conveniente utilizar desnormalización con documentos embebidos cuando:
    - Los datos contenidos en el documento embebido son un contenedor adicional de información relacionada con el documento superior.
    - Existen relaciones uno a mucho entre entidades (aunque no es una solución para todos los casos de uso).
- En definitiva, en el diseño del modelo de datos en MongoDB es imprescindible considerar los patrones de acceso a los datos (consultas, actualizaciones y procesado de los datos) de la aplicación.
    - La normalización proporcionará una actualización de datos eficiente. La desnormalización hará que la lectura de datos sea eficiente.
    - Y en general:
        - Embeber:
            - Pequeños subdocumentos.
            - Los datos no cambian regularmente.
            - Se requiere consistencia.
            - Los documentos crecen poco.
            - Se necesitan los datos embebidos para realizar una segunda consulta.
            - Lecturas rápidas
        - Referenciar:
            - Subdocumentos grandes.
            - Los datos cambian frecuentemente.
            - No es necesario mantener la consistencia.
            - Los documentos crecen bastante.
            - Escrituras rápidas.

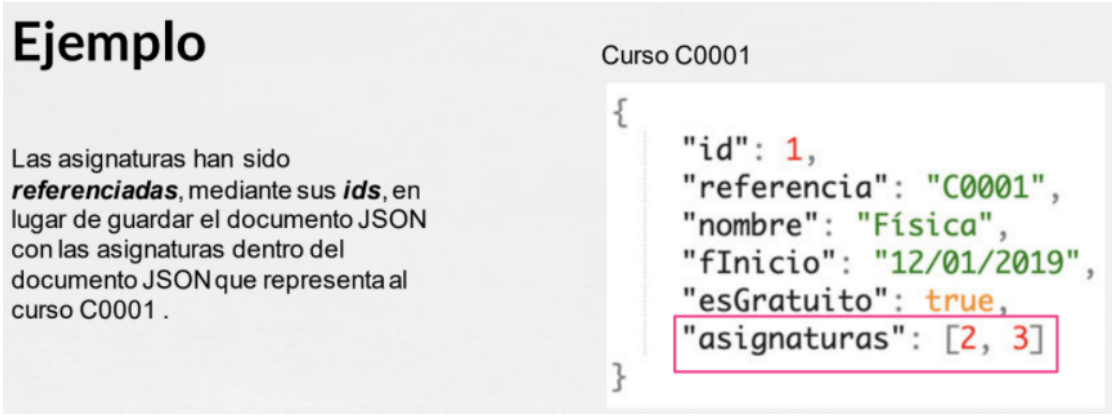
Atomicidad de las operaciones de escritura

- En MongoDB, una operación de escritura es atómica a nivel de un solo documento, incluso si la operación modifica varios documentos incrustados dentro de un solo documento.
- Un modelo de datos desnormalizado con datos incrustados combina todos los datos relacionados en un solo documento en lugar de la normalización en múltiples documentos y colecciones. Este modelo de datos desnormalizado facilita las operaciones atómicas.
- Cuando una sola operación de escritura (por ejemplo, db.collection.updateMany()) modifica varios documentos, la modificación de cada documento es atómica, pero la operación en su conjunto no es atómica.

DOCUMENTOS EMBEBIDOS



DOCUMENTOS REFERENCIADOS



Vemos un ejemplo sencillo realizado embebiendo y referenciando

A continuación tenemos en cuenta la relación entre el usuario y la dirección del usuario siguiente.

Un usuario puede tener múltiples direcciones, es de uno a muchos.

El siguiente es un documento de configuración de usuario sencilla:

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "name": "Tom Hanks",
   "contact": "987654321",
   "dob": "01-01-1991"
}
```

El siguiente es un documento electrónico estructura simple:

```
{
   "_id":ObjectId("52ffc4a5d85242602e000000"),
   "building": "22 A, Indiana Apt",
   "pincode": 123456,
   "city": "Los Angeles",
   "state": "California"
}
```

Con documentos embebidos

Utilizando el método incorporado, que puede incrustar la dirección del usuario para el documento del usuario:

```
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "name": "Tom Benzamin",
   "address": [
      {
         "building": "22 A, Indiana Apt",
         "pincode": 123456,
         "city": "Los Angeles",
         "state": "California"
      },
      {
         "building": "170 A, Acropolis Apt",
         "pincode": 456789,
         "city": "Chicago",
         "state": "Illinois"
      }]
}
```

Los datos anteriores se almacena en un solo documento, puede acceder más fácilmente y mantener datos. Podrías consultar en la dirección del usuario:

```
>db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

Nota: La consulta anterior indica db base de datos y los usuarios y colecciones.

La desventaja de esta estructura de datos es que si los usuarios y abordan la creciente cantidad de datos se hace más grande y más grande, afectará a la lectura y escritura.

Documentos referenciados

Referencia Tipo de relación se utiliza a menudo en el enfoque de diseño de base de datos que los archivos de datos de usuario y archivo de datos de direcciones de usuario por separado para construir relaciones de documento de referencia campoid.

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
}
```

El ejemplo anterior, el documento que contiene el campo address_ids objeto de la identificación del usuario de la matriz de direcciones de usuario (ObjectId).

Podemos leer estos objetos Identificación del domicilio del usuario (ObjectId) para obtener información detallada acerca de la dirección del usuario.

Este método requiere dos consultas, la dirección del primer Identificación del objeto de consulta del usuario (OBJECTID), la segunda dirección para obtener más información sobre la consulta del usuario por ID.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

DECISIÓN EN BASE AL TIPO DE RELACIÓN

Análisis Relaciones 1-1

- Ejemplo de esta relación: paciente e historical clínico.
- Esta relación suele modelarse embebiendo, pero también es posible separar los datos en dos colecciones distintas y unirlas mediante un identificador. Todo depende de cómo se van a acceder a los datos. Concretamente es necesario estudiar:
    - Patrones de acceso a los datos desde la aplicación.
    - Frecuencia de acceso.
    - Tamaño de los datos: teniendo en cuenta la limitación de 16 MB por documento, puede ser conveniente separar los datos si los documentos son grandes.
    - Atomicidad requerida para la actualización de los documentos (embeber es más recomendable en estos casos).
    - Crecimiento de los documentos: si un documento crece con el tiempo, quizás es preferible utilizar dos colecciones con referencias.

Análisis Relaciones 1-N

- Ejemplo de esta relación: persona y direcciones de viviendas.
- Para relaciones 1-N existen diferentes estrategias de modelado:
    - Si la entidad N consta de pocos documentos, entonces se pueden embeber como un array de documentos (o al revés, embebiendo un documento dentro de la entidad N), aunque puede ser mala práctica si se repite la información.
    - Si la entidad N consta de muchos documentos, es conveniente separarlos en dos colecciones.
- Embeber los datos en un array de documentos es preferible cuando es necesario devolver el campo embebido como de tipo array.

```
/* colección people */
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
    {
      street: "123 Fake Street",
      city: "Faketon",
      state: "MA",
      zip: "12345"
    },
    {
      street: "1 Some Other Street",
      city: "Boston",
      state: "MA",
      zip: "12345"
    }
  ]
 }
```

- Embeber un documento dentro de la entidad many puede ser mala práctica porque la información se repite, aunque a veces puede ser deseable.
    - Esta técnica es menos utilizada que la anterior, aunque puede ser útil cuando se realizan muchas consultas sobre el lado many.
    - Un posible ejemplo de duplicación deseable de elementos es el carrito de compra de una tienda online. Agregar los productos a comprar en la lista duplica la información (porque varios clientes pueden comprar los mismos productos), pero puede ser deseable porque se guardaría el estado de los productos en el momento en el que el cliente lo agrega en su carrito de compra.
    - Otro posible ejemplo es con respecto a las distintas direcciones de envío de los productos a los clientes.

```
/* colección streets */
{
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345",
  person: {
    _id: "joe",
    name: "Joe Bookreader"
  }
}

{
  street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345",
  person: {
    _id: "joe",
    name: "Joe Bookreader"
  }
}
```

- Cuando se utilizan dos colecciones, una solución recomendable es apuntar de N a 1 a través de un campo (generalmente un _id de la entidad 1), aunque es necesario tener cuidado por las posibles inconsistencias que pueden producirse.

```
/* utilizando referencias (normalizando de N a 1) */

/* colección people */
{
  _id: "joe",
  name: "Joe Bookreader"
}
```

```
/* colección streets */
{
  patron_id: "joe",
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}

{
  patron_id: "joe",
  street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345"
}
```

- También es posible apuntar de 1 a N utilizando un array de referencias.

```
/* utilizando referencias (normalizando de 1 a N) */

/* colección people */
{
  _id: "joe",
  name: "Joe Bookreader",
  addresses: [ObjectId("5e56f329a51e010eacb4b417"),
ObjectId("5e56f329a51e010eacb4b418")]
}

/* colección streets */
{
  _id: ObjectId("5e56f329a51e010eacb4b417"),
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}

{
  _id: ObjectId("5e56f329a51e010eacb4b418"),
  street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345"
}
```

Análisis Relaciones N - N

- Ejemplo de esta relación: profesor y estudiante, o libro y autor.

- La estrategia a seguir es distinta en función de si la relación es pocos a pocos o muchos a muchos. Puede utilizarse un array de identificadores que apunten a documentos de la otra colección.
- También es posible definir un array de referencias en los documentos de ambas colecciones, aunque no siempre es una buena idea.
- Embeber en este caso no es una buena decisión porque se estaría duplicando la información.

```
/* normalizando N-M con doble referencia */

/* colección students */
{
  _id: 1,
  name: "Joe",
  teachers: [1, 2]
}

/* colección teachers */
{
  _id: 1,
  name: "John",
  students: [1]
}

{
  _id: 2,
  name: "Carl",
  students: [1]
}
```

Relación "uno a pocos"

- En este caso es preferible embeber el documento.

```
db.users.save({
  name: 'Kate Monster',
  ssn: '123-456-7890',
  addresses : [
     { street: '123 Sesame St', city: 'Anytown', cc: 'USA' },
     { street: '123 Avenue Q', city: 'New York', cc: 'USA' }
  ]
})

db.users.find({"addresses.city": "Anytown"})
```

- La principal ventaja de embeber es evitar realizar una consulta independiente a otro colección.
- La principal desventaja es que no existe manera de acceder a los detalles incrustados como entidades independientes.

Relación "uno a muchos"

- La cantidad "muchos" puede interpretarse como cientos de documentos, pero no más de mil.
- Para este caso podría emplearse mejor una segunda colección. En la colección principal se encontraría un array de referencias con ObjectIDs vinculados a documentos de la segunda colección.

```
id = ObjectId();

// products
db.products.save({
  name : 'left-handed smoke shifter',
  manufacturer : 'Acme Corp',
  catalog_number: 1234,

  // para simplificar se utilizan ObjectID de 2 bytes, pero en la práctica sería
conveniente utilizar los _id de mongo
  parts : [
    id
  ]
});

// parts
db.parts.save({
  _id : id,
  partno : '123-aff-456',
  name : '#4 grommet',
  qty: 94,
  cost: 0.94,
  price: 3.99
});
```

- Para obtener los documentos de la segunda colección es necesario efectuar dos operaciones de consulta.

```
product = db.products.findOne({catalog_number: 1234});
product_parts = db.parts.find({_id: { $in : product.parts } } );
```

- Para optimizar la consulta es conveniente establecer un índice en el campo products.catalog_number.
- La ventaja de este diseño es principalmente la facilidad de búsqueda y actualización de documentos de la segunda colección.
- La desventaja es la necesidad de ejecutar una segunda consulta para obtener los detalles de la segunda colección.
- Este tipo de diseño no solo permitiría relaciones 1-N, sino también N-M. En el ejemplo anterior, un producto puede tener asociado varias piezas, pero también una pieza puede tener asociado varios productos.

Relación "uno a muchísimos"

- Debido a que el tamaño máximo de un documento en Mongo son 16 MB, las dos estrategias presentadas en secciones anteriores podrían presentar problemas en colecciones muy grandes con arrays de documentos embebidos ("uno a pocos") o aunque simplemente se almacenen como un array de ObjectId ("uno a muchos").
- Para esta relación de "uno a muchísimos", la segunda colección debería tener un campo de tipo ObjectId que apunte a un único documento de la primera colección. De esta forma, ambas colecciones podrían crecer sin problemas y sería más difícil alcanzar la limitación de las 16 MB.

Desnormalización en detalle

- Con la desnormalización se evita realizar una consulta adicional para obtener los valores de los campos de otras documentos en colecciones que están vinculadas.
- La desnormalización solo tiene sentido cuando hay una alta proporción de lecturas con respecto a las actualizaciones.
    - Si se van a leer los datos desnormalizados con frecuencia, pero solo se actualizan con poca frecuencia, a menudo tiene sentido pagar el precio de actualizaciones más lentas y más complejas para obtener consultas más eficientes.
    - A medida que las actualizaciones se vuelven más frecuentes en relación con las consultas, las ventajas de la desnormalización disminuyen.
- También es conveniente recordar que con la desnormalización se puede perder consistencia ya que una actualización de datos posiblemente requiera de la modificación en documentos de dos colecciones. No es posible atomicidad en estos casos porque la atomicidad en MongoDB se encuentra a nivel de documento.
- La desnormalización también es útil cuando se pretende evitar tener que unir los datos a nivel de aplicación.

Resumen de recomendaciones

- Favorecer la incrustación o embebido de documentos a menos que haya una razón de peso para no hacerlo.
- La incrustación es muy útil cuando se pretende realizar consultas u operaciones atómicas sobre un único documento.
- La principal desventaja de la incrustación es que no hay forma de acceder a los detalles integrados como entidades independientes.
- Si se pretende acceder a los datos embebidos como entidades independientes es preferible utilizar una segunda colección y unir ambas mediante referencias.
- Es importante considerar si la relación es "uno a pocos", "uno a muchos" o "uno a muchísimos".
- Cuando hay muchísimos elementos a referenciar en una relación 1-N, es preferible usar una referencia hacia el lado único en los objetos del lado N.
- Los arrays no deberían crecer sin límite. Si hay más de un par de cientos de documentos, no deben incrustarse, sino que debería emplearse un array de referencias con ObjectID. Con miles de documentos ya no es conveniente tampoco utilizar un array de referencias.
- Para la desnormalización de campos se debe evaluar la relación lectura/escritura. Si un campo va a ser leído principalmente y se actualiza poco, éste puede ser un buen candidato para la desnormalización. En cambio, si el campo se actualiza con frecuencia, quizás ya no sea conveniente desnormalizarlo.

Ejemplo Relaciones Entre Documentos

La referencia Usuario con Comentario se ve claramente que debe ser una referencia entre documentos, ya que el Usuario puede cambiar sus datos en cualquier momento sin que el Comentario se de cuenta, por lo que es importante enlazarlos con una relación aunque perdamos un nivel de indirección a la hora de consultar.

En la colección Usuario sólo se insertarán los datos de los usuarios. Ejemplo:

```
> db.usuario.insertMany([{nombre: "Pepe", edad:34},{nombre: "Juan", edad:45}])
> db.usuario.find().pretty()
[
  {
    _id: ObjectId("63678123b3fe3f6bae82c77c"),
    nombre: 'Pepe',
    edad: 34
  },
  {
    _id: ObjectId("63678123b3fe3f6bae82c77d"),
    nombre: 'Juan',
    edad: 45
  }
]
```

Según el uso y modelado que hemos definido ahora los usuarios crearán sus comentarios.

En la **colección Comentario**, tendríamos los datos de los comentarios que serán sobre un producto referenciado y realizados por un usuario:

```
> db.comentario.insertOne(
{"usuario" : ObjectId("63678123b3fe3f6bae82c77c"), fecha: ISODate("2022-10-12"),
texto:"Me gusta"})

> db.comentario.insertOne(
{"usuario" : ObjectId("63678123b3fe3f6bae82c77d"), fecha: ISODate("2022-10-22"),
texto:"Me gusta regular"})

> db.comentario.find().pretty()
[
  {
    _id: ObjectId("636801d6b3fe3f6bae82c787"),
    usuario: ObjectId("63678123b3fe3f6bae82c77c"),
    fecha: ISODate("2022-10-12T00:00:00.000Z"),
    texto: 'Me gusta'
  },
  {
    _id: ObjectId("636801fcb3fe3f6bae82c788"),
    usuario: ObjectId("63678123b3fe3f6bae82c77d"),
    fecha: ISODate("2022-10-22T00:00:00.000Z"),
    texto: 'Me gusta regular'
  }
]
```

Podemos hacer una búsqueda sobre la segunda colección:

```
> var comentario = db.comentario.findOne({"texto":/regular/});
> db.usuario.find({"_id":comentario.usuario})
[
  {
    _id: ObjectId("63678123b3fe3f6bae82c77d"),
    nombre: 'Juan',
    edad: 45
  }
]
```

Sin embargo la relación entre producto y sus comentarios debemos analizarla: Este tipo de decisiones se deben de tomar cuando se tienen **relaciones Uno a Muchos** donde la parte de muchos **siempre aparecen o son visionados desde el contexto de sus documentos padres**. En este caso merece la pena llevar a los comentarios en una nueva colección en vez de embeberlos, pero si siempre se van a ver los comentarios desde producto, habría que incluir las referencias de los comentarios en los productos, si no fuera así en el comentario pondríamos la referencia del producto.

```
db.producto.insertOne(
{
"nombre":"Capucchino",
"precio":20,
"marca":"Nescafé",
"comentarios": [
ObjectId("63678123b3fe3f6bae82c77c"), ObjectId("63678123b3fe3f6bae82c77d") ]
})

> db.producto.find().pretty()
[
  {
    _id: ObjectId("6368064db3fe3f6bae82c789"),
    nombre: 'Capucchino',
    precio: 20,
    marca: 'Nescafé',
    comentarios: [
          ObjectId("6367d75ab3fe3f6bae82c782"),
          ObjectId("6367d788b3fe3f6bae82c783")
    ]
  }
]
```

Más adelante también podremos consultar los datos del producto con la información de todos los comentarios (texto del comentario, nombre del usuario y edad del usuario que ha realizado el comentario)