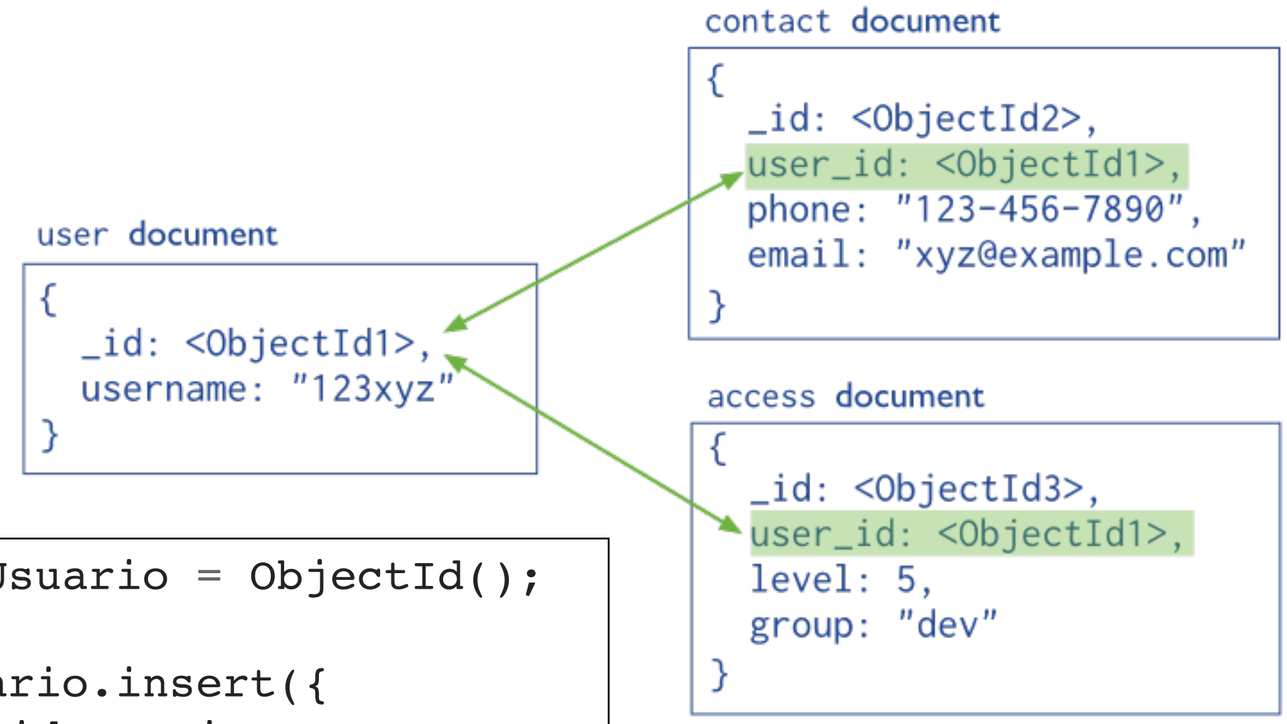
Diseño del esquema

- Enfoque diferente al relacional
- No 3FN → tendencia a denormalizar
- ~~*MongoDB* no soporta transacciones~~
  - Asegura que las operaciones son atómicas
  - Solución:
    1. Restructurar el código para que toda la información esté contenida en un único documento.
    2. Implementar un sistema de bloqueo por software (semáforo, etc…).
    3. Tolerar un grado de inconsistencia en el sistema.

- **Denormalizar** los datos para minimizar la redundancia pero facilitando que mediante operaciones atómicas se mantenga la integridad de los datos

Referencias manuales

- Almacenar el campo `_id` como clave ajena
- La aplicación realiza una 2ª consulta para obtener los datos relacionados.
- Son sencillas y suficientes para la mayoría de casos de uso




```
var idUsuario = ObjectId();

db.usuario.insert({
  _id: idUsuario,
  nombre: "123xyz"
});

db.contacto.insert({
  usuario_id: idUsuario,
  telefono:  "123-456-7890",
  email: "xyz@ejemplo.com"
});
```

DBRef

- Objeto que representa una referencia de un documento a otro mediante el valor del campo `_id`, el nombre de la colección y, opcionalmente, el nombre de la base de datos
- `{ "$ref" : <nombreColeccion>, "$id" : <valorCampo_id>, "$db" : <nombreBaseDatos> }`
- Permite referenciar documentos localizados en diferentes colecciones.
- En *Java*, mediante la clase `DBRef`

```
db.contacto.insert({
  usuario_id: new DBRef("usuario", idUsuario),
  telefono:  “123 456 7890",
  email: "xyz@ejemplo.com"
});
```

Datos embebidos

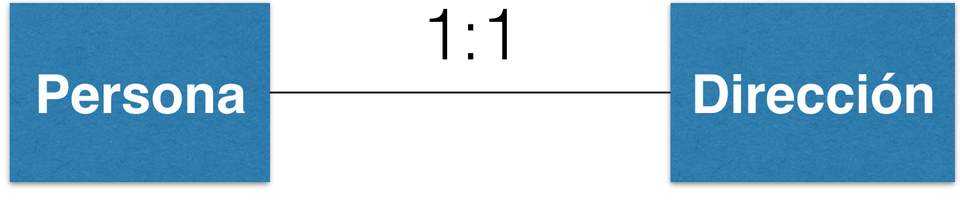
- Mediante sub-documentos
- Dentro de un atributo o un array
- Permite obtener todos los datos con un acceso
- Se recomienda su uso en relaciones:
  - contiene
  - uno a uno
  - uno a pocos

⚠️ Un documento BSON puede contener un máximo de 16MB

- Si un documento crece mucho → usar referencias o *GridFS*

Relaciones - 1:1

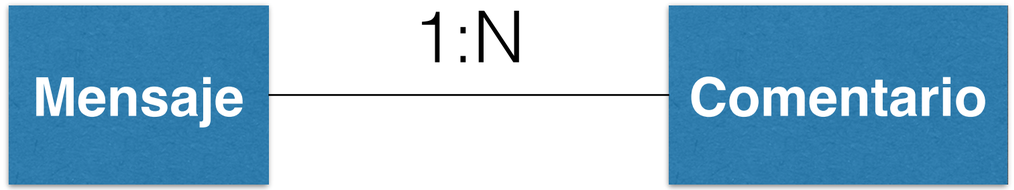
- Embeber un documento dentro de otro
- Motivos para no embeber:
  - Frecuencia de acceso.
    - Si a uno de ellos se accede muy poco
    - Al separarlos → se libera memoria
  - Tamaño de los elementos.
    - Si hay uno que es mucho más grande que el otro
    - O uno lo modificamos muchas más veces que el otro




```
{
  nombre: "Aitor",
  edad: 38,
  direccion: {
    calle: "Mayor",
    ciudad: "Elx"
  }
}
```

Relaciones 1:N - *1 a pocos*

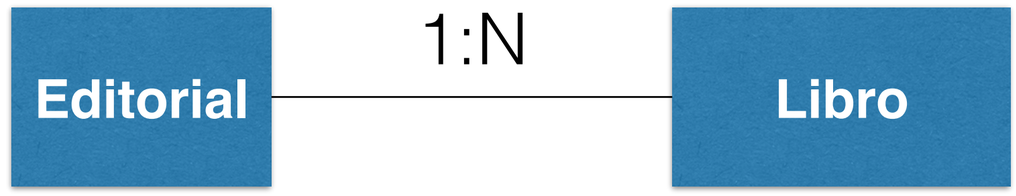
- Embeber los datos
- Crear un array dentro de la entidad 1
  - El `Mensaje` contiene un array de `Comentario`




```
{
  titulo: "La broma asesina",
  url: "http://es.wikipedia.org/wiki/Batman:_The_Killing_Joke",
  text: "La dualidad de Batman y Joker",
  comentarios: [
    {
      autor: "Bruce Wayne",
      fecha: ISODate("2015-04-01T09:31:32Z"),
      comentario: "A mi me encantó"
    }, {
      autor: "Bruno Díaz",
      fecha: ISODate("2015-04-03T10:07:28Z"),
      comentario: "El mejor"
    }
  ]
}
```

Relaciones 1:N - *1 a muchos*

- Referencia de N a 1
  - Igual que clave ajena
- Restricción 16MB BSON
- Se pueden emplear documentos embebidos con redundancia de datos cuando la información que interesa es la que contiene en un momento determinado
  - Productos (pvp) de un Pedido
  - Dirección (de envío) de un Cliente




```
{
  _id: 1,
  nombre: "O'Reilly",
  pais: "EE.UU."
}
```

Editorial

```
{
  _id: 1234,
  titulo: "MongoDB: The Definitive Guide",
  autor: [ "Kristina Chodorow", "Mike Dirolf" ],
  numPaginas: 216,
  editorial_id: 1,
}
{
  _id: 1235,
  titulo: "50 Tips and Tricks for MongoDB Developer",
  autor: "Kristina Chodorow",
  numPaginas: 68,
  editorial_id: 1,
}
```

Libro

N:M

- Suelen ser relaciones pocos a pocos
- 3 posibilidades
  1. Enfoque relacional con colección intermedia
     - Desaconsejado → 3 consultas
  2. Dos documentos, cada uno con un array que contenga los ids del otro documento (*2 Way Embedding*).
     - Vigilar la inconsistencia de datos
  3. Embeber un documento dentro de otro (*One Way Embedding*)
     - No se recomienda si alguno de los documentos puede crecer mucho
     - Revisar si un documento depende de otro para su creación


```
{
  _id: 1,
  titulo: "La historia interminable",
  anyo: 1979,
  autores: [1]
},{
  _id: 2,
  titulo: "Momo",
  anyo: 1973,
  autores: [1]
}
```

```
{
  _id: 1,
  nombre: "Michael Ende",
  pais: "Alemania",
  libros: [1,2]
}
```

2 way embedding

Consejos de Rendimiento I

- Si se realizan más lecturas que escrituras → denormalizar los datos para usar **datos embebidos** y así con sólo una lectura obtener más información.
- Si se realizan muchas inserciones y sobretodo actualizaciones, será conveniente usar referencias con dos documentos.
- El mayor beneficio de embeber documentos es el rendimiento, sobretodo el de lectura.
- El acceso a disco es la parte más lenta, pero una vez la aguja se ha colocado en el sector adecuado, la información se obtiene muy rápidamente (alto ancho de banda).
- Si toda la información a recuperar esta almacenada de manera secuencial favorece que el rendimiento de lectura sea muy alto

  Sólo se hace un acceso a la BBDD.

Consejos de Rendimiento II

- Planteamiento inicial para modelar los datos es basarse en **unidades de aplicación**
  - Petición al *backend* → click de un botón, carga de los datos para un gráfico.
  - Cada unidad de aplicación se debe poder conseguir con una única consulta → datos embebidos.
- Si necesitamos consistencia de datos → normalizar y usar referencias.
  - Puede que un sistema NoSQL no sea la elección correcta
  - Necesitemos dos o más lecturas para obtener la información deseada.
- Si la consistencia es secundaria, duplicar los datos (pero de manera limitada)
  - El espacio en disco es más barato que el tiempo de computación

Concurrencia

- *MongoDB* emplea bloqueos de lectura/escritura que permiten acceso concurrente para las lecturas de un recurso (ya sea una base de datos o una colección),
- Sólo da acceso exclusivo para cada operación de escritura.
- Gestión de bloqueos:
  - Puede haber un número ilimitado de lecturas simultáneas a un base de datos.
  - En un momento dado, sólo puede haber un escritor en cualquier colección en cualquier base de datos.
  - Una vez recibida una petición de escritura, el escritor bloquea a todos los lectores.

- > v2.2 → bloqueo a nivel de base de datos
- > v3 → sólo se bloquean los documentos implicados en la operación de escritura.

Motor de almacenamiento

- Define cómo se almacenan los datos
- Restringe el nivel de bloqueo de la concurrencia
- 2 posibilidades:
  - **MMAPv1**: Almacenamiento por defecto en v2. Emplea bloqueos a nivel de colección.
  - **WiredTiger**: Nuevo motor de almacenamiento desde v3. Ofrece bloqueo a nivel de documento y permite compresión de los datos → múltiples clientes pueden modificar más de un documento de una misma colección al mismo tiempo.
- Se indica al arrancar el demonio

```
mongod --storageEngine wiredTiger
```

Índices

- Estructura de datos que almacena información sobre los valores de determinados campos de los documentos de una colección.
- Permite recorrer los datos y ordenarlos de manera muy rápida
- Se utilizan en las consultas:
  - buscar
  - ordenar
- Todas las colecciones contienen un índice sobre el campo `_id`

```
> db.students.findOne()
{
  "_id" : 0,
  "name" : "aimee Zank",
  "scores" : [
    {
      "type" : "exam",
      "score" : 1.463179736705023
    }, {
      "type" : "quiz",
      "score" : 11.78273309957772
    }, {
      "type" : "homework",
      "score" : 6.676176060654615
    }, {
      "type" : "homework",
      "score" : 35.8740349954354
    }
  ]
}
```

Plan de Ejecución I

- Consulta por un campo sin indexar

```
> db.students.find({"name":"Kaila Deibler”}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "expertojava.students",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "name" : { "$eq" : "Kaila Deibler" }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "name" : { "$eq" : "Kaila Deibler" }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
```

```
"executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 2,
    "executionTimeMillis" : 1,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 200,
    "executionStages" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "name" : { "$eq" : "Kaila Deibler" }
      },
      "nReturned" : 2,
      "executionTimeMillisEstimate" : 0,
      "works" : 204,
      "advanced" : 2,
      "needTime" : 199,
      "needYield" : 2,
      "saveState" : 2,
      "restoreState" : 2,
      "isEOF" : 1,
      "invalidates" : 0,
      "direction" : "forward",
      "docsExamined" : 200
    }
  },
  "serverInfo" : {
    "host" : "MacBook-Air-de-Aitor.local",
    "port" : 27017,
    "version" : "3.2.1",
    "gitVersion" : “a14d559…”
  },
  "ok" : 1
}
```

Plan de Ejecución II

- Consulta por `_id` (siempre está indexado)

```
> db.students.find({_id:30}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    ...
    "winningPlan" : {
      "stage" : "IDHACK"
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 1,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 1,
    "totalDocsExamined" : 1,
    "executionStages" : {
      ...
    }
  },
  "serverInfo" : {
    ...
  },
  "ok" : 1
}
```

Índices Simples

- **`createIndex`**`({atributo:orden})`

```
db.students.createIndex( {name:1} )
```

- Orden de los índices (`1` para ascendente, `-1` para descendente)
  - No importa para un índice sencillo
  - Si que tendrá un impacto en los índices compuestos cuando se utilizan para ordenar o con una condición de rango.

```
> db.students.find({"name" : "Kaila Deibler"}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    ...
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : { "name" : 1 },
        "indexName" : "name_1",
        "isMultiKey" : false,
        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 1,
        "direction" : "forward",
        "indexBounds" : {
          "name" : ["[\"Kaila Deibler\", \"Kaila Deibler\"]"]
      }
    }
   },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 2,
    "executionTimeMillis" : 1,
    "totalKeysExamined" : 2,
    "totalDocsExamined" : 2,
    "executionStages" : {
    ...
   }
  },
  …
}
```

Información de los índices

- Toda la información relativa a los índices creados se almacenan en la colección `system.indexes`
- Podemos obtener los índices de una determinada colección mediante el método `getIndexes()`.
- Para borrar un índice emplearemos el método `dropIndex(campo)`.

```
db.system.indexes.find() // muestra los índices existentes
db.students.getIndexes() // muestra los índices de la colección students
db.students.dropIndex( {"name":1} ) // borra el índice que existe sobre la propiedad name
```

Propiedades de los índices

- Se pasan como segundo parámetro
- `unique` → Sólo permiten valores únicos en una propiedad. No puede haber valores repetidos y una vez creado no permitirá insertar valores duplicados.

```
db.students.createIndex( {students_id:1}, {unique:1} )
```

- `sparse` → Si queremos añadir un índice sobre una propiedad que no aparece en todos los documentos, necesitamos crear un índice *sparse*. Se crea para el conjunto de claves que tienen valor.

```
db.students.createIndex( {size:1}, {sparse:1} )
```

- `partialFilterExpression` → Permite indexar sólo los documentos que cumplen un criterio. La consulta debe realizarse por el campo del índice, y cumplir con un subconjunto de la expresión de filtrado

```
db.students.createIndex({“scores.type":1},
 {partialFilterExpression:{"scores.score":{$gt:95}}} )
```

Índices Compuestos

- Se aplican sobre más de una propiedad de manera simultánea

```
db.students.createIndex({name:1,scores.type:1})
```

- El **orden** de los índices **importa**
- Los índices se usan con los subconjuntos por la izquierda (prefijos) de los índices compuestos.
  - Si creamos un índice sobre los campos (A,B,C), el índice se va a utilizar para las búsquedas sobre A, sobre la dupla (A,B) y sobre el trio (A,B,C).
- Si tenemos varios índices candidatos a la hora de ejecutar, el optimizador de consultas los usará en paralelo y se quedará con el resultado del primero que finalice

Índices Multiclave

- Al indexar una propiedad que es un array se crea un índice multiclave para todos los valores del array de todos los documentos.
  - Aceleran las consultas sobre documentos embebidos

```
db.students.createIndex({"teachers":1})
db.students.find({"teachers":{"$all":[1,3]}})
```

- Se pueden crear índices tanto en propiedades básicas, como en propiedades internas de un array, mediante la notación de `.`:

```
db.students.createIndex({“adresses.phones":1})
```

- Sólo se pueden crear índices compuestos multiclave cuando únicamente una de las propiedades del índice compuesto es un array → no puede haber dos propiedades array en un índice compuesto.

Rendimiento

- Por defecto, los índices se crean en *foreground* → al crear un índice se van a bloquear a todos los *writers*.
- Para crearlos en *background* (de 2 a 5 veces más lento), segundo parámetro `background`:

```
db.students.createIndex( { twitter: 1}, {background: true} )
```

- Operadores que no utilizan los índices eficientemente son: `$where`, `$nin` y `$exists`.
- Al emplearlos en una consulta hay que tener en mente un posible cuelo de botella cuando el tamaño de los datos incremente.

Gestión de la memoria

- **Los índices tienen que caber en memoria.**
- Si están en disco, pese a ser algorítmicamente mejores que no tener, al ser más grandes que la RAM disponible, no se obtienen beneficios por la penalización de la paginación.

```
db.students.stats()  // obtiene estadísticas de la colección
db.students.totalIndexSize()  // obtiene el tamaño del índice (bytes)
```

- Mucho cuidado con los índices *multikeys* porque crecen mucho y si el documento tiene que moverse en disco, el cambio supone tener que cambiar todos los puntos de índice del array.
- Aunque sea más responsabilidad de un DBA, los desarrolladores debemos saber si el índice va a caber en memoria.

- Si no usamos un índice o al usarlo su rendimiento es peor, es mejor borrarlo → `dropIndex`

```
db.students.dropIndex('nombreDeIndice')
```

Hints

- Fuerzan el uso de un índice
- Se ejecutan sobre un cursor, pasándole un parámetro con el campo → `.hint({campo:1})`

```
db.people.find({nombre:"Aitor Medrano",twitter:"aitormedrano"}).hint({twitter:1}})
```

- Si queremos no usar índices, le pasaremos el operador `$natural`:

```
db.people.find({nombre:"Aitor Medrano",twitter:"aitormedrano"}).hint({$natural:1}})
```

- Los operadores `$gt`, `$lt`, `$ne`, … provocan un uso ineficiente de los índices, ya que la consulta tiene que recorrer toda la colección de índices.
- Si hacemos una consulta sobre varios atributos y en uno de ellos usamos `$gt`, `$lt` o similar, es mejor hacer un *hint* sobre el resto de atributos que tienen una selección directa.

```
db.grades.find({ score:{$gt:95, $lte:98}, type:"exam" }).hint('type')
```