MongoDB

Miguel Rodríguez Penabad

Laboratorio de Bases de Datos

Universidade da Coruña

28 de marzo do 2025

Índice

Características principais

"MongoDB is a powerful, flexible, and scalable general-purpose database."
(Chodorow [1], cap. 1)

- Non é un sistema relacional. En concreto
  - Non ten soporte completo de joins. Hai soporte parcial (v4.0+)
  - Hai limitacións no soporte de transaccións. Hai atomicidade a nivel de documento; transaccións multidocumento (v4.0+) e multi-shard (v4.2+)
  - Non soporta restricións (en particular, claves foráneas ou check)
    (É posible a validación –pero pode saltarse– de documentos para checks)
- É un almacén de datos baseado en Documentos (JSON/BSON)
  - Representación de obxectos complexos desde o punto de vista do desenvolvedor
  - Esquemas flexibles: distintos tipos de documentos na mesma colección
- Ofrece alta dispoñibilidade e tolerancia a fallos (replica sets)
- Permite facilmente escalar horizontalmente o sistema (sharding)
- Un dos seus obxectivos é a alta eficiencia
- Permite indexar as coleccións
- Permite consultas complexas, incluíndo un *framework* para a agregación de datos e un optimizador de consultas

Instalación e execución

Descarga de ficheiros

NOTA: A empresa ten tendencia a cambiar o empaquetado e modificar as URLs de descarga, polo que poida que estas instrucións non sexan exactas en algún momento.

- Pode descargarse de `https://www.mongodb.com/try`.
- Hai binarios para Windows, GNU/Linux e MacOS.
  Debe seleccionarse a opción *On premises* e descargar a versión *MongoDB Community Edition*.
  No momento de escribir este documento era a versión 8.0.3.

  Adicionalmente, en *Tools* debe descargarse *MongoDB Shell* e *MongoDB Database Tools*.
  Escolleremos a versión do noso SO, e o empaquetado `tgz` ou `zip` (interfire menos cos demáis servizos do noso equipo).
- Non require "instalación", son un conxunto de binarios que se poden executar directamente.
  - `mongod`: O servidor.
  - `mongos`: Mongo Shard, é o servizo de enrutamento de consultas (query router) para *sharded clusters*.
  - `mongosh`: A *shell*. É o cliente, que ademais é un intérprete de JavaScript completo.
  - Outras utilidades de BD, entre eles os usados para exportar/importar (`mongoimport`, `mongoexport`, `mongodump`, `mongorestore`)

Instalación e execución

Arranque do servizo `mongod`

Inicialmente, para arrancar o servidor de MongoDB só hai que arrancar un proceso de `mongod`. Veremos máis adiante configuracións en cluster para *Replica Sets* e *Sharded Clusters*.

- `mongod` pode lanzarse directamente indicando unha serie de parámetros, ou usar un ficheiro de configuración.

  Parámetros importantes: porto (27017), directorio de datos (`/data/db` ou `C:\data\db`), path do ficheiro de log.

- Arranque en GNU/Linux ou MacOS (con paso de parámetros por liña de comando):

```
mongod --port 27017 --dbpath /home/user/data --logpath /home/user/logs/mongodb.log --fork
```

- Arranque en Windows (con paso de parámetros por liña de comando):

  Usando o *Simbolo do sistema* (`cmd.exe`):

```
start /b mongod.exe --port 27017 --dbpath c:\mongo\datos --logpath c:\mongo\mongod.log
```

  Usando *PowerShell*:

```
cmd /c "start /b mongod.exe --port 27017 --dbpath c:\mongo\datos --logpath c:\mongo\mongod.log"
```

  ou

```
Start-Process mongod.exe
   -ArgumentList "--dbpath c:\mongo\datos --logpath c:\mongo\mongod.log" -NoNewWindow
```

Instalación e execución

Arranque do servizo `mongod`

É posible utilizar un ficheiro de configuración para indicar os parámetros (en formato YAML).

Exemplo de ficheiros `mongodb.conf`, para ser invocados con `mongod -f mongodb.conf` (os directorios referenciados deben existir previamente):

Para GNU/Linux ou MacOS

```
storage:
  dbPath: /home/user/data
net:
  port: 27017
systemLog:
  destination: file
  path: /home/user/logs/mongodb.log
  logAppend: true
processManagement:
  fork: true
```

Para MS Windows

```
storage:
  dbPath: C:/mongo/datos
net:
  port: 27017
systemLog:
  destination: file
  path: c:/mongo/mongod.log
  logAppend: true
```

Instalación e execución

Alternativa 1: Usar os instaladores oficiais

Existe a posibilidade de instalar MongoDB usando paquetes oficiais para diversas plataformas.

Neste caso, a instalación pode crear servizos que se arrancan automaticamente, e veñen cunha configuración predeterminada (portos, paths, ...).

Enlace xeral: `https://www.mongodb.com/docs/manual/installation/`

- GNU/Linux: distribucións soportadas: Debian, Ubuntu, Suse, RedHat/CentOS, Amazon Linux. Hai información específica para cada distribución (a veces para versións específicas da distribución).
- Microsoft Windows.
- MacOS.
- Instalación con Docker.

Instalación e execución

Alternativa 2: Instalación con Docker

Nota: estas instrucións son algo diferentes das que ofrece o propio MongoDB.

1. Descarga da imaxe e creación do contedor. Só hai que executalo unha vez.

   ```
   penabad@sil:~$ docker run --name mongo8 -d mongo:8.0.0
   ```

2. Arranque e parada do contedor: Se reiniciamos o equipo (ou paramos o contedor) teremos que arrancalo antes de conectarnos a el.

   Ó arrancar o contedor, o servizo de MongoDB arranca automaticamente.

   ```
   penabad@sil:~$ docker start mongo8
   penabad@sil:~$ docker stop mongo8
   ```

3. Conectarnos ó contedor: Isto permite a execución da shell `mongosh` e outras utilidades como `mongoimport`.

   ```
   penabad@sil:~$ docker exec -it mongo8 bash
   root@51d5ba08ec5e:/# mongosh
   Current Mongosh Log ID: 66fd50c31c9e117c78964032
   Connecting to: mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.1
   Using MongoDB: 8.0.0
   Using Mongosh: 2.3.1
   ...
   test>
   ```

   Podemos copiar ficheiros (por exemplo, para logo importar datos) desde o host local ó contedor:

   ```
   penabad@sil:~$ docker cp contactos_gl.json mongo8:/tmp
   Successfully copied 3.52MB to mongo8:/tmp
   ```

Instalación e execución

O cliente `mongosh`

A aplicación cliente ou shell interactiva é o programa `mongosh` (en versións anteriores á 5.0 había outro: `mongo`).

- Axuda: `mongosh --help`.
- Conexión (exemplos simples)

  Documentación máis completa do string de conexión a mongodb:
  `https://www.mongodb.com/docs/manual/reference/connection-string/`:

```
mongosh                        -- Conecta a localhost, porto 27017, BDd test
mongosh bd1                    -- Conecta a localhost, porto 27017, BDd bd1
mongosh 127.0.1.14:27018/bd2 -- Conecta a 127.0.1.14 no porto 27018, BD bd2
```

- En modo interactivo:

```
> show databases           -- Mostra as bases de datos dispoñibles no servidor
                           -- ó que estamos conectados
> db                       -- Contén o nome da BD actual
> use bd2                  -- Cambia á base de datos bd2
> show collections         -- Mostra as coleccións da bd actual
> show tables              -- Alias para show collections
```

Instalación e execución

Importación e exportación de datos

- `mongoimport` permite importar datos de texto nunha colección de MongoDB.
- `mongoexport` permite exportar unha colección de MongoDB a un ficheiro de texto.
- Admite formatos JSON, CSV (*Comma Separated Values*) e TSV (*Tab Separated Values*)
- Exemplos:

```
mongoimport --help
   Axuda de mongoimport

mongoimport -d mai -c contactos contactos_gl.json
  Importa na BD mai, Colección: contactos
  Ficheiro de entrada: contactos_gl.json.
  Formato (predeterminado): JSON.

mongoimport -d test -c col1 --type=CSV --headerline --drop datos.csv
  Importa na BD test, Colección: col1
  Ficheiro de entrada: datos.csv. Formato: CSV. A primeira liña contén os nomes dos campos
  Se a colección existe, elimínase e volve a crearse antes de importar os datos.

mongoimport produtos.json
  Importa na BD (predeterminada) test, Colección (baseada no nome do ficheiro): produtos
  Ficheiro de entrada: produtos.json. Formato (predeterminado): JSON.

mongoexport -d bd3 -c col5 -o col5.json
  -- Exporta a colección col5 da base de datos bd3 no ficheiro col5.json
  -- Usa o formato preterminado JSON
```

- `mongodump` e `mongorestore` tamén se poden utilizar para exportar/importar datos ou facer backups (en formato binario: BSON).

Índice

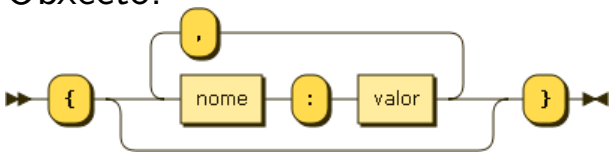
JSON

- JSON (Java Script Object Notation) é un formato de intercambio de datos.
- Descríbese no estándar ECMA-404.
- Deriva orixinalmente do ECMAScript (JavaScript), pero é independente da linguaxe a usar.
- O elemento principal é un *obxecto* (conxunto desordenado de elementos *nome*:*valor*).
  O nome é un string.
- Os valores poden ser de varios tipos, incluídos obxectos ou arrays que fan que se poidan representar obxectos arbitrariamente complexos:
  - Un string (vai entre comiñas).
  - Un número (non pode ir entre comiñas). Non hai subtipos numéricos (real, int, etc.).
    Nota: Os valores `42` e `"42"` serían distintos!
  - Un *array*: colección ordenada de valores.
  - Un obxecto.
  - As constantes `true`, `false` e `null` son tamén valores válidos.
- Notación con punto (dot notation) que usa MongoDB:
  - Acceso a campos en subdocumentos: `"doc.campo"`. Ex: `"cliente.nome"`
  - Acceso a elementos dun array: `"array.index"`. Ex: `"liñas.0"`
    Caso especial: `"array.$"` usado en operacións de actualización.

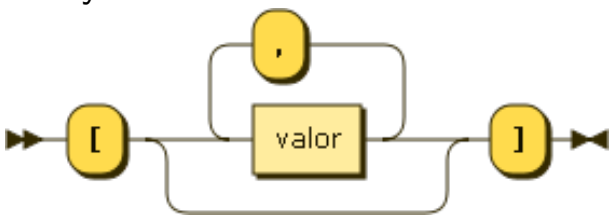
JSON

Gramática (www.json.org)

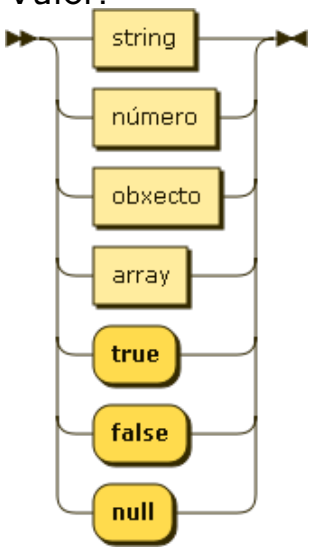
- Obxecto:



- Array:



- Valor:



Exemplos

```
/* Obxecto baleiro*/
{ }

/* Obxecto simple (persoa) */
{
 "nome" : "Pedro",
 "profesion" : "astronauta",
 "salario" : 50000
}

/* Aumentamos a complexidade  (factura) */
{
 "id": 1,
 "cliente": {
             "nome" : "Juan Pérez",
             "NIF": "123345678A",
             "enderezo": {
                          "rua": "Real",
                          "numero" : "12",
                          "poboacion" : "A Coruña"
                         }
            }
 "pagada" : true,
 "liñas" : [
            {"concepto": "pan", "prezo" : 0.9 },
            {"concepto": "viño", "litros" : 1, "prezo": 6.95 }
           ]
}
```

Documentos de MongoDB

- Un documento é basicamente un obxecto JSON.
- Hai algunha diferencia
  - Oficialmente os obxectos son coleccións ordenadas de pares nome:valor (pero os drivers poden desordenalos). Non é demasiado relevante.
  - Utilizando a shell `mongosh`, non é necesario entrecomillar os nomes se usan só caracteres "válidos como nome de variable" (si é obrigado para os valores que son strings).
    Ex: `{nome : "Pepe"}`
- Almacénase e transfírese usando o formato BSON (Binary JSON)
  - Incrementa os tipos de datos (21 tipos –5 deles considereado obsoletos– na versión 4.0), incluíndo Double, Date, ObjectId, Boolean, Binary Data, Regular Expression etc. (`http://docs.mongodb.org/manual/reference/bson-types/`)
  - Ten un tamaño máximo de 16MB por documento (Don Quijote de la Mancha no Proxecto Gutenberg ocupa 2.1MB)
  - MongoDB permite utilizar `GridFS` para almacenar obxectos de tamaño grande (por exemplo multimedia).
- Os documentos almacénanse en *coleccións* (concepto análogo a táboa relacional).
- As coleccións organízanse en bases de datos.
- Unha instancia de MongoDB pode xestionar varias bases de datos.

Documentos de MongoDB (cont.)

- Todo documento ten un campo `_id`
  - É único dentro da colección
  - É inmutable.
  - Pode ser de calquera tipo de datos excepto un array, incluso un documento complexo.
    O tipo predeterminado é `ObjectId`.
  - Se non se indica este campo, MongoDB asígnalle un automaticamente (de tipo `ObjectId`)

```
> var doc = { nome : "Pepe" }
> db.proba.insertOne(doc)
  { acknowledged: true, insertedId: ObjectId("6101070b3f95d03444e58667") }
> doc
  { nome: 'Pepe' }
> db.proba.find()
  [ { _id: ObjectId("6101070b3f95d03444e58667"), nome: 'Pepe' } ]
```

- Esquemas flexibles: Nunha colección pode haber documentos con distintos esquemas.
- Posibilidade de validar esquemas usando unha sintaxe similar á que se verá para facer búsquedas.
  - Existencia de campos.
  - Comprobación de tipos.
  - Restricións adicionais (valores válidos).

```
db.createCollection("emps",
{ validator: {
      nome: {$type: "string"},
      salario: {$gt: 950},
      email: { $regex: /@empresa\.com$/ }
     }
})
```

Índice

Operacións sobre os datos

- CRUD: ***C**reate*, ***R**ead*, ***U**pdate*, ***D**elete*.
  Inserción, consulta, actualización e borrado de documentos nunha colección.
  - Desde a shell (`mongosh`) podemos utilizar unha serie de métodos da clase colección.
    - Inserción: `insertOne(doc)` / `insertMany(array)` (tamén actualizacións con `{upsert : true}`).
    - Consulta: `find()`, `findOne()`
    - Actualización: `updateOne()` / `updateMany()`.
    - Borrado: `deleteOne()` / `deleteMany()`(`drop()` para eliminar completamente a colección).
  - Desde linguaxes de programación, os drivers implementan as mesmas funcinalidades (a sintaxe varía coa linguaxe).
- *Aggregation framework*: Permite facer consultas agregadas, por exemplo consultas similares ás de SQL con `group by`: `aggregate()`
- Índices: Podemos crear índices sobre os campos dos documentos.
- `db` é unha variable que en `mongosh` contén a base de datos actual.

  As operacións invócanse mediante `db.<colección>.<operación>(<parámetros>)`.

  Ex: `db.proba.insertOne({a:1})` realiza na colección `proba` da BD actual a operación `insertOne()`, insertando o documento que se pasa como parámetro (`{a:1}`).

Inserción de datos

- Sintaxe:
  - `db.<colección>.insertOne(<doc>)` inserta un documento.
  - `db.<colección>.insertMany(<array>)` inserta un array de documentos.
  - `db.<colección>.insert(<documento / array de documentos>)` (obsoleta)
- Recordemos que `db` é a base de datos actual.
  - Non é necesario que exista. Se non existe créase automaticamente ó crear a primeira colección.
- `<colección>` é o nome da colección onde se insertará o documento.
  - Non é necesario que exista. Se non existe créase automaticamente cando se engade o primeiro documento.
  - Tamén podería crearse antes con `db.createCollection()`
    - Útil para tipos especiais de coleccións como *capped collections*.
    - Se queremos validar o esquema desde o inicio.
  - A colección non ten esquema fixo, polo que pode conter documentos con distinta estrutura.
    A partir da versión 3.2 MongoDB permite validar o esquema dos documentos.
- Todo documento almacenado terá un campo `_id`.
  Se non se indica de forma explícita, MongoDB engade un `_id` de tipo `ObjectId`.
- Parámetros adicionais opcionais: `writeConcern` e `ordered`.

Inserción de datos

Exemplos

(Nótese a forma especial de insertar un campo de tipo temporal como a data de nacemento)

```
> db.usuarios.insertOne({ _id : "miguel.penabad",
                          nome : "Miguel Rodríguez Penabad",
                          email : "miguel.penabad@udc.es",
                          roles : ["administrador", "profesor"]});

{ "acknowledged" : true, "insertedId" : "miguel.penabad" }


> db.usuarios.insertMany([ { nome : "Pepe Gotera", datanac: new Date("1966-04-02") },
                           { nome : "Otilio"} ]);
{
        "acknowledged" : true,
        "insertedIds" : {
                0 : ObjectId("57e0e19d8b059d73cb8812ff"),
                1 : ObjectId("57e0e19d8b059d73cb881300")
        }
}

> db.usuarios.insert({_id: "usuario", nome: "Usuario"})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': "usuario" }
}
```

Selección de datos

- Sintaxe:
  `db.<colección>.find(<criterios>, <proxección>)`
- Devolve un *cursor*.
  - Na shell devolve os 20 primeiros documentos. Tecleando `it` recuperamos os seguintes.
  - Poderíamos iterar sobre o cursor utilizando JavaScript e unha función (pode usarse a notación con =>).

```
> db.usuarios.find().forEach(
     function (doc) {
        print("Usuario: ", doc.nome)
      }
)
```

```
> db.usuarios.find().forEach(
     (doc) => {print("Usuario: ", doc.nome)}
   )
```

```
Usuario:  Miguel Rodríguez Penabad
Usuario:  Pepe Gotera
Usuario:  Otilio
Usuario:  Usuario
```

- `db.<colección>.findOne(<criterios>, <proxección>)` devolve un só documento que satisfaga os criterios (non se sabe cal deles).
- `<criterios>` e `<proxección>` son documentos JSON. Ambos son opcionais.

Selección de datos

- Os criterios de selección:
  - Sen `<criterios>` (ou sendo o documento baleiro `{}`) obtén todos os documentos.
  - Veremos por separado distintos tipos de criterios e exemplos.
- A proxección especifica os campos que se devolven. era
  - `<proxección>` terá o formato `{campo1:1, campo2:1, _id:0}`
    (1 ou true = visible; 0 ou false = invisible)
  - Omitindo a proxección, móstranse todos os campos.
  - Se só especificamos `{campo:0}` mostra todos os demáis campos (especialmente útil para indicar `{_id:0}`).
  - Se se activa algún campo (`{campo:1}` deben activarse todos os que se desexen ver (coa excepción de que `_id` está a 1 de forma predeterminada).

```
> db.usuarios.find()
```

```
{ "_id" : "miguel.penabad", "nome" : "Miguel Rodríguez Penabad",
   "email" : "miguel.penabad@udc.es", "roles" : [ "administrador", "profesor" ] }
{ "_id" : ObjectId("5600297758cc02907aa22b46"), "nome" : "Pepe Gotera",
  "datanac" : ISODate("1966-04-02T00:00:00Z") }
{ "_id" : ObjectId("5600297758cc02907aa22b47"), "nome" : "Otilio" }
{ "_id" : "usuario", "nome": "Usuario" }
```

Se queremos obter todos os documentos pero especificar a proxección de certos campos, debemos indicar os criterios con `{}`, xa que a proxección é o segundo argumento da función.

```
> db.usuarios.findOne({}, {_id:0, nome:1})
```

```
{ "nome" : "Miguel Rodríguez Penabad" }
```

Selección de datos

Colección de exemplo

Usaremos a colección `contactos` para poñer exemplos de consultas. Non todos os contactos teñen toda a información. Exemplo:

```
{
        "_id" : ObjectId("562e6241bbc7db1bc09134a3"),
        "nome" : "Leire",
        "apelidos" : "Ramos Posada",
        "sexo" : "M",
        "email" : "lramos@gmail.com",
        "datanac" : ISODate("1982-09-14T00:00:00.000Z"),
        "enderezo" : {
                "poboacion" : "Barcelona",
                "cp" : "08493",
                "provincia" : "Barcelona",
                "rua" : "Paseo Lagarella,159 2ºF"
        },
        "tels" : [
                {
                        "tipo" : "móbil",
                        "numero" : "620691334"
                },
                {
                        "tipo" : "fixo",
                        "numero" : "939985342"
                },
                {
                        "tipo" : "móbil",
                        "numero" : "627150926"
                }
        ],
        "chamadas" : 35
}
```

Selección de datos

Criterios

- Formato básico dun criterio:
  - `{campo: <constante>}`: Busca por igualdade. Exemplo: `{nome: "Leire"}`.
  - `{campo: <documento>}`: Permite aplicar un operador, ou aplicar varias condicións sobre o mesmo campo. Ex: `{nome: {$eq: "Leire"}}`.
- Igualdade (non usamos habitualmente o operador eq)
  - Contactos de nome "Leire": `db.contactos.find({nome:"Leire"})`
  - Contactos con 5 chamadas: `db.contactos.find({chamadas:5})`
  - Contactos nacidos en determinado día: `db.contactos.find({datanac:new Date("1982-09-14")})` (no noso exemplo, as datas teñen todas "cero horas")
  - Contactos da provincia de Lugo: `db.contactos.find({"enderezo.provincia":"Lugo"})` (`"enderezo.provincia"` debe ir entre comiñas obrigatoriamente).
  - Contactos con móbil: `db.contactos.find({"tels.tipo":"móbil"})` (Busca nos elementos de `tels`, que é un array)
- Existencia: `$exists`, `null`.
  - Contactos que teñen o campo co número de chamadas: `db.contactos.find({chamadas:{$exists:1}})`
  - Contactos que non teñen o campo co número de chamadas: `db.contactos.find({chamadas:{$exists:0}})`
    A opción `db.contactos.find({chamadas:null})` obtería aqueles documentos onde `chamadas` non existe, ou si que existe pero ten valor `null`.

Selección de datos

Criterios

- Desigualdade:
  - `$lt` (*less than*, $<$)
  - `$lte` (*less than or equal*, $\leq$)
  - `$gt` (*greater than*, $>$)
  - `$gte` (*greater than or equal*, $\geq$)
  - `$ne` (*not equal*, $\neq$)
  - Para igualdade, podemos poñer o valor directamente ou usar `$eq`.
  - Exemplos:
    - Contactos con máis de 5 chamadas: `db.contactos.find({chamadas:{$gt:5}})`
    - Contactos con entre 5 e 10 chamadas (ambas inclusive):
      `db.contactos.find({chamadas:{$gte:5, $lte:10}})`
    - Contactos que non son da provincia de Ourense:
      `db.contactos.find({"enderezo.provincia":{$ne:"Ourense"}})`
      Nota: aparecerán tamén as persoas que non teñan `enderezo.provincia`.

- Expresións regulares: `{$regex : patrón}` ou `/patron/`.
  - Usa expresións regulares compatibles con Perl (PCRE v8.42).
    Emails do dominio `gmail.com`:
    `db.contactos.find({email : /@gmail.com$/}, {email:1, _id:0})`
    `db.contactos.find({email : {$regex:'@gmail.com$'}}, {email:1, _id:0})`
    Nomes de persoas que empezan por `A` ou por `S` e que acaben en a:
    `db.contactos.find({nome: /^(A|S).*a$/},{nome:1, _id:0})`

Un inciso...

Expresións regulares

- Búsqueda con `/patrón/opcións` ou `{$regex:"patrón", $options:"opcións"}`
- Opcións: Son opcionais, poden omitirse. A máis relevante é `i` que fai que a busca sexa insensible a maiúsculas ou minúsculas.

Caracteres significativos no patrón:

Carácter	***Significado***
`^`	*Inicio da liña*
`$`	*Fin da liña*
`.`	*Calquera carácter*
`(...)`	*Paréntesis, permiten agrupar expresións*
`|`	*Operador "Or"*
`[...]`	*Especificación dun conxunto de caracteres separados por coma (lista) ou guión (rango)*
`?`	*(Multiplicidade) cero ou un do anterior*
`*`	*(Multiplicidade) cero ou máis do anterior*
`+`	*(Multiplicidade) un ou máis do anterior*
`\`	*Carácter de escape*

Exemplos:

Patrón	***Texto que sigue o patrón***			
`^a`	*a*	*abcd*		
`S$`	*S*	*caS*		
`ab*c`	*ac*	*abc*	*abbc*	*abbbc*
`^a.S$`	*aaS*	*abS*		
`^a.+S$`	*aaS*	*abcS*	*axxxxxS*	
`^a.*S$`	*aS*	*aaS*	*axxxxxS*	
`^(a|A)na`	*ana*	*Ana*		
`c(an)?ana`	*cana*	*canana*	*xzcananaww*	
`[a-c,h]W`	*aW*	*xxxbWx*	*cW*	*hW*

Un inciso...

Expresións regulares

Exemplos finais:

Emails que empecen por `d` e que sexan de `hotmail.com` ou `gmail.com`:

```
db.contactos.find({"email": /^d.*@(hotmail|gmail)\.com$/  }, {_id:0, email:1})
```

Examinando a expresión regular, vemos que:

- `^d`: empeza por `d`
- `.*`: cero ou máis letras
- `@`: Literalmente `@`
- `(hotmail|gmail)`: Texto `hotmail` ou `gmail`.
- `\.`: Un punto (usando o carácter de escape para eliminar o significado de "calquera carácter" asociado ó punto)
- `com$`: Remata por `com`

Contactos da provincia de Lugo, escribindo calquera das súas letras en maiúsculas ou minúsculas

```
db.contactos.find({"enderezo.provincia": /lugo/i}, {_id:0, "enderezo.provincia":1})
db.contactos.find({"enderezo.provincia": {$regex:"lugo",$options:"i"}},
                  {_id:0, "enderezo.provincia":1})
```

Selección de datos

Criterios

- Conectivas lóxicas: `$and`, `$or`, `$not`
  - Se poñemos varios criterios separados por comas, hai un `AND` implícito.
    Persoas con correo en Gmail *e* que teñan máis de 20 chamadas.
    `db.contactos.find({email : /@gmail.com/, chamadas:{$gt:20}})`
  - `$and : [array de criterios]` e `$or : [array de criterios]`
    Persoas con correo en Gmail *ou* máis de 20 chamadas:
    `db.contactos.find({$or:[{email : /@gmail.com/}, {chamadas:{$gt:20}}]})`
  - `$and` e `$or` son operadores de primeiro nivel, non se aplican a un atributo como por exemplo `$lt` ou `$regex`. Ademais, poden anidarse para construir expresións booleanas complexas.
  - Pode utilizarse a negación con `$not` (aplicable a unha expresión –documento– ou a unha regex). Emails que non sexan de Gmail:
    `db.contactos.find({email : {$not:/@gmail.com/}}, {email:1, _id:0})`

    Ollo! Ó contrario do comportamento dunha BD relacional con SQL, un documento co campo email ausente ou a `null` si que verificaría a condición.

    Podemos excluir eses nulos ou campos ausentes así:
    `db.contactos.find({email : {$not:/@gmail.com/, $ne:null}}, {email:1, _id:0})`

Selección de datos

Criterios

- Consultas con arrays
  - Recordemos o exemplo xa visto: contactos con teléfono móbil
    `db.contactos.find({"tels.tipo":"móbil"})`
  - Acceso a un índice concreto do array: notación con puntos. O índice empeza en 0.
    Contactos que teñan o fixo como primeiro teléfono:
    `db.contactos.find({"tels.0.tipo":"fixo"})`
  - Varias condicións sobre o mesmo elemento do array: `$elemMatch`
    Considera esta colección `test`:

```
{ "a" : 1, "b" : [ { "x" : 1, "z" : 1 }, { "x" : 2, "z" : 4 } ] }
{ "a" : 2, "b" : [ { "x" : 1, "z" : 2 }, { "x" : 2, "z" : 1 } ] }
```

    Documentos que teñan tanto `x` como `z` igual a 1 no mesmo elemento do array:

    Incorrecto: `db.test.find({"b.x":1, "b.z":1})` (devolve os 2 documentos)

    Correcto: `db.test.find({ b : { $elemMatch : { x:1 , z:1} } } )`

Selección de datos

Criterios

- Consultas con arrays (cont.)
  - Uso de arrays como alternativa a `$or`: `$in`, `$nin`

    Concactos con 2 ou 4 chamadas: Pode facerse con `$or`:
    `db.contactos.find({ $or : [ {chamadas:2}, {chamadas:4} ] })`

    Ou pode usarse o operador `$in` e construir un array cos valores posibles:
    `db.contactos.find({chamadas: {$in: [2, 4]}})`

    O complementario do anterior pode obterse con `$nin` (not in):
    `db.contactos.find({chamadas: {$nin: [2, 4]}})`

    Debemos ter en conta que devolve tamén os contactos co campo `chamadas` ausente ou nulo.

  - Uso de arrays como alternativa a `$and`: `$all`
    Contactos con móbil e fixo:

    Opción **non** válida:

    ```
    db.contactos.find({"tels.tipo":"móbil", "tels.tipo":"fixo"})
    ```

    (se repetimos a clave "tels.tipo", só se considera a segunda)

    Correctas:

    - `db.contactos.find({$and:[{"tels.tipo":"móbil"}, {"tels.tipo":"fixo"}]})`
    - `db.contactos.find({"tels.tipo":{$all:["móbil", "fixo"]}})`

Selección de datos

Criterios

- Consultas con JavaScript (`$where`)
  Hai consultas que MongoDB non permite facer con `find()`, ou cando menos non tan directamente como poderíamos esperar. Algunha moi simple, como comparar 2 campos do mesmo obxecto (documento).
  Exemplo de colección:

```
> db.test2.find({},{_id:0})
{ "a" : 1, "b" : 1 }
{ "a" : 2, "b" : 1 }
```

  Queremos obter os documentos con a igual a b.

```
> db.test2.find({a:b})
2015-11-17T11:41:43.415+0100 E QUERY    ReferenceError: b is not defined
    at (shell):1:18
```

  Podemos resolvelo co predicado `$where : string`, onde `string` é unha expresión ou unha función JavaScript completa. É extremadamente flexible, pero tamén é **moi ineficiente** (nunca usa índices).

```
db.test2.find( { $where: "this.a == this.b" } )
db.test2.find( { $where: "function() { return this.a == this.b } "} )
```

NOTA: *A consulta realmente si que pode facerse, usando o operador* `$expr`*:*

```
db.test2.find({$expr: {$eq: ["$a", "$b"]}})
```

Falaremos deste operador e outros conceptos relacionados no apartado do Framework de agregación.

Selección de datos

Cursores

- `db.`*coleccion*`.findOne()` devolve un só documento.
- `db.`*coleccion*`.find()` devolve un cursor sobre unha colección de documentos.

  O cursor permite contar, ordernar, saltar e/ou limitar os documentos obtidos.
  - Contar: `count()`
  - Ordenar: `sort({campo: orde, ...})`
    Pode ordenarse sucesivamente por varios campos; `orde` pode ser `1` (asecendente) ou `-1` (descendente). Os nulos son "menores" que os demais valores.
    Pode ordenarse utilizando `$natural` como se fose un campo, e significa a ordenación por orde de inserción.
  - Saltar: `skip(numero)`
  - Limitar: `limit(numero)`

```
> var c = db.contactos.find()
> c.count()
20
> c.sort({chamadas:1})
... [20 documentos ordenados por chamadas]

> db.contactos.find({sexo : "M"}).sort({datanac:1}).skip(3).limit(5)
... [5 documentos ordenados por datanac, saltando os 3 primeiros]
```

Selección de datos

Cursores - JavaScript

É posible utilizar JavaScript para recorrer o cursor que devolve `find()`.
Exemplos:

```
db.contactos.find({"enderezo.provincia": 'A Coruña'}).forEach(
    function(doc){
        print(doc.nome, doc.apelidos)
    })

------

contactos_prov = function(prov){
    cursor = db.contactos.find({"enderezo.provincia": prov})
    while (cursor.hasNext()){
        doc = cursor.next()
        print(doc.nome, doc.apelidos)
    }
}

> contactos_prov("A Coruña")
```

Modificación de datos

- Posibilidades de modificación:
  - Un documento, reemplazándoo por outro (excepto o `_id` que é inmutable).
  - Un ou varios documentos, modificando os documentos: pódense modificar, engadir ou eliminar campos.
- Sintaxe:
  - `db.<coleccion>.replaceOne(criterios, actualización, opcións)`: Reemplaza un documento.
  - `db.<coleccion>.updateOne(criterios, actualización, opcións)`: Modifica un documento.
  - `db.<coleccion>.updateMany(criterios, actualización, opcións)`: Modifica varios documentos.
  - `db.<coleccion>.update(criterios, actualización, opcións)`: Obsoleto, serviría para as 3.
- Os `criterios` son como os de selección con `find()`.
- A `actualización` é un documento que especifica os cambios a realizar (reemplazo ou modificación).
- As `opcións` son un documento que permite especificar:
  - `upsert : false|true`: (predeterminado: false). Se non hai ningún documento coincidente e upsert é true, insértase un documento novo.
  - `writeconcern`: Explicarase cando se describan os *replica sets*.
  - `multi : false|true`: (predeterminado: false). Obsoleto, só con `update()`. Indicaría se modifca un documento ou todos. Recoméndase usar `updateOne()` ou `updateMany()`.

Modificación de datos

- Sustitución do documento completo (excepto `_id`) con `replaceOne()`

```
test> db.usuarios.findOne({_id:"usuario"})
{ _id: 'usuario', nome: 'Usuario' }

test> db.usuarios.replaceOne({_id:"usuario"}, {login:'usuario', contrasinal:'secret'})
{ acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 1, upsertedCount: 0 }

test> db.usuarios.findOne({_id:"usuario"})
{ _id: 'usuario', login: 'usuario', contrasinal: 'secret' }
```

- Modificacións sobre o documento (especifícanse en forma de documento JSON).

  Podes ver a lista completa en `https://docs.mongodb.org/manual/reference/operator/update/`.
  Exemplos básicos:

  - `$set: {campo: valor, ...}`: Establece o valor dun campo, creándoo se non existe.
  - `$unset: {campo: "", ...}`: Elimina os campos. O valor utilizado (neste caso `""`) é irrelevante.
  - `$rename: {campo: novo_nome, ...}`: Renomea un campo.
  - `$inc : {campo : cantidade}, ...` Incrementa o campo (debe ser numérico) na cantidade indicada, ou establece esa cantidade como valor se non existe.
  - `$mul: {campo : cantidade, ...}` é similar, pero multiplica por ese valor se existe o campo, e establece o valor a 0 se non.

  Podemos combinar varios operadores na mesma actualización.

Modificación de datos

- Exemplos de modificacións básicas.

  `Documento inicial: { b : 2, c : 2 }`

  - Establecemos un valor novo para `c` e creamos o campo `d`.

```
> db.test.updateOne({}, {$set: {c:3, d:1}  })
{ b : 2, c : 3, d : 1 }
```

  - Eliminamos os campos `c` e `x` (este non existe, pero non da erro). Nota os valores irrelevantes que usamos en `$unset`.

```
> db.test.updateOne({}, {$unset: {c : '', x: 3} })
{ b : 2, d : 1 }
```

  - Incrementa o campo `b` en 7 unidades e `f` en 3. Se non existen, establece eses valores iniciais nos campos. Ó mesmo tempo, multiplica `d` por 3 e `g` por 11. Nota que se non existe o campo quedará con valor 0.

```
> db.test.updateOne({}, {$inc: {b:7, f:3}, $mul : {d:3, g:11} })
{ b : 9, d : 3, f : 3, g : 0 }
```

  - Renomea o campo `f` como `novoF`

```
> db.test.updateOne({}, {$rename: {'f':'novoF'}})
{ b: 9, d: 3, g: 0, novoF: 3 }
```

Modificación de datos

- Actualización de arrays
  - `$push: {array: elemento}` engade un elemento ó array.
  - `$addToSet: {array: elemento}` só o engade se non está xa presente.

    Para ambos, se `elemento` fose un array, engadiríase como un único elemento.
    Se queremos engadir o contido como ítems separados, usaremos o operador `{$each:elemento}`.

    Ex:

    ```
    db.test.insertOne({_id:0, arr: [1,2]})

    a) db.test.updateOne({_id:0}, {$push: {arr : [3,4]}})          --- Quedaría arr: [1, 2, [3, 4] ]
    b) db.test.updateOne({_id:0}, {$push: {arr : {$each:[3,4]}}})  --- Quedaría arr: [1, 2, 3, 4 ]
    ```

  - `$pull: {array : elemento/condición}`: Elimina elementos.
  - Notación posicional $: Permite modificar un valor dentro dun array utilizando o operador `$set`.

    O primeiro documento de `updateOne()`/`updateMany()` localiza o elemento, e o documento de actualización utiliza $.

    ```
    updateMany({array: criterio}, {$set: {array.$ : valor}})
    ```

Modificación de datos

- Exemplos de actualización en arrays Documento inicial

```
> db.arrays.insertOne({froitas: ["pera", "mazá"], moedas: ["euro", "dólar"]})
```

Engadimos a froita `pera` e, se non existe, cada moeda do array `["dracma", "dinar", "euro"]`. Como se ve, agora `pera` aparece duplicado, pero non se engade `euro` no array de moedas.

```
> db.arrays.updateOne({},
        {  $push : {froitas: "pera"},
           $addToSet: {moedas: {$each: ["dracma", "dinar", "euro"]}}
        })

 { "froitas" : [ "pera", "mazá", "pera" ], "moedas" : [ "euro", "dólar", "dracma", "dinar" ] }
```

Eliminamos a froita `pera` (todas as súas ocurrencias) e tamén as moedas `dólar` e `euro`.

```
 > db.arrays.updateOne({}, { $pull : {froitas: "pera"}})
{ "froitas" : [ "mazá" ], "moedas" : [ "euro", "dólar", "dracma", "dinar" ] }

 > db.arrays.updateOne({}, { $pull : {moedas: {$in: ["dólar", "euro"]}} })
{ "froitas" : [ "mazá" ], "moedas" : [ "dracma", "dinar" ] }
```

Cambiar a moeda `dracma` por `euro` (usando notación posicional $).

```
 > db.arrays.updateOne({moedas:"dracma"}, {  $set : {"moedas.$": "euro"} })
{ "froitas" : [ "mazá" ], "moedas" : [ "euro", "dinar" ] }
```

Modificación de datos

- Uso da opción `{upsert:true}`
  - É válido para as 3 funcións: `replaceOne()`, `updateOne()` e `updateMany()`
  - Se os criterios da función non atopan ningún documento, `{upsert:true}` fai que se cree un novo documento coas modificacións indicadas.
  - Se o criterio é de igualdade: con `updateOne()` e `updateMany()` inclúese no documento creado. Con `replaceOne()` non se inclúe.
- Exemplos de upsert (`testupsert` inicialmente non ten documentos):
  - Sen `{upsert:true}` non inserta un novo documento.

```
> db.testupsert.replaceOne({a:1},{b:1})
{ acknowledged: true, insertedId: null, matchedCount: 0, modifiedCount: 0, upsertedCount: 0 }

> db.testupsert.updateOne({a:1},{$set:{b:1}})
{ acknowledged: true, insertedId: null, matchedCount: 0, modifiedCount: 0, upsertedCount: 0 }
```

Modificación de datos

- Exemplos de upsert (cont.):
  - Con `{upsert:true}` engádese o documento.

    Nótese a diferencia entre `replaceOne()` que non engade o criterio, a pesar de ser igualdades, mentras que `updateOne()` si que o inclúe.

```
> db.testupsert.replaceOne({a:1}, {b:1}, {upsert:true})
{acknowledged: true, insertedId: ObjectId("6102799867feac616d3fd4e6"),
  matchedCount: 0, modifiedCount: 0, ... upsertedCount: 1 }

> db.testupsert.updateOne({a:2}, {$set:{b:2}}, {upsert:true})
{... upsertedCount: 1 }

t> db.testupsert.find({},{_id:0})
  { b: 1 },      -- replaceOne() non incluíu o criterio ({a:1}) no documento
  { a: 2, b: 2 } -- updateOne() si incluíu o criterio {a:2}
```

  - Upsert con criterios que non usan igualdades.

```
> db.testupsert.replaceOne({a:{$gt:20}}, {e:1}, {upsert:true})
> db.testupsert.updateOne({a:{$gt:20}}, {$set:{f:1}}, {upsert:true})
> db.testupsert.updateMany({a:{$gt:20}}, {$set:{g:1}}, {upsert:true})

test> db.testupsert.find({},{_id:0})
{ b: 1 },
{ a: 2, b: 2 }
{ e: 1 }       -- Ningunha das 3 funcións
{ f: 1 }       -- engadiu o criterio,
{ g: 1 }       -- por non ser igualdades.
```

Borrado de datos

- Borrado de documentos
  - `db.<colección>.deleteOne(<criterios>)`: Borra un documento que satisface os `criterios`.
  - `db.<colección>.deleteMany(<criterios>)`: Borra todos os documentos que satisfacen os `criterios`.
    - Os criterios son obrigatorios. Para indicar todos os documentos: `{}`.
    - Só borra os documentos. A colección pode quedar baleira.

```
> db.testupsert.deleteOne({})
{ acknowledged: true, deletedCount: 1 }

> db.testupsert.deleteMany({})
{ acknowledged: true, deletedCount: 4 }

> db.testupsert.find()
   --- [non hai documentos ] ---
```

  - `db.<colección>.remove(<criterios>, justOne)`: Obsoleta.
    Borra un documento ou todos, dependendo de se o segundo parámetro é `true` ou `false`.
- Eliminación de colecciónss por completo
  - `db.<colección>.drop()`
  - Se hai índices sobre a colección, tamén son eliminados.
  - Devolve como resultado (true/false) se borrou a colección ou non existía.

```
> db.test.drop()
   true

> show collections
-- [test non aparece] --
```

Índice

Agregación

Introdución

- Moitos sistemas NoSQL non permiten a realización directa de consultas complexas, entre elas consultas de agregación (como sumas de ventas por zona, etc.). Algúns recurren ó paradigma *Map-Reduce* para obter este tipo de resultados.
- MongoDB implementa operacións *Map-Reduce* mediante a función `db.`*colección*`.mapReduce(<mapFunction>, <reduceFunction>, <optional params>)`.

  MongoDB 5.0+ con `mongosh` indica que é obsoleta e recomenda usar o framework de agregación.

  Parámetros opcionais principais:

  - Saída: `output: "nome_coleccion"` ou `output: {inline:1}`
  - Filtro (mesma sintaxe que `find`): `query: {filtro}`
- Map-Reduce implica programación en JavaScript (ou utilizar outras ferramentas como Hadoop).

  Exemplo: Para cada un dos sexos dos contactos indica o total de chamadas.

  Usamos os 2 parámetros opcionais, un para indicar que queremos os resultados nunha nova colección `saida` e outro para filtrar os contactos sen chamadas (sen ese filtro, que exclúe 3 contactos sen chamadas,os totais serían ambos `NaN`).

```
> db.contactos.mapReduce(
    function(){emit(this.sexo, this.chamadas)},
    function(key, values) {return Array.sum(values)},
    {out:"saida",
     query:{chamadas:{$exists:true}}} )
```

```
> db.saida.find()
  { "_id" : "H", "value" : 164 }
  { "_id" : "M", "value" : 103 }
```

Agregación

Introdución

- Ademais, MongoDB ofrece o *Aggregation Framework*, para facer estas consultas de forma máis simple, utilizando a función `db.`*colección*`.aggregate()`.

  Tamén hai funcións de agregación de propósito específico, como son `countDocuments()` ou `distinct("campo")`.

```
> db.contactos.aggregate([
   { $group: {_id:"$sexo", suma: {$sum : "$chamadas"} } }
   ])
{ "_id" : "H", "suma" : 164 }
{ "_id" : "M", "suma" : 103 }

> db.contactos.distinct("sexo")
[ "M", "H" ]

> db.contactos.countDocuments()       -- db.contactos.count(): obsoleto
20

> db.contactos.countDocuments({sexo:"M"})
9
```

  Nota: para contar documentos tamén se pode utilizar o método `count()` do obxecto `cursor` que devolve a función `find()`.

```
> db.contactos.find().count()
20
```

Agregación

Conceptos xerais

- A función `aggregate()` procesa os datos usando *pipelines*.

```
> db.<colección>.aggregate(
   [
    {<etapa 1>},
    {<etapa 2>},
    ...
    {<etapa n>}
   ]
  )
```

- Defínese unha serie de etapas (cada unha realizando unha operación), e a saída dunha etapa será a entrada da seguinte.
- Existen distintos operadores a utilizar como etapas de `aggregate()`. Cando o operador sexa similar a algún de SQL, indicarase.
- A saída de `aggregate()` será un cursor sobre unha colección de documentos.
- En algunha etapa (depende do operador) xenéranse valores a partir de **expresións** que habitualmente usan campos dos documentos de entrada a esa etapa.

Agregación

Expresións

Unha expresión en MongoDB pode ser, entre outras cousas, unha das seguintes:

- Un ***field path***, representado por `"$<nome_campo>"`.
  Representa o valor do campo no documento de entrada á etapa actual.
  Ex: `"$chamadas"`, `"$_id"`, `"$enderezo.rua"`.

- Un valor **literal** (unha constante).

  Pode escribirse directamente ou, se é necesario, usando a sintaxe `{$literal: valor}`

  (Ex: se a constante é un texto que empeza por $, para evitar que se interprete como field path).

  Ex: 3, `"A Coruña"`, `{$literal: "$27.05"}`

- Un operador: `{<operador> : <argumento>}` ou `{<operador> : [<arg1>, ..., <argn>]}`.
  Isto inclúe:
  - Operadores matemáticos: `{$abs: -5}`, `{$add: [3,4]}`, `{$subtract:[5,2]}`, ...
  - Operadores sobre arrays: `{$in: [12,13]}`, `{$avg: array}`, `{$size: "$array"}`, ...
  - Comparacións: `{$eq: [1,1]}`, `{$gt: [2,1]}`, `{$expr: {$eq: ["$a", "$b"]}}` ...
  - Condicionais: `{$cond: [ {$gt: ["$a", "$b"]}, "a > b", "a <= b" ]}`, ...
  - Acumuladores. Verémolos a continuación, na etapa `$group`.
  - Sobre datas, strings, conxuntos, trigonometría, ...

Nota: *Neste documento non utilizaremos todos estes tipos de expresión. Esta transparencia sirve basicamente de referencia.*

Agregación

Expresións

Tamén se poden crear "Obxectos expresión", co formato `{campo: <expresión>}`

Sirven para, por exemplo, crear campos novos.

Exemplo (A etapa `$project` explícase máis adiante):

```
> db.test.insertOne({ _id: 1, a: 10, b: 5, arr: [2, 3, 4] })

> db.test.aggregate(
            {$project: {
                 amaiorb: { $cond: [{ $gt: ["$a", "$b"] }, "si", "non"] },
                 sumaarray: {$sum: "$arr"},
                 tamanoarr: {$size:"$arr"},
                 mediaarr: {$avg:"$arr"}
                 } } )

[ { _id: 1, amaiorb: 'si',   sumaarray: 9, tamanoarr: 3, mediaarr: 3 } ]
```

Referencia máis completa:

`https://www.mongodb.com/docs/manual/meta/aggregation-quick-reference/#std-label-aggregation-expressions`

Agregación

Etapas: Agrupación (`$group`)

Esta etapa é similar ó `group by` de SQL.

```
$group : {_id:expresion,  campo1 : {acumulador1 : expresion1}, ...} }
```

Exemplo:

```
{ $group: { _id:"$sexo", suma: {$sum : "$chamadas"} } }
```

- Sempre debe haber un `_id`, que será o **criterio de agrupación**.
  Pode ser un campo ou un documento máis complexo.
  ```
  _id : "$chamadas"
  _id:{c:"$enderezo.rua", p:"$enderezo.provincia"}
  ```
- Con `_id:null` (ou calquera constante) facemos un único grupo con todos os documentos.
- Os **valores agregados** son obxectos de expresións creadas con acumuladores e field paths (fp).
  Acumuladores máis comúns:
  - Operacións numéricas: `{$sum:"$fp"}`, `{$avg:"$fp"}` (ignoran valores non numéricos)
  - Comparación: `{$max:"$fp"}`, `{$min:"$fp"}`
  - Contar: `{$count:{}}` ou `{$sum:1}`
  - Engadir a arrays: con duplicados (`{$push:"$fp"`) ou sen duplicados nin orde (`{$addToSet:"$fp"}`)

Agregación

Etapas: Agrupación ($group)

Ex: Agrupa os contactos por provincia e poboación, e para cada grupo di cantos contactos hai, o total de chamadas, e a lista de nomes dos contactos dese grupo, sen que se repitan.

```
db.contactos.aggregate([
  { $group: {
             _id: {prov:"$enderezo.provincia", pob:"$enderezo.poboacion"},
             num_contactos: {$count:{}},   /* Tamén: num_contactos: {$sum:1},  */
             total_chamadas: {$sum:"$chamadas"},
             nomes: {$addToSet:"$nome"}
            }
  }
])
```

Agregación

Etapas: Filtrado (`$match`)

Pode corresponder co `where` ou `having` de SQL:

```
{ $match : { <query> } }
```

Exemplo:

```
{ $match: {chamadas:3} }
```

- Filtra documentos, que poden ser os da colección orixinal ou da etapa anterior no pipeline de agregación.
- O formato é igual ó utilizado coa función `find()`, pero permite o uso de epxresións con field paths.

Exemplo: Mostra os contactos que teñan email e teléfono(s), e que teñan un número de chamadas igual á cantidade de teléfonos que teñen.

```
db.contactos.aggregate([
  { $match: {
              email : {$exists:true},
              tels: {$exists: true},
              $expr: {$eq: ["$chamadas", {$size: "$tels"}]}
             }
  }])
```

Nota: O operador de comparación `$expr` pode usarse na etapa `$match` e algunha outra, pero non en todo tipo de etapas.

Agregación

Etapas: Filtrado (`$match`)

Ex: Para os contactos con email, agrupa por sexo e mostra o total de chamadas e o número de contactos, só se no grupo hai máis de 5 contactos.

```
db.contactos.aggregate([
  { $match: { email : {$exists:true} } },
  { $group: {
             _id: "$sexo",
             num_contactos: {$sum:1},
             total_chamadas: {$sum:"$chamadas"},
            }   },
  { $match: { num_contactos:{$gt:5} } }
])
```

Como se ve no exemplo, encadeamos varias etapas no pipeline de agregación.

- O primeiro operador `$match` sería equivalente a un `where` de SQL, xa que filtra os documentos orixinais.
- O segundo sería equivalente a un `having`, filtrando grupos de contactos (falamos de grupos a nivel conceptual, porque a saída do `$group` é unha colección de documentos).

Agregación

Etapas: Reformateado de documentos (`$project`)

Podería corresponder coas expresións seleccionadas nun `SELECT` de SQL.

```
{ $project : { especificación } }
```

Exemplo:

```
{ $project: {_id:0, nome:1, apelidos:1, correo:"$email"} }
```

Permite reestruturar os documentos de saída, incluíndo filtrado de campos e creación de campos novos.

Filtrado de campos: Especificación dos campos que pasan ou non á seguinte etapa:

- Para incluir calquera campo: `campo:1` ou `campo:true`.
- Para excluir calquera campo: `campo:0` ou `campo:false`.
- O `_id` inclúese de xeito predeterminado.
- Se excluímos un campo, os demáis mostraranse, pero se incluímos explícitamente un campo, só se verá ese campo (e máis o `_id`).

Exemplos:

```
-- Colección orixinal
[ { _id: 1, a: 2, b: 3, c: 4 } ]

    > db.testpr.aggregate({$project:{_id:0}})      ==> [ {         a: 2, b: 3, c: 4 } ]
    > db.testpr.aggregate({$project:{a:0}})        ==> [ { _id: 1,       b: 3, c: 4 } ]
    > db.testpr.aggregate({$project:{_id:1}})      ==> [ { _id: 1                     } ]
    > db.testpr.aggregate({$project:{a:1}})        ==> [ { _id: 1, a: 2               } ]
    > db.testpr.aggregate({$project:{_id:0, a:1}}) ==> [ {         a: 2               } ]
```

Agregación

Etapas: Reformateado de documentos (`$project`)

Reestruturación dos documentos:
Usaremos obxectos expresión, normalmente involucrando field paths: `campo:<expresión>`

- Podemos aplanar documentos con `novocampo:"$subdoc.campo"`.
- Podemos engadir subdocumentos con `"doc.novocampo":"$campo"` ou con `doc: {novocampo: "$campo"}`.

Exemplo:

```
db.contactos.aggregate([
{$match: {tels: {$exists:true}}},
    {$project: {
        _id:0,
        apelidosenome: {$concat: ["$apelidos",
                                  ", ", "$nome"]},
        "nome.pila": "$nome",
        "nome.apelidos": "$apelidos",
        email:1,
        provincia: "$enderezo.provincia",
        telefonos: {
            cantidade: {$size: "$tels"},
            lista: "$tels"
            } }} ])
```

```
{
  nome: { pila: 'Leire', apelidos: 'Ramos Posada' },
  email: 'lramos@gmail.com',
  apelidosenome: 'Ramos Posada, Leire',
  provincia: 'Barcelona',
  telefonos: {
    cantidade: 3,
    lista: [
      { tipo: 'móbil', numero: 620691334 },
      { tipo: 'fixo', numero: 939985342 },
      { tipo: 'móbil', numero: 627150926 }
    ]
  }
}
```

Agregación

Etapas: Reformateado de documentos (`$addFields`)

Como se indicou anteriormente, usando `$project`, se se especifica que queremos ver un campo (ou se lle da un valor cunha expresión), o resto dos campos (excepto o `_id`) omítense.

```
> db.contactos.aggregate([ { $project: {algo:{$literal:23}}}])
```

```
  { _id: ObjectId("562e6241bbc7db1bc09134a4"), algo: 23 },
```

O operador (etapa) `addFields` permite engadir campos novos, mantendo visibles todos os campos dos documentos que chegan á etapa acutal (a alternativa sería usar `$project` pero poñendo explicitamente a 1 todos os campos que queremos manter).

A sintaxe para construir os campos é a mesma que a de `$project`.

```
> db.contactos.aggregate([ { $addFields: {algo:{$literal:23}}}])
```

```
{  _id: ObjectId("562e6241bbc7db1bc09134a4"),
    nome: 'Daniel',
    apelidos: 'Gesto Ledo',
    sexo: 'H',
    email: 'danielgl@hotmail.com',
    datanac: ISODate("1941-10-09T00:00:00.000Z"),
    tels: [ { tipo: 'móbil', numero: 678401842 }, { tipo: 'móbil', numero: 671139760 }  ],
    chamadas: 4,
     algo: 23    }
```

Agregación

Etapas: Desaniñar arrays ($unwind)

```
{ $unwind : "$fieldpath" }.
```

Exemplo:

```
{ $unwind: "$tels" }
```

- Desaniña un array creando varios documentos.
- Crea un documento para cada elemento do array (utilizando como nome de campo o do array). Os documentos serán idénticos no resto dos campos.
- $unwind non produce documentos de saída para un documento no que o array non existe, é nulo ou está baleiro ([])

```
> db.testunwind.find()
{ "_id" : 1, "a" : 1, "v" : [ "a", "b", "c" ] }
{ "_id" : 2, "a" : 4 }
{ "_id" : 3, "a" : 5 , "v" : [ ] }

> db.testunwind.aggregate([{$unwind:"$v"}])
{ "_id" : 1, "a" : 1, "v" : "a" }
{ "_id" : 1, "a" : 1, "v" : "b" }
{ "_id" : 1, "a" : 1, "v" : "c" }
```

Ex: Conta cantos teléfonos móbiles aparecen na colección de contactos.

```
db.contactos.aggregate([
      {$unwind: "$tels"},
      {$match: {"tels.tipo":"móbil"}},
      {$group: {_id:null, numero: {$count:{}}}}
])
```

```
-- OLLO á ordenación de $unwind e $match

==>  { "_id" : null, "numero" : 25 }
```

Agregación

Etapas: Outer join (`$lookup`)

```
$lookup : { from: <colección a vincular>,
            localField: <campo na colección local>,
            foreignField: <campo na colección "from" a vincular>,
            as: <campo array co resultado> }
```

- Implementa un Join exterior con outra colección (que ata a versión 5.1 non podía estar particionada).
- Compara por igualdade `localField` e `foreignField`. Se non existen, para a comparación son tratados como `null`.
- Deposita o resultado no array especificado en `as` (sobreescribe o campo se existe na colección local).
- Omitimos nestas transparencias as cláusulas `let` e `pipeline` que permitirían facer consultas máis complexas.

Ex: considera as seguintes colecciónss `emps` e `depts`:

```
{ "_id" : 1, "nome" : "Pepe", "salario" : 2000, "dept" : 10 }
{ "_id" : 2, "nome" : "Ana", "salario" : 2100, "dept" : 10 }
{ "_id" : 3, "nome" : "Suso", "dept" : 20 }
```

```
{ "_id" : 10, "nomedep" : "Compras" }
{ "_id" : 20, "nomedep" : "Ventas" }
```

Obtén para cada departamento todos os datos dos seus empregados, no array "emps"

```
> db.depts.aggregate([
    {$lookup: { from: "emps",  localField: "_id", foreignField: "dept",   as: "emps"       } }  ])
```

```
{ "_id" : 10, "nomedep" : "Compras", "emps" :
          [ { "_id" : 1, "salario" : 2000, "dept" : 10, "nome" : "Pepe" },
            { "_id" : 2, "salario" : 2100, "dept" : 10, "nome" : "Ana" } ] }
{ "_id" : 20, "nomedep" : "Ventas", "emps" :
          [ { "_id" : 3, "dept" : 20, "nome" : "Suso" } ] }
```

Agregación

Etapas: Outer join ($lookup)

Obtén o nome, salario, e nome do departamento de cada empregado.
Coa seguinte consulta obtemos os datos do departamento, pero son todos os datos, e están nun array.

```
> db.emps.aggregate([
   {$lookup: {from: "depts",
              localField: "dept",
              foreignField:"_id",
              as: "depto"}}  ])
```

```
{ "_id" : 1, "nome" : "Pepe", "salario" : 2000, "dept" : 10,
             "depto" : [ { "_id" : 10, "nomedep" : "Compras" } ] }
{ "_id" : 2, "nome" : "Ana", "salario" : 2100, "dept" : 10,
             "depto" : [ { "_id" : 10, "nomedep" : "Compras" } ] }
{ "_id" : 3, "nome" : "Suso", "dept" : 20,
                      "depto" : [ { "_id" : 20, "nomedep" : "Ventas" } ] }
```

Finalmente, desaniñamos o array "depto"e quedamos só cos campos que nos interesa:

```
> db.emps.aggregate([
       {$lookup: {from: "depts",
                  localField: "dept",
                  foreignField:"_id",
                  as: "depto"}},
       {$unwind: "$depto"},
       {$project: {_id:0, nome: 1, salario: 1, nomedep: "$depto.nomedep"}}  ])
```

```
{ "salario" : 2000,   "nome" : "Pepe", "nomedep" : "Compras" }
{ "salario" : 2100,   "nome" : "Ana", "nomedep" : "Compras" }
{ "nome" : "Suso", "nomedep" : "Ventas" }
```

Agregación

Etapas: Outros operadores

- Ordenación: `{ $sort: criterios }`
- Salta documentos: `{ $skip: enteiro_positivo }`
- Limita o número de documentos xenerados: `{ $limit: enteiro_positivo }`

Agrupa os contactos por provincia, conta cantos hai, ordena o resultado por provincias, salta o primeiro e obtén os 2 documentos seguintes.

```
db.contactos.aggregate([
      {$group: {_id:"$enderezo.provincia", numero: {$count:{}}}},
      {$sort: {_id:1}},
      {$skip:1},
      {$limit:2}
])
{ "_id" : "A Coruña", "numero" : 4 }
{ "_id" : "Almería", "numero" : 1 }
```

Agregación

Unha nota sobre eficiencia

O framework de agregación é unha ferramenta moi potente para moitos tipos de consultas.

En algún caso, debido á natureza das propias consultas, ó deseño das coleccións (esquema dos documentos), ou limitacións da versión de MongoDB que usemos, é posible que non se poida usar `aggregate()` como desexamos. Por exemplo, en versións anteriores á 5.1, o operador `$lookup` non funcionaba para buscar datos na colección `from` se esta estaba particionada (*sharded collection*).

Unha alternativa é implementar a consulta no cliente, por exemplo en `mongosh` usnado código JavaScript. Evidentemente, a diferencia na eficiencia é *moi* elevada.

A continuación vexamos 2 consultas que son equivalentes, e vexamos o tempo de execución:

- Arternativa 1: `aggregate()`. **Tempo: 1,989s**

```
db.emps.aggregate([
   {$lookup: {from:"provincias", localField:"enderezo.provincia", foreignField:"_id", as: "prov"}},
   {$unwind: "$prov"},
   {$project: {_id:0, nome:1, apelidos:1, com:"$prov.comunidade"}},
   {$out: "col1"}
])
```

- Alternativa 2: Códiga JavaScript no cliente. **Tempo: 14,774s**

```
db.emps.find().forEach(
    (doc) => {
        p = db.provincias.findOne({_id:doc.enderezo.provincia})
        if(p) db.col2.insertOne({nome: doc.nome, apelidos: doc.apelidos, com: p.comunidade})
        }
)
```

Índice

Replica sets

Introdución

MongoDB utiliza replica sets para ofrecer alta dispoñibilidade:

- Tolerancia a fallos (ante caídas dun servidor), con recuperación automática (*failover*).
- Opcionalmente, aumento da capacidade (só de lectura).

Un replica set está composto por varios nodos, que poden ser:

- Primario:
  - Almacena datos. É o único que recibe peticións de escritura.
  - Só pode haber un por replica set en cada instante.
- Secundario:
  - Almacena datos, neste caso replicando os datos do primario (normalmente).
  - Pode haber varios.
  - Non reciben peticións de escritura dos clientes.
  - Normalmente non son consultados (peticións de lectura). Pode activarse a opción a nivel de sesión do cliente, pero a replicación non é completamente síncrona.
  - Poden usarse para propósitos específicos: backups, recuperación de desastres, consultas analíticas, etc.
- Árbitro:
  - Non almacena datos.
  - Só se usan na votación que ten lugar no proceso de *failover*.

Replica sets

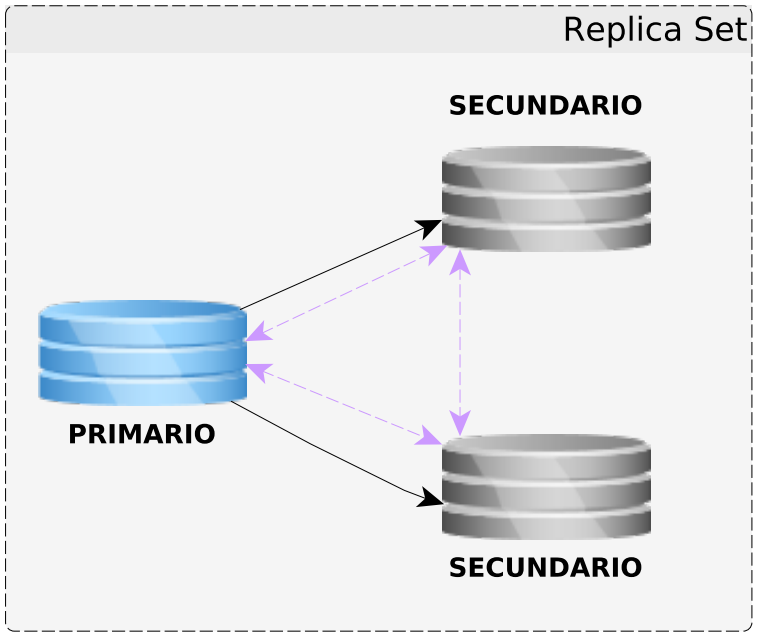
Configuración habitual

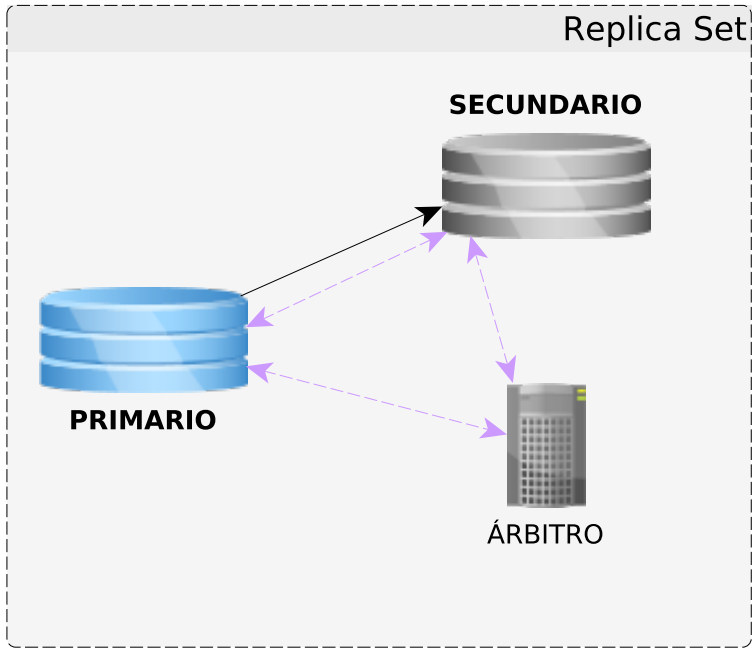
Límites e recomendacións:

- Número máximo de nodos: 50.
- Número máximo de nodos que poden votar: 7.
- Recoméndase un número impar de nodos que poidan votar, para evitar o *split-brain problem*.

Configuracións habituais con 3 nodos:

- Un primario e 2 secundarios: maior replicación (3 copias de cada dato).
- Un primario e 1 secundario (2 copias) máis un árbitro para as votacións.



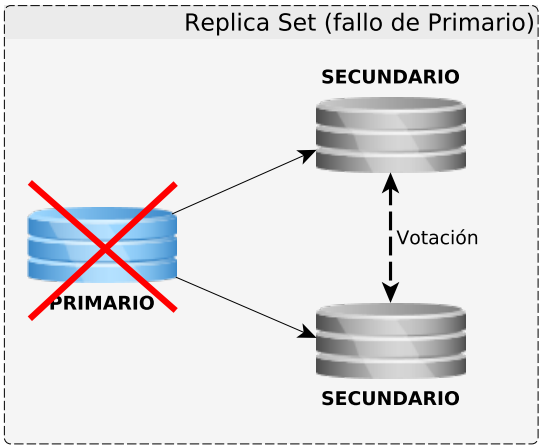


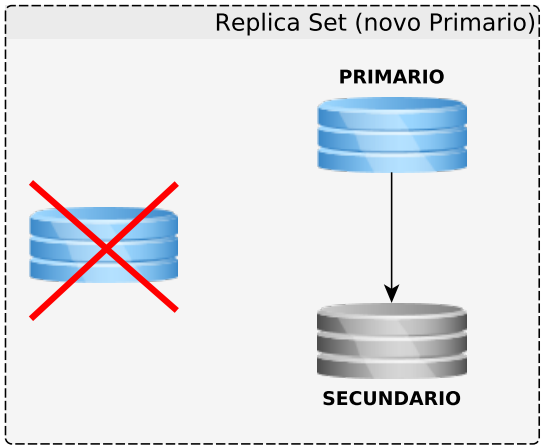
Liñas continuas: Replicación (datos). Liñas discontinuas: Comprobación de dispoñibilidade dos nodos (heartbeat).

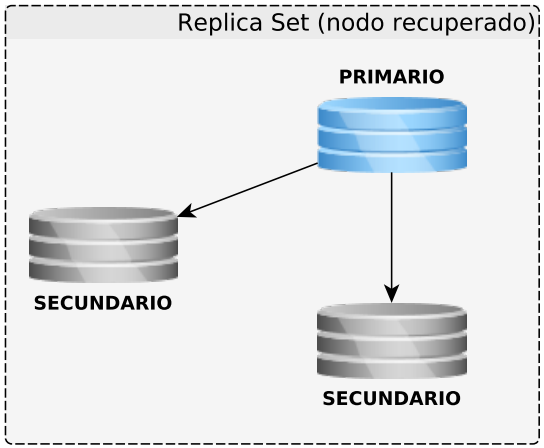
Replica sets

Automatic failover

- A caída dun secundario (ou árbitro) non ten ningún efecto automático. As lecturas e escrituras poden seguir facéndose con normalidade sobre o primario.
- A caída do primario impediría escrituras (e en moitos casos lecturas).
- Automatic failover: Proceso de recuperación automática ante a caída do primario.
  1. Detéctase que o primario está caído (comprobacións *heartbeat*).
     Os restantes nodos (secundarios e árbitros, só aqueles "con dereito a voto") votan para escoller un novo primario entre os secundarios.
     Requírese unha maioría cualificada ($> 50\,\%$ dos votos) para escoller un novo primario, por iso se recomenda un número impar de nodos (evita o *split-brain problem*).
  2. Activado o novo nodo primario, a replica set está de novo completamente funcional.
  3. Se o antigo primario se recupera, agora será un nodo secundario. Resincronizará automaticamente os datos co novo primario.







Replica sets

Preferencias de escritura

- Cando un cliente realiza unha escritura contra o replica set, devolverá a confirmación de que a escritura foi correcta, ou un erro. O nivel de confirmación especifícase co **write concern**: `{ w: <valor>, j: <boolean>, wtimeout: <número>}`.
- Parámetros:
  - O parámetro `w` especifica a cantos nodos (instancias de mongod) se propagou a escritura.
    Un número máis alto implica maior dispoñibilidade e tolerancia a fallos, pero pode producir máis erros se hai nodos caídos, con latencia alta, ou se usan replicación con retardo.
    - `w:0` Non hai confirmación.
      Ex: gravar un log (non ten impacto no usuario, non se faría nada diferente en caso de erro)
    - `w:1` Confirma a escritura no primario.
      Para datos non críticos. Importa máis *write availability*.
    - `w:"majority"` Confírmase a escritura en máis do 50 % dos nodos.
      Usado para case todos os casos, aseguramos máis a dispoñibilidade de datos.
      Correspóndese co nivel `quorum` que ofrece Cassandra.
    - `w:<n>` Confírmase a escritura en *n* nodos.
      Só sirve en replica set (non mongod standalone). *n* é como moito o nº de nodos do replica set.
    - `w: <tag>` Pode usarse se engadimos tags ós servidores.
  - O parámetro `j` especifica se hai que esperar confirmación de que se escribiu no *journal* (o journal úsase para recuperación dun nodo que aborta de xeito inesperado).
  - O parámetro `wtimeout` (para valores `w` > 1) especifica un timeout en milisegundos.
- O write concern pode especificarse a nivel de operación (polo cliente), e tamén se poden especificar valores predeterminados a nivel de replica set.

Replica sets

Preferencias de lectura

- O habitual é que un cliente lea os datos do primario.
- A réplica de datos é asíncrona. Polo tanto, se lemos dun secundario poida que esteamos lendo datos non actualizados (hai *eventual consistency*).
- Os drivers de MongoDB dan soporte a varias preferencias de lectura:
  - `primary` (Predeterminado): Só le do primario.
  - `primaryPreferred`: Le do primario na maioría dos casos, pero se non está dispoñible le do secundario.
  - `secondary`: Le só dun secundario.
  - `secondaryPreferred`: Le dun secundario na maioría dos casos, pero se non hai ningún dispoñible le do primario.
  - `nearest`: Le do membro máis cercano (o que teña a menor latencia de rede nese instante).
- Read concern:
  - `local`: Le o valor máis recente (no primario).
  - `majority`: Le o valor máis reciente que está replicado na maioría dos nodos do replica set.
    Require usar o motor *wiredTiger* e algún outro axuste.

Índice

Sharding

Introdución

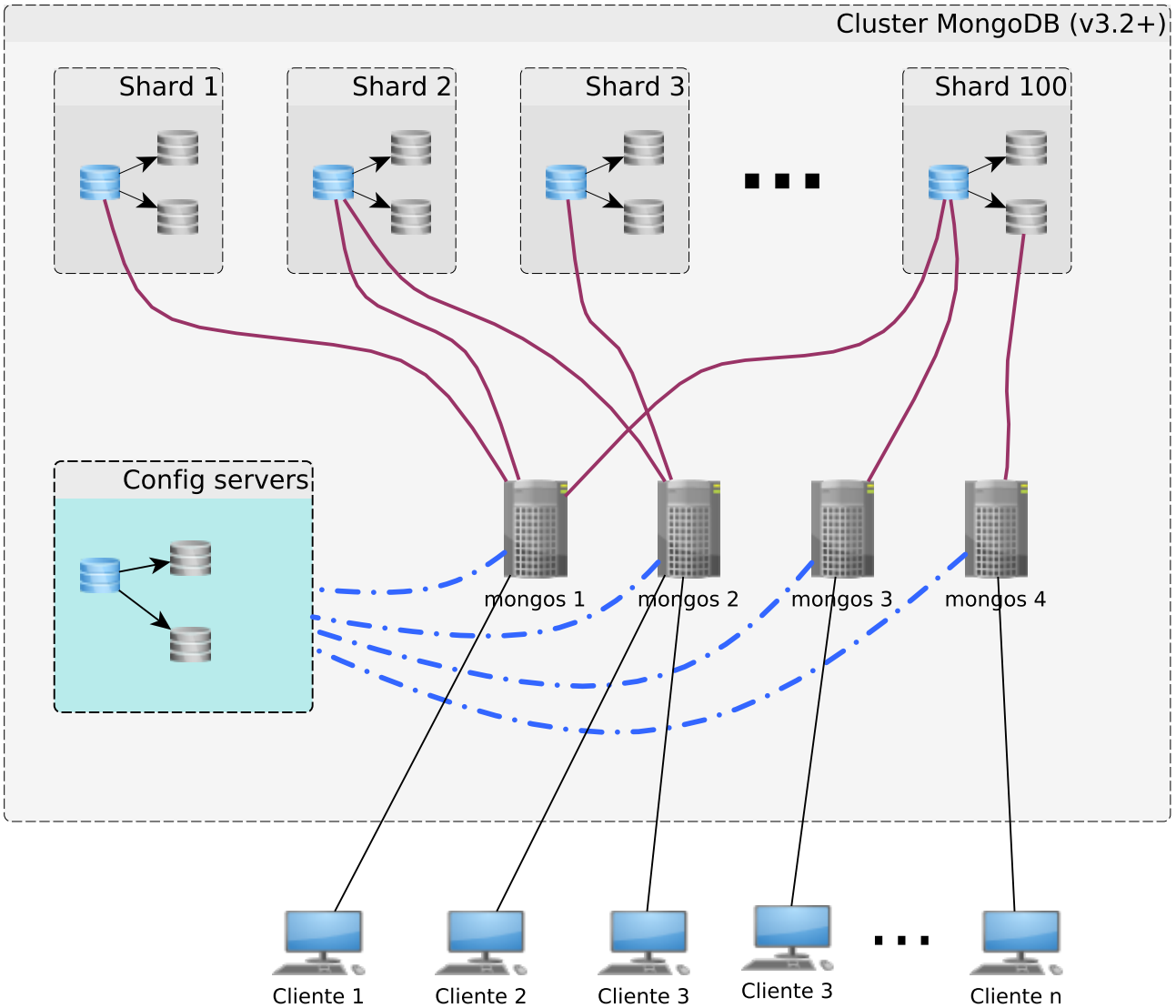
Un dos motivos fundamentais para facer *sharding* ou particionado e distribución dos datos sobre un cluster de servidores é poder escalar o sistema.

Escalado do sistema

- Vertical: Engadir máis capacidade a un servidor. "Compra un servidor máis grande".
  - Hardware caro.
  - Que pasa se o servidor máis grande non é suficiente, ou non é posible compralo?
  - *Single point of failure (SPOF)*: Se falla o (único) servidor, todo o sistema queda inutilizado.
  - Non favorece a natureza distribuída dos datos de certas aplicacións ou organizacións.
  - A latencia da rede de comunicacións pode facelo inviable.
- Horizontal: Engadir máis servidores.
  - Os datos distribúense sobre múltiples servidores (que almacenan *shards* ou particións). A unión de todas as particións fai a base de datos lóxica completa.
  - Os servidores poden ser *commodity hardware*, non son necesarios servidores demasiado caros.
  - Redúcese a cantidade de datos e o número de operacións por nodo.
  - A capacidade do sistema medra de forma (aproximadamente) lineal co número de servidores.

Sharding

Configuración habitual dun cluster



Sharding

Elementos dun cluster MongoDB

- **Shards** ou particións: Almacenan os datos particionados. Poden ser replica sets (o máis habitual) ou instancias `mongod` independentes. O número de shards dependerá de cada caso (tamaño, distribución, potencia dos servidores, ...)

- **Query routers:** Instancias `mongos`.

  Os clientes conéctanse a instancias `mongos` para realizar cada operación, e estes *routers* redirixen as consultas ós shards axeitados para levar a cabo a operación. Opcionalmente poden integrar datos de varios shards para devolvelos ó cliente (ex: se hai que devolver documentos de máis de 1 shard).

  Pode haber tantos query routers como se desexe. Unha opción relativamente común é ter unha instancia mongos instalada no mesmo servidor que a aplicación cliente que se conecta a ela.

- **Config servers:** Almacenan os metadatos de todo o cluster, incluída a información de particionado (asignación dos datos do cluster ós distintos shards).

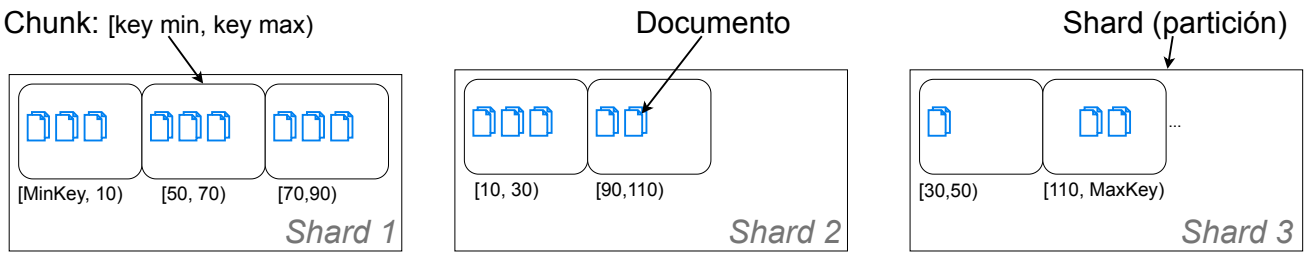
  Versións previas usaban servidores replicados sincronamente, pero desde a V3.2 usa Config Server Replica Set (CSRS): utiliza un Replica Set específico:

  - Pode ter (como calquera RS) ata 50 membros.
  - Sen árbitros nin *delayed members* (replicados asincronamente con retardo).
  - Debe crear índices.
  - Escribe con write concern `"majority"` e le con read concern `"majority"`.
  - Se non pode escoller primario, os metadatos quedan de só lectura (pode haber lectura e escritura de datos, pero non pode haber splits nin migración de chunks).

Sharding

Particionado de coleccións

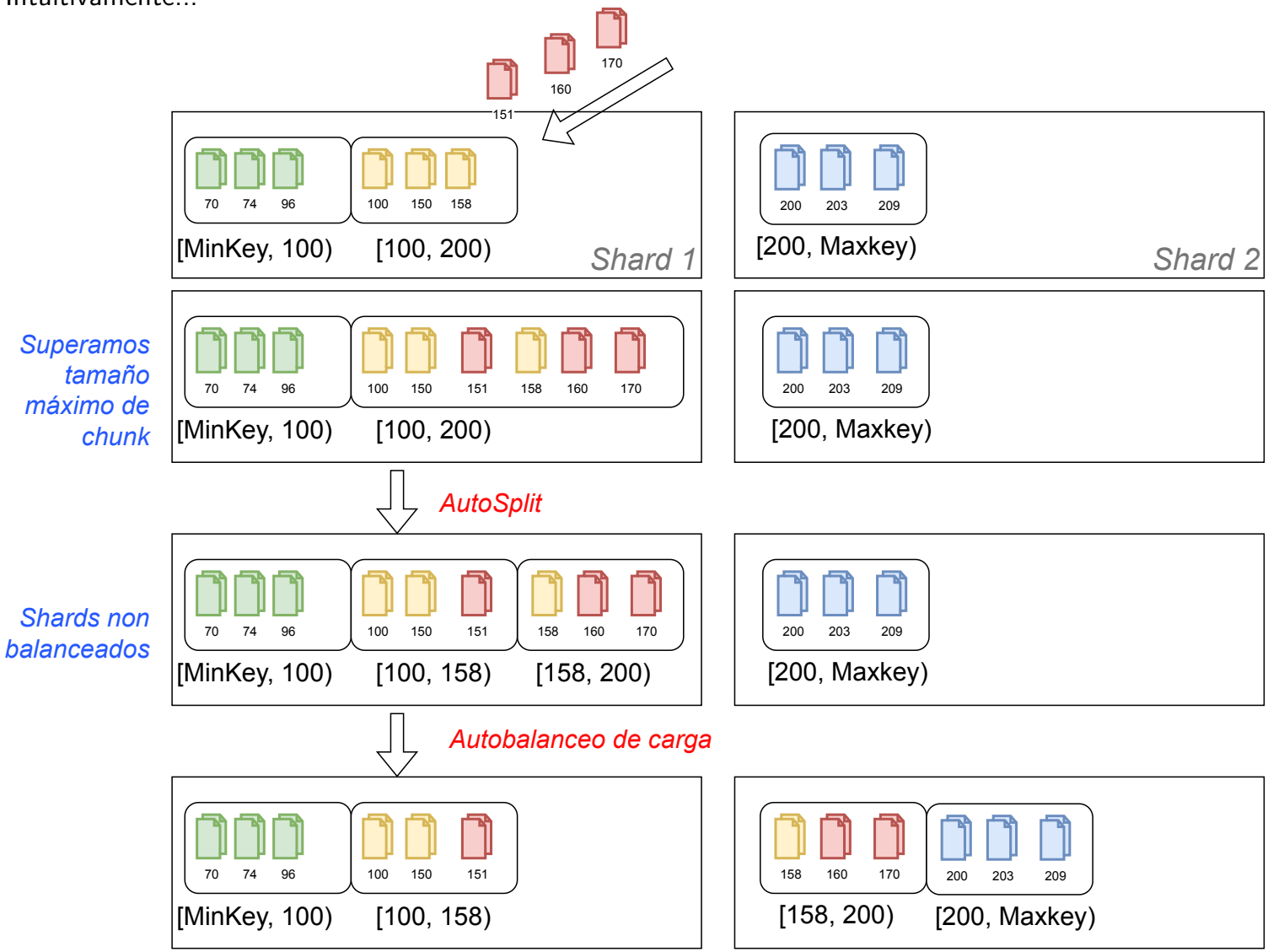
- MongoDB permite particionar os datos a nivel de colección.
- Hai varios tipos de particionado:
  - Autosharding (particionado automático) por rangos, normalmente con balanceo de carga. Pode combinarse con preparticionado manual.
  - Outros: Hash, uso de tags, ...
- Crear un cluster de MongoDB non particiona os datos de xeito automático. Debe:
  1. Habilitarse o particionado a nivel de base de datos: `db.enableSharding("bd1")`
  2. Especificar o particionado dunha colección, indicando a **clave de particionado** ou *shard key*. `sh.shardCollection("bd1.coleccion1", <shard key>)`.
- Cada documento almacénase, evidentemente, nunha única partición, en función do valor da clave.
- Os documentos agrúpanse de forma lóxica en *chunks*.
  - Un chunk é unha colección lóxica que corresponde a un rango de valores [mín, máx) asociados á clave de particionado. Os documentos non teñen que estar fisicamente contiguos.
  - Ten un tamaño predeterminado de 128 MB (o tamaño é configurable).
  - É o elemento principal para o particionado e movemento de datos entre shards.



Sharding

Autosharding

Intuitivamente...



Sharding

Autosharding

Con autosharding, o particionado, a distribución e o balanceo dos datos entre os shards do cluster realízase de xeito automático.

Para o autosharding "normal", por rangos (oposto ó sharding que usa hash):

- A *shard key* (clave de particionado) debe estar indexada. Se o índice non existe, créase automaticamente.
- Os rangos de datos que definen os chunks serán contiguos e ordenados según a shard key.
- O número de chunks dependerá da cantidade e tamaño dos documentos.
- Cando un chunk sobrepasa o tamaño máximo, divídese en 2 (operación `split` automática).
- Tamén é posible crear rangos de xeito manual (`splitAt()`). Pode utilizarse para *preparticionar* a colección.
- Para balancear a carga móvense chunks (*chunk migration*) entre os shards, non documentos individuais.
  - O balanceo ten en conta o número de chunks, non de documentos.
  - O umbral de diferencia entre os chunks dos shards que provoca o movemento de chunks depende do número total de chunks.

Hai que ter especial coidado coa selección da clave de particionado:

- Sobre a inmutabilidade:

  Ata a versión 5 de MongoDB a clave de particionado era inmutable en 2 sentidos: non podía modificarse o seu valor nun documento, nin podía cambiarse a clave de particionado dunha colección (os campos que a forman).

  A partir da versión 5 ambos cambios son posibles:

  - Para modificar o valor, a sentencia `updateOne()/updateMany()` debe incluir a clave de particionado na parte da query (filtrado) da sentencia de modificación.

    (Nota: o campo `_id` de calquera colección sempre é inmutable)

  - Cambiar (os campos que forman) a clave de particionado provoca un *resharding*: unha costosísima operación que redistribúe a colección usando unha nova clave de particionado.

- A clave debe ter unha **cardinalidade alta**, para poder crear un número suficiente de chunks.

- Unha clave **monotonamente crecente** (ídem decrecente) como un `_id` autoxenerado é moitas veces unha **mala opción**, xa que todas as inserciónss se dirixirían ó shard que ten o último chunk, e sempre será ese último chunk o que se divide para balancear.

Sharding

Outros tipos de particionado

É bastante habitual usar autosharding con rangos, pero MongoDB ofrece outras posibilidades:

- **Particionado hash:**

  Asígnase un documento a un shard utilizando unha función hash sobre a shard key (a colección debe ter creado previamente un índice hash).

  Evita os problemas das claves monotomamente crecentes como timestamps ou ObjectIds, creando unha mellor distribución, máis uniforme.

  Ten desventaxas desde o punto de vista da optimización de consultas por rangos de valores (os rangos non estarán ordenados pola clave de particionado, polo que unha consulta non poderá usar só parte dos shards).

- **Particionado por zonas:**

  Utilízase, por exemplo, para colocar datos xeograficamente cerca de onde se usan.

  Créanse unha ou varias *Zona*s asociadas a rangos de datos da clave de particionado.

  Logo, cada shard asóciase tamén a unha ou varias zonas.

  A migración de chunks realízase só entre os shards da zona á que pertencen os datos.
  Exemplo, usando unha dirección IP como shard key:

```
Zona "UDC": 193.144.51.0--255
Zona "USC": 193.144.75.0--255
Zona "UVigo": 193.146.32.0--255
```

```
shard1: zona "UDC"
shard2: zona "UDC", "USC"
shard3: zona "USC", "UVigo"
shard4: zona "UVigo"
```

  Os datos da zona `UDC` poderían moverse entre os shards 1 e 2. Os da zona `USC` entre os shards 2 e 3, e `UVigo` entre os shards 3 e 4.

Sharding

Consideracións sobre o particionado de datos

- Normalmente usaremos sharding con MongoDB para:
  - Incrementar a RAM dispoñible.
  - Incrementar o espazo en disco dispoñible.
  - Reducir a carga dos servidores.
  - Obter maior rendemento que cunha soa instancia de `mongod`.
- Poden engadirse ou eliminarse shards dun cluster operativo (non é necesario paralo).
- Un cluster con moi poucos shards (2, por exemplo) pode ser menos eficiente que un único nodo, pola sobrecarga da xestión do particionado e movemento de datos entre os nodos.
- Recorda que só crear o cluster non particiona nada. Hai que habilitar o particionado a nivel de base de datos, e particionar cada colección que se desexe usando unha clave de particionado concreta.
- Aínda que con novas versións se melloran aspectos, o sharding pode ter limitacións.

  Unha limitación actual (versión 8): a unicidade verifícase a nivel de shard, non de colección.

  Isto implica que unha colección particionada só garante a unicidade do `_id` a nivel de shard (a nivel colección completa: control desde as aplicacións).

Índice

Comparación cos modelos clásicos

- Modelo E-R e relacional: *data-driven*, dirixido polos datos.
  - Importan os datos por si mesmos.
  - Modélase a base de datos sen ter en conta as aplicacións que a van usar (aínda que logo poidan realizarse optimizacións).
  - A linguaxe SQL permite realizar consultas complexas vinculando as táboas necesarias.
  - Os xestores de bases de datos inclúen potentes optimizadores de consultas.
- MongoDB (e outros NoSQL): *application-driven*.
  - Parten do concepto "unha base de datos, unha aplicación".
  - Importan as "preguntas", o uso dos datos desde as aplicacións.
  - Modélase a base de datos con agregados baseados nesas preguntas.
  - Polo tanto, é fundamental conocer os **patróns de acceso** ós datos.

Modelado de datos

Introdución

Características de MongoDB que inflúen no modelado

- Soporte de documentos incrustados (admite ata 100 niveis de aniñamento de subdocumentos/arrays).
- Hai un límite *hardcoded* de 16 MB por documento.
  (Aínda que existe `gridfs` para tamaños maiores, é habitualmente usado para almacenar ficheiros, especialmente multimedia: audios, vídeos etc.)
- Atomicidade de operacións a nivel de documento.
- Soporte de transaccións (incluíndo replica sets e sharded clusters):
  - Non é obrigatorio utilizalas.
  - Inflúe negativamente no rendemento.
- Non hai "claves foráneas".
  Se usamos referencias, a integridade debe garantirse a través das aplicacións.

Alternativas de modelado: *embedding & linking/referencing*

- Embedding (incrustar): incluímos un documento (ou array) dentro doutro.
- Linking/Referencing (enlazar/referenciar): incluímos nun documento unha referencia a outro documento (normalmente ó `_id`).

Modelado de datos

Incrustar documentos (*Embedding*)

- Os datos están preagregados, o que permite unha alta eficiencia de lectura.
- A escritura dos documentos internos pode ser ineficiente (máis lento que enlazando).
  Ler post, agregar comentario, gardar post completo vs. Insertar un comentario nunha colección de comentarios.
- A integridade de datos debe ser xestionada polo usuario ou aplicación.
  Ver elementos resaltados no exemplo.
- Transaccionalidade implícita: hai atomicidade a nivel de documento.
- Non se pode sobrepasar o límite de 16MB por documento.

```
POSTS:
{
  _id: 1,
  titulo: 'Discusión sobre modelado',
  autor: 'Miguel Rodríguez Penabad',
  tags: ['modelado', 'NoSQL'],
  comentarios: [
           {usuario: 'anon1', email: 'anon1@server.com', texto: 'Non hai modelado formal. Está todo mal' },
           {usuario: 'anon2', texto: 'É fantástico. Da moita flexibilidade'}
           ]
},
{
  _id: 2,
  titulo: 'Segunda discusión',
  autor: 'Miguel R. Penabad',
  email: 'miguel.penabad@udc.es',
  tags: ['nosql'],
  comentarios: [ ...     ]
}
```

Modelado de datos

Enlazar/Referenciar documentos (*Linking/Referencing*)

- Un documento referencia outro a través do seu `_id` (en teoría, calquera campo que asegure unicidade). (Recorda: en sharded clusters, a unicidade garántese a nivel de shard)
- É flexible para representar/modelar información.
- Permite "normalizar" as coleccióńs ó estilo relacional.
- A integridade de datos é xestionada polo usuario ou aplicación (non hai "claves foráneas").
- Consultas máis complexas e lentas: os "joins" deben facerse utilizando o framework de agregación (`$lookup`) ou a nivel de aplicación cando os datos están separados en varios documentos/coleccióńs.
- A escritura é normalmente máis eficiente que modificar subdocumentos incrustados.
- Consideracións sobre aspectos transaccionais.

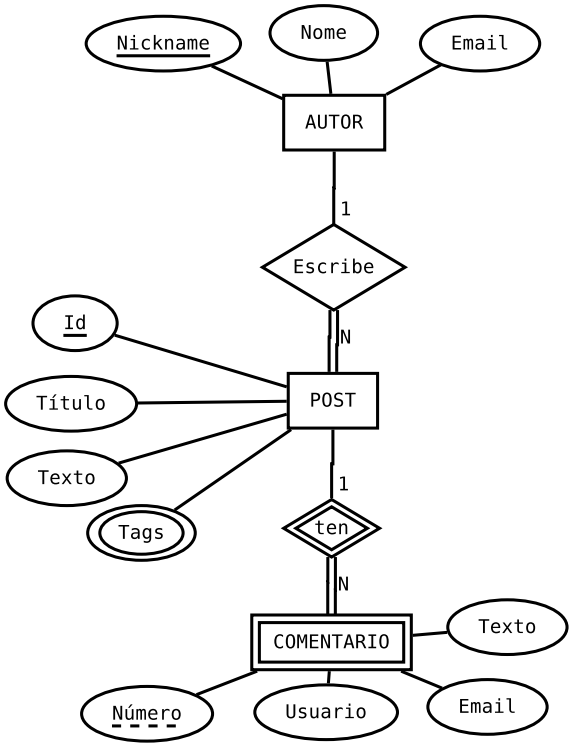
```
POSTS:
{
  _id: 1,
  titulo: 'Discusión sobre modelado',
  autor: 'penabad',
  ...
},
{
  _id: 2,
  titulo: 'Segunda discusión',
  autor: 'penabad',
  ...
}
```

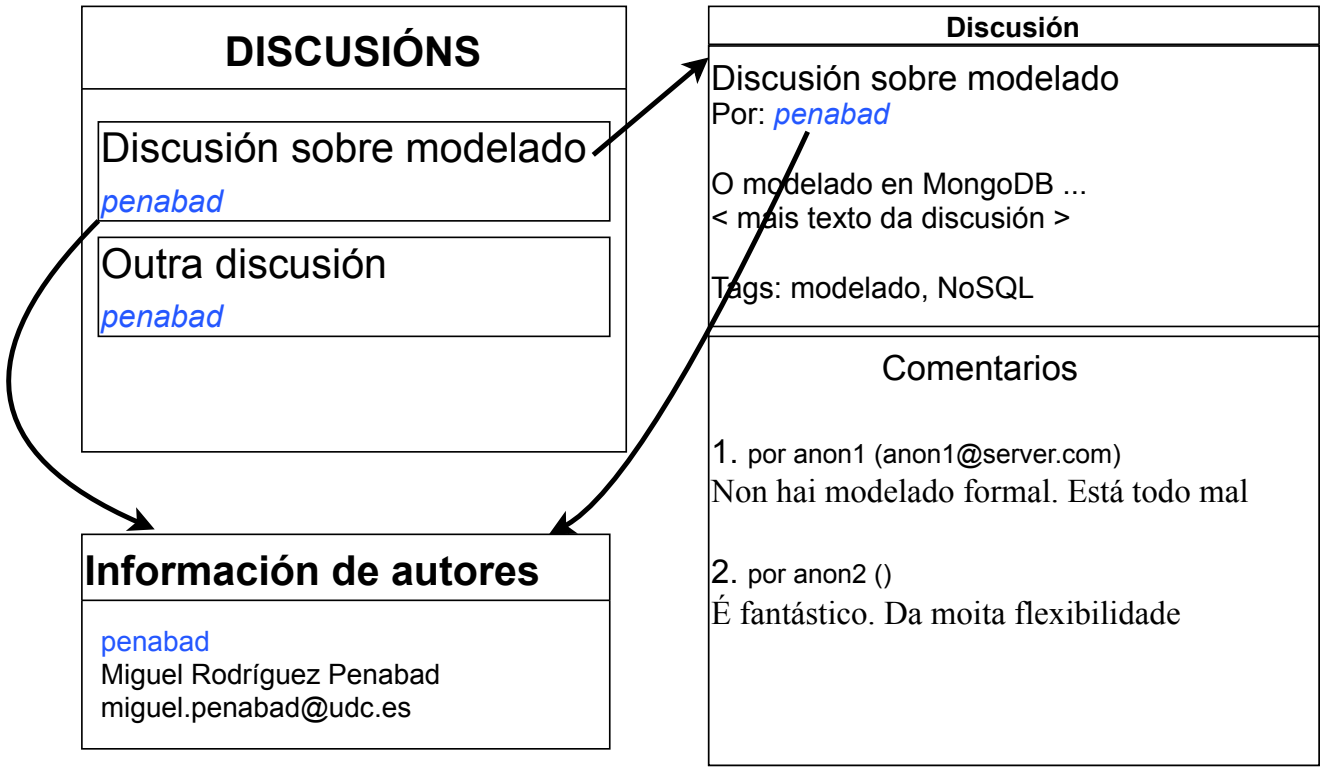
```
AUTORES:
{
  _id: 'penabad',
  nome: 'Miguel Rodríguez Penabad',
  email: 'miguel.penabad@udc.es'
}
```

Modelado de datos

Alternativas de modelado: Exemplo de toma de decisións

Supoñamos que temos o modelo conceptual e tamén os patróns de acceso (aquí visibles moi informalmente co esbozo dunha web).





Modelado de datos

Alternativas de modelado: Exemplo de toma de decisións

- A lista de posts inclúe só o título e o nome de usuario do autor, que quedarán na colección de `posts`.
- Na páxina que amosa un post vese o título, o texto, os tags e comentarios. Aparentemente non se accede ós comentarios nin ós tags de forma independente do post ó que pertencen.

  Isto implica, por exemplo, que non se vexa unha "lista de tags" (nin se verifica a uniformidade na forma de escribilos).

  Sería complexo cambiar un tag concreto en varios (ou todos os) posts.

  Beneficia a eficiencia de lectura ó estar todos os datos necesarios para mostrar o post no mesmo documento (unha única lectura).

  Non se espera superar o límite de 16MB por documento, polo que incluír a información completa dos posts, incrustando tags e comentarios, parece boa idea.
- Os datos persoais do autor do post non son accedidos nin na lista nin no detalle do post. Só aparece o nome de usuario como un link para acceder a esa información noutra páxina.

  Polo tanto, podemos incluir o `autor` na colección `posts` como unha referencia. Cando se vaia á páxina de detalle do autor, mediante unha soa lectura, accédese ós datos persoais.

  Nótese que na páxina dos autores non hai ningunha referencia (número ou listaxe) ós seus posts, polo que non se incluirá nada máis na colección de autores.

Modelado de datos

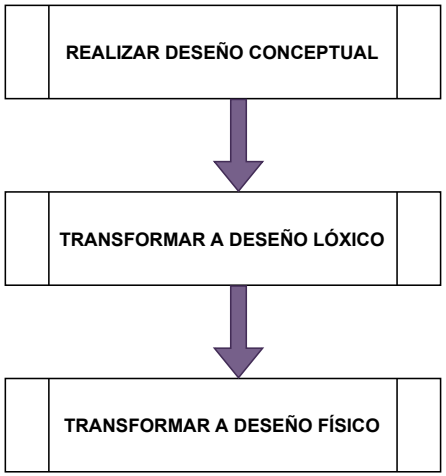
Alternativas de modelado: Exemplo de toma de decisións

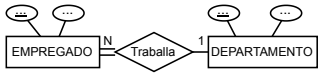
Usando o modelo e os patróns de acceso do exemplo, a seguinte parece unha modelización axeitada: Nótese que na colección de autores, o campo que no E-R aparece como `Nickname` pasa a ser agora o campo `_id`.

```
POSTS:
  {
    _id: 1,
    titulo: 'Discusión sobre modelado',
    texto: ...,
    autor: 'penabad',
    tags: ['modelado', 'NoSQL'],
    comentarios: [
             {usuario: 'anon1', email: 'anon1@server.com',
             texto: 'Non hai modelado formal. Está todo mal' },
             {usuario: 'anon2', texto: 'É fantástico. Da moita flexibilidade'}
             ]
  }

AUTORES:
  {
    _id: 'penabad',
    nome: 'Miguel Rodríguez Penabad',
    email: 'miguel.penabad@udc.es'
  }
```

Modelado de datos

Metodoloxía clásica de deseño de BDs





DEPT(DEPTNO, DNAME, ...)

EMP (EMPNO, ..., DEPTNO)

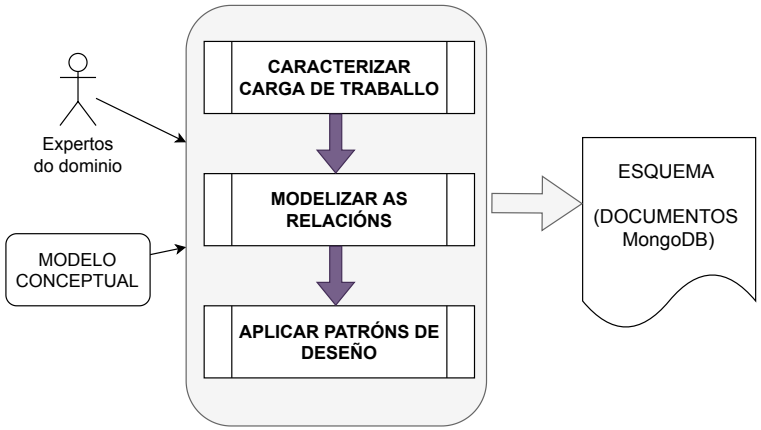
```
CREATE TABLE DEPT(
  DEPTNO NUMERIC(3)
CONSTRAINT PK_DEPT PRIMARY KEY....)

CREATE TABLE EMP (
... )
```

1. Deseño conceptual. Usaremos un modelo gráfico, no noso caso diagramas EER.
   É independente do modelo lóxico e da implementación física.
   En moitos casos podemos ter o modelo de datos e non saber se imos usar MongoDB ou outro sistema (relacional, NoSQL, ...)
2. Deseño lóxico: transformación do deseño conceptual a un deseño lóxico.
   Frecuentemente úsase o Modelo Relacional.
3. Deseño físico: partindo do modelo lóxico, realizar a implementación física, especificando onde almacenar os datos, tamaños, índices, ...
   En BDs relacionais utilízase SQL.

Modelado de datos

Metodoloxía de deseño para MongoDB

Principio básico: *"Data that is accessed together should be stored together"*



1. Caracterizar carga de traballo
   - Identificar consultas
   - Cualificar consultas (R/W)
   - Cuantificar consultas (frecuencia, latencia máxima, ...)
2. Modelizar as relacións
   - Relacións 1:1, 1:N, N:N.
   - Decisión: incrustar ou enlazar
3. Aplicar patróns de deseño
   - Polimorfismo, referencia extendida, versionado, ...
   - Hai 12 patróns de deseño recomendados.
   - Existen tamén *antipatróns* que debemos evitar.

Notas:

- En MongoDB non hai unha separación demasiado clara entre o modelo lóxico e o físico (documentos).
- Os índices son un elemento esencial para a eficiencia das consultas, pero non os trataremos.
- Dada a filosofía "unha BD, unha aplicación", debemos ter coñecemento sobre o que realiza a aplicación. Poden delegarse na aplicación algúns aspectos (ex: comprobación de claves foráneas que os SXBD relacionais fan automaticamente).

Modelado de datos

Caracterización da carga de traballo

Punto de partida: Temos

- Coñecemento sobre o dominio. Idealmente, teremos algún tipo de modelo conceptual.
- Expertos do dominio que nos describan posibles escenarios (reais ou estimados) de uso da aplicación/BD (ex: a partir de documentos de requisitos).
- (Se existe un sistema actual) Logs ou estatísticas de uso.

O *tamaño* dos datos é importante (tamaño de cada documento, tamaño das coleccións).

1. Identificar as consultas

- Debemos identificar as consultas máis frecuentes (idealmente poden ser todas as operacións CRUD, pero as máis frecuentes son máis relevantes).

Operación	Descrición
Ver un post	Mostra o título e o contido dun post, incluíndo os tags e comentarios se os houbese.
Engadir un post	Engade un post (para o usuario activo). Pode incluir tags, pero non comentarios.
Buscar información e posts dun usuario	Lista toda a información dun usuario, xunto co listado dos títulos dos seus posts.

2. Cualificar as consultas

- Debemos indicar o tipo de consulta (R/W: lectura ou escritura)
- Debemos especificar a información afectada
- Outra información: escritura crítica ou non, admisión ou non de datos obsoletos, ...

Operación	Desc.	Tipo	Información
Ver un post	...	R	Título, texto, nick do autor, tags, comentarios
Engadir un post	...	W	Título, texto, nick do autor[, tags]
Buscar información e posts dun usuario	...	R	Nick, nome e email do autor, título de post

3. Cuantificar as consultas

- Parámetros temporais: frecuencia, latencia máxima, ...
- Tamaño da resposta.

Operación	Desc.	Tipo	Información	Frecuencia	Tamaño
Ver un post	...	R	Título, texto, nick do autor, tags, comentarios	25/s	<100KB
Engadir un post	...	W	Título, texto, nick do autor[, tags]	1/min	<100KB
Buscar información e posts dun usuario	...	R	Nick, nome e email do autor, título de post	50/min	10KB

Información do tamaño do dataset para este exemplo:

- 2.000.000 posts
- 30.000 usuarios (promedio de posts por usuario: 67)
- [Tamaño total e expectativas de crecemento]

Resume

- Como entrada para esta fase dispoñemos de:
  - Coñecemento sobre os datos (modelo conceptual).
  - Coñecemento sobre o dominio (expertos no dominio, escenarios reais ou estimados, quizás estatísticas).
  - Tamaño (real ou estimado) dos datos.
- Como saída obtemos:
  - Unha definición da carga de traballo, coas consultas máis frecuentes descritas, cualificadas e cuantificadas.
  - Opcionalmente poderíamos ter *mockups* de pantallas/páxinas web que mostraran prototipos da aplicación.

Nesta materia faremos unha estimación máis xeral da carga de traballo, incluíndo as consultas (patróns de acceso a datos) máis frecuentes, ordenando por frecuencia ou importancia as consultas, ou con porcentaxes de uso de cada unha.

Modelado de datos

Modelado das relacións

Consideracións:

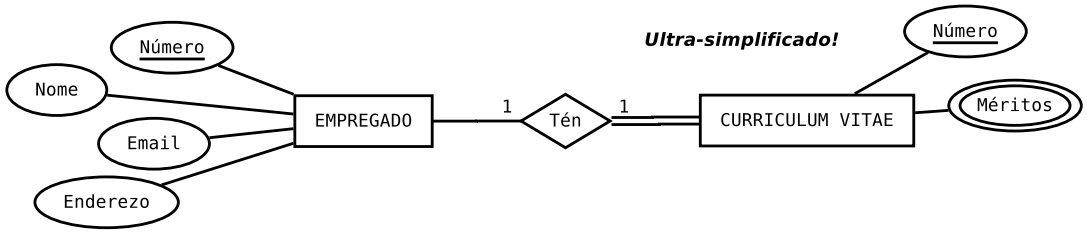
- Para modelar as relacións, partiremos de diagramas Entidade-Relación.
- Só consideraremos relacións binarias (as de grado > 2 poderían transformarse).
- A participación pode ser importante na toma de decisións.
- Considerando a cardinalidade máxima, temos relacións 1:1, 1:N e N:N.
- Nos diagramas sempre aparecerá unha *N* co significado de "moitos", pero os valores mínimo, medio e máximo poden ser importantes para a toma de decisións.
  En algúns diagramas de MongoDB utilizan a notación `[min, med, max]`. Por exemplo, un usuario pode crear `[0, 67,1000]` posts.
- A caracterización da carga de traballo da fase anterior ten un peso determinante na elección de como modelar a relación.

Nesta fase obtense a decisión de modelar a relación *incrustando* un documento (ou array) nunha colección, ou *enlazando/referenciando* un documento dunha colección desde outro documento (normalmente doutra colección).

Na documentación oficial de MongoDB dan habitualmente preferencia a *incrustar* en ver de *enlazar*.

Modelado de datos

Modelado das relacións: Relacións 1:1



Podemos incrustar?

Podemos incrustar CV en Empregado?

- Se non superamos os 16MB.
- Se o patrón de acceso ós datos do empregado e o CV é similar.
- Aumenta a integridade (operacións atómicas a nivel de documento).

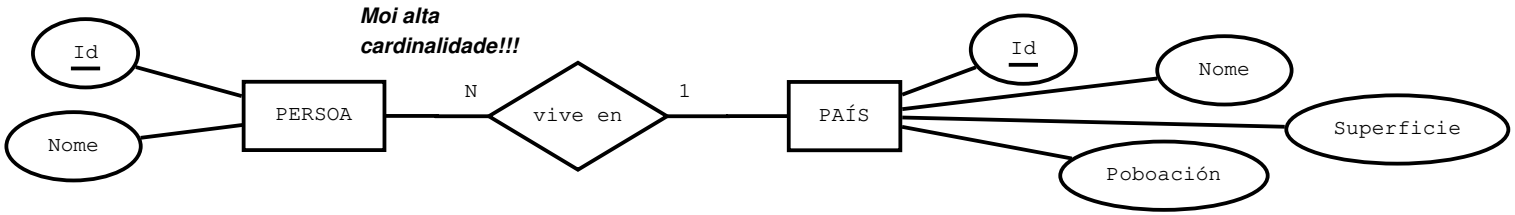
Podemos incrustar Empregado en CV?

- **Non** se pode incrustar Empregado en CV, debido á participación parcial de Empregado.

Podemos enlazar?

Poderíamos enlazar: Empregado $\rightarrow$ CV, CV $\rightarrow$ Empregado ou incluso ambas:

- Sería obrigatorio se superamos os 16 MB.
- Sería aconsellable se o patrón de acceso (frecuencia, tipo de acceso) é diferente.
  Ex: Lemos moitas veces datos do Empregado, poucas o CV, modificamos CV pero non Empregado, ...



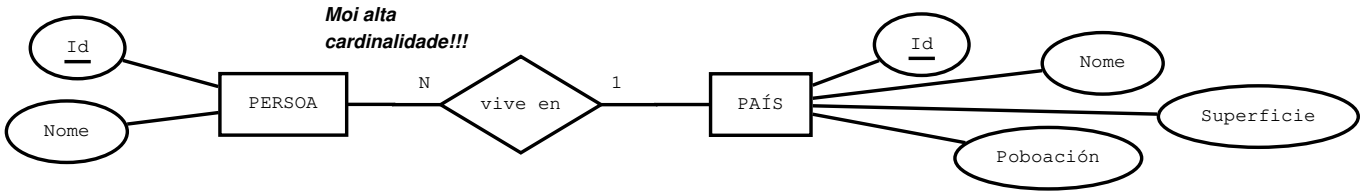
Podemos incrustar?

Incrustar a parte 1 en todos os documentos da parte N (incrustar País en cada Persoa)?

- Se hai participación parcial, non podemos (como neste exemplo).
- Se hai participación total (todo país ten persoas): deben terse en conta dous aspectos, xa que hai duplicidade de datos:
  - Espazo ocupado.
  - Xestión de posibles inconsistencias se hai que modificar datos.
    Neste exemplo, a poboación do país cambia (non sabemos a frecuancia coa que se actualiza este dato). Se modifico a poboación debo facelo en todas as copias (consideracións transaccionais).
    Se omitimos a poboación, o resto é información basicamente estática, co que este problema case desaparece.

  Se garantimos que xestionaremos a duplicidade de datos correctamente, e se unha consulta importante é "datos de persoas incluíndo o nome ou superficie do país", sería unha boa opción.

Máis adiante falaremos do patrón *extended reference*, relacionado con este tipo de situacións.

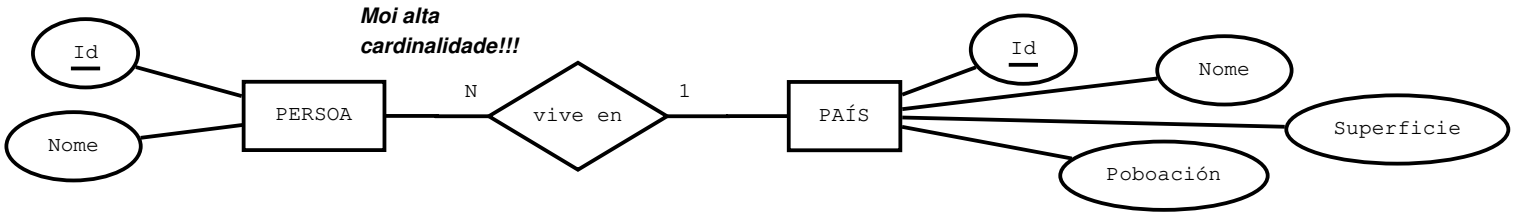


Podemos incrustar?

Incrustar a parte N como un array no documento da parte 1 (incrustar un array de Persoas en País)?

- Só pode facerse se hai participación total (se hai participación parcial, como neste exemplo, persoas sen país asociado, non podemos).
- Consideracións sobre a cardinalidade da parte N:
  - Incrustar un array de N documentos na parte 1 pode ser boa idea se a cardinalidade é baixa: a relación é "un a poucos", non "un a moitos". Ex: comentarios dun post (poucos)
  - Normalmente será mala idea se a cardinalidade é moi alta. Ex: persoas que viven nun país (moitas).
  - Especificar a cardinalidade con `[min, med, max]` pode axudar á toma de decisións.
  - Está relacionado con 2 *antipatróns*: *bloated document* e *unbounded array*.
- Se a parte N ten relacións propias de tipo 1:N ou N:N (nunha teórica transformación a relacional a táboa sería destino dunha clave foránea) e incrustamos os documentos, despois é posible que non sexamos capaces de referencialos desde outras coleccións.

  Ex: Sobre a BD de posts: Se creo a colección `autores` incrustando `posts` e logo crease a colección `comentarios`, desde un comentario non podería referenciar de forma única un post.



Podemos enlazar?

Enlazar a parte 1 desde cada documento da parte N

- É habitual facelo cando a cardinalidade é alta: Cada Persoa referencia o seu País.
- Recoméndase o uso de campos `_id` representativos na parte 1.
  Ex: Poñer o nome do país (sabemos que é único) como `_id` na colección de Países.

Enlazar a parte N desde a parte 1 (array de referencias)

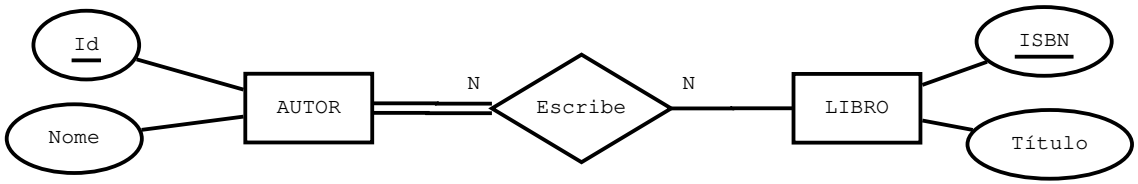
- Se a cardinalidade é baixa, pode enlazarse desde a parte 1 mediante un array de referencias á parte N.
- Neste exemplo (ter un array de referencias a Persoa desde cada País) non parece recomendable.
- No exemplo da BD de posts, sería factible que un Usuario tivese un array de referencias ós seus Post, que estarían noutra colección.

Recordemos que a carga de traballo é determinante.

Por exemplo, a consulta "Buscar información e posts dun usuario" desaconsella modelar a relación 1:N poñendo en Usuario un array de referencias ós seus Posts.

Modelado de datos

Modelado das relacións: Relacións N:N

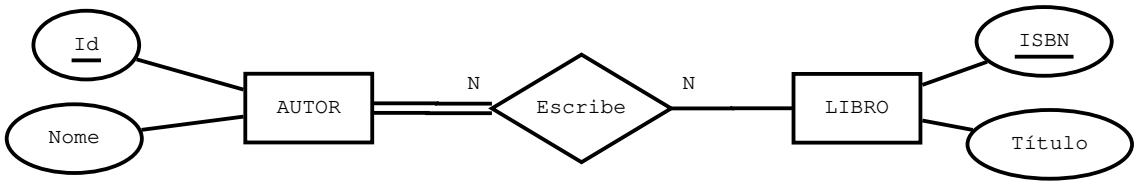


Podemos incrustar?

- De novo, hai datos duplicados (probablemente en maior grado que nas relacións 1:N)
- Sería interesante incrustar se:
  - Podemos permitirnos gastar o espazo consumido polos datos duplicados
  - Podemos garantir (por aplicación) a consistencia dos datos.
    Se os datos son case inmutables (como no exemplo) é unha ventaxa.
- Temos que prestar atención á participación (non podemos incrustar Libro en Autor)
- Temos que prestar atención á cardinalidade (se é moi alta, probablemente non sexa boa idea incrustar)
- Considerar os patróns de acceso.
  Se buscamos normalmente por libro, considerando que non vai ter un número moi elevado de autores, e tendo en conta que a información dos autores non cambia habitualmente, incrustar un array de Autores en cada Libro pode ser boa idea.

Modelado de datos

Modelado das relacións: Relacións N:N



Podemos enlazar?

- Se as multiplicidades son baixas (ex: Libros-Autores), poden usarse arrays de referencias nas 2 coleccións (ou incluso só nunha delas).
  - En Libros, array de referencias a Autores; en Autores, arrays de referencias a Libros.
  - Considerar os patróns de acceso.
    Se a consulta habitual é "autores dun libro" e apenas se busca "libros dun autor":
    só array de referencias a Autores na colección de Libros.
- Se a cardinalidade é *moi* grande, esta opción pode ser mala idea.
  De novo, a especificación da cardinalidade como `[min, med, max]` pode axudar.
- Alternativa (similar ó modelo relacional): crear unha colección para a relación (`escribe`) con referencias a `libros` e `autores`.
- Para todos os casos: debemos recordar que non hai "claves foráneas" xestionadas por MongoDB.
  A coherencia dos datos queda baixo a responsabilidade da aplicación.

De forma similar ós patróns de deseño en desenvolvemento de software, existen patróns de deseño para crear o esquema dunha base de datos en MongoDB.

Un *patrón de deseño* de MongoDB é unha solución reusable para moitos casos de uso que se atopan cando deseñamos unha aplicación que garda os seus datos en MongoDB.

En total existen 12 patróns de deseño ben coñecidos [2, 3], pero imos centrarnos nos seguintes:

- Polimorfimo (*polymorphism: inheritance, single collection*)
- Campo calculado (*computed*)
- Referencia extendida (*extended reference*)

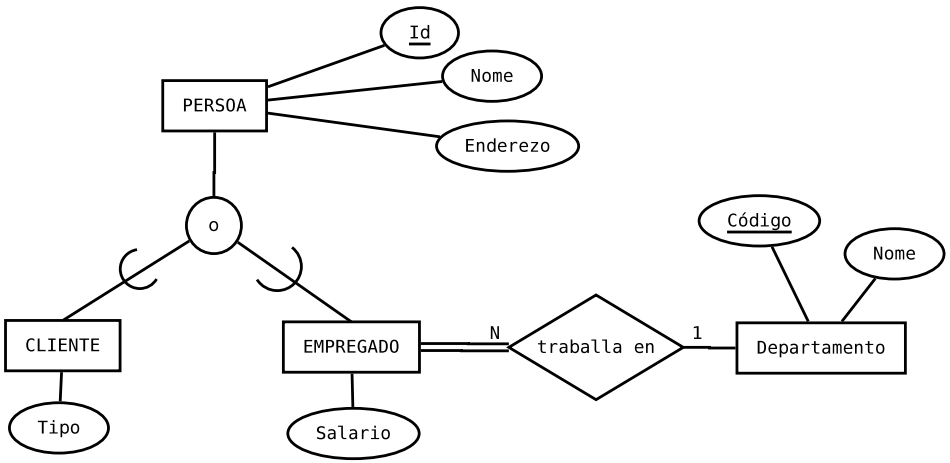
Modelado de datos

Patróns de deseño: polimorfismo

Está baseado no feito de que MongodDB é *schemaless*, os documentos dunha colección poden ter un esquema diferente.

Sirve para transformar as xerarquías e retículas de xeneralización/especialización:

- Dado que os documentos non teñen esquema fixo, podemos usar unha única colección.
  - Os datos específicos da subclase poden ir nun subdocumento.
  - Uso dun atributo para especificar a clase á que pertence (un array para especializacións solapadas).
- Non se ten que usar sempre:
  Se as subclases teñen patróns de acceso diferentes, e a xerarquía é total e disxunta, pode ser beneficioso crear unha colección para cada subclase (control de `_id` único por aplicación).




```
PERSOAS:
 {
  _id:1,
  nome: "Filemón Pi",
  enderezo: "Rue do percebe, 13",
  tipo: ['cliente', 'empregado'],
  cliente: {tipo: 'VIP'},
  empregado: {salario: 2000, departamento: 'I+D'}
 }
DEPARTAMENTOS:
 {
   _id: 'I+D',
   nome: 'Investigación e Desenvolvemento'
 }
```

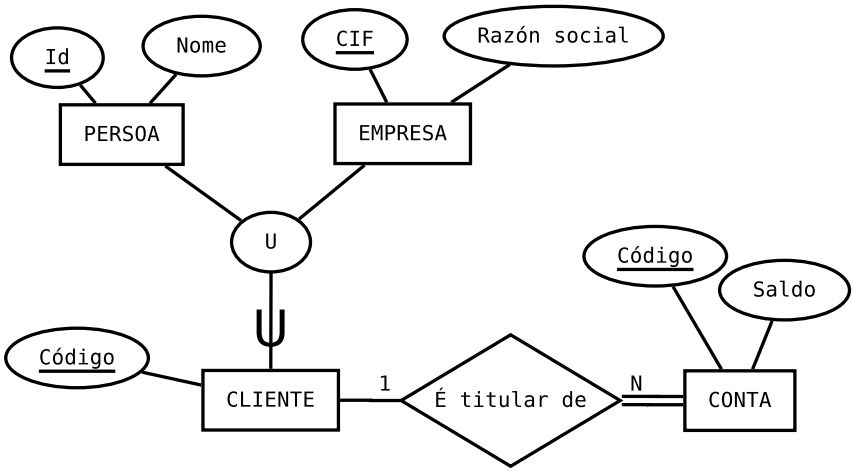
Modelado de datos

Patróns de deseño: polimorfismo

Relacionado co anterior: como transformamos unha **Categoría** do modelo EER?

Ó non existir a integridade referencial, as categorías non teñen moito sentido en MongoDB.

A aplicación é a encargada de controlar a coherencia dos datos, polo que podería comprobar directamente nas clases involucradas, sen crear a categoría.



Se a categoría tivese atributos (obviando ese `código` subrogado), poderían incluirse en `persoas` e `empresas` (só para aqueles documentos que representen `clientes`).

No caso do `código` identificador, asegurándose de que é único, podería uarse como referencia desde `contas`.

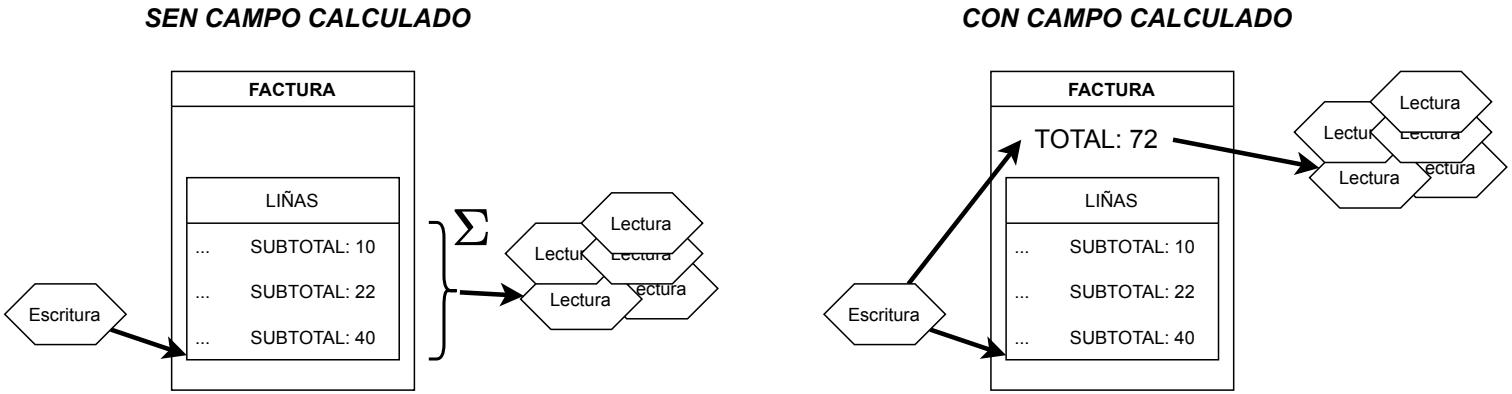
```
PERSOAS:
 { _id:1,
 nome: 'Mortadelo'
 }
EMPRESAS:
 {
  _id: 'B1534889K',
  razonsocial: 'Explosivos ACME'
 }
CONTAS:
 {
  _id: '1234 5678 9012 3456',
  saldo: 0.97,
  persoa: 1
 }
 {
   _id: '9876 5432 1098 7654',
   saldo: 28256321.03,
   empresa: 'B1534889K'
 }
```

Modelado de datos

Patróns de deseño: campo calculado (*computed*)

Utilízase habitualmente cando hai moitas máis lecturas que escrituras, que consultan un valor agregado (suma, media, ...). Exemplos:

- Total dunha factura (suma do subtotal das súas liñas)
- Número de comentarios dun post (contar os comentarios)
- Valoración media dun artigo (media das valoracións)



Aspectos a ter en conta:

- Implica escrituras adicionais, pero acelera moito as lecturas.
- Pode escribirse o campo calculado sincronamente, ou periodicamente (por aplicación cada X insercións, procesos batch periódicos)
- A segunda opción implicaría datos obsoletos. Poden ser admisibles ou non.
  Non tería sentido para as facturas, pero non é demasiado importante para o número de comentarios ou valoración media.

(Nota: Esta técnica é tamén habitual en SQL, con campos almacenados e triggers, ou vistas materializadas.)

Modelado de datos

Patróns de deseño: referencia extendida (*extended reference*)

Volvendo ó exemplo dos posts, centrándonos na operación "Ver un post", poderíamos:

- **Incrustar** os datos de Autor como un subdocumento de Post.
- **Enlazar (referenciar)** Autor desde Post, vinculando o seu `_id` (que debería ser significativo).

Pero, que pasa se na carga de traballo atopamos isto?

Operación	Desc.	Tipo	Información
Ver un post	...	R	Título, texto, nick **e nome** do autor, tags, comentarios

O patrón Referencia Extendida é algo intermedio: consiste en referenciar un documento, pero *extendendo* esa referencia incluíndo outros campos dese documento que son requeridos por unha consulta habitual.

```
POSTS:
{
  _id: 1,
  titulo: 'Discusión sobre modelado',
  autor: {
     _id: 'penabad',
     nome: 'Miguel Rodríguez Penabad'
     },
  ...
}
```

```
AUTORES:
{
  _id: 'penabad',
  nome: 'Miguel Rodríguez Penabad',
  email: 'miguel.penabad@udc.es'
}
```

Ten as ventaxas de incrustar documentos (evitamos `$lookup` para buscar información adicional).

Ten os mesmos inconvenientes, derivados da duplicidade: espazo usado e xestión da coherencia das copias.

Modelado de datos

Antipatróns

Hai algúns erros, bastante comúns, que poden aparecer no deseño dos esquemas de MongoDB.

Frecuentemente estes erros só se detectan cando se escala a BD a un tamaño grande.

A estes erros a comunidade de MongoDB chámalles *antipatróns*.

Algúns dos máis comúns son os seguintes (unha colección máis completa pode atoparse en [4]):

- *Massive/Unbounded arrays*: Arrays, incluídos nun documento, que poden ser demasiado grandes.
- *Bloated documents*: Documentos inchados, sobrecargados, excesivamente grandes.

Modelado de datos

Antipatróns: Massive/Unbounded arrays

- O problema aparece cando un documento ten un array con demasiados elementos.
- Poderíamos exceder o límite de 16MB por documento.
- Os índices funcionan peor con arrays grandes.
- As modificacións que afectan ó array implican ler o documento, modificar o array e gardar o documento completo. É ineficiente.
- Exemplo: Incrustar Persoas en Países (no exemplo da relación 1:N).
  Solución: Incrustar a información do País en cada Persoa.

```
// Países
{
 _id: 1,
 nome: 'España',
 superficie: 505990,
 poboacion: 48692804,
 persoas: [
    {_id:1, nome: 'Mortadelo'},
    {_id:2, nome: 'Filemón'},
 ...
 ]
}
```

```
// Persoas
{
 _id: 1,  nome: 'Mortadelo',
 pais: { _id: 1,
   nome: 'España',
   superficie: 505990,
   poboacion: 48692804
  }
}
{
 _id: 2, nome: 'Filemón',
 pais: { _id: 1,
   nome: 'España',
   superficie: 505990,
   poboacion: 48692804
  }
}
```

Modelado de datos

Antipatróns: Bloated documents

"Data that is accessed together should be stored together" but not necessarily data that is related to eachother should be stored together. [4]

- Ocorre cando os documentos dunha colección son moi grandes (aínda que non cheguen ós 16MB).
- Xenera problemas de ineficiencia porque caben menos documentos na cache de MongoDB (onde están os índices e os documentos máis frecuentemente usados).
- Pode xenerar problemas de *split* e balanceo en clusters particionados (*jumbo chunks*).
- Pode estar relacionado co antipatrón de arrays masivos.
- Pode evitarse separando a información que non se usa xunta, separando en documentos e referenciando.
  - En relacións 1:1 (ex: Empregado e CV, con distintos patróns de acceso)
  - Incluso na mesma entidade do modelo conceptual, non todos os atributos son accedidos á vez.
    Ex: Na colección de Contactos, podemos mover para unha colección `detalles_contactos` os datos pouco frecuentemente usados (data de nacemento, enderezo) e referencialos co mesmo `_id`.
- Existe un patrón de deseño que podería aplicarse: *Subset*.
  Ex: Na colección de Posts, gardar no documento só os últimos 5 comentarios, e mover os demáis a documentos separados, nunha colección diferente (que referencie ó Post).

Índice

Anexo: Validación de documentos

Ó longo deste documento indicouse varias veces que certos controis (como as referencias, o equivalente a claves foráneas do modelo relacional) deben ser controladas pola aplicación.

Tamén se indicou claramente que MongoDB non ten esquema fixo: cada documento dunha colección podería ter un esquema diferente.

Para poder exercer certo control dentro dunha colección, MongoDB permite validar os seus documentos:

- Pode validarse o esquema dos documentos que se inserten ou modifiquen.
- Configuración da validación:
  - Creando a colección: `db.createCollection("<col>", { <validación> })`.
  - Modificando a colección: `db.runCommand( {"collMod" : "<col>" , <validación> })`
- A especificación da validación contén:
  - Especificación de restricións: `{validator : { <restricións> } }`.
    As restricións son un documento JSON que pode conter:
    - Unha especificación `$jsonSchema`
    - Condicións análogas ás de `find()`
    - Unha combinación de ambos.
  - `validationLevel` (opcional): Posibles valores son: `off` (non valida), `strict` (a predeterminada, valida toda inserción e update), ou `moderate` (valida insercións e update de documentos que eran válidos, non valida os que xa eran inválidos).
  - `validationAction` (opcional): `error` (predeterminado) ou `warn`.

Anexo: Validación de documentos

Exemplo (`$jsonSchema`)

Na colección de autores de posts, o nome é un string obrigatorio. O email é opcional, pero se existe é un string cun determinado formato.

Úsase unha versión simplificada, onde \S na expresión regular significa "non-whitespace character".

```
db.createCollection("autores", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         title: "Autor Object Validation",
         required: [ "nome" ],
         properties: {
            nome: {
               bsonType: "string",
               description: "'nome' é requerido e debe ser un string"
            },
            email: {
               bsonType: ["string"],
               pattern: "^\\S+@\\S+\\.\\S+$"
            }
         }
      }
   }
} )
```

Anexo: Validación de documentos

Exemplo (condicións)

Na colección emps, debe haber un campo dni ou un campo nss, de tipo string.

```
db.createCollection("emps",
         {  validator: { $or:
                     [{dni: {$type:"string"}},  {nss: {$type:"string"}} ]
         }  } )
```

```
> db.emps.insertOne({})
```

```
MongoServerError: Document failed validation
Additional information: {
  failingDocumentId: ObjectId("633d9b16d48442491e01e65c"),
  details: {
    operatorName: '$or',
    clausesNotSatisfied: [
      { index: 0,
        details: {
          operatorName: '$type',
          specifiedAs: { dni: { '$type': 'string' } },
          reason: 'field was missing' } },
      { index: 1,
        details: {
          operatorName: '$type',
          specifiedAs: { nss: { '$type': 'string' } },
          reason: 'field was missing' } } ]
 }
```

```
> db.emps.insertOne({dni:"12345678Z"})
```

```
  {"acknowledged" : true, "insertedId" : ObjectId("57e259d5ae18b34c8d2b43ee") }
```

Anexo: Time Series

Introdución

Time series data, para MongoDB, é unha secuencia de puntos no tempo, con informaciòn asociada.

Normalmente considera 3 compoñentes:

- **Tempo:** O instante de tempo.
- **Metadatos:** Normalmente identifica a procedencia dos datos. Ex: datos dun sensor, ubicación xeográfica.
  Almacénanse no denominado `metaField`. Deberían ser valores que raramente cambian e que sirven para filtrar documentos.
- **Medidas:** Métricas ou valores que se miden ó longo do tempo. Ex: Temperatura, salinidade, potencia, consumo, ...

Utilízase de xeito interno técnicas de *bucketing* para disminuir o espazo de disco usado e aumentar a eficiencia das consultas.

Exemplo de documento:

`https://www.mongodb.com/docs/manual/core/timeseries/timeseries-procedures/`

```
{
  metadata: { sensorId: 5578, type: "temperature" },
  timestamp: ISODate("2021-05-18T00:00:00.000Z"),
  temp: 12
}
```

Anexo: Time Series

Creación da colección

Exemplo:

```
db.createCollection(
   "weather",
   {
         timeseries: {
             timeField: "timestamp",
             metaField: "metadata",
             granularity: "seconds"
         },
         expireAfterSeconds: 86400
    })
```

Especificación do campo `timeseries`:

- `timeField` (obrigatorio): O instante de tempo.
- Granularidade (opcional): Pode especificarse de 2 maneiras:
  - `granularity: "seconds"` (predeterminado), `"minutes"` ou `"hours"`
  - `bucketRoundingSeconds` e `bucketMaxSpanSeconds` (ambos deben ter o mesmo valor)
- `metaField` (opcional): Metadatos para identificar e filtrar as medicións.

Adicionalmente, `expireAfterSeconds` (opcional) activa o borrado automático de documentos que superan ese número de segundos.

Bibliografía

[1] Kristina Chodorow. *MongoDB: The Definitive Guide*. 2ª ed. O'Reilly Media, Inc., 2013.

[2] Daniel Coupal e Ken W. Alger. *Building with Patterns: A Summary*. https://www.mongodb.com/blog/post/building-with-patterns-a-summary.

[3] Daniel Coupal, Steve Hoberman e Pascal Desmarets. *MongoDB data modeling and schema design*. Sedona: Technics Publications, 2023. ISBN: 9781634621984.

[4] Lauren Schaefer e Daniel Coupal. *A Summary of Schema Design Anti-Patterns and How to Spot Them*. https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-summary/.

[5] "The JSON Data Interchange Format". En: *Standard ECMA-404* (2003).

[6] *The MongoDB Manual*. https://www.mongodb.com/docs/manual/.