



UNIVERSIDAD
DE GRANADA



Cloud Computing: Servicios y Aplicaciones

Curso 2016 - 2017

Tutorial MongoDB.

Tutorial MongoDB.	1
1. Introducción	2
2. Conceptos clave	3
3. Elementos de los Documentos	3
4. Ejemplo de documento (schema-less)	4
5. Documentación e instalación	4
6. Utilidades de MongoDB	5
7. Herramientas gráficas	5
8. Mongo Shell	6
9. Correspondencia SQL vs MongoDB	7
10. Operaciones CRUD → IFUR	8
10.1 Insert (Añadir)	8
10.2 Find (Consulta)	9
10.3 Update (Actualizar)	12
10.4 Remove (Eliminar)	13
10.5 Resumen CRUD. MongoDB vs SQL	13
11. Índices	13
12. Pipeline de Agregación	16
Acumuladores sobre grupos:	18
13. MapReduce en MongoDB	19
13.1 Ejemplo 1	19
13.2 Ejemplo 2	21
13.3 Ejemplo 3	21
13.4 Ejercicio propuesto	23

1. Introducción

MongoDB (de la palabra en inglés “humongous” que significa enorme) es un sistema de base de datos NoSQL “open source” orientado a documentos escrito en C++.

Existen versiones para instalar en diversas plataformas: Windows, Linux, OS X.

También está disponible como servicio empresarial en la nube (MMS) a través de <https://mms.mongodb.com>, y que se puede integrar con Amazon Web Services (AWS)

Como característica principal, destacamos que es una BBDD **basada en documentos**. Los documentos (objetos) se acoplan perfectamente en los tipos de datos de los lenguajes de programación. Los documentos incrustados y las colecciones reducen la necesidad de reuniones. Además dispone de un esquema dinámico que facilita el polimorfismo.

Posee un **alto rendimiento** debido al carácter compuesto de los documentos, que pueden incluir otros documentos y colecciones relacionadas, hace más rápidas las escrituras y las lecturas. Los índices pueden incluir claves definidas sobre los documentos incrustados y sobre las colecciones. Adicionalmente, existe una opción de escritura de flujos de datos sin protocolo de reconocimiento.

Otras dos características importantes son su **alta disponibilidad**, con servidores replicados con restablecimiento automático maestro, y su **fácil escalabilidad**. El “sharding” automático distribuye una colección de datos entre diferentes máquinas. Por último, las lecturas eventualmente-consistentes se pueden distribuir a través de servidores replicados.

Las **consultas** se realizan de manera “ad hoc” ya que MongoDB soporta la búsqueda por campos, consultas de rangos y expresiones regulares. Las consultas pueden devolver un campo específico del documento pero también pueden ser una función JavaScript definida por el usuario.

Cualquier campo en un documento de MongoDB puede ser **indexado**, al igual que es posible hacer índices secundarios. El concepto de índices en MongoDB es similar a los encontrados en base de datos relacionales.

MongoDB soporta el tipo de **replicación** maestro-esclavo. El maestro puede ejecutar comandos de lectura y escritura. El esclavo puede copiar los datos del maestro y sólo se puede usar para lectura o para copia de seguridad, pero no se pueden realizar escrituras. El esclavo puede elegir un nuevo maestro en caso del que se caiga el servicio con el maestro actual.

Para el **balanceo de carga**, MongoDB escala de forma horizontal usando el concepto de “shard”. El desarrollador elige una llave shard, la cual determina cómo serán distribuidos los datos de una colección. Los datos son divididos en rangos (basado en la llave shard) y distribuidos a través de múltiples shard. MongoDB tiene la capacidad de ejecutarse en múltiples servidores, balanceando la carga y/o duplicando los datos para poder mantener el sistema funcionando en caso que exista un fallo de hardware.

MongoDB puede ser utilizado con un **sistema de archivos** (GridFS), con balanceo de carga y la replicación de datos utilizando múltiples servidores para el almacenamiento de archivos. Esta función (que es llamada GridFS) está incluida en los drivers de MongoDB y disponible para los lenguajes de programación que soporta MongoDB.

La función MapReduce y el operador **aggregate()** pueden ser utilizados para el procesamiento por lotes de datos y operaciones de agregación. Estos mecanismos permiten que los usuarios puedan obtener el tipo de resultado que se obtiene cuando se utiliza el comando SQL "group-by".

Por último, MongoDB tiene la capacidad de realizar consultas utilizando **JavaScript**, haciendo que estas sean enviadas directamente a la base de datos para ser ejecutadas: `db.system.js.save` (<http://docs.mongodb.org/manual/tutorial/store-javascript-function-on-server/>)

2. Conceptos clave

1. MongoDB tiene el concepto de **"base de datos"** con el que estamos familiarizados (schema en el mundo relacional). Dentro de un servidor MongoDB podemos tener 0 o más BBDD, cada una actuando como un contenedor de todo lo demás.
2. Una base de datos puede tener una o más **"colecciones"**, equivalente en el mundo relacional a una "tabla".
3. Las colecciones están hechas de 0 o más **"documentos"**, donde un documento puede considerarse equivalente a una fila de una tabla de un RDBMS.
4. Un documento está compuesto de uno o varios **"campos"** que son equivalentes a las columnas de una fila. La estructura está compuesta por "key-value pairs" parecido a las matrices asociativas en un lenguaje de programación.
5. Los **"índices"** en MongoDB funcionan como los de los RDBMS.
6. Los **"cursores"** son utilizados para acceder progresivamente a los datos recuperados con una consulta. Pueden usarse para contar o moverse hacia delante entre los datos

3. Elementos de los Documentos

Además de los datos de un documento, MongoDB siempre introduce un campo adicional `_id`. Todo documento tiene que tener un campo `_id` único, que puede contener un valor de cualquier tipo BSON, excepto un array (<http://docs.mongodb.org/manual/core/document/>)

Podemos generar el identificador nosotros como `x = "55674321R"` o `y = ObjectId("507f191e810c19729de860ea")`, o bien dejarle a MongoDB que lo haga, eso lo hace si no asignamos un valor a `_id`. Es preferible que MongoDB lo genere por nosotros.

Tal como se especifica arriba, el tipo de ese campo es `ObjectId` (<http://docs.mongodb.org/manual/reference/object-id/>). Es un tipo de datos de 12 bytes, donde 4 bytes representan un timestamp, 3 un identificador de máquina, 2 el identificador del proceso y 3 restantes un contador. Tiene el atributo `str` y los métodos `getTimeStamp()` y `toString()`

El campo `_id` es indexado lo que explica que se guardan sus detalles en la colección del sistema `system.indexes`

4. Ejemplo de documento (schema-less)

```
{
  "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
  "Last Name": "PELLERIN",
  "First Name": "Franck",
  "Age": 29,
  "Address": {
    "Street": "1 chemin des Loges",
    "City": "VERSAILLES"
  }
}
```

5. Documentación e instalación

La documentación completa de MongoDB puede encontrarse en:

<http://docs.mongodb.org/manual/>

Instrucciones para instalar MongoDB en Linux:

<http://docs.mongodb.org/manual/administration/install-on-linux/>

Si deseamos usar el servicio de docker podemos utilizar el contenedor oficial de mongo (`docker pull mongo`) o una versión más ligera (`docker pull mvertes/alpine-mongo`). La configuración del primero puede ser más complicado, así que se recomienda el segundo.

Para lanzarlo sólo tenemos que ejecutar la siguiente orden:

```
docker run -d --name mongo -p 140XX:27017 -v
/somewhere/onmyhost/mydatabase:/data/db /mvertes/alpine-mongo
```

Fijaos que el puerto 140XX corresponde a uno de vuestros puertos asignados. La opción `-v` indica la carpeta donde queréis que se almacenen los datos de MongoDB dentro de vuestro *home*.

Para acceder al contenedor con mongoDB, solamente tendremos que ejecutar:

```
$ docker exec -ti mongo sh
```

Si bien podemos lanzar directamente el cliente Shell de mongo

```
$ docker exec -ti mongo mongo
```

O lanzarlo también en su propio contenedor:

```
$ docker run -ti --rm --name mongoshell mongo host:port/db
```

6. Utilidades de MongoDB

Utilidades disponibles para el manejo y la administración del sistema de base de datos:

- **mongo**: es un Shell interactivo que permite a los desarrolladores ver, insertar, eliminar y actualizar datos en su base de datos. Entre otras funciones, también permite la replicación de información, configurar los Shards, apagar los servidores y ejecutar JavaScript.
- **mongostat**: es una herramienta de línea de comandos que muestra una lista de estadísticas de una instancia de MongoDB en ejecución.
- **mongotop**: es una herramienta de línea de comandos que provee un método para mostrar la cantidad de tiempo empleado por lectura o escritura de datos en una instancia.
- **mongosniff**: es una herramienta de línea de comandos que provee un sniffing en la base de datos haciendo un sniffing en el tráfico de la red que va desde y hacia MongoDB.
- **mongoimport/mongoexport**: es una herramienta de línea de comandos que facilita la importación/exportación de contenido desde JSON, CSV o TSV.
- **mongodump/mongorestore**: es una herramienta de línea de comandos para la creación de una exportación binaria del contenido de la base de datos.

No aparecen instaladas por defecto en el contenedor “light” de mongoDB sugerido en el punto anterior. Para poder hacer uso de ellas (en particular “mongoimport”) deberemos seguir los siguientes pasos:

```
apk update && \ apk upgrade && \ apk add mongoddb-tools
```

Nota: para copiar ficheros del host al contenedor con la orden “cp”, tendremos que tener instalado docker versión 1.8. Una forma “particular” de alcanzar el mismo fin sería la siguiente:

```
docker exec -i <NOMBRE> sh -c 'cat > /home/test.txt' < test.txt
```

7. Herramientas gráficas

En el siguiente enlace se encuentra una recopilación de herramientas gráficas para la administración y uso de MongoDB: <http://docs.mongodb.org/ecosystem/tools/administration-interfaces/>

Adicionalmente, MongoDB tiene una interfaz administrativa accesible yendo a: <http://localhost:28017/> siempre que se haya iniciado con la opción: `mongod --rest` o se haya añadido al fichero de configuración la línea `rest=true`.

UMongo (<http://edgytech.com/umongo>) es una aplicación (open source) de sobremesa para navegar y administrar un clúster MongoDB.

RoboMongo (<http://robomongo.org/> open source). Incluye una shell completamente compatible con la shell mongo.

8. Mongo Shell

El shell Mongo se comporta como un buen shell de UNIX. En prime lugar, ofrece autocompletado (usando el tabulador). Permite moverte por la historia de comandos con el cursor para arriba o abajo o moverte al primer o último comando con las combinaciones de teclas (CTRL-a y CTRL-e).

Ofrece un objeto implícito llamado `db` que representa a la base de datos. Usa por defecto la BD `test` salvo que establezcamos otra mediante el comando `use <database-name>`.

Las colecciones se crean automáticamente cuando insertamos el primer documento en ellas

Es un shell en JavaScript que no distingue entre enteros y números en coma flotante, todo número se representa en JavaScript como un número de coma flotante de 64 bits.

Conectarse a nuestro servidor mongod: `mongo <database> -u <user> -p <clave> [--authenticationDatabase admin]`

Comandos útiles:

- `help` – muestra ayuda.
- `db.help()` – muestra ayuda de los métodos de la BD .
- `db.<collection>.help()` – detalla qué métodos se pueden aplicar a una colección.
- `show dbs` – imprime una lista de las bases de datos del servidor.
- `use <database-name>` – cambia la base de datos a `<db>`, haciendo que `db` apunte la BD seleccionada.
- `show collections` – imprime todas las colecciones de la base de datos actual.
- `show users` – imprime los usuarios de la BD.

Referencia comandos mongo Shell:
<http://docs.mongodb.org/manual/reference/method/>

9. Correspondencia SQL vs MongoDB

MYSQL EXECUTABLE	ORACLE EXECUTABLE	MONGODB EXECUTABLE
mysqld	oracle	mongod
mysql	sqlplus	mongo

SQL TERM	MONGODB TERM
database	database
table	collection
index	index
row	document
column	field
joining	embedding & linking

SQL	MONGODB
CREATE TABLE users (name VARCHAR(128), age NUMBER)	db.createCollection("users")
INSERT INTO users VALUES ('Bob', 32)	db.users.insert({name: "Bob", age: 32})
SELECT * FROM users	db.users.find()
SELECT name, age FROM users	db.users.find({}, {name: 1, age: 1, _id:0})
SELECT name, age FROM users WHERE age = 33	db.users.find({age: 33}, {name: 1, age: 1, _id:0})
SELECT * FROM users WHERE age > 33	db.users.find({age: {\$gt: 33}})
SELECT * FROM users WHERE age <= 33	db.users.find({age: {\$lte: 33}})

SQL	MONGODB
SELECT * FROM users WHERE age > 33 AND age < 40	db.users.find({age: {\$gt: 33, \$lt: 40}})
SELECT * FROM users WHERE age = 32 AND name = 'Bob'	db.users.find({age: 32, name: "Bob"})
SELECT * FROM users WHERE age = 33 OR name = 'Bob'	db.users.find({\$or:[{age:33}, {name: "Bob"}]})
SELECT * FROM users WHERE age = 33 ORDER BY name ASC	db.users.find({age: 33}).sort({name: 1})
SELECT * FROM users ORDER BY name DESC	db.users.find().sort({name: -1})
SELECT * FROM users WHERE name LIKE 'Joe%'	db.users.find({name: /Joe/})
SELECT * FROM users WHERE name LIKE '^Joe%'	db.users.find({name: /^Joe/})
SELECT * FROM users LIMIT 10 SKIP 20	db.users.find().skip(20).limit(10)
SELECT * FROM users LIMIT 1	db.users.findOne()

SELECT DISTINCT name FROM users	db.users.distinct("name")
SELECT COUNT(*) FROM users	db.users.count()
SELECT COUNT(*) FROM users WHERE AGE > 30	db.users.find({age: {\$gt: 30}}).count()
SELECT COUNT(AGE) FROM users	db.users.find({age: {\$exists: true}}).count()
UPDATE users SET age = 33 WHERE name = 'Bob'	db.users.update({name: "Bob"}, {\$set: {age: 33}}, {multi: true})
UPDATE users SET age = age + 2 WHERE name = 'Bob'	db.users.update({name: "Bob"}, {\$inc: {age: 2}}, {multi: true})
DELETE FROM users WHERE name = 'Bob'	db.users.remove({name: "Bob"})
CREATE INDEX ON users (name ASC)	db.users.ensureIndex({name: 1})
CREATE INDEX ON users (name ASC, age DESC)	db.users.ensureIndex({name: 1, age: -1})
EXPLAIN SELECT * FROM users WHERE age = 32	db.users.find({age: 32}).explain()

10. Operaciones CRUD → IFUR

Operaciones CRUD: Create (insertar) , Read (consultar), Update (actualizar) and Delete (eliminar) “records”.

CRUD vs IFUR:

Create	=>	Insert
Read	=>	Find
Update	=>	Update
Delete	=>	Remove

10.1 Insert (Añadir)

BD ejemplo de Restaurantes. Para importar la base de datos, accedemos a `hadoop.ugr.es` y escribimos el siguiente comando:

```
mongoimport -u <user> -p <clave> --db <vuestraBD> --collection
restaurants --type json --drop --file
/tmp/mongo/restaurants.json
```

Para INSERTAR documentos en una colección se usa el método `insert()`:

```
db.restaurants.insert(
{
  "address" : {
    "street" : "2 Avenue",
    "zipcode" : "10075",
    "building" : "1480",
    "coord" : [ -73.9557413, 40.7720266 ]
  },
  "borough" : "Manhattan",
  "cuisine" : "Italian",
```



```

    "grades" : [
      {
        "date" : ISODate("2014-10-01T00:00:00Z"),
        "grade" : "A",
        "score" : 11
      },
      {
        "date" : ISODate("2014-01-16T00:00:00Z"),
        "grade" : "B",
        "score" : 17
      }
    ],
    "name" : "Vella",
    "restaurant_id" : "41704620"
  }
)

```

Como prueba, inserta en la colección `restaurants`, de la base de datos (**db**) un documento con los valores que se indican anteriormente. El método devuelve un objeto *WriteResult* con el estado de la operación:

```
WriteResult({ "nInserted" : 1 })
```

La colección `restaurants` se crearía automáticamente cuando insertamos un documento en ella, si no existiese previamente.

db hace referencia a la base de datos sobre la que estamos actuando (si no se cambia mediante `use`, por defecto se usa la base de datos `test`). Recordad que si se inserta sin el campo `_id`, mongod inserta un valor para ese campo que es único para cada documento de la colección

10.2 Find (Consulta)

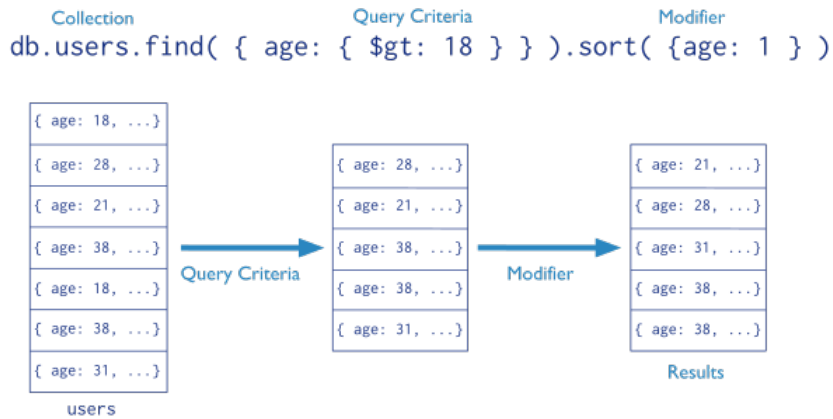
Se utiliza el método `find`:

```

db.users.find(
  { age: { $gt: 18 } },
  { name: 1, address: 1 }
).limit(5)

```

 **collection**
 **query criteria**
 **projection**
 **cursor modifier**



Se utiliza el modificador con el valor “1” para orden ascendente, “-1” para descendente.

Los operadores para los criterios de consulta serían los siguientes:

Operador	Descripción	Ejemplo de consulta
\$eq	Igual que	db.ships.find({class:\$eq:'P'})
\$gt	Mayor que	db.ships.find({class:\$gt:'P'})
\$gte	Mayor o igual que	db.ships.find({class:\$gte:'P'})
\$lt	Menor que	db.ships.find({class:\$lt:'P'})
\$lte	Menor o igual que	db.ships.find({class:\$lte:'P'})
\$ne	Distinto a	db.ships.find({class:\$ne:'P'})
\$in	Igual a alguno de los elementos de un array	db.ships.find({class:\$in:['P', 'Q']})
\$exists	Si un atributo existe o no	db.ships.find({type:\$exists:true})
\$regex	Expresiones regulares tipo Perl	db.ships.find({name : {\$regex:'^USS\\sE'}})
\$type	Busca el tipo de un determinado campo de un documento	db.ships.find({name : {\$type:2}})

El operador `$exists` se utiliza para encontrar qué documentos contienen o no el campo indicado. Por ejemplo:

```
db.restaurants.find({address: {$exists: true}})
```

Si queremos utilizar el operador booleano OR tenemos que hacer uso del operador `$or` y asociarle un array de tuplas clave/valor sobre los que realizar el OR:

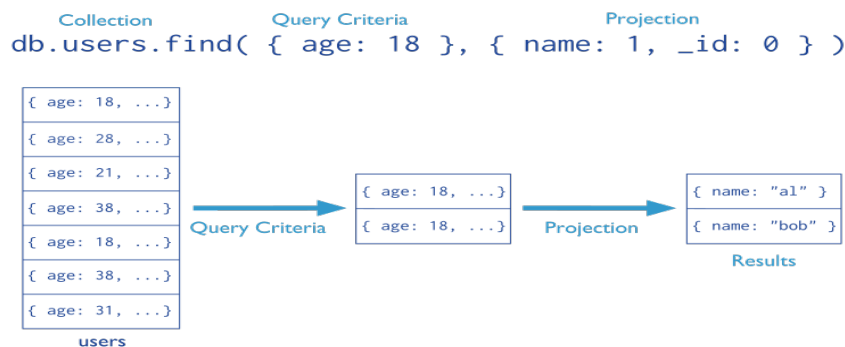
```
db.restaurants.find( { $or:[{ "cuisine": "Italian" }, {
"address.zipcode": "10075" }]} )
```

Dado que los arrays en MongoDB son objetos de primera categoría se puede comprobar la inclusión de un elemento dentro de un array al igual que si comparáramos con un único valor:

```
db.restaurants.find( {grades : { "date" : ISODate("2014-10-
01T00:00:00Z"), "grade" : "A", "score" : 11}} )
```

devolverá los restaurantes que incluyan el grado descrito por ({date:, grade, :score:})

La **proyección** se determina mediante el segundo argumento de `find()`. El valor 1 indica que se muestre el campo y el valor 0 que se excluya el campo de los resultados mostrados:



Ejemplo: `db.restaurants.find({ "cuisine": "Italian" ,
"address.zipcode": "10075" }, {name:1,address:1})`

El resultado de una consulta (`find()`) se devuelve mediante un objeto **cursor**. Éste puede ser asignado a una variable. En caso contrario, mongo shell muestra los 20 primeros documentos. Se puede iterar usando este procedimiento:

```
var myCursor = db.restaurants.find({ "cuisine": "Italian"})
while (myCursor.hasNext()) {
    printjson(myCursor.next());
}
```

Otro modo de iterarlo: `myCursor.forEach(printjson);`

Se puede realizar una visualización más cómoda mediante la **paginación**. Se permite con los métodos de cursor `skip` y `limit`

```
db.restaurants.find(
{ "cuisine": "Italian" }).sort({name:1}).limit(2).skip(1)
```

Del mismo modo que en SQL estándar, aquí también se pueden contabilizar resultados:


```
db.restaurants.count({ "grades.score": { $lt: 5 } } )

db.restaurants.find({ "grades.score": { $lt: 5 } } ).count()
```

10.3 Update (Actualizar)

por defecto actualiza sólo un documento, si se pone multi: true se actualizan todos los documentos que cumplan el criterio.

```
db.users.update(
  { age: { $gt: 18 } },
  { $set: { status: "A" } },
  { multi: true }
)
```



```
db.restaurants.update(
  { "restaurant_id" : "41156888" },
  { $set: { "address.street": "East 31st Street" } })
```

El operador \$set hace que se mantengan los valores para el resto de los campos en los documentos existentes y sólo se cambie el del campo al que se aplica el operador.

El operador \$unset elimina un campo del documento de salida:

```
db.books.update( { _id: 1 }, { $unset: { tags: 1 } } )
```

El operador \$inc se usa para incrementar el campo por una cantidad positiva o negativa:

```
db.unicorns.update({name: 'Pilot'}, {$inc: {vampires: -2}})
```

Podemos añadir una nueva valoración para un restaurante a través del modificador \$push:

```
db.restaurants.update({ "restaurant_id" : "41156888" },{$push:
{grades: { "date" : ISODate("2016-01-02T00:00:00.000Z"),
"grade" : "A", "score" : 14}}})
```

Si queremos que se cree un nuevo documento cuando intentamos actualizar uno no existente (upsert) colocamos un tercer parámetro a true

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}},
{upsert: true});
```

Si queremos que update() actualice todos los documentos que cumplen una expresión, el 4º parámetro tiene que ponerse a true

```
db.restaurants.update( { "address.zipcode": "10016", cuisine:
"Other" },{$set: { cuisine: "Category To Be Determined" },
$currentDate: { "lastModified": true }}, { multi: true})
```

10.4 Remove (Eliminar)

Por defecto elimina todos los documentos que cumplan el criterio, si se pone la opción: `justOne: true`, se elimina sólo uno.

```
db.users.remove(
  { status: "D" }
)
```

← collection
← remove criteria

La modificación de un solo documento siempre es atómica, incluso si la operación de escritura modifica varios documentos incrustados dentro de ese documento.

Ninguna otra operación es atómica. Se puede intentar aislar una operación de escritura que afecta múltiples documentos utilizando el operador `$isolated`.

```
db.foo.update(
  { status: "A" , $isolated: 1 },
  { $inc: { count: 1 } },
  { multi: true }
)
```

10.5 Resumen CRUD. MongoDB vs SQL

<pre>db.users.insert ({ name: "sue", age: 26, status: "A" })</pre> <p>← collection ← field: value ← field: value ← field: value } document</p>	<pre>INSERT INTO users (name, age, status) VALUES ("sue", 26, "A")</pre> <p>← table ← columns ← values/row</p>
<pre>db.users.update({ age: { \$gt: 18 } }, { \$set: { status: "A" } }, { multi: true })</pre> <p>← collection ← update criteria ← update action ← update option</p>	<pre>UPDATE users SET status = 'A' WHERE age > 18</pre> <p>← table ← update action ← update criteria</p>
<pre>db.users.remove({ status: "D" })</pre> <p>← collection ← remove criteria</p>	<pre>DELETE FROM users WHERE status = 'D'</pre> <p>← table ← delete criteria</p>

11. Índices

Los índices operan de modo similar a los de los RDBMS, mejorando el rendimiento de las consultas y operaciones de ordenación en MongoDB. Se crean con la sentencia

`ensureIndex()`, identificando el sentido de ordenación por campo ascendente (1) o descendente (-1):

```
db.restaurants.ensureIndex({name: 1});
```

Para conocer los índices de la colección, escribir:

```
db.restaurants.getIndexes();
```

Y se eliminan con `dropIndex()`:

```
db.restaurants.dropIndex({name: 1});
```

Para asegurarnos que el índice es único, usamos el atributo `unique`:

```
db.restaurants.ensureIndex({name: 1}, {unique: true});
```

Los índices también pueden ser compuestos:

```
db.restaurants.ensureIndex({cuisine: 1, name: -1});
```

Para saber si se está usando un índice o no, usamos `explain()`:

```
db.restaurants.find().explain()
```

Si la salida indica que el plan utilizado (*winningplan*) utiliza una etapa "COLLSCAN" entonces no se han utilizado los índices en la búsqueda. En ese sentido, se escanearán más documentos y tardarán más las búsquedas u ordenaciones. Por otra parte, si en el plan aparece marcado como "IXSCAN" entonces sí que se ha utilizado el índice.

```
db.restaurants.find().explain()
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "alberto.restaurants",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "$and" : [ ]
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "$and" : [ ]
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "serverInfo" : {
    "host" : "2c322c6f2207",
    "port" : 27017,
    "version" : "3.2.10",
```

```

        "gitVersion" :
"79d9b3ab5ce20f51c272b4411202710a082d0317"
    },
    "ok" : 1
}

db.restaurants.find({name : {$regex:'^A'}}).explain()
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "alberto.restaurants",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "name" : /^A/
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "name" : 1
        },
        "indexName" : "name_1",
        "isMultiKey" : false,
        "isUnique" : false,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 1,
        "direction" : "forward",
        "indexBounds" : {
          "name" : [
            "[\"A\", \"B\"]",
            "[/^A/, /^A/]"
          ]
        }
      },
      "rejectedPlans" : [ ]
    },
    "serverInfo" : {
      "host" : "2c322c6f2207",
      "port" : 27017,
      "version" : "3.2.10",
      "gitVersion" :
"79d9b3ab5ce20f51c272b4411202710a082d0317"
    },
    "ok" : 1
  }
}

```

La colección `db.system.indexes` contiene detalles de todos los índices de una BD de MongoDB:

```
db.system.indexes.find()
```

Resumen de operaciones de indexación:

Index	Comando
Creación de un índice	<code>db.universe.ensureIndex({galaxy : 1})</code>
Eliminación de un índice	<code>db.universe.dropIndex({galaxy : 1})</code>
Creación de un índice compuesto	<code>db.universe.ensureIndex({galaxy : 1, quadrant : 1, planet : 0})</code>
Eliminar un índice compuesto	<code>db.universe.dropIndex({galaxy : 1, quadrant : 1, planet : 0})</code>
Creación de un índice compuesto en "background"	<code>db.universe.ensureIndex({galaxy : 1, quadrant : 1, planet : 0},{unique : true, background : true})</code>

12. Pipeline de Agregación

Las operaciones de agregación procesan registros de datos y devuelven resultados. Es una técnica para simplificar código de aplicación y limitar el uso de recursos, ya que usa operaciones nativas en C++

(<http://docs.mongodb.org/manual/reference/operator/aggregation/interface/>)

MongoDB 2.2 introdujo un framework de agregación modelado en torno al concepto de pipelines de procesamiento de datos. A través de varios pasos los documentos son transformados en resultados agregados. Filtros, transformaciones, agrupamientos, ordenación y cálculos son algunas de las operaciones. Finalmente, MongoDB ofrece un conjunto de operadores sencillos de agregación como `count()`, `distinct()` o `group()`. Indicar que MongoDB también permite el uso de MapReduce para hacer agregación, como veremos en la siguiente sección.

Nombre	Descripción
<code>db.collection.aggregate()</code>	Proporciona acceso a la pipeline de agregación.
<code>db.collection.group()</code>	Agrupar documentos en una colección mediante una clave especificada y realiza una agregación simple.
<code>db.collection.mapReduce()</code>	Realiza agregación map-reduce para conjuntos de datos extensos.

MongoDB incorpora un mecanismo para poder agregar datos de documentos en diferentes pasos. Cada paso toma como entrada un conjunto de documentos y como salida produce un conjunto de documentos. Hay operaciones que en un paso dado mantendrán el mismo número de documentos de entrada pero hay otras que los pueden reducir (filtrar).

La siguiente tabla muestra equivalencias entre cláusulas de agregación, incluyendo operaciones de filtrado, agrupación, proyección, ordenación y cálculo, de SQL al framework de agregación de MongoDB:

SQL	MongoDB
WHERE	\$match
GROUP BY	\$group
HAVING	\$match
SELECT	\$project
ORDER BY	\$sort
LIMIT	\$limit
SUM	\$sum
COUNT()	db.records.count()

Operaciones:

- \$project – cambia el conjunto de documentos modificando claves y valores.
- \$match – es una operación de filtrado que reduce el conjunto de documentos generando un nuevo conjunto de los mismos que cumple alguna condición, ej. { \$match: { <query> } }
- \$group – hace el agrupamiento en base a claves o campos indexados, reduciendo el número de documentos.
- \$sort – ordenar en ascendente o descendente, dado que es computacionalmente costoso debería ser uno de los últimos pasos de la agregación .
- \$skip – permite saltar entre el conjunto de documentos de entrada, por ejemplo avanzar hasta el décimo documento de entrada. Se suele usar junto con \$limit
- \$limit – limita el número de documentos a procesar.
- \$unwind – desagrega los elementos de un array en un conjunto de documentos. Esta operación incrementa el número de documentos para el siguiente paso.

Ejemplos y documentación de estos operadores en:

<http://docs.mongodb.org/manual/reference/operator/aggregation-pipeline/>

<http://blog.findemor.es/2015/06/aprender-a-usar-mongodb-guia-7/>

A menudo agrupamos documentos para agregar valores en subconjuntos de documentos.

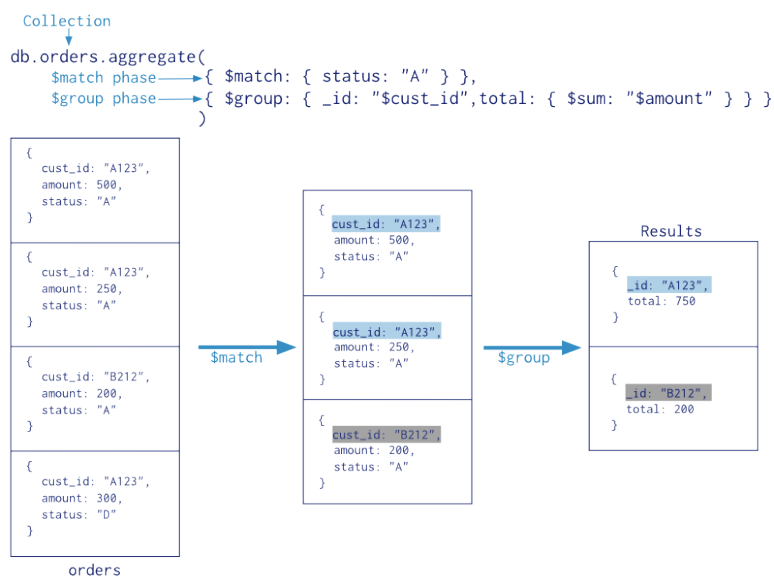
Ejemplo, dado un conjunto de artículos:

```
db.article.aggregate(
  { $group : {
    _id : "$author",
    docsPerAuthor : { $sum : 1 },
    viewsPerAuthor : { $sum : "$pageViews" }    } } );
```

Acumuladores sobre grupos:

Descripción	Ejemplo
\$sum Suma sobre el valor definido de todos los documentos de la colección.	<code>db.ships.aggregate([{\$group : {_id : "\$operator", num_ships : {\$sum : "\$crew"}}}])</code>
\$avg Calcula la media sobre los valores seleccionados de todos los documentos.	<code>db.ships.aggregate([{\$group : {_id : "\$operator", num_ships : {\$avg : "\$crew"}}}])</code>
\$min Obtiene el mínimo de los valores seleccionados de todos los documentos.	<code>db.ships.aggregate([{\$group : {_id : "\$operator", num_ships : {\$min : "\$crew"}}}])</code>
\$max Obtiene el máximo de los valores seleccionados de todos los documentos.	<code>db.ships.aggregate([{\$group : {_id : "\$operator", num_ships : {\$max : "\$crew"}}}])</code>
\$push Devuelve un array de todos los valores que resultan de la aplicación de una expresión a cada documento en un grupo de documentos que comparten el mismo grupo por clave.	<code>db.ships.aggregate([{\$group : {_id : "\$operator", classes : {\$push : "\$class"}}}])</code>
\$addToSet Lo mismo que \$push pero sin valores duplicados	<code>db.ships.aggregate([{\$group : {_id : "\$operator", classes : {\$addToSet : "\$class"}}}])</code>
\$first Obtiene el primer documento de un conjunto, sólo tiene sentido después de un \$sort.	<code>db.ships.aggregate([{\$group : {_id : "\$operator", first_class : {\$first : "\$class"}}}])</code>
\$last Obtiene el último documento de un conjunto, sólo tiene sentido después de un \$sort.	<code>db.ships.aggregate([{\$group : {_id : "\$operator", last_class : {\$last : "\$class"}}}])</code>

Pipeline de agregación:

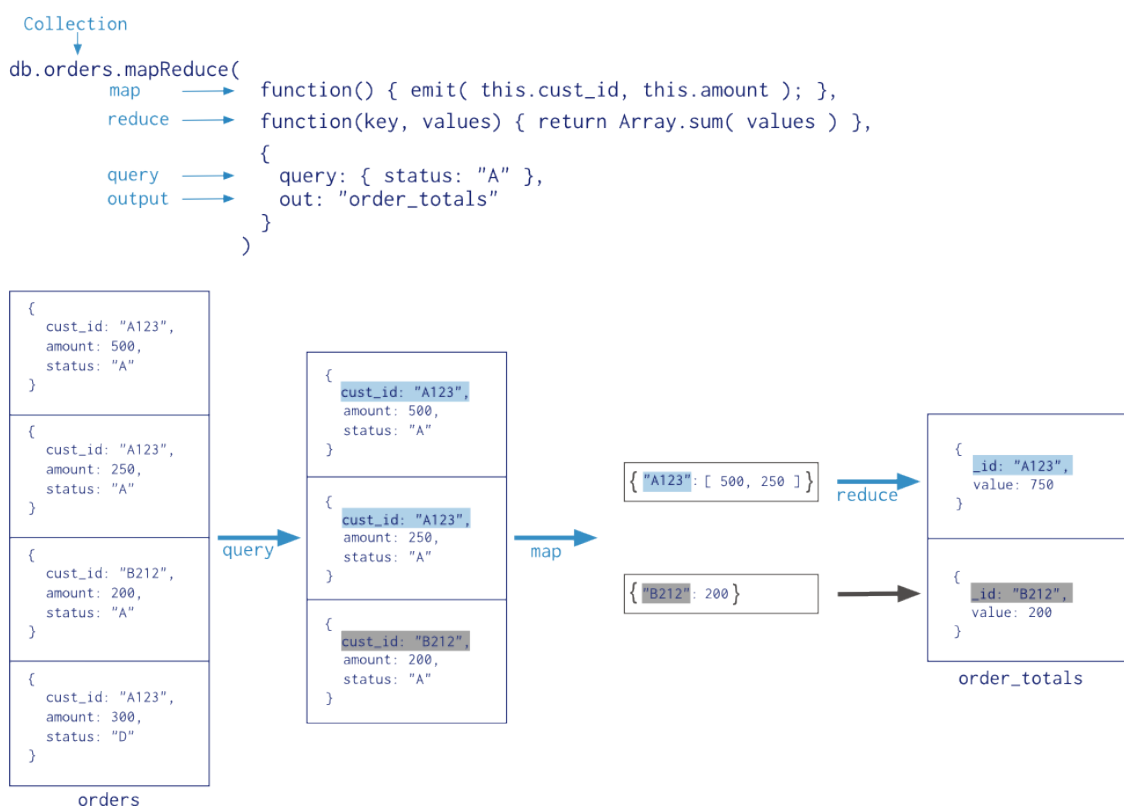


13. MapReduce en MongoDB

MapReduce es un enfoque para procesar datos que tiene dos beneficios: (1) Puede ser paralelizado permitiendo que grandes volúmenes de datos sean procesados a través de varios cores/CPU's y máquinas; y (2) el procesamiento se puede describir mediante el uso de JavaScript.

Se materializa a través de dos pasos:

1. Mapear los datos transformando los documentos de entrada en pares clave/valor.
2. Reducir las entradas conformadas por pares clave y array de valores asociados a esa clave para producir el resultado final.



13.1 Ejemplo 1

Vamos a realizar un ejemplo donde contaremos el número de hits por día en un portal web. La documentación completa se encuentra en

<http://docs.mongodb.org/manual/reference/method/db.collection.mapReduce/>

Cada hit del portal Web estará representado con un log como:

```

resource    date
index       Jan 20 2010 4:30
index       Jan 20 2010 5:30
...         ...

```

Generando tras el procesamiento la siguiente salida:

```
resource year month day count
index    2010 1      20  2
...
```

Para la función *map* emitiremos pares compuestos por una clave compuesta (*resource*, *year*, *month*, *day*) y un valor 1, generando datos como:

```
{resource: 'index', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1}]
```

La función *reduce* recoge cada dato intermedio y genera un resultado final:

```
{resource: 'index', year: 2010, month: 0, day: 20} => {count: 3}
```

Creamos la colección de entrada con un conjunto de documentos:

```
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 4, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 5, 30)});
...
```

La función *map*:

```
var map = function() {
    var key = {resource: this.resource, year:
this.date.getFullYear(), month: this.date.getMonth(), day:
this.date.getDate()};

    emit(key, {count: 1});
};
```

La función *reduce*:

```
var reduce = function(key, values) {
    var sum = 0;
    values.forEach(function(value) {
        sum += value['count']; //sum se incrementa con el
valor del campo
                                //count
    });
    return {count: sum};
};
```

Ejecutamos `mapReduce()`. *out*: indica la colección que almacenará el resultado, *{inline:1}* indica que la operación `mapReduce` se hará en memoria):

```
db.hits.mapReduce(map, reduce, {out: {inline:1}})
```

13.2 Ejemplo 2

Considere la colección orders (pedidos) que contiene los documentos de acuerdo con siguiente prototipo:

```
{ _id: ObjectId("50a8240b927d5d8b5891743c"),
  cust_id: "abc123",
  ord_date: new Date("Oct 04, 2012"),
  status: 'A',
  price: 25,
  items: [ { sku: "mmm", qty: 5, price: 2.5 },
            { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

Vamos a calcular el total del pedido para cada cliente.

```
var mapFunction1 = function() {
    emit(this.cust_id, this.price);
};

var reduceFunction1 = function(keyCustId, valuesPrices) {
    return Array.sum(valuesPrices);
};

db.orders.mapReduce(
    mapFunction1,
    reduceFunction1,
    { out: "map_reduce_ejemplo" }
)
```

Resultado:

```
db.orders.find()

db.map_reduce_ejemplo.find()
```

13.3 Ejemplo 3

Vamos a usar mapReduce en la colección de orders (pedidos) tomando los documentos que tienen un valor de ord_date mayor que 01/01/2012. Agruparemos por el campo item.sku , y calcularemos la cantidad de pedidos y la cantidad total pedida para cada sku (artículo). La operación concluye mediante el cálculo de la cantidad media por pedido para cada valor de sku.

```
var mapFunction2 = function() {
    for (var idx = 0; idx < this.items.length; idx++) {
        var key = this.items[idx].sku;
        var value = {
            count: 1,
            qty: this.items[idx].qty
        }
    }
}
```

```

        };
        emit(key, value);
    }
};

```

Se define la función de *reduce* correspondiente con dos argumentos *keySKU* y *countObjVals*. *countObjVals* es un array cuyos elementos son los objetos agrupados por *keySKU* pasados por la función **map** a la función **reduce**. La función reduce el array *countObjVals* a un sólo objeto *reducedValue* que contiene el recuento y los campos *qty*.

En *reducedVal*, el campo *count* contiene la suma de los campos *count* de cada elemento del array, y el campo *qty* contiene la suma de los campos *qty* de cada elemento del array.

```

var reduceFunction2 = function(keySKU, countObjVals) {
    reducedVal = { count: 0, qty: 0 };
    for (var idx = 0; idx < countObjVals.length; idx++) {
        reducedVal.count += countObjVals[idx].count;
        reducedVal.qty += countObjVals[idx].qty;
    }
    return reducedVal;
};

```

Se define la función de **finalize** que modifica el objeto *reducedVal* para añadir un campo calculado denominado *avg* y devuelve el objeto modificado.

```

var finalizeFunction2 = function (key, reducedVal) {
    reducedVal.avg = reducedVal.qty/reducedVal.count;
    return reducedVal;
};

```

Se realiza la operación MapReduce sobre la colección *orders* usando las funciones *mapFunction2*, *reduceFunction2* y *finalizeFunction2*.

MapReduce:

```

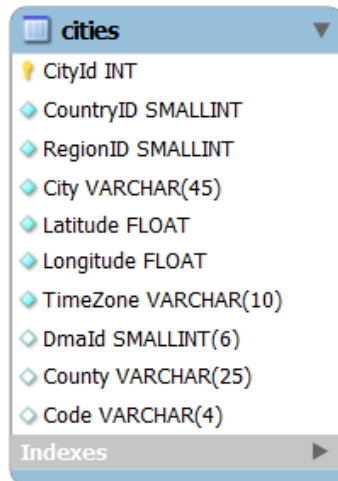
db.orders.mapReduce( mapFunction2,reduceFunction2,
    {
        out: { merge: "map_reduce_ejemplo" },
        query: { ord_date:
            { $gt: new Date('01/01/2012') }
        },
        finalize: finalizeFunction2
    }
)

```

Esta operación utiliza el campo de query para seleccionar sólo aquellos documentos con `ord_date` superior a new Date (01/01/2012). El resultado se guarda en la colección `map_reduce_ejemplo`. Si ya existe la colección `map_reduce_example`, la operación combinará los contenidos existentes con los resultados de esta operación MapReduce.

13.4 Ejercicio propuesto

Vamos a utilizar la base de datos libre GeoWorldMap de GeoBytes. Es una base de datos de países, con sus estados/regiones y ciudades importantes. El esquema de la tabla ciudades es:



Sobre esta Base de datos **vamos a obtener el par de ciudades que se encuentran más cercanas en cada país, excluyendo a los EEUU**. Para ello vamos a cotejar dos enfoques: SQL sobre una representación relacional de esa BD y MapReduce sobre MongoDB.

Para cada ciudad la BD almacena sus coordenadas geográficas en términos de latitud y de longitud. En aras de la simplicidad, vamos a representar la tierra como un plano 2D. La distancia entre dos puntos P1 (x_1, y_1) y P2 (x_2, y_2) en un plano 2D se calcula como la raíz cuadrada de $\{(x_1-x_2)^2 + (y_1-y_2)^2\}$. Siendo y_i la latitud y x_i la longitud.

En un contexto SQL estándar, los pasos a seguir serían los siguientes:

1. Crear una vista que almacene el cuadrado de la distancia para cada par de ciudades de cada país, salvo EEUU:

```
/* QUERY1 - VIEW: city_dist */
CREATE VIEW city_dist AS
SELECT c1.CountryID,
       c1.CityId, c1.City,
       c2.CityId AS CityId2, c2.City AS City2,
       POWER(c1.Latitude-c2.Latitude,2) +
       POWER(c1.Longitude-c2.Longitude,2) as Dist
FROM cities c1 , cities c2
WHERE c1.CountryID = c2.CountryID /* Del mismo país*/
AND c1.CityId < c2.CityId /* Cada par de ciudades una sola vez */
```

```
AND c1.CountryID <> 254 /* No se incluyen las ciudades de
EEUU */;
```

2. Agrupamos estos datos por país y seleccionamos las 2 ciudades que tienen el menor valor para el campo "Dist", siendo mayor que cero dicha distancia.

```
/* QUERY 2 */
SELECT c.CountryID,c.City,c.City2,round(sqrt(c.DIST),5) AS
Distancia
FROM (
    SELECT CountryID, min(Dist) AS MinDist
    FROM city_dist
    WHERE Dist > 0 /* Evitamos ciudades que compartan
latitud y
                                longitud */
    GROUP BY CountryID
) a ,city_dist c
WHERE a.CountryID = c.CountryID
    AND a.MinDist = c.Dist;
```

Es importante tener en cuenta los pasos que seguimos. En el primer paso se realizaron todos los cálculos (calculando la distancia entre cada 2 ciudades de cada país). En el siguiente paso se agruparon los resultados por país y se seleccionaron aquellas 2 ciudades en las que el valor de la distancia era menor.

Para realizar la consulta en MongoDB mediante el modelo MapReduce, seguiremos los siguientes pasos.

1. En primer lugar, vamos a importar en nuestra BD de MongoDB un archivo con 37245 ciudades del mundo que está en formato csv (/tmp/mongo/Cities.csv)

```
mongoimport -u <user> -p <clave> --db <bd> --collection
cities --type csv --headerline --file
/tmp/mongo/Cities.csv
```

2. A continuación, vamos a implementar el código para resolver el problema sobre la recién creada colección mediante un enfoque MapReduce conforme a los pasos que hemos ilustrado anteriormente. En primer lugar, debemos considerar la función **Map**

- a. Esta función se utiliza para dividir los datos en grupos en base a un valor deseado (llamado *Key*). Esto es similar a la Etapa 2 de la solución de SQL anterior. El paso de Map se implementa escribiendo una función en JavaScript, cuyo formato es el siguiente:

```
function /*void*/ MapCode() { }
```


La función Map se invoca por cada documento de la colección como un método. Con *"this"* se puede acceder a cualquier dato del documento actual

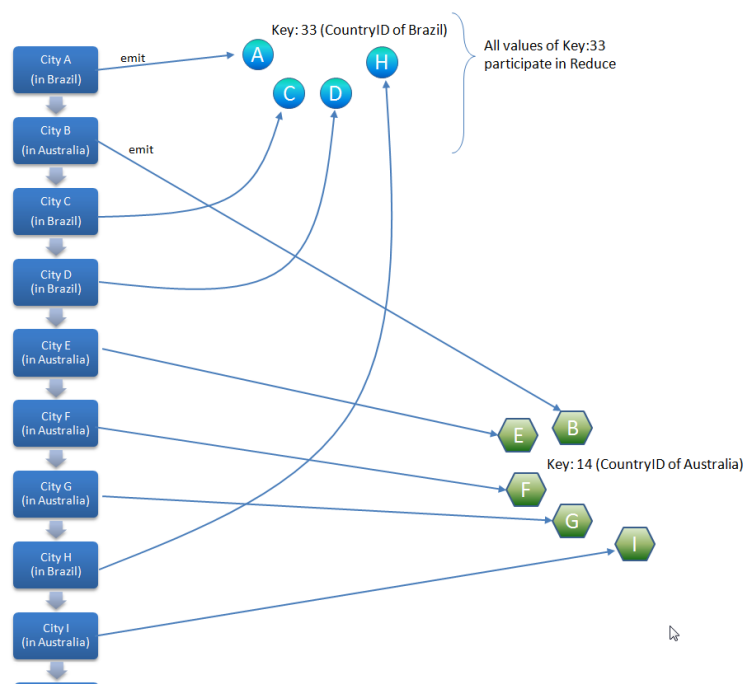
Otro elemento que está disponible es la función *"emit"* que dispone de dos argumentos: el primero, la clave sobre la que desea agrupar los datos; el segundo argumento son los datos que desea agrupar.

b. Aspectos a considerar al escribir la función Map:

- ¿Cómo queremos dividir o agrupar los datos? En otras palabras, ¿cuál es nuestra clave? Que es lo que se debe pasar como primer parámetro a la función *"emit"*.
- ¿Qué datos necesitaremos para el procesamiento subsiguiente? Esto ayuda a determinar que se incluye en el segundo parámetro de la función *"emit"*.
- En qué formato o estructura necesitaremos nuestros datos. Esto nos ayuda a refinar el segundo parámetro de la función de *"emit"*.
- En nuestro ejemplo los datos deben agruparse según el código de país: *"CountryId"*. Así que éste será el primer parámetro de la función *"emit"*

```
MapCode function () { emit (this.CountryID, ...); }
```

- Los datos que precisaremos para el procesamiento posterior serán: City, Latitude y Longitude. Por ello el segundo parámetro será un array que contenga todos los valores para cada ciudad de ese país.
- Después de completada la etapa Map, se obtienen un conjunto de pares clave-valor. En nuestro caso, en el par clave-valor la clave es *CountryId* y el valor es un objeto JSON como se muestra en la siguiente imagen:



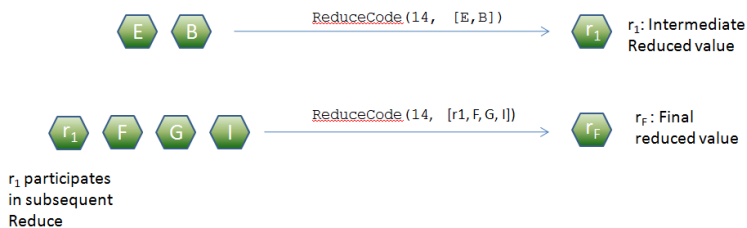
3. La operación Reduce

- a. Agrega los diferentes valores para cada clave dada usando una función definida por el usuario. En otras palabras, Reduce recorrerá cada valor de la clave (*CountryId*) y recogiendo todos sus valores (en nuestro caso) objetos JSON creados a partir de la etapa **Map** y luego los procesará uno por uno usando una lógica personalizada definida.

```
function /*object*/ ReduceCode(key, arr_values) { }
```

- b. Reduce toma 2 parámetros - 1) Clave 2) un array de valores (emitidos desde el paso Map). La salida de Reduce es un objeto. Es importante tener en cuenta que Reduce se puede invocar varias veces desde un mismo valor de la clave. Considerar un caso en el que la cantidad de datos es enorme y se encuentran en 2 servidores diferentes. Sería ideal realizar un Reduce para una clave dada en el primer servidor, y luego realizar un Reduce para la misma clave en el segundo servidor. Y después realizar un Reduce sobre los resultados de estos dos valores reducidos.

Lets consider the Reduce of Key: 14 (Australia)



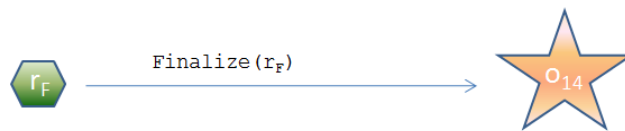
- c. No sabemos el orden y la forma en que se aplicarían esos pasos de Reduce, dependerá de como esté configurado MongoDB. Lo que sí sabemos es que si Reduce se ejecuta más de una vez, entonces el valor devuelto por la cada invocación de Reduce será usado en una subsecuente invocación de reduce como parte de la entrada de dicha función.

4. El siguiente paso Finalize.

- a. se usa para realizar aquellas transformaciones que se precisen sobre la salida final de Reduce. La signatura de la función es:

```
function /*object*/ FinalizeCode(key, value) { }
```

- b. La función Finalize toma cada par clave-valor, y emite un objeto. La salida de Finalize para todas las claves se inserta en una colección, y esta colección es el resultado del proceso MapReduce. Se puede dar el nombre que se desee, y si se deja sin especificar, MongoDB asigna un nombre de colección.



Finalize can be used to transform the output of each Key's reduced value. The result is added to the output collection.

- c. En nuestro ejercicio usaremos *Finalize* para encontrar la dos ciudades más próximas en cada país.
5. Por último, ejecutamos el comando mapReduce usando [db.runCommand](#)(comando). Cuya sintaxis general para mapReduce es:

```
db.runCommand (
    {
        mapReduce: <collection>,
        map: <function>,
        reduce: <function>,
        finalize: <function>,
        out: <output>,
        query: <document>,
        sort: <document>,
        limit: <number>,
        scope: <document>,
        jsMode: <boolean>,
        verbose: <boolean>
    }
)
```