Introdução ao MongoDB

Operações básicas

Elaine Naomi Watanabe

elainew@ime.usp.br

http://www.ime.usp.br/~elainew/

Profa. Dra. Kelly Rosa Braghetto

Departamento de Ciência da Computação – Instituto de Matemática e Estatística
Universidade de São Paulo

Setembro/2015

Este material é baseado:

- Na documentação do MongoDB [MongoDB 2015b]
- Nas aulas do curso M101P - MongoDB for Developers [University 2015b]
- Nas aulas do curso M101J - MongoDB for Java Developers [University 2015a]
- Nas aulas do curso M102 - MongoDB for DBAs [University 2015c]
- No livro MongoDB - Construa novas aplicações com novas tecnologias de Fernando Boaglio [Boaglio 2015]

Introdução

O que é o MongoDB?

- Banco de dados não-relacional de documentos JSON (BSON)
- Open-source (`https://github.com/mongodb/mongo`)
- Possui um esquema flexível (*schemaless*)
- Sua API de consulta é baseada em JavaScript
- Seu nome vem do inglês *humongous* ("gigantesco")

O que é JSON?

JSON [json.org 2015]

- *JavaScript Object Notation* [json.org 2015]
- Notação criada para o armazenamento e a troca de dados
- Baseada em JavaScript, mas armazenada como texto simples (*plain text*)
- Fácil de ser:
  - lida e escrita por humanos
  - analisada e gerada por máquinas
- É basicamente uma chave + um valor
  Ex.: {"chave":"valor"}

Tipos de dados

O padrão JSON aceita como valor os seguintes **tipos de dados**:

- strings
- numbers
- booleans(true/false)
- null
- arrays
- objects/documents

Exemplo de JSON

```
{
    "nome":"João", //string
    "idade":21, //number
    "eleitor":true, //boolean
    "escolaridade": null, //null
    "hobbies":["tênis","xadrez"], //array
    "endereço":{
         "cidade":"São Paulo",
         "estado":"SP"
    } //object/document
}
```

Para visualizar arquivos json: `http://jsonlint.com/`

BSON (Binary JSON)

BSON [BSON 2015] é uma representação binária do JSON, utilizada pelo MongoDB para armazenar seus documentos.
É utilizado devido a:

- **Rápida escaneabilidade** (*fast scanability*), ou seja, torna possível varrer um documento procurando um valor rapidamente.
- **Novos tipos de dados** (*Date*, *ObjectId*, *Binary Data*)

Todo documento tem um campo chamado **_id** obrigatoriamente. Por padrão, é um campo do tipo *ObjectId* [MongoDB 2015c], mas pode ser definido como qualquer outro tipo de dado.

Características do MongoDB

O MongoDB foi projetado para ser **escalável**.

Por isso, algumas funcionalidades **NÃO** foram incorporadas ao seu sistema, como:

- Junções (JOINs) entre coleções
- Transações ACID [Elmasri and Navathe 2014]

Características do MongoDB

E como viver em um mundo sem junções e transações?

- **Aninhando** documentos (*embedded documents*)
- Criando **links artificiais** (textual, não é uma chave estrangeira)
- Lembrando que as operações de escritas são **atômicas** no nível de **um único documento** [MongoDB 2015a], mesmo quando uma operação modificar vários documentos.

Atomicidade: as outras requisições para irão recuperar o documento já modificado completamente ou antes da modificação, nunca durante.

Problemas?

O que pode acontecer com os nossos dados?

- A consistência pode ser eventual
- Pode haver duplicação e inconsistência entre campos usados como links de documentos

Em compensação, ganhamos desempenho e escalabilidade.
A **aplicação** torna-se responsável pela integridade dos dados duplicados.

Principais Componentes do MongoDB

O MongoDB é composto (basicamente) por:

- **mongo**: **cliente** do MongoDB (Mongo Shell)
- **mongod**: **servidor** do MongoDB
- mongod --replSet "nomeReplica": servidor de réplica do MongoDB
- mongod --configsvr: serviço de configuração do particionamento de dados (ele processa as consultas do cliente e determina a localização dos dados)
- mongos: serviço de roteamento de consultas do cliente para o serviço de configuração do cluster

Lista de drivers:

`http://docs.mongodb.org/ecosystem/drivers/`

Relembrando...

- **Sharding** (Particionamento de dados): É um dos métodos utilizados para a obtenção de escalabilidade horizontal. O conjunto de dados é dividido em **diferentes partes** (*shards*) e essas partes são distribuídas em servidores distintos.
- **ReplicaSet** (Conjunto de Réplicas): É um grupo de servidores (mongod) que hospedam/armazenam o **mesmo conjunto de dados**.

MongoDB x SQL

MongoDB	SQL
Documentos	Linhas
Campos/chaves/atributos	Colunas
Coleções	Tabelas

Mongo Shell

Conectando ao servidor

```
$ mongo
```

Caso o banco de dados não seja especificado na conexão, o mongo sempre se conectará ao banco de dados *test*.

Além disso, por padrão, a **autenticação** do MongoDB **não vem habilitada**.

Conectando ao servidor da disciplina MAC0439

```
$ mongo db_NUSP --host data.ime.usp.br --port 25134
-u u_NUSP -p
```

A senha é o seu NUSP.

No banco de dados da disciplina, a autenticação está habilitada. Por isso, precisamos **especificar** o banco de dados de autenticação (normalmente o admin) ou um banco de dados que tenhamos acesso.

Trocando a senha

```
use db_NUSP;
db.runCommand(
    { updateUser: "u_NUSP",
      pwd: "new_passwd"
    }
)
```

Vendo seus privilégios

```
use db_NUSP;
db.runCommand(
  {
    usersInfo:"u_NUSP",
    showPrivileges:true
  }
);
```

Comando	Descrição
help	Exibe os principais comandos do Mongo Shell
show dbs	Lista todos os BDs do servidor mongod
show collections	Lista todas as coleções de um BD
use <db_name>	Conecta ao BD especificado
rs.help()	Exibe os comandos relacionados ao *ReplicaSet*
sh.help()	Exibe os comandos relacionados a *Sharding*

O **Help** pode ser acessado em diferentes níveis no Mongo Shell:

```
//geral
help;
//nível do banco de dados
db.help();
//nível de uma coleção
db.<nome_coleção>.help();
//nível do comando find()
db.<nome_coleção>.find().help();
//definição de uma função
db.<nome_coleção>.find
```

<nome_coleção> é o nome da coleção dentro do banco de dados selecionado com use <nome_banco_de_dados>

A API do MongoDB é baseada em JavaScript. Por isso, podemos executar scripts js no Mongo Shell...

```
for(i=0; i < 3; i++){
 print ("hello, " + i);
}
```

Podemos definir variáveis...

```
var test = "abc";
print (test);
test
```

Manipular dicionários (documentos JSON)...

```
dict = {"a":1, "b":2};
dict
dict.a
dict["a"]
w = "a"
dict[w]
```

Criando um banco de dados chamado my_database
(se o usuário tiver o privilégio para isso)

```
use my_database
```

Exibindo o banco de dados em uso:

```
//ponteiro para o banco de dados atual
db;
```

Listando todos os banco de dados do servidor (conforme o privilégio do usuário):

```
show dbs
```

Para criar coleções, basta inserir um documento à nova coleção. O comando abaixo cria uma nova coleção chamada minha_nova_colecao com o documento {a:1}.

```
db.minha_nova_colecao.save({a:1})
```

Ou podemos criar apenas a coleção:

```
db.createCollection("minha_nova_colecao");
```

Podemos organizar as coleções em grupos, criando namespaces para as coleções e utilizando a notação ponto (*dot notation*). Exemplo:

```
db.mac0439.alunos.insert({"a":1});
db.createCollection("mac0439.alunos");
db.createCollection("mac0439.topicos");
```

Para localizar o documento inserido, podemos usar os comando .find() e .findOne():

```
db.minha_nova_colecao.find({a:1});
db.minha_nova_colecao.findOne();
```

Listando todas as coleções de um banco de dados:

```
show collections;
```

Excluindo uma coleção:

```
//exclui my_collection
db.my_collection.drop();
```

Excluindo um banco de dados:

```
use my_db;
//exclui my_db
db.dropDatabase();
```

Restrições

- O nome de um banco de dados **NÃO** pode:
  - conter os seguintes caracteres no UNIX: /\ . "$
  - conter os seguintes caracteres no WINDOWS: /\. "$*<>:|?
  - conter caracteres nulos ou ser vazio
  - conter mais de 64 caracteres
- O nome de uma coleção **NÃO** pode:
  - conter o caracter $ ou ser uma string vazia
  - começar com o prefixo *system.* (ele é reservado para o sistema)
- O nome de um campo (chave) **NÃO** pode:
  - conter pontos (.) ou ser uma string vazia
  - não pode começar com $

Restrições

Caso seja necessário usar o $ como nome de um atributo, veja:
`http://docs.mongodb.org/master/faq/developers/#faq-dollar-sign-escaping`.

Cada documento JSON/BSON pode ter no **máximo 16MB**.
Caso ultrapasse esse tamanho, utilize o GridFS:
`http://docs.mongodb.org/manual/core/gridfs/`.

Veja mais restrições em:
`http://docs.mongodb.org/master/reference/limits/`.

Saindo do Mongo Shell:

```
exit;
```

Operações básicas

CRUD	SQL	MongoDB
Create	Insert	Insert
Read	Select	Find
Update	Update	Update
Delete	Delete	Remove

Não existe uma linguagem separada para descrever as operações de CRUD no MongoDB.

As operaçõs existem como métodos/funções dentro da API.

Cada documento deve conter um id único associado, especificado pelo campo **_id**.

Caso o usuário não especifique um valor para esse campo, ele é gerado automaticamente pelo MongoDB, definido como um ObjectId().

O MongoDB cria também um índice para o campo **_id**, a fim de tornar as consultas mais eficientes.

Inserindo um documento

Protótipo: db.<nome_coleção>.insert(<json>)

```
use db_NUSP;
db.pessoas.insert({
"nome":"Maria",
"profissão":"Estudante"
});

db.pessoas.insert({
"nome":"Beatriz",
"profissão":"Estudante"
});
```

Outra forma de inserir um documento:

```
use db_NUSP;
doc = {"nome":"Ana", "profissão":"Professora"}
db.pessoas.insert(doc)
```

Explorando a ausência de esquema

```
db.pessoas.insert({"a":1});
db.pessoas.insert(
  {"frutas":
    ["maçã", "banana", "mamão"]
  });
db.pessoas.insert({"alunos":["João", "Carlos"]});
db.pessoas.insert(
{"alunos":[
  {"nome":"Maria","profissão": "Estudante"},
  {"nome":"Beatriz","profissão": "Estudante"}
]});
```

Podemos especificar um _id (que será único na coleção):

```
db.pessoas.insert({_id:1, a:1})
```

Localizando documentos

A condição, passada como parâmetro, é um documento JSON.

```
//localiza todos
db.pessoas.find();
db.pessoas.find({});

//localiza conforme um critério de busca
db.pessoas.find({"nome":"Maria"})
db.pessoas.find({"nome":"Maria",
"profissão":"Estudante"})
```

Todos os parâmetros passados para as funções da API são documentos JSON.

Localizando documentos

Projeção: selecionando atributos/campos a serem exibidos no resultado.

```
//localiza conforme um critério de busca
db.pessoas.find({"nome":"Maria"},
{"nome":1});
db.pessoas.find({"nome":"Maria"},
{_id:0, "nome":1});
db.pessoas.find({"nome":"Maria"},
{_id:false, "nome":true});
```

Todos os parâmetros passados para as funções da API são documentos JSON.

E como localizar documentos aninhados e arrays?
Temos, por exemplo:

```
{"frutas":["maçã", "banana", "mamão"]}

{"alunos":[
  {"nome":"Maria","profissão": "Estudante"},
  {"nome":"Beatriz","profissão": "Estudante"}
]}
```

Usaremos a notação ponto (*dot-notation*):

```
db.pessoas.find({"frutas":"maçã"});
db.pessoas.find({"alunos.nome":"Beatriz"});
```

.findOne() retorna apenas o primeiro documento da lista de documentos que seriam o resultado. É equivalente a .find().limit(1).

```
//retornando o primeiro documento de uma coleção
db.pessoas.findOne();
db.pessoas.find().limit(1);

//adicionando uma condição
db.pessoas.findOne({"profissão":"Estudante"});
db.pessoas.find({"profissão":"Estudante"
}).limit(1).pretty();
db.pessoas.find({"profissão":"Estudante"
}).limit(1).toArray();
```

```
//projeção do resultado
db.pessoas.findOne({"profissão":"Estudante"},
{nome:true});
db.pessoas.find({"profissão":"Estudante"},
{nome:true}).limit(1).pretty();
db.pessoas.find({"profissão":"Estudante"},
{nome:true}).limit(1).toArray();
```

Criando uma nova coleção com mais documentos:

```
for(i=0;i<1000;i++){
atividades=["prova","projeto", "exercícios"];
for(j=0;j<3;j++){
db.notas.insert({"estudante":i,
"tipo": atividades[j],
nota: Math.round(Math.random()*100)});
}
}
```

Pronto! 3000 documentos inseridos!

Cursores

Um cursor permite que você navegue por um conjunto de dados.

Permite que limitemos, por ex., a lista de documentos em uma consulta:

```
//limitando
db.notas.find().limit(5)
//exibindo de modo "legível"
db.notas.find().limit(5).pretty();
db.notas.find().limit(5).toArray();
```

É possível exibir os valores de um cursor, um a um:

```
var myCursor = db.notas.find( { tipo: 'prova' } );

while (myCursor.hasNext()) {
   printjson(myCursor.next());
}

//ou
myCursor.forEach(printjson);
```

Outros exemplos:

```
var myCursor = db.notas.find( { tipo: 'prova' } );
var documentArray = myCursor.toArray();
var myDocument = documentArray[3];
myDocument;
myCursor.toArray() [3]; //equivalente
```

Podemos manipular o cursor da nossa consulta, exibindo, por ex., os resultados de 5 em 5 documentos.

Manualmente fazendo isso:

```
//limitando
db.notas.find().limit(5);
db.notas.find().skip(5).limit(5);
db.notas.find().skip(10).limit(5);
```

Mais métodos: `http://docs.mongodb.org/manual/reference/method/js-cursor/`

Operadores

- **$gt**: verifica se um atributo é maior que um valor
- **$gte**: verifica se um atributo é maior ou igual que um valor
- **$lt**: verifica se um atributo é menor que um valor
- **$lte**: verifica se um atributo é menor ou igual que um valor
- **$exists**: verifica se um atributo existe
- **$type**: verifica se existe um atributo de um tipo determinado
- **$regex**: se o atributo corresponde à expressão regular
- **$or**: compara duas condições com o operador ou
- **$and**: compara duas condições com o operador and
- **$in**: verifica se um atributo contém um dos valores de um array
- **$all**: verifica se um atributo contém todos os valores de um array

Operadores $gt, $gte, $lt, $lte

**$gt, $gte, $lt, $lte**: {campo: {$operador:valor}}

```
db.notas.find({nota:{$gt:95})
db.notas.find({nota:{$gte:95, $lte:98},
tipo:"exercicio"})
db.notas.find({score:{$lt:95})
```

Comparando strings com $gt, $gte, $lt, $lte

Suponha que tenhamos a seguinte coleção:

```
db.pessoas.insert({nome:"Ana"});
db.pessoas.insert({nome:"Beatriz"});
db.pessoas.insert({nome:"Carlos"});
db.pessoas.insert({nome:"Daniel"});
db.pessoas.insert({nome:"Érica", "idade":20,
"profissão":"estudante"});
db.pessoas.insert({nome:"Fernando", "idade":30,
"profissão":"web developer"});
db.pessoas.insert({nome:42});
```

Caso use os comparadores para strings, eles só analisarão chaves do valores do tipo string, ignorando os números.
Os valores são ordenados conforme o padrão UTF-8.

```
db.pessoas.find({nome:{$lt:"D"}}); //de A a C
db.pessoas.find({nome:{$lt:"D", $gt:"B"}}); //C
```

Operadores $exists e $type

$exists: retorna os documentos que contenha o atributo especificado.

```
db.pessoas.find({"profissão":{$exists:true}});
```

$type: retorna os documentos cujo atributo seja do tipo especificado (de acordo com a notação BSON).

```
db.pessoas.find({nome:{$type:2}});
```

Número	Tipo
1	Double
2	String
3	Object
4	Array
5	Binary Data
6	Undefined (Deprecated)
7	Object id
8	Boolean
9	Date
10	Null

Ver lista completa: `http://docs.mongodb.org/manual/reference/operator/query/type/`

Lembrando que não existe, em JSON, chave sem um valor (mesmo que nulo) e que um documento será sempre composto por uma chave e um valor. Não esqueçam de colocar o símbolo {}.

Exemplos inválidos:
{"chave"}, {"chave":operador:10}, "chave"

Exemplos válidos:
{"chave":null},{"chave":"conteudo"}, {"chave":{operador:10}}

$regex: retorna os documentos que satisfazem a expressão regular

```
// nomes que possuam a letra a em qualquer posição
// Em SQL... WHERE (nome like %a%)
db.pessoas.find({"nome":{$regex:"a"}});

// nomes que terminem com a
// Em SQL... WHERE (nome like %A)
db.pessoas.find({nome:{$regex:"a\$"}});

// nomes que comecem com A
// Em SQL... WHERE (nome like A%)
db.pessoas.find({nome:{$regex:"^A"}});
```

Tutoriais sobre REGEX:

`http://aurelio.net/regex/guia/`

`http://regexone.com/`

$or: retorna todos os documentos que atendam a uma duas condições.

```
db.estoque.find({$or: [{qtde: {$lt: 20}},
 {preço: 10}]})
```

$and: retorna todos os documentos que atendam às duas condições.

```
//explicitamente (reparem no [])
db.pessoas.find( { $and: [ { nome: { $gt: "C"} },
{ nome: { $regex: "a"} } ] } );
//implicitamente
db.pessoas.find( { nome: { $gt: "C", $regex: "a"}});

//explicitamente (reparem no [])
db.notas.find( { $and: [ { nota: { $gt: 50} },
 { tipo: "prova"} ] } );
//implicitamente
db.notas.find( { nota: { $gt: 50}, tipo:"prova"}});
```

Mas, tome cuidado! Observe essa seguinte consulta:

```
db.notas.find({nota:{ $gt : 50 }, nota:{$lt : 60}});
```

Um documento cuja nota seja 40, será retornado?

Combinando **$or** e **$and**:

```
db.estoque.find( {
    $and : [
        { $or : [{preço: 0.99},{preço: 1.99}]},
        { $or : [ {a_venda : true},
         { qtde:{$lt:20}}]}
    ]
} )
```

Considere os seguintes documentos:

```
db.pessoas.insert({name:"João",
favoritos:["refrigerante","pizza"]});
db.pessoas.insert({name:"Pedro",
favoritos:["sorvete","pizza"]});
db.pessoas.insert({name:"Luís",
favoritos:["pizza","pipoca", "refrigerante"]});
```

Como poderíamos fazer buscas em favoritos (array)?

$in: retorna todos os documentos cujo atributo contenha pelo menos um dos valores especificados no array

$all: retorna todos os documentos cujo atributo contenha todos os valores especificados no array

```
// todos os documentos que tenham pizza como favorito
db.pessoas.find({favoritos:"pizza"});
// retorna João, Pedro e Luís

// todos os documentos que contenham pizza
// OU refrigerante como favorito
db.pessoas.find({favoritos:
{$in:["pizza","refrigerante"]}});
// retorna João, Pedro e Luís

// todos os documentos que contenham pizza
// E refrigerante como favorito
db.pessoas.find({favoritos:
{$all:["pizza","refrigerante"]}});//Pedro e Luís
```

Outros operadores:

- **$nin**: seleciona documentos em que o valor do campo especificado não está no array ou documentos que não possuam o campo
- **$eq**: seleciona documentos em que o valor do campo é igual ao valor especificado.
- **$ne**: seleciona documentos em que o valor é diferente (*not equal*) do valor especificado
- **$not**: operador NOT, seleciona documentos que não satisfazem a expressão especificada, incluindo documentos que não contenham o atributo especificado

- **$nor**: seleciona os documentos que não satisfazem a lista de condições.
  Por exemplo: db.estoque.find({ $nor: [{ preço: 1.99 }, { a_venda: true }]})
  Retorna os documentos:
  - cujo preço não seja igual a 1.99 e atributo a_venda não seja true;
  - cujo preço não seja igual a 1.99 e que não contenham o atributo a_venda;
  - não contenham o atributo preço e o atributo a_venda seja diferente de true;
  - não contenham o atributo preço ou a_venda.

Mais informação em: `https://docs.mongodb.org/manual/reference/operator/query-comparison/`

Protótipo: db.<collection>.find(<condição>).sort(<ordenação>)

<ordenação>: {nomecampo: direção}
Direção:

- ASC: 1
- DESC: -1

```
db.notas.find().sort({"estudante":1})
db.notas.find().sort({"estudante":-1})
db.notas.find({tipo:"prova", nota:{$gte:50}},
{_id:0, estudante:1}).sort({"estudante":-1})
```

.count(): conta o número de documentos retornados na consulta

```
db.pessoas.count()
db.pessoas.find().count()
db.pessoas.find({"nome":"Ana"}).count()
db.pessoas.find({"profissão":"Estudante"}).count()
```

.distinct(<atributo>): retorna os valores distintos para um atributo. Por exemplo, temos:

```
db.estoque.insert({ "_id": 1, "prod": "A",
"item": { "qtde": "111", "color": "red" },
"sizes": [ "P", "M" ] });
db.estoque.insert({ "_id": 2, "prod": "A",
"item": { "qtde": "111", "color": "blue" },
"sizes": [ "M", "G" ] });
db.estoque.insert({ "_id": 3, "prod": "B",
"item":{"qtde":"222", "color":"blue"},"sizes":"P"});
db.estoque.insert({ "_id": 4, "prod": "A",
"item": { "qtde": "333", "color": "black" },
"sizes": [ "P" ] });
```

E queremos saber o nome dos produtos em estoque.
Usando distinct:

```
db.estoque.distinct("prod");
```

Atualização de documentos

```
db.<nome_coleção>.update(
    <condição>,
    <doc_completo_ou_parcial>,
    {
      upsert: <boolean>,
      multi: <boolean>,
      writeConcern: <document>
    }
)
```

Atualizando documentos

Podemos atualizar um documento inteiro ou parte de um ou mais documentos. Por padrão, uma operação de atualização modifica **apenas o primeiro documento** que ela encontrar (que satisfaça a condição de busca).

writeConcern: descreve o nível de garantia que o MongoDB deve fornecer para cada operação para que esta seja considerada "executada com sucesso". Define, por exemplo, o número de réplicas que devem responder para o servidor mestre , afirmando que receberam a atualização de um documento. Ex: writeConcern:{w:'majority'} ou writeConcern: { w: 2, wtimeout: 5000 }.

Upsert: permite definir se um documento será adicionando à coleção, caso nenhum dos documentos existentes satisfaçam à condição de busca.

Multi: permite controlar se o update (atualização) será aplicado a apenas um documento ou a todos que satisfaçam à condição de busca.

Por padrão, upsert e multi são false. O atributo _id é o único que não pode ser sobreescrito.

Suponha que nossa coleção aula tem os seguintes documentos:

```
db.aula.insert({_id:1, "a":1,"b":2});
db.aula.insert({_id:2, "a":2,"b":1});
```

E executamos:

```
db.aula.update({"a":2},{"a":3});
```

O que acontece? Será que atualiza só o atributo a?
O documento {"a":2,"b":1} será totalmente reescrito.
O novo documento será {_id:2,"a":3}.

Para atualizar parcialmente um documento, podemos criar uma variável no Mongo Shell para reescrever ou adicionar um atributo de um documento.

```
var myObj = db.aula.findOne({a:1});
myObj.c = 123; //adicionou um novo campo
db.aula.update({_id:myObj._id},myObj);

//ou
var cursor = db.aula.find({a:1}).limit(1);
var obj = cursor.next();
db.aula.update({_id:obj._id},obj);
```

Upsert: insere um novo documento (com os atributos do update) quando o critério de busca não for satisfeito por nenhum documento da coleção.

```
db.pessoa.update(
   { nome: "André" },
   {
      nome: "André",
      classificação: 1,
      nota: 100
   },
   { upsert: true }
)
// se nenhum documento for encontrado, será inserido
// {"nome":"André", "classificação":1, "nota":100}
```

Multi: se true, permite que todos os documentos da coleção, que satisfaçam a condição, sejam alterados.

```
db.notas.update({nota:{$lt:50}},
{$set:{"recuperação":true}}, {multi:true});
```

.save(): é uma função do mongo shell apenas, um atalho para upsert. Caso o documento não exista, insira-o. Senão, atualize-o.

```
var myObj = db.aula.findOne(); //um documento
myObj.y = 400; //adicionou um novo campo
db.aula.save({_id:myObj._id},myObj);
```

Para ver a definição, digite:

```
db.aula.save
```

Outras formas de update

Para atributos simples:

- **$set**: Define um valor para um atributo específico
- **$unset**: Remove um atributo específico de um documento
- **$inc**: Incrementa o valor de um atributo
- **$rename**: Renomeia um atributo

Para arrays:

- **$addToSet**: Adic. elemento a um array se não existir
- **$pop**: Remove o primeiro ou último item de um array
- **$pull**: Remove de um array os valores especificados.
- **$pullAll**: Remove de um array todos os valores especificados.
- **$push**: Adic. um elemento a um array (mesmo se ele já existir)
- **$pushAll**: Adic. todos os elemento especificados a um array (*deprecated*)

Outros operadores: `http://docs.mongodb.org/manual/reference/operator/update/`

Operador $set e $unset

Modificando apenas um atributo de um documento com **$set**:

```
//adicionando
db.aula.update({_id:2},{$set:{d:100}});
//alterando
db.aula.update({_id:2},{$set:{d:200}});
```

Removendo um atributo com **$unset**:

```
//removendo
db.aula.update({_id:2},{$unset:{d:""}});
```

Incrementando o valor de um atributo com **$inc**:

```
db.funcionarios.update(
{"_id" : 6124109} ,
{ $inc: { "salario" : 1500}});

//Aumento de 1500 reais para o _id: 6124109! :)
```

$rename é utilizado para renomear atributos.
É possível passar uma lista de atributos a serem renomeados.

```
db.students.insert({_id:1, name:"Ana",
nickname: "Aninha", celular:"99999-9999"});
db.students.update( { _id: 1 },
{$rename: {'nickname': 'alias', 'cell': 'mobile'}});
```

E se precisamos modificar um elemento de um array?

```
db.aula.insert({_id:3, a:[1,2,3,4]});
```

Podemos especificar a posição do array com a notação ponto.

```
//a.2 é o terceiro elemento
db.aula.update({_id:0},{$set:{"a.2":5}});

// E teríamos como resultado:
// {_id:3,a:[1,2,5,4]}
```

Outras formas de manipular arrays

Usando o **$addToSet**:

```
db.aula.update(
   { _id: 1 },
   { $addToSet: {letras: [ "c", "d" ] } }
);
```

O que acontece se executarmos duas vezes?
O documento {_id:1, letras:["c","d"]} não será modificado novamente.

E se usássemos o **$push**, seria:

```
db.aula.update(
    { _id: 1 },
    { $push: {letras: "e"}}
);
```

O que aconteceria se executássemos duas vezes?
O documento _id:1 seria:
{_id:1, letras:["a","b","c", "d","e","e"]}

E se quiséssemos adicionar mais de um elemento ao array de uma vez?

```
db.aula.update({_id:1},
{$pushAll:{letras:["f","g","h"]}});
```

O **$pushAll** também **NÃO** verifica se o elemento já existe.

Para evitar a inserção de elementos repetidos em um update, podemos usar o operador **$each** + **$addToSet**.

```
db.aula.update(
   { _id: 2 },
   { $addToSet: {letras:{$each:[ "f", "g", "h" ]}}}
 )
```

Removendo um valor de um array com **$pull**. Por exemplo, temos a seguinte coleção:

```
db.lojas.insert({
    _id: 1,
    frutas: [ "maçãs",
      "peras",
      "laranjas",
      "uvas",
      "bananas" ],
    legumes: [ "cenouras",
      "aipos",
      "abóboras",
      "cenouras" ]
});
```

```
db.lojas.insert({
    _id: 2,
    frutas: [
     "ameixas",
     "kiwis",
     "laranjas",
     "bananas",
     "maçãs" ],
    legumes: [
     "brócolis",
     "abobrinhas",
     "cenouras",
     "cebolas" ]
});
```

Se executássemos...

```
db.lojas.update(
    {},
    {$pull: {frutas: {$in: ["maçãs","laranjas"]},
    legumes: "cenouras" } },
    {multi: true}
)
```

Seria removido de todos os documentos todos os elementos "maçãs"e "laranjas"do array de frutas e todos os elementos "cenouras"do array de legumes.

$pushAll: remove todos os elementos que estejam dentro do array ou documento especificado.

```
db.notas.insert({ _id: 1, notas:[0,2,5,5,1,0]});
db.notas.update({ _id: 1},{$pullAll:{notas:[0,5]}});
```

Removendo elementos de um array com **$pop**:

Podemos remover um elemento do final:

```
db.aula.update({_id:1},{$pop:{letras:1}})
```

ou do começo:

```
db.aula.update({_id:1},{$pop:{letras:-1}})
```

Excluindo documentos

Protótipo: db.<nome_coleção>.remove(<condição>)

Ao contrário do update, ele apaga **TODOS** os documentos que satisfaçam a condição.

```
db.pessoas.remove({_id:100})
db.pessoas.remove({}) //apaga tudo!
```

Dica: teste sempre a condição e veja quantos documentos são retornados.

```
db.pessoas.find({_id:100})
db.pessoas.find({_id:100}).count()
```

Veja mais em:

https://www.mongodb.com/lp/misc/quick-reference-cards
http://www.tutorialspoint.com/mongodb
http://docs.mongodb.org/manual/reference/sql-comparison/
http://docs.mongodb.org/manual/reference/
sql-aggregation-comparison/
https://www.mongodb.com/compare/mongodb-mysql

Material em:
https://mega.nz/#F!OUcgXLaA!iOaHOInXGzh-1lw5vFiaFA

Referências I


Boaglio, F. (2015).
MongoDB: Construa novas aplicações com novas tecnologias.
Casa do Código.


BSON (2015).
Bson.
`http://bsonspec.org`.
Data de acesso: 04/07/2015.


Elmasri, R. and Navathe, S. B. (2014).
Fundamentals of database systems.
Pearson.

Referências II


json.org (2015).
Introducing json.
http://json.org.
Data de acesso: 04/07/2015.


MongoDB (2015a).
Atomicity and transactions.
http://docs.mongodb.org/v3.0/core/write-operations-atomicity/.
Data de acesso: 04/07/2015.

Referências III


MongoDB (2015b).
The mongodb 3.0 manual.
http://docs.mongodb.org/manual.
Data de acesso: 04/07/2015.


MongoDB (2015c).
Objectid.
http://docs.mongodb.org/manual/reference/object-id/.
Data de acesso: 04/07/2015.

Referências IV

University, M. (2015a).
M101j - mongodb for java developers.
https://university.mongodb.com/courses/M101J/about.

Data de acesso: 04/07/2015.

University, M. (2015b).
M101p: Mongodb for developers.
https://university.mongodb.com/courses/M101P/about.

Data de acesso: 04/07/2015.

Referências V


University, M. (2015c).
M102 - mongodb for dbas.
https://university.mongodb.com/courses/M102/about.
Data de acesso: 04/07/2015.