

NOSQL - MONGODB

MARCIO JASINSKI

APICE

1. INTRODUÇÃO

...

INTRODUÇÃO

- Banco de dados “orientado a documentos”. Não há tabelas e regras de integridade. Além disso, os relacionamentos são diferentes...
- O modelo foi pensado para atender demandas de escala e dados complexos
- Um banco orientado a documentos, remove o conceito de linhas e colunas por um modelo de “documento”. Esse documento pode ter uma estrutura dinâmica e com subelementos e/ou arrays de dados.
- O conceito de documentos permite representar dados complexos como hierarquias e estruturas relacionadas de forma “auto-contida” em um único registro.
- O modelo tem muita aderência com o que os desenvolvedores precisam no dia a dia. Em geral, o modelo de documento é a típica representação de uma estrutura computacional como objetos.

INTRODUÇÃO

- Uma das principais características é a ausência de um schema pré definido
 - Não é necessário informar tipos
 - Não existe limitação de tamanho para os campos
 - É simples adicionar ou remover campos
- Modelo ideal para experimentos/prototipação
- Esse é um dos principais benefícios para aplicações estão mudando bastante;
- A validação dos dados e estruturas fica para a aplicação

INTRODUÇÃO

- Uma das principais características do é seu modelo de escala, voltado para escala horizontal: pela adição de servidores (scale-out).
- Nascido com foco em commodity-servers, em geral não exige nenhum equipamento específico para novos nós no cluster.
- Como os documentos são “auto-contidos” é mais fácil espalhar os dados entre vários servidores.
- O MongoDB é projetado para fazer esse balanceamento da carga no cluster, redistribuindo os documentos automaticamente e redirecionando as requisições para cada grupo de máquinas.
- O ponto positivo, é que se a aplicação demandar mais escala, é mais fácil atender com mais nós no cluster
- O ponto negativo é que com o tempo se torna bem mais desafiador gerenciar centenas de nós de que apenas um banco de dados...
- Mas em geral, existem serviços que já oferecem o banco “as a service”.

FEATURES

- **CRUD** – Operações para manipulação de documentos (Create, Read, Update, Delete)
- **Indexing** – É possível indexar qualquer campo de qualquer documento
- **Aggregation** – Operação com dados para dados computados e usados em pipelines.
- **Special Collection types** – Suporte a coleções para dados que devem expirar ou de tamanho fixo (rotativo)
- **File Storage** – Capacidade de armazenar grandes arquivos e metadados
- **Replication** – Modelo master slave onde muitos slaves podem escalar para leitura.
- **Sharding** – Suporta replicação de coleções em “shards” (pedaços)

O QUE VOCÊ PERDE...

- Para quem utiliza um banco relacional, é comum sentir falta de elementos como:
- Possibilidade de fazer operações como “joins”
- Transações entre várias linhas com possibilidade de rollback
- Facilidade de ver os dados tabulados
- Operações sobre colunas
- Manipulação dos dados com SQL
- Formalismo (as vezes isso é bom)
 - **Exemplo de uma busca com coluna não existente VS chave não existente**

Conceitos Básicos

CONCEITOS - DOCUMENTO

- No centro de tudo para o MongoDB está o documento em formato JSON.
- Embora a noção de documento possa variar, o JSON é um padrão muito comum para diversas linguagens.
 - JSON - JavaScript Object Notion
 - Se tornou um padrão no transporte de dados
 - Modelo fácil de compreender
 - Composto por chaves e valores
 - Suporta elementos aninhados
 - Simples nos tipos: string, number, object, array, boolean

```
{  
  "banda" : "Iron Maiden",  
  "albuns" : [  
    "Iron Maiden",  
    "The Killers",  
    "The number of the beast"  
  ]  
}
```

CONCEITOS - DOCUMENTO

- No MongoDB, um documento segue o formato JSON onde:
 - As chaves são sempre strings
 - Qualquer caractere UTF-8 pode ser parte da chave exceto \0
 - Além disso, o ponto “.” e dolar “\$” são caracteres especiais
- O armazenamento não é em JSON, mas sim em BSON

CONCEITOS - DOCUMENTO

- Mongo é type-sensitive e case-sensitive. Por exemplo, os documentos abaixo são diferentes:

```
{"foo" : 3} {"foo" : "3"}
```

```
{"FOO" : 3} {"Foo" : 3}
```

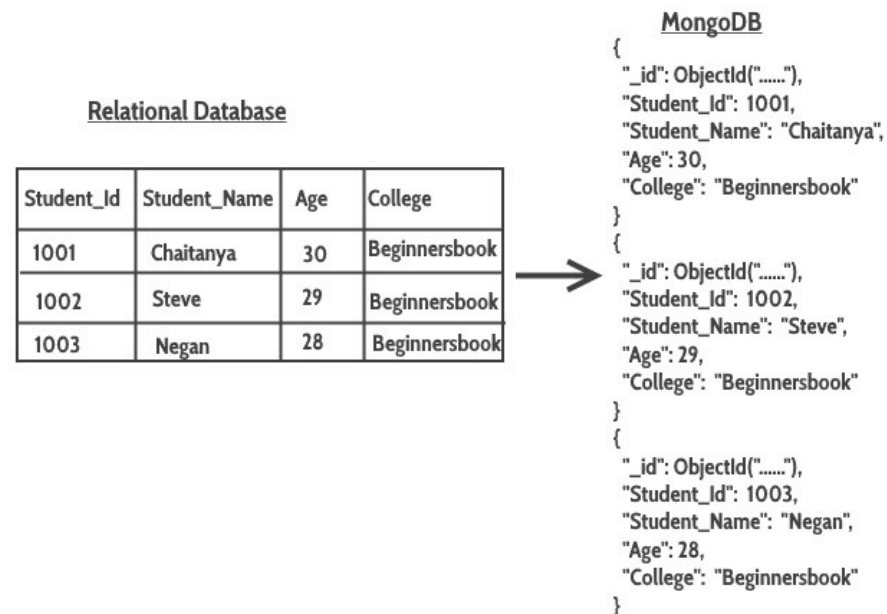
- Chaves não podem ser duplicadas, mas podem ocorrer em diferentes níveis

```
{  
  "greeting" : "Hello, world!",  
  "greeting" : "Hello, MongoDB!"  
}
```

```
{  
  "greeting" : "Hello, world!",  
  "langs": [{  
    "lang": "pt_BR",  
    "greeting" : "Olá, MongoDB!"  
  }]  
}
```

CONCEITOS - DOCUMENTO

- Um documento pode ser pensado como a linha de uma tabela em um banco relacional



- Os pares de chave e valor são ordenados, então `{"x" : 1, "y" : 2}` não é a mesma coisa que `{"y" : 2, "x" : 1}`.
- Mas a ordem dos campos não deve ser relevante para sua aplicação. Apenas não dependa disso

COLLECTIONS

- Uma coleção é um grupo de documentos. A coleção pode ser pensada como uma tabela em um banco relacional.
- A principal diferença é que coleções podem ter esquemas diferentes em cada documento.
- Ou seja, cada documento em uma mesma coleção pode ter os pares chave/valor que forem de fato relevantes;
- Esse modelo contrapõe um dos problemas com bases relacionais em tabelas esparsas que precisam ser remodeladas

Collection of runners

```
{“name” : “Jon Foo“, “age”: “27“, “track”: “5km“, “cpf”: “045.457.147-98“, “category”: “1B”},  
{“name” : “Doggy“, “age”: “6“, “track”: “5km”},  
{“name” : “Phill“, “age”: “22“, “track”: “15km“, “cpf”: “045.457.147-98“, “category”: “1A”}
```

COLLECTIONS

- **Se Podemos armazenar dados diferentes, por separar os dados em mais coleções?**
- Para administrar os dados ou mesmo para uma aplicação, a separação de coleções é essencial em termos de organização. Uma aplicação precisa “saber” que os dados que estão sendo retornados tem uma semântica, mesmo que a estrutura seja flexível.
- É mais rápido obter uma lista de objetos de uma coleção organizada do que filtrar o resultado por tipos.
- O agrupamento de objetos em uma coleção é um princípio de “data locality”, ou seja, foco em computar o que é preciso ao invés de transferir dados de forma desnecessária para uma aplicação.
- Quando há muitos documentos e campos, é preciso utilizar técnicas mais avançadas de busca, como índices. Os índices são alocados por coleção e por isso, é bem importante ter documentos que contenham os campos indexados.

COLLECTIONS

- Uma coleção é identificada por um nome, que deve ser padrão UTF-8 com poucas restrições:
- Não é permitido coleção com nome vazio ""
- Não pode conter o caractere \0 (null) pois isso é usada para o final da string
- Não deve começar com a palavra system, pois é reservado para uso interno do MongoDB como
 - system.users – coleção do Sistema que gerencia usuários do MongoDB
 - System.namespaces – coleção interna que tem dados dos databases
- Não podem conter o caractere \$, embora alguns drivers permitam, não utilizar pois existem coleções do MongoDB com esse padrão e pode gerar problemas. Use apenas se estiver acessando uma coleção interna do banco.

SUBCOLLECTIONS

- Uma convenção para organizar coleções é utilizando um namespace com o ponto “.”
- Por exemplo:
 - “blog.sites”, “blog.users” ,
 - “blog.dev.posts”, “blog.dev.users”,
 - “blog.ux.posts”, “blog.ux.users”
- O padrão é apenas organizacional e não garante nenhum tipo de relacionamento entre as coleções.
- Não existe nem mesmo obrigação de um namespace pai existir para a criação de um namespace filho.
- Subcoleções são um padrão adotado por ferramentas no ecossistema do MongoDB:
 - GridFS – Protocolo para armazenamento de arquivos grandes – utiliza subcoleções para metadados e conteúdo

DATABASES

- Assim como collection agrupam documentos, databases agrupam coleções.
- Uma simples instância de MongoDB pode ter diversos databases, cada um com zero ou mais coleções
- A necessidade de um database existe pois cada database tem permissões específicas e será armazenada de forma separada. Isso permite que aplicações diferentes ou mesmo modelos multitenant utilizem databases separados.
- Cada database é identificado pelo seu nome, novamente UTF-8 onde
 - String vazia "" não é permitida
 - Deve ser ASCII - Não pode ter nenhum dos seguintes caracteres /, \, ., ", *, <, >, :, |, ?, \$, (a single space), or \0 (the null character
 - Databases são case-sensitive, mesmo em filesystems que não sejam case-sensitive (padrão é usar apenas minúsculas)
 - Nome do database não pode ultrapassar 64 bytes

DATABASES

- O nome do database é exatamente o mesmo que você verá no filesystem
- Existem alguns databases que são do Sistema e eventualmente podem ser acessados:

- *admin*

O database utilizado pelo MongoDB para autenticação master. Alguns comandos do banco precisam rodar através desse database, como a listagem de todos os database do banco.

- *local*

Database usado para armazenar coleções locais, que não devem ser replicadas.

- *config*

Utilizado para compartilhar as configurações compartilhadas quando sharding está configurado.

DATABASES

- É possível concatenar o nome do database com a collection para ter o nome completo (namespace) do recurso. Por exemplo: `cms.blog.posts` para o database: `cms` e collection: `blog.posts`.
- Namespaces são limitados a 121 bytes em tamanho.

MongoDB

Database portal

Collection blog

Document: dev

{fields}

Document: ux

{fields}

Collection blog.posts

Document: node

{fields}

Document: python

{fields}

Collection blog.users

Document: jon

{fields}

Document: mary

{fields}

Database chatbot

Collection crm

Document: talk1

{fields}

Document: talk2

{fields}

Collection erp

Document: talk1

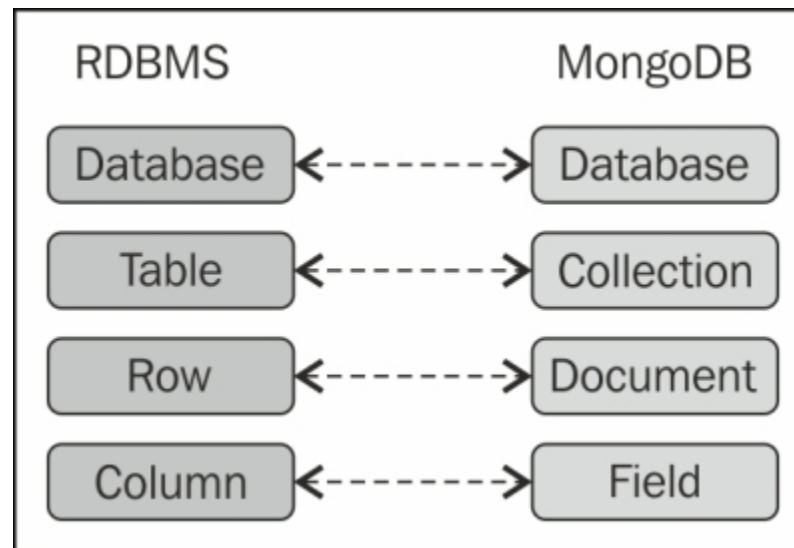
{fields}

Document: talk2

{fields}

RDBMS <-> MONGODB

- Há uma distância grande entre as funções de ambos, mas a comparação é quase inevitável...
- Uma visão que ajuda quem domina o modelo de SQL é entender as estruturas quase “equivalentes”
- Mas jamais use um banco de documentos como um banco relacional...



INSTALANDO O MONGODB

- Windows

```
iwr -useb get.scoop.sh | iex  
scoop install mongodb
```

- Linux

```
wget https://fastdl.mongodb.org/linux/mongodb-linux-x86\_64-ubuntu1604-4.2.3.tgz  
tar -zxvf mongodb-linux-x86\_64-ubuntu1604-4.2.3.tgz
```

- Pelo site:

<https://www.mongodb.com/download-center/community>

STARTING MONGODB

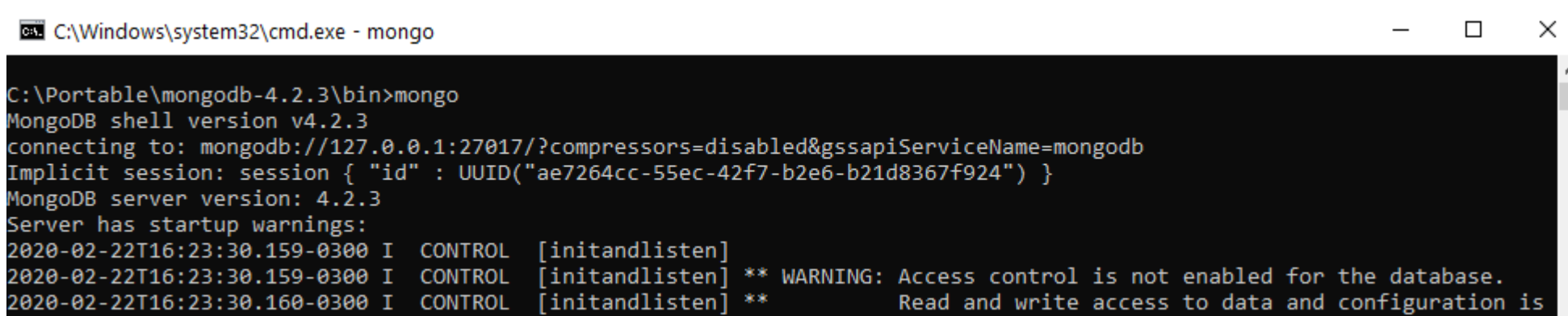
- A instalação do MongoDB é bastante simples:
 - Existem pacotes prontos para Windows, Linux e Mac
 - Mas basta descompactar um MongoDB para seu SO e iniciar o binário **mongod**

```
C:\Windows\system32\cmd.exe - mongod.exe

C:\Portable\mongodb-4.2.3\bin>mongod.exe
2020-02-22T16:23:29.251-0300 I CONTROL [main] Automatically disabling TLS 1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'
2020-02-22T16:23:29.262-0300 I CONTROL [initandlisten] MongoDB starting : pid=62788 port=27017 dbpath=C:\data\db\ 64-bit host=nbbnu010160
2020-02-22T16:23:29.262-0300 I CONTROL [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2020-02-22T16:23:29.263-0300 I CONTROL [initandlisten] db version v4.2.3
2020-02-22T16:23:29.264-0300 I CONTROL [initandlisten] git version: 6874650b362138df74be53d366bbefc321ea32d4
2020-02-22T16:23:29.265-0300 I CONTROL [initandlisten] allocator: tcmalloc
2020-02-22T16:23:29.266-0300 I CONTROL [initandlisten] modules: none
2020-02-22T16:23:29.266-0300 I CONTROL [initandlisten] build environment:
2020-02-22T16:23:29.267-0300 I CONTROL [initandlisten] distmod: 2012plus
2020-02-22T16:23:29.269-0300 I CONTROL [initandlisten] distarch: x86_64
2020-02-22T16:23:29.270-0300 I CONTROL [initandlisten] target_arch: x86_64
2020-02-22T16:23:29.272-0300 I CONTROL [initandlisten] options: {}
2020-02-22T16:23:29.286-0300 I STORAGE [initandlisten] Detected data files in C:\data\db\ created by the 'wiredTiger' storage engine, so setting the active storage engine to 'wiredTiger'.
2020-02-22T16:23:29.288-0300 I STORAGE [initandlisten] wiredtiger_open config: create,cache_size=3505M,cache_overflow=(file_max=0M),session_max=33000,eviction=(threads_min=4,threads_max=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manager=(close_idle_time=100000,close_scan_interval=10,close_handle_minimum=250),statistics_log=(wait=0),verbose=[recovery_progress,checkpoint_progress],
2020-02-22T16:23:29.439-0300 I STORAGE [initandlisten] WiredTiger message [1582399409:438622][62788:140732919009888], txn-recover: Recovering log 2 through 3
2020-02-22T16:23:29.529-0300 I STORAGE [initandlisten] WiredTiger message [1582399409:528381][62788:140732919009888], txn-recover: Recovering log 3 through 3
2020-02-22T16:23:29.645-0300 I STORAGE [initandlisten] WiredTiger message [1582399409:645069][62788:140732919009888], txn-recover: Main recovery loop: starting at 2/4480 to 3/256
2020-02-22T16:23:29.859-0300 I STORAGE [initandlisten] WiredTiger message [1582399409:858499][62788:140732919009888], txn-recover: Recovering log 2 through 3
```

STARTING MONGODB

- Ao iniciar, o mongodb permite definir o local dos databases pelo parâmetro “dbpath”
- Se executado sem argumentos, o os diretórios , /data/db/ serão procurados pelo MongoDB. Caso não sejam encontrados, o banco falha na inicialização.
- Por padrão, o serviço fica ouvindo conexões na porta **27017**.
- O server pode ser encerrado com um Ctrl-C no shell
- Com o servidor iniciado (**mongod**) abra outro terminal com o binário **mongo**



```
C:\Windows\system32\cmd.exe - mongo
C:\Portable\mongodb-4.2.3\bin>mongo
MongoDB shell version v4.2.3
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("ae7264cc-55ec-42f7-b2e6-b21d8367f924") }
MongoDB server version: 4.2.3
Server has startup warnings:
2020-02-22T16:23:30.159-0300 I  CONTROL  [initandlisten]
2020-02-22T16:23:30.159-0300 I  CONTROL  [initandlisten] ** WARNING: Access control is not enabled for the database.
2020-02-22T16:23:30.160-0300 I  CONTROL  [initandlisten] **           Read and write access to data and configuration is
```

MONGO SHELL

- O shell é um ambiente capaz de processar comandos e funções JavaScript:

```
$ mongo
```

```
MongoDB shell version: 2.4.0
```

```
connecting to: test >
```

```
> x = 200
```

```
> x / 5;
```

```
40
```

```
> Math.sin(Math.PI / 2);
```

```
1
```

```
> new Date("2010/1/1");
```

```
"Fri Jan 01 2010 00:00:00 GMT-0500 (EST)"
```

```
> "Hello, World!".replace("World", "MongoDB"); Hello,  
MongoDB!
```


MONGO SHELL

- Exemplo de função

```
$ mongo
MongoDB shell version: 2.4.0
connecting to: test >
```

```
> function factorial (n) {
... if (n <= 1) return 1;
... return n * factorial(n - 1); ... }
> factorial(5);
120
```

O shell sempre identifica se o statement JavaScript está completo. Se estiver, ele retorna, senão, permite continuar com indicando as reticências (...). Enter 3x cancela o comando “pela metade”.

MONGODB CLIENT

- Na inicialização, o shell se conecta na base test e a conexão será associada à variável **db**. Essa variável é o ponto de acesso primário para o MongoDB server via shell.

```
> db  
test
```

- O shell tem comandos que não existem no JavaScript padrão, mas que são familiares para quem trabalha com outras tecnologias de banco de dados. Por exemplo, para trocar de database, utilizamos o **use**

```
> use foobar  
switched to db foobar
```

```
> db  
foobar
```

As coleções, são acessadas pela variável db. Por exemplo, **db.baz** retorna a coleção **baz** no banco de dados em uso (atual).

CRUD BASIC - INSERT

- A função insert adiciona um documento em uma coleção.

```
> post = {"title" : "My Blog Post",  
... "content" : "Here's my blog post.", ... "date" : new Date()}
```

```
{  
"title" : "My Blog Post", "content" : "Here's my blog post.",  
"date" : ISODate("2012-08-24T21:12:09.982Z")  
}
```

```
> db.blog.insert(post)
```

CRUD BASIC - READ

- Para consultar, utilizamos a função find:

```
> db.blog.find()
{
  "_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
  "title" : "My Blog Post",
  "content" : "Here's my blog post.",
  "date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

```
> db.blog.findOne()
{
  "_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
  "title" : "My Blog Post",
  "content" : "Here's my blog post.",
  "date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

CRUD BASIC - UPDATE

- O update utiliza pelo menos 2 parâmetros:

1. Critério para encontrar o documento
2. O novo documento.

```
> post.comments = []
```

```
[]
```

```
> db.blog.update({title : "My Blog Post"}, post)
```

```
> db.blog.find()
```

```
{  
  "_id" : ObjectId("5037ee4a1084eb3ffeef7228"),  
  "title" : "My Blog Post",  
  "content" : "Here's my blog post.",  
  "date" : ISODate("2012-08-24T21:12:09.982Z"),  
  "comments" : []  
}
```

CRUD BASIC – “DELETE”

- No mongoDB, utilizamos o commando remove para operação de apagar dados de forma permanente.
- Cuidado, se chamado sem parâmetros, limpa a coleção! (bom para fazer na sexta feira)
- Pode receber um critério como parâmetro identificar o que será removido

```
> db.blog.remove({title : "My Blog Post"})
```

Data Types

DATA TYPES

- Os documentos no MongoDB, podem ser pensados como objetos JSON (modelo fácil de absorver para quem usa JavaScript)
- Mas um JSON tradicional tem algumas limitações com os tipos null, boolean, numeric, string, array, e object
- Para armazenamento, manipulação, o MongoDB define alguns padrões não formalizados em JSON
- Os principais tipos são
 - Data
 - Tipos numéricos
 - Timestamp
 - Minkey, Maxkey
 - ObjectId
 - Regular Expressions
 - Binários

DATA TYPES

null

`{"x" : null}` *boolean*

O tipo boolean é usado para valores true e false: `{"x" : true}`

number

No shell, o padrão é sempre números float de 64bits. Por exemplo ambos os casos serão o mesmo tipo:

`{"x" : 3.14}` ou:

`{"x" : 3}`

Para números inteiros utilize `NumberInt` ou `NumberLong` classes, que usam representação de 4-byte ou 8-byte signed

`{"x" : NumberInt("3")}`

`{"x" : NumberLong("3")}`

string

Qualquer string UTF-8 é sempre representada/armazenada no MongoDB

`{"x" : "foobar"}`

DATA TYPES

date

Datas são armazenados em UTC. Se necessitar timezone, deve ser tratado na aplicação.

```
{"x" : new Date()}
```

regular expression

Utilizado para queries:

```
{"x" : /foobar/i}
```

array

Conjuntos de dados:

```
{"x" : ["a", "b", "c"]}
```

embedded document

Documentos podem ter documentos embutidos na estrutura:

```
{"x" : {"foo" : "bar"}}
```

DATA TYPES - DATE

- Sempre utilizar o **new** Date() e não Date()
 - > Date()
Sat Feb 22 2020 22:41:04 GMT-0300 (Hora oficial do Brasil)
 - > new Date()
ISODate("2020-02-23T01:41:08.286Z")
- Qual a diferença?

DATA TYPES

object id

Tipo especial do MongoDB para identificador de objetos composto de 12 bytes:

```
{"x" : ObjectId()}
```

4-bytes timestamp value, representing the ObjectId's creation, measured in seconds since the Unix epoch

5-bytes random value

3-bytes incrementing counter, initialized to a random value

binary data

Não pode ser manipulado via shell, mas sim via drivers.

Único tipo capaz de manipular dados não UTF-8

code

Queries podem ter funções JavaScript:

```
{"x" : function() { /* ... */ }}
```

DATA TYPES - ARRAY

- Arrays podem ser manipulados de forma ordenada (como lista) ou não ordenada (sets)
- Os dados não precisam ser iguais, podendo misturar tipos e mesmo assim permitir buscas sobre valores do array
`{"things" : ["pie", 3.14]}`
- O MongoDB vai “compreender” a estrutura do array permitindo operações sobre o conteúdo, e criar índices. No exemplo acima, é possível buscar pelos documentos que contem 3.14 como elemento de “things” e atualizar “pie” para “pi”
- A atualização sobre o conteúdo do array é atômica.

DATA TYPES – EMBEDDED DOCUMENTS

- Um documento todo pode ser o valor de uma chave. Nesse caso o conteúdo de address é um documento embutido:

```
{ "name" : "John Doe",  
  "address" : {  
    "street" : "123 Park Street",  
    "city" : "Anytown",  
    "state" : "NY"  
  }  
}
```

- O Mongo é capaz de operar sobre “subdocumentos” buscando valores, criando índices e fazendo updates.
- Em bancos relacionais, é um exemplo clássico de tabelas separadas que necessitam join para montagem do registro.
- Como contrapartida, é possível ter dados repetidos entre vários documentos.

DATA TYPES – OBJECTID

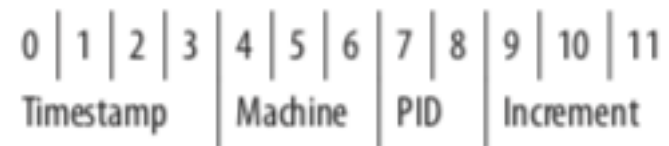
- Todo documento armazenado precisa ter uma “_id”. O “_id” pode ser de qualquer tipo, mas por padrão será ObjectId
- Em uma mesma coleção, o “_id” de cada documento deve ser único
- O ObjectId é uma classe desenvolvida para ser leve e ainda capaz de gerar ids únicos de forma global em diferentes máquinas.
- A principal razão para o uso de ObjectId é o fator de distribuição em diversos nós. Por isso, um modelo de auto incremento não é utilizado.
- Caso for controlar o “_id” é necessário fazer a gestão/geração dos identificadores com cuidado.

```
{ "_id" : "012.147.557-87",  
  "name" : "John Doe",  
    "address" : {  
      "street" : "123 Park Street",  
      "city" : "Anytown",  
      "state" : "NY"  
    }  
}
```

DATA TYPES – OBJECTID

- O ObjectId tem uma estrutura que em geral mostra os dados em ordem de inserção (timestamp no início)
- São 12 bytes de armazenamento, mas que em “formato legível” são 24 caracteres hexadecimais
- Os primeiros nove bytes visam a unicidade entre servidores. Já os últimos, visam garantir que o dados será único dentro de um segundo pelos processos locais.
- Não é preciso informar o “_id”, sempre que não existir essa chave, ela será adicionada pelo MongoDB

ObjectId



Shell Parte 2

SHELL

- Para conectar em um MongoDB rodando em outro servidor/porta, basta especificar o servidor, porta e database

```
$ mongo some-host:30000/myDB
MongoDB shell version: 2.4.0 connecting to: some-
host:30000/myDB
```

- É possível iniciar o shell sem conectar em um banco de dados e depois montar a conexão.

```
$ mongo --nodb
MongoDB shell version: 2.4.0
> conn = new Mongo("some-host:30000")
connection to some-host:30000

> db = conn.getDB("myDB")
myDB
```

SHELL

- Na dúvida, além do StackOverflow, Google, há sempre um help no shell 😊

> help

db.help()	help on db methods
db.mycoll.help()	help on collection methods
sh.help()	sharding helpers
...	
show dbs	show database names
show collections	show collections in current database
show users	show users in current database
...	

SHELL

- Javascript tem algumas vantagens em facilitar o conhecimento sobre código fonte
- Para saber o que uma função específica faz, basta digitar a mesma sem os parêntesis

```
> db.foo.update
function (query, obj, upsert, multi) {
    assert(query, "need a query");
    assert(obj, "need an object");
    this._validateObject(obj);
    this._mongo.update(this._fullName, query, obj,
        upsert ? true : false, multi ? true : false);
}
```

SHELL

- É possível passar scripts para serem invocados pelo shell:

```
$ mongo script1.js script2.js script3.js
```

```
MongoDB shell version: 2.4.0
```

```
connecting to: test
```

```
I am script1.js
```

```
I am script2.js
```

```
I am script3.js
```

- Para automação ou acompanhamento, nem sempre o banner de welcome do shell é desejado. Nesses casos, basta omitir o mesmo com o `--quiet`

```
$ mongo --quiet server-1:30000/foo script1.js script2.js script3.js
```

SHELL

- Também é possível carregar conteúdo diretamente do shell

```
// defineConnectTo.js
```

```
var connectTo = function(port, dbname) {  
  if (!port) {  
    port = 27017;  
  }  
  if (!dbname) {  
    dbname = "test";  
  }  
  db = connect("localhost:"+port+"/"+dbname);  
  return db;  
};
```

```
> typeof connectTo  
undefined
```

```
> load('defineConnectTo.js')
```

```
> typeof connectTo  
function
```

SHELL - .MONGORC.JS

- Existe um arquivo no \$HOME chamado *mongorc.js* file.
- É possível adicionar scripts de inicialização do shell:

```
// mongorc.js
```

```
var compliment = ["attractive", "intelligent", "like Batman"];
```

```
var index = Math.floor(Math.random()*3);
```

```
print("Hello, you're looking particularly "+compliment[index]+" today!");
```

```
$ mongo
```

```
MongoDB shell version: 2.4.0-pre-
```

```
connecting to: test
```

```
Hello, you're looking particularly like Batman today!
```

SHELL - .MONGORC.JS

- Esse arquivo pode ser um bom aliado para evitar “acidentes”:

```
var no = function() {  
    print("Not on my watch.");  
};
```

```
db.dropDatabase = DB.prototype.dropDatabase = no;  
DBCollection.prototype.drop = no;  
DBCollection.prototype.dropIndex = no;
```


Operações

CRUD II

INSERTING AND SAVING DOCUMENTS

- O insert permite adicionar um ou mais documentos
- O método tem 3 parâmetros:
 - O documento a ser inserido ou Array
 - Write concern (forma de ACK para assegurar o insert)
 - Indicação se a inserção de array deve ser ordenada
- Opções do Write concern
 - Majority (Exemplo 3 nos – Majority 2)
 - W Option
 - Custom

```
db.collection.insert(  
  <document or array of documents>,  
  {  
    writeConcern: <document>,  
    ordered: <boolean>  
  }  
)
```

SETTING A WRITE CONCERN

- *Write concern é basicamente uma configuração que define o quão Seguro uma operação de write deve ocorrer antes da aplicação seguir em frente.*
- Opções do Write concern
 - Majority (Exemplo 3 nos – Majority 2)
 - W Option
 - Custom
- Em 2012, o Write Concer padrão foi alterado, e por isso versões antigas do Mongo podem funcionar de forma diferente.
- Atualmente, o padrão é com acknowledgement w: 1 (standalone)

INSERTING AND SAVING DOCUMENTS

- Variações do Insert
 - insertOne
 - insertMany
- Por padrão, há um limite de writes com o InsertMany
- Por padrão, esse limite é de 100 mil writes
- Alguns drivers gerenciam isso automaticamente
- Para evitar o limite de write, é possível usar bulks

```
db.collection.insertOne(  
  <document>,  
  {  
    writeConcern: <document>  
  }  
)
```

```
db.collection.insertMany(  
  [ <document 1> , <document 2>, ... ],  
  {  
    writeConcern: <document>,  
    ordered: <boolean>  
  }  
)
```

INSERTING AND SAVING DOCUMENTS

- Há uma limitação de 16MB para cada documento.
- Inserções em volumes devem utilizar o Bulk
- A quantidade de insertes por bulk não deve passar de 1000.

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.insert( { item: "abc123", defaultQty: 100, status: "A", points: 100 } );
bulk.insert( { item: "ijk123", defaultQty: 200, status: "A", points: 200 } );
bulk.insert( { item: "mop123", defaultQty: 0, status: "P", points: 0 } );
bulk.execute();
```

INSERTING AND SAVING DOCUMENTS

- Para arquivos maiores que 16MB, deve-se utilizar o GridFS (através de drivers)
- Na prática, o arquivo será “quebrado” em pedaços para armazenamento dentro do MongoDB (256kb)
- Em geral, não é uma prática muito comum armazenar grandes arquivos no MongoDB
- O GridFS utiliza duas coleções para armazenar os dados
 - Metadados
 - File Chunks
- O GridFS não carrega o conteúdo em memória. Então mesmo que o arquivo não tenha 16MB, ainda pode ser uma ferramenta a ser explorada.

EXCECUÇÃO DO SCRIPT

- Execução do Script italian-people.js
- Adicionar 5 pessoas manualmente via shell



REMOVING DOCUMENTS

- Remoção possui 2 sintaxes. A mais comum é com 2 parâmetros
 - Query de filtro
 - Indicação se somente 1 documento deve ser afetado.

```
db.collection.remove(  
  <query>,  
  <justOne>  
)
```

- Sempre que nenhuma query for informada, todos os documentos da coleção são removidos
- Com o remove() os metadados são mantidos
- Para remoção da coleção, usa-se o drop

> db.foo.remove()

> db.foo.remove({"opt-out" : true})

> db.foo.drop()

```
db.collection.remove(  
  <query>,  
  {  
    justOne: <boolean>,  
    writeConcern: <document>,  
    collation: <document>  
  }  
)
```


REMOVING DOCUMENTS

- Vamos remover apenas uma pessoa que tenha mãe com idade na casa de 80-90 anos

```
db.pessoas.find({"mother.age": 81}).count();
```

```
db.pessoas.remove({"mother.age": 81}, true);
```

- Collation permite especificar regras específicas de linguagem e caracteres como:

- Case sensitive
- Acentuação



```
db.collection.remove(  
  <query>,  
  <justOne>  
)
```

```
db.collection.remove(  
  <query>,  
  {  
    justOne: <boolean>,  
    writeConcern: <document>,  
    collation: <document>  
  }  
)
```

UPDATE DOCUMENTS

- O update pode ser “espartano”: substituir o documento anterior por um novo.
- É possível atualizar apenas atributos através com o <update>
- Os campos do <update> são operadores:
 - \$set
 - \$unset
 - \$inc
 - \$mul
 - \$rename
 - \$max/\$min
 - \$setOnInsert

```
db.collection.update(  
  <query>,  
  <update>,  
  {  
    upsert: <boolean>,  
    multi: <boolean>,  
    writeConcern: <document>,  
    collation: <document>,  
    arrayFilters: [ <filterdocument1>, ... ],  
    hint: <document|string>           // Available starting in MongoDB 4.2  
  }  
)
```

UPDATE DOCUMENTS

- Atualização por substituição

```
{
  "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
  "name" : "joe",
  "friends" : 32,
  "enemies" : 2
}
```

- Como jogar **friends** e **enemies** para um novo objeto chamado **relationships**?

```
> var joe = db.users.findOne({"name" : "joe"});
> joe.relationships = {"friends" : joe.friends, "enemies" :
joe.enemies};
{ "friends" : 32, "enemies" : 2 }
```

```
> joe.username = joe.name;
"joe"
```

```
> delete joe.friends;
true
```

```
> delete joe.enemies;
true
```

```
> db.users.update({"name" : "joe"}, joe);
```

UPDATE DOCUMENTS

- É importante cuidar no update para evitar erros de chave duplicada se existem mais documentos.

```
> joe = db.people.findOne({"name" : "joe", "age" : 20});
```

```
{ "_id" : ObjectId("4b2b9f67a1f631733d917a7c"),  
  "name" : "joe",  
  "age" : 20 }
```

```
>joe.age++;
```

```
> db.people.update({"name" : "joe"}, joe);  
E11001 duplicate key on update
```

```
> db.people.update({"_id" : ObjectId("4b2b9f67a1f631733d917a7c")}, joe)
```

UPDATE DOCUMENTS - MODIFIERS

- **Atomic update modifiers** - os modificadores permitem fazer operações complexas de update, incrementando, adicionando ou removendo campos.
- Um exemplo é incrementar pageviews em websites. Considere o seguinte objeto:

```
{ "_id" : ObjectId("4b253b067525f35f94b60a31"), "url" :  
  "www.example.com", "pageviews" : 52 }
```

- A cada visita ao site, é necessário atualizar o valor de forma atômica:

```
db.analytics.update({"url" : "www.example.com"}, ... {  
  "$inc" : {"pageviews" : 1}  
})
```

UPDATE DOCUMENTS - MODIFIERS

- "\$set" sets the value of a field. If the field does not yet exist, it will be created.

```
> db.users.findOne()  
  { "_id" : ObjectId("4b253b067525f35f94b60a31"),  
    "name" : "joe",  
    "age" : 30,  
    "sex" : "male",  
    "location" : "Wisconsin" }  
  
> db.users.update({"_id" :  
  ObjectId("4b253b067525f35f94b60a31")}, ... {"$set" : {"favorite  
  book" : "War and Peace"}})
```

UPDATE DOCUMENTS - MODIFIERS

- "\$set" pode ser usado para alterar valores

```
> db.users.update({"name" : "joe"}, ... {  
    "$set" : {"favorite book" : "Green Eggs and Ham"}  
})
```

- "\$set" inclusive mudando o tipo

```
> db.users.update({"name" : "joe"}, ... {  
    "$set" : {"favorite book" : ... ["Cat's Cradle", "Foundation Trilogy", "Ender's Game"]}  
})
```

UPDATE DOCUMENTS - MODIFIERS

- "\$unset" é usado para remover uma chave

```
> db.users.update({"name" : "joe"}, ... {  
    "$unset" : {"favorite book" : 1}  
})
```

- Será que Podemos melhorar nosso update “espartano” do joe?
- Como jogar **friends** e **enemies** para um novo objeto chamado **relationships**?

UPDATE DOCUMENTS

- Atualização com modificadores

```
{  
  "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),  
  "name" : "joe",  
  "friends" : 32,  
  "enemies" : 2  
}
```

- Como jogar **friends** e **enemies** para um novo objeto chamado **relationships**?

```
> var joe = db.users.findOne({"name" : "joe"});
```

```
> relationships = {"friends" : joe.friends, "enemies" : joe.enemies};
```

```
> db.pessoas.update({ "name": "joe" }, {  
  $unset: { friends: "", enemies: "" },  
  $set: { relationships: relationships }  
})
```

UPDATE DOCUMENTS - MODIFIERS

- Atenção! Sempre utilize os modificadores \$ para adicionar, modificar ou remover chaves de algum documento. Um erro comum é acreditar que o update para adicionar um campo pode ser feito com a chave/valor informada sem modificadores:

```
> db.coll.update(criteria, {"foo" : "bar"})
```

- O que faz essa estrutura?

O método acima está substituindo o(s) documento(s) que tem *match* pelo *criteria* pelo novo documento especificado no Segundo parâmetro {"foo" : "bar"}

Por isso, o update deve ser sempre com os modificadores.

UPDATE DOCUMENTS - MODIFIERS

- O \$inc é semelhante ao \$set, mas é projetado para incrementar/decrementar números de forma atômica
- O \$inc só pode ser usado com tipos integer, long ou double.

```
> db.foo.insert({"count" : "1"})
```

```
> db.foo.update({}, {"$inc" : {"count" : 1}})
```

Cannot apply \$inc modifier to non-number



Incremente em um, a idade de todas as pessoas do nosso grupo de italianos 😊

Quantos registros foram alterados?

```
db.italians.update({}, {"$inc": {"age": 1}}, {multi: true});
```

UPDATE DOCUMENTS – UPSERTS

- **Updating Multiple Documents** – É possível atualizar múltiplos documentos sem passar a chave, mas nesse caso deve se manter a ordem dos parâmetros.

```
> db.users.update({"birthday" : "10/13/1978"},  
... {"$set" : {"gift" : "Happy Birthday!"}}, false, true)
```

- Para saber o resultado da operação sobre múltiplos documentos, é possível utilizar a função `getLastError`

```
> db.runCommand({getLastError : 1})  
{  
  "err" : null, "updatedExisting" : true,  
  "n" : 5,  
  "ok" : true  
}
```

UPDATE DOCUMENTS – ARRAY MODIFIERS

- O \$push permite adicionar elementos a um array e se o array não existir, cria o mesmo.
- De forma semelhante \$pull é usado para tirar elementos

```
> db.blog.posts.findOne() {  
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),  
  "title" : "A blog post", "content" : "..."  
}
```

```
> db.blog.posts.update({"title" : "A blog post"}, ...  
  {"$push" : {"comments" : ...  
  {"name" : "joe", "email" : "joe@example.com", ... "content" : "nice post."}}})
```

UPDATE DOCUMENTS – ARRAY MODIFIERS

- Adicione morango ao primeiro italiano do grupo...

```
db.italians.update({}, { $push: {favFruits: "Morango" } })  
db.italians.findOne()
```

- Agora adicione Pera e Limão ao mesmo tempo... Em seguida, veja se ficou certo...

```
db.italians.update({}, { $push: {favFruits: { $each: [ "Pera", "Limão" ]}}})  
db.italians.findOne()
```



UPDATE DOCUMENTS – ARRAY MODIFIERS

- Algumas operações exigem a utilização de suboperadores como é o caso do "\$each"
- O \$each indica que a operação deve ser feita para cada elemento informado

```
> db.stock.ticker.update({"_id" : "GOOG"},  
    ... {"$push" : {"hourly" : {"$each" : [562.776, 562.790, 559.123]}}})
```

- Sub operadores de push/pull:
 - \$each
 - \$slice
 - \$sort
 - \$position

UPDATE DOCUMENTS – ARRAY MODIFIERS

- O \$slice permite limitar o tamanho de um array. Isso pode ser feito durante o push/pull
- Basta adicionar o suboperador junto com o modificador

```
> db.movies.find({"genre" : "horror"},  
  ... {"$push" : {"top10" : {  
  ... "$each" : ["Nightmare on Elm Street", "Saw"], "$slice" : -10}}})
```



Adicione frutas ao primeiro italiano até ter 5. Em seguida execute o slice “sem adicionar nenhuma fruta”

```
db.italians.update({}, { $push: {favFruits: { $each: [], $slice:  
-3 }}})
```


UPDATE DOCUMENTS – ARRAY MODIFIERS

- É possível ordenar antes de limitar o tamanho do array com \$slice e \$sort

```
db.italians.update({}, { "$push" : { movies : { $each : [  
  { "title" : "Nightmare on Elm Street", "rating" : 3.6},  
  { "title" : "Saw", "rating" : 4.3 },  
  { "title" : "God father", "rating" : 4.9} ],  
  $slice : 3,  
  $sort : { "rating" : -1}}}}))
```

- Nesse caso, todos os elementos do array serão ordenados e de forma decrescente e os top 3 serão mantidos
- Importante observar que sempre é necessário o uso do \$each para aplicar o "\$slice" ou "\$sort" com o "\$push".

UPDATE DOCUMENTS – ARRAY MODIFIERS

- É possível tratar os arrays como um set, somente adicionando valores que não existam no array
- Isso é feito com o operador **\$ne** na querie

```
> db.papers.update({"authors cited" : {"$ne" : "Richie"}},  
  ... {$push : {"authors cited" : "Richie"}})
```

- Ou através do **\$addToSet**, que de certa forma é mais expressivo

```
{ "_id" : ObjectId("4b2d75476cc613d5ee930164"), "username" : "joe",  
  "emails" : [ "joe@example.com", "joe@gmail.com", "joe@yahoo.com" ] }
```

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},  
  ... {"$addToSet" : {"emails" : "joe@gmail.com"}})
```

UPDATE DOCUMENTS – ARRAY MODIFIERS

- É possível adicionar vários elementos com o "\$addToSet" junto com o "\$each"

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")}, {"$addToSet" :  
  ... {"emails" : {"$each" :  
    ... ["joe@php.net", "joe@example.com", "joe@python.org"]}}})
```

```
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")}) { "_id" :  
  ObjectId("4b2d75476cc613d5ee930164"), "username" : "joe", "emails" : [  
    "joe@example.com", "joe@gmail.com", "joe@yahoo.com", "joe@hotmail.com"  
    "joe@php.net" "joe@python.org" ] }
```

UPDATE DOCUMENTS – ARRAY MODIFIERS

- O operador "\$pop" é utilizado para remoção de itens das extremidades de um array.
- Por exemplo {"\$pop" : {"key" : 1}} remove o ultimo item e {"\$pop" : {"key" : -1}} remove do início

```
> db.lists.insert({"todo" : ["dishes", "laundry", "dry cleaning"]})
```

```
> db.lists.update({}, {"$pop" : {"todo" : 1}})
```

```
> db.lists.find()  
  { "_id" : ObjectId("4b2d75476cc613d5ee930164"),  
    "todo" : [ "dishes", "laundry" ] }
```

UPDATE DOCUMENTS – ARRAY MODIFIERS

- O operador "\$pull" pode ser usado para remover usando um filtro como critério

```
> db.lists.insert({"todo" : ["dishes", "laundry", "dry cleaning"]})
```

```
> db.lists.update({}, {"$pull" : {"todo" : "laundry"}})
```

```
> db.lists.find()  
  { "_id" : ObjectId("4b2d75476cc613d5ee930164"),  
    "todo" : [ "dishes", "dry cleaning" ] }
```

- Cuidado, o pull vai remover todos os itens que derem “match”

UPDATE DOCUMENTS – ARRAY MODIFIERS

- Existem duas formas de manipular os valores de um array: pela posição ou utilizando o operador posicional \$

```
> db.blog.posts.findOne()  
  { "_id" : ObjectId("4b329a216cc613d5ee930192"),  
    "content" : "...",  
    "comments" : [  
      { "comment" : "good post", "author" : "John", "votes" : 0 }, ← 0  
      { "comment" : "i thought it was too short", "author" : "Claire", "votes" : 3 }, ← 1  
      { "comment" : "free watches", "author" : "Alice", "votes" : -1 } ← 2  
    ] }  
  }
```

- Para incrementar a quantidade de votos do primeiro comentário

```
> db.blog.update({"post" : post_id,  
... {"$inc" : {"comments.0.votes" : 1}})
```

UPDATE DOCUMENTS – ARRAY MODIFIERS

- Mas é incomum saber em uma aplicação o índice de um determinado dado dentro de um array
- Por isso, é mais prático utilizar o operador posicional \$ que identifica a posição pelo critério
- Por exemplo, caso um usuário utilizava um nome “IvanTheTerribleOne” e agora mudou apenas para Ivan

```
db.blog.update({"comments.author" : "IvanTheTerribleOne"},  
... {"$set" : {"comments.$.author" : "Ivan"}})
```

- O operador posicional, atualiza apenas o primeiro match. Ou seja, se existirem mais comentários, a operação precisa ser realizada mais vezes

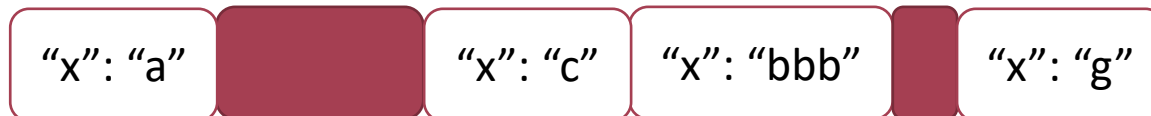
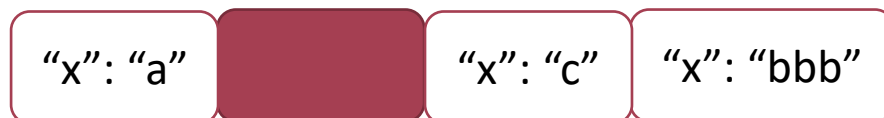
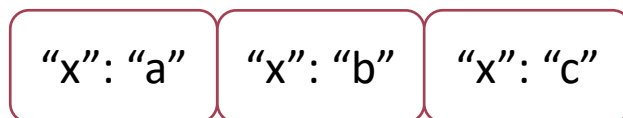
UPDATE DOCUMENTS – MODIFIER SPEED

- O impacto dos modificadores varia conforme a mudança estrutura que ele podem “causar”
- Por exemplo, \$inc é um operador extremamente rápido, pois opera sobre um valor e não muda o tamanho do documento
- Já operadores que fazem alteração no tamanho do documento podem ser “lentos”
- O \$set é um exemplo que pode ter impacto semelhante aos operadores de array.
- **Por que isso acontece?**

A principal razão é que o MongoDB armazena os documentos um ao lado do outro (save space strategy). Então se forem feitas mudanças estruturais muito grandes, o documento pode não “caber mais” onde estava e precisará ser movido para outro local.

UPDATE DOCUMENTS – MODIFIER SPEED

- Mover documentos não é uma atividade barata e com o tempo se torna ainda mais “cara”
- Nas versões mais novas do MongoDB, esses problemas não tem aparecido devido a mudança no Storage Engine



- Se o banco ficar lento, existem comandos de diagnóstico para avaliar a distribuição dos dados

UPDATE DOCUMENTS – MODIFIER SPEED

- As execuções que mais tem chance de impactar na estrutura de documentos é o \$push sobre arrays. Operações seguidas assim podem forçar mudar o documento e até mesmo exigir executar ferramentas como o compact.
- O MongoDB opera com um padding factor dinâmico. Quando o tamanho dos documentos aumenta seguidas vezes, o padding factor é ajustado para cima. Se isso para de ocorrer, ele vai sendo reduzido.
- Operações como \$push podem ser feitas normalmente, principalmente nas novas versões do MongoDB. Porém, em alguns casos pode ser interessante fazer esse processamento a nível de aplicação.

UPDATE DOCUMENTS – MODIFIER SPEED

```
> db.testers.insert({"x" : 1})
> var timeInc = function() {
... var start = (new Date()).getTime();
...
... for (var i=0; i<100000; i++) {
... db.testers.update({}, {"$inc" : {"x" : 1}});
... db.getLastError();
... }
...
... var timeDiff = (new Date()).getTime() - start;
... print("Updates took: "+timeDiff+"ms");
... }
> timeInc()
```

111432ms

```
db.testers.update({}, {"$push" : {"x" : 1}})
```

209687ms

UPDATE DOCUMENTS – UPSERTS

- O upsert é apenas um parâmetro especial de update. Se o documento existir, será atualizado. Senão, será inserido

```
blog = db.analytics.findOne({url : "/blog"})
```

```
if (blog) {  
    blog.pageviews++;  
    db.analytics.save(blog);  
} else {  
    db.analytics.save({url : "/blog", pageviews : 1})  
}
```

```
db.analytics.update({"url" : "/blog"}, {"$inc" : {"pageviews" : 1}}, true)
```

upsert

UPDATE DOCUMENTS – UPSERTS

- As operações com upserts são garantidas, mesmo que o documento não exista

```
> db.users.update({"rep" : 25}, {"$inc" : {"rep" : 3}}, true)
```

```
> db.users.findOne()
```

```
{  
  "_id" : ObjectId("4b3295f26cc613d5ee93018f"),  
  "rep" : 28  
}
```

- Se o documento não existir, um novo será criado e em seguida o incremento é aplicado;
- Caso o upsert não fosse informado, o efeito seria: não modificar nada se um documento não existisse;

UPDATE DOCUMENTS – UPSERTS

- Outra operação interessante a usar é o `$setOnInsert`
- É um modificador que só tem efeito em caso de um insert, por exemplo, adicionar data de criação do registro

```
> db.users.update({"username": "joe"}, {"$setOnInsert" : {"createdAt" : new Date()}}, true)
> db.users.findOne()
{
  "_id" : ObjectId("512b8aefae74c67969e404ca"),
  "username": "joe"
  "createdAt" : ISODate("2013-02-25T16:01:50.742Z")
}
```

- Se for executado mais de uma vez, as execuções subsequentes não tem efeito.
- Mas não use nada como `"createdAt"`, pois o `ObjectId` tem essa informação: `ObjectId("...").getTimestamp()`

Querying

FIND

- O método find() é provavelmente um dos métodos mais usados no dia a dia
- Ele permite fazer as consultas (queries) no MongoDB.
- Cada query retorna um subconjunto de documentos de uma coleção.

`db.collection.find(query, projection)`

- Query – Define os filtros da busca
- Projection – Define o que será retornado

FIND

- Tal como os filtros vistos para update, o find funciona de forma semelhante

```
> db.c.findOne()
```

```
> db.c.find()
```

```
> db.users.find({"age" : 27})
```

```
> db.users.find({"username" : "joe"})
```

```
> db.users.find({"username" : "joe", "age" : 27})
```

FIND - PROJECTION

- É comum em consultas de bancos de dados que sejam informados apenas os campos de interesse.
- A projeção é feita através do segundo parâmetro informando 1 para trazer o campo;

```
> db.italians.find({"surname": "Rossi"}, {"firstname" : 1, "email" : 1})
```

- O campo "_id" é sempre parte do resultado a menos que seja especificado na projeção para ser omitido.

```
> db.italians.find({"surname": "Rossi"}, {"firstname" : 1, "email" : 1, "_id": 0})
```

FIND - QUERY CRITERIA

- Tal como no SQL, o MongoDB também possui operadores de comparação equivalentes a <, <=, >, and >=,
 - **\$lt**, **\$lte** – “less than” e “less than or equal to”
 - **\$gt**, **\$gte** – “greater than” e “greater than or equal to”
 - **\$ne** – not equal

```
> db.italians.find({"age" : {"$gte" : 50, "$lte" : 80}})
```

```
> start = new Date("01/01/2012")  
> db.italians.find({"registeredDate" : {"$lt" : start}})
```

```
> db.italians.find({"surname" : {"$ne" : "Rossi"}})
```

FIND - QUERY CRITERIA

- Existem duas formas de fazer um OR no MongoDB:
- **\$in** e **\$nin** podem ser usadas para procurar por uma série de valores em uma chave
- **\$or** é mais genérico e pode ser usado para fazer queries sobre múltiplas chaves
- Por exemplo, verificar quem está na idade do alistamento ou não

```
> db.italians.find({"age" : {"$in" : [18, 19]}})
```

```
> db.italians.find({"age" : {"$nin" : [18, 19]}})
```

FIND - QUERY CRITERIA

- O \$or é uma construção que pode receber vários campos em um array.
- Por exemplo, vamos verificar quem está na idade de 18 anos e pode ser um doador universal

```
> db.italians.find({"$or" : [{"age" : 18}, {"bloodType" : "O-"}]}))
```

- Uma vantagem do "\$or" é compor mais condicionais:

```
> db.italians.find({"$or" : [{"age" : {"$in" : [18, 19]}}, {"bloodType" : "O-"}]}))
```

FIND - QUERY CRITERIA

- **\$not** é um metaconditional: pode ser colocado no topo de um critério:
- **\$mod** obtém o modulo no formato [divisor, resto]

```
> db.users.find({"ticketNumber" : {"$mod" : [17, 0]}})
```

- Para fazer a negção disso, basta trazer o \$not acima da construção

```
> db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})
```

FIND - CONDITIONAL SEMANTICS

- Múltiplas condições podem ser colocadas em uma mesma chamada

```
> db.italians.find({"age" : {"$lt" : 30, "$gt" : 20}})
```

- E os “meta-operators” podem ser colocados acima: “\$and”, “\$or”, and “\$nor”:

```
> db.italians.find({"$and" : [{"age" : {"$gt" : 18}}, {"bloodType" : "O-"}]}))
```



Liste todos as pessoas com mais de 65 anos que tenham sangue AB-

```
db.italians.find({"$and" : [{"age" : {"$gt" : 65}}, {"bloodType" : "AB-"}]}))
```

FIND - NULL

- O null no MongoDB segue o padrão JavaScript: se comporta de forma “estranha”
- A verdade é que o match para null funciona também como um “não existe”

```
> db.c.find()  
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }  
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }  
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

```
> db.c.find({"y" : null})  
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
```

Conforme o esperado, só temos um item null para y nessa coleção

FIND - NULL

- O null no MongoDB segue o padrão JavaScript: se comporta de forma “estranha”
- A verdade é que o match para null funciona também como um “não existe”

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

```
> db.c.find({"z" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null } {
  "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

Nesse caso, z não existe para nenhum documento. Então é considerado null
Por isso, essa busca traz todos os documentos acima.

FIND - NULL

- O null no MongoDB segue o padrão JavaScript: se comporta de forma “estranha”
- A verdade é que o match para null funciona também como um “não existe”

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
{ "_id" : ObjectId("4ba0f143d22aa494fd523624"), "r" : null }
```

```
> db.c.find({"r" : {"$in" : [null], "$exists" : true}})
{ "_id" : ObjectId("4ba0f143d22aa494fd523624"), "r" : null }
```

Vai retornar todos os documentos

```
> db.c.find({"r" : null})
```

FIND – REGULAR EXPRESSIONS

- Expressões regulares são usadas para facilitar a busca por strings.
- Por exemplo, busca pelo nome de forma independente de maiúsculas/minúsculas:

```
> db.italians.find({"firstname" : /ele/i})
```

```
> db.italians.find({"name" : /ele?/i})
```

- As expressões regulares no mongo seguem o Perl Compatible Regular Expression (PCRE)

```
{ <field>: { $regex: /pattern/, $options: '<options>' } }  
{ <field>: { $regex: 'pattern', $options: '<options>' } }  
{ <field>: { $regex: /pattern/<options> } }
```

```
{ <field>: /pattern/<options> }
```

FIND – REGULAR EXPRESSIONS

- Expressões regulares são usadas para facilitar a busca por strings.
- Por exemplo, busca pelo nome de forma independente de maiúsculas/minúsculas:

```
> db.italians.find({"firstname" : /ele/i})
```

```
> db.italians.find({"name" : /ele?/i})
```

- As expressões regulares no mongo seguem o Perl Compatible Regular Expression (PCRE)



Utilize uma expressão regular para listar pessoas com a sílaba “gi” no nome ou sobrenome. Em seguida veja quantos tem gi em ambos nome e sobrenome

```
db.italians.find({$and: [ {"firstname" : /gi/i}, {"surname": /gi/i} ]}).count()
```

FIND – QUERYING ARRAYS

- A execução de consultas sobre arrays é feito com alguns operadores especiais

```
> db.food.insert({"fruit" : ["apple", "banana", "peach"]})
```

```
> db.food.find({"fruit" : "banana"})
```

- O "\$all" permite fazer a busca sobre mais de um element do array

```
> db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})
```

```
> db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})
```

```
> db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})
```

```
> db.food.find({fruit : {$all : ["cherry", "banana"]}})
{"_id" : 3, "fruit" : ["cherry", "banana", "apple"] }
```

FIND – QUERYING ARRAYS

- É possível consultar o valor de um índice específico:

```
> db.food.find({"fruit.2" : "peach"})
```

- E utilizar o "\$size" para retornar um array pelo tamanho

```
> db.food.find({"fruit" : {"$size" : 3}})
```



Procure no nosso grupo de italianos pessoas quem tem 3 frutas favoritas e 3 filmes favoritos

```
db.italians.find({$and: [ {"favFruits" : {"$size" : 4}},  
{"movies" : {"$size" : 3}}).count()
```

QUERYING ON EMBEDDED DOCUMENTS

- Existem duas formas de consultar valores de sub documentos

- Mantendo a estrutura do JSON na busca
- Utilizando a navegação com o ponto “.”

```
{  
  "name" : {  
    "first" : "Joe",  
    "last" : "Schmoe"  
  },  
  "age" : 45 }
```

```
> db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})
```

```
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
```

- Atenção que as consultas são sensíveis a ordem! {"last" : "Schmoe", "first" : "Joe"} não vai achar nada...

QUERYING ON EMBEDDED DOCUMENTS

- O ponto funciona como uma navegação entre os documentos, por isso são reservados
- Mas existem limitações nesse tipo de busca. Não é possível trazer dados de dois documentos no estilo “or”

```
> db.blog.find()  
{ "content" : "...",  
  "comments" : [  
    { "author" : "joe", "score" : 3, "comment" : "nice post" },  
    { "author" : "mary", "score" : 6, "comment" : "terrible post" }  
  ]  
}
```

- Por exemplo a busca abaixo não traz nenhum resultado...

```
> db.blog.find({ "comments" : { "author" : "joe", "score" : { "$gte" : 5 } } })
```


QUERYING ON EMBEDDED DOCUMENTS

- Para esses casos, devemos utilizar o \$elemMatch
- Esse atributo permite agrupar os critérios quando existem diferentes chaves em subdocumentos

```
> db.blog.find(  
  { "content" : "...",  
    "comments" : [  
      { "author" : "joe", "score" : 3, "comment" : "nice post" },  
      { "author" : "mary", "score" : 6, "comment" : "terrible post" }  
    ]  
  }  
)
```

```
> db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe", "score" : {"$gte" : 5}}}})
```

\$WHERE QUERIES

- Chave valor são interessantes mas algumas coisas precisam de mais semântica...
- Embora o \$where seja bastante lento, ele pode ser mais expressivo (não usa índices)
- Principalmente por poder operar com funções javascript

```
> db.bank.insert({"id" : 1, "credits" : 6000, "debits" : 3000})  
> db.bank.insert({"id" : 2, "credits" : 4500, "debits" : 400})  
> db.bank.insert({"id" : 3, "credits" : 4244, "debits" : 4244})  
> db.bank.insert({"id" : 4, "credits" : 2345, "debits" : 447})
```

```
> db.bank.find( { $where: "this.credits == this.debits" } );  
> db.bank.find( { $where: "this.credits <= this.debits" } );  
> db.bank.find( { $where: "this.credits >= this.debits" } );
```

\$WHERE QUERIES

- O \$where no mongo é algo a ser usado em últimos casos
- Mesmo assim, permite extrair dados embora isso não seja necessariamente simples



Procure no nosso grupo de italianos pessoas o mesmo primeiro nome de seu pai (father)

Dica: nem todos tem um “father”

```
db.italians.find( { $and: [ { father: { $exists: true } },  
  { $where: "this.firstname == this.father.firstname" } ] },  
  {"father.firstname": 1, "firstname": 1})
```

Se o where é um código javascript, talvez isso já significa que podes tratar na aplicação ou usar o aggregation

LIMITS, SKIPS, AND SORTS

- Limites servem para evitar trazer dados de forma desnecessária

```
> db.c.find().limit(3)
```

- O Skip permite pular alguns registros

```
> db.c.find().skip(3)
```

- Sort recebe um objeto chave valor com as indicações da ordenação 1 (ascending) or -1 (descending).

```
> db.c.find().sort({username : 1, age : -1})
```

```
> db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
```

CURSORS

- Como os bancos relacionais, o MongoDB também oferece cursors
- A principal vantagem é poder processar um documento de cada vez

```
> var cursor = db.people.find();  
> cursor.forEach(function(x) {  
... print(x.name);  
...});
```

- Quando o find é chamado, ele não executa a query imediatamente. Somente quando o resultado é processado que a consulta será enviada ao servidor

CURSORS

- Quase todos os métodos dos cursores, retornam cursores 😊

```
> var cursor = db.foo.find().sort({"x" : 1}).limit(1).skip(10);  
> var cursor = db.foo.find().limit(1).sort({"x" : 1}).skip(10);  
> var cursor = db.foo.find().skip(10).limit(1).sort({"x" : 1});
```

- Até aqui, nenhuma consulta executada de fato...

Somente agora o banco processou a requisição

```
> cursor.hasNext()
```

Indexing

INDEXING

- O principal objetivo dos índices é ter resultados de busca mais rápidos
- Uma consulta que não usa índices é chamada de *table scan* (o mesmo dos bancos relacionais)

INDEXING

- A função para entender o que acontece em uma query é a explain()

```
> db.italians.find({username: "user101"}).explain()
{
  "cursor" : "BasicCursor",
  "nscanned" : 1000000,
  "nscannedObjects" : 1000000,
  "n" : 1,
  "millis" : 721,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {
  }
}
```

INDEXING

- Podemos ver uma otimização limitando os resultados

```
> db.users.find({username: "user101"}).limit(1).explain() {  
  "cursor" : "BasicCursor",  
  "nscanned" : 102,  
  "nscannedObjects" : 102,  
  "n" : 1,  
  "millis" : 2,  
  "nYields" : 0,  
  "nChunkSkips" : 0,  
  "isMultiKey" : false,  
  "indexOnly" : false,  
  "indexBounds" : { } }
```

INDEXING

- Criar um índice é possível com a função `ensureIndex`

```
> db.users.ensureIndex({"username" : 1})
```

```
> db.users.find({"username" : "user101"}).explain() {  
  "cursor" : "BtreeCursor username_1", "nscanned" : 1,  
  "nscannedObjects" : 1,  
  "n" : 1,  
  "millis" : 3,  
  "nYields" : 0,  
  "nChunkSkips" : 0,  
  "isMultiKey" : false,  
  "indexOnly" : false,  
  "indexBounds" : {  
    "username" : [ [ "user101",  
      "user101"  
    ] ]  
  }  
}
```

COMPOUND INDEXES

- Os índices podem ser compostos pois as vezes existem consultas comuns que fazem um índice apenas não ser suficiente

```
> db.users.find().sort({"age" : 1, "username" : 1})
```

- Para otimizar, é possível criar indice sobre ambos as chaves "age" *and* "username":

```
> db.users.ensureIndex({"age" : 1, "username" : 1})
```

COMPOUND INDEXES

- **Indexing arrays** - You can also index arrays, which allows you to use the index to search for specific array elements efficiently.
- Suppose we have a collection of blog posts where each document was a post. Each post has a "comments" field, which is an array of comment subdocuments. If we want to be able to find the most-recently-commented-on blog posts, we could create an index on the "date" key in the array of embedded "comments" documents of our blog post collection

```
> db.blog.ensureIndex({"comments.date" : 1})
```

- Indexing an array creates an index entry for each element of the array, so if a post had 20 comments, it would have 20 index entries. This makes array indexes more expensive

EXPLAIN E HINT

- Existem basicamente dois tipos de saídas no explain
 - indexed – Geralmente associado a um Btree
 - non-indexed queries – Geralmente está associado a um "BasicCursor".
- Se o Mongo não estiver usando o índice corretamente, é possível enviar hints

```
> db.c.find({"age" : 14, "username" : /.*/}).hint({"username" : 1, "age" : 1})
```

EXPLAIN E HINT

```
> db.users.find({"age" : 42}).explain() {
  "cursor" : "BtreeCursor age_1_username_1",
  "isMultiKey" : false,
  "n" : 8332,
  "nscannedObjects" : 8332,
  "nscanned" : 8332,
  "nscannedObjectsAllPlans" : 8332,
  "nscannedAllPlans" : 8332,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 91,
  "indexBounds" : {
    "age" : [[ 42, 42 ]],
    "username" : [[ { "$minElement" : 1 }, { "$maxElement" : 1 } ] ]
  },
  "server" : "ubuntu:27017" }
```

EXPLAIN E HINT

- "cursor" : "BtreeCursor age_1_username_1" BtreeCursor means that an index was used, specifically, the index on age and username: {"age" : 1, "username" : 1}
- "isMultiKey" : false If this query used a multikey index
- "n" : 8332 - This is the number of documents returned by the query.
- "nscannedObjects" : 8332 This is a count of the number of times MongoDB had to follow an index pointer to the actual document on disk
- "nscanned" : 8332 The number of index entries looked at if an index was used. If this was a table scan, it is the number of documents examined.
- "scanAndOrder" : false If MongoDB had to sort results in memory.
- "indexOnly" : false If MongoDB was able to fulfill this query using only the index entries

EXPLAIN E HINT

- Os índices são uma escolha ruim se suas buscas sempre trazem muitos resultados. Pois com índice são mais operações de lookups
- Em geral, se as queries retornam em média 30% ou mais dos dados de uma coleção, já pode ser interessante comparar o uso do índice vs table scan
- É possível fazer hint para o table scan com {"\$natural" : 1}.

Índices são recomendados	Table Scan funciona bem
Coleções grandes	Coleções pequenas
Documentos grandes	Documentos pequenos
Consultas bem seletivas	Consultas pouco seletivas

Aggregation

AGGREGATE

- O Aggregation framework entrega o conceito de query pipeline. É possível “ligar” uma coleção no início e adicionar transformações por uma série de operações, e eventualmente trazer um resultado como saída (snigger).
- Por exemplo, é possível pegar um resultset, filtrar, agrupar por um campo e somar valores de um grupo em particular. Por exemplo, estimar a população de um estado pelo códigos postais
- Um pipeline pode ser visto como uma cadeia que tranfere JSON
- Uma regra simples é utilizar o find sempre que possível pois consultas simples ele é suficiente
- Sempre que precisar fazer vários finds para tratar uma informação, aggregate é o comando ideal

AGGREGATE

Collection



```
db.orders.aggregate( [  
  { $match: { status: "A" } },  
  { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }  
])
```

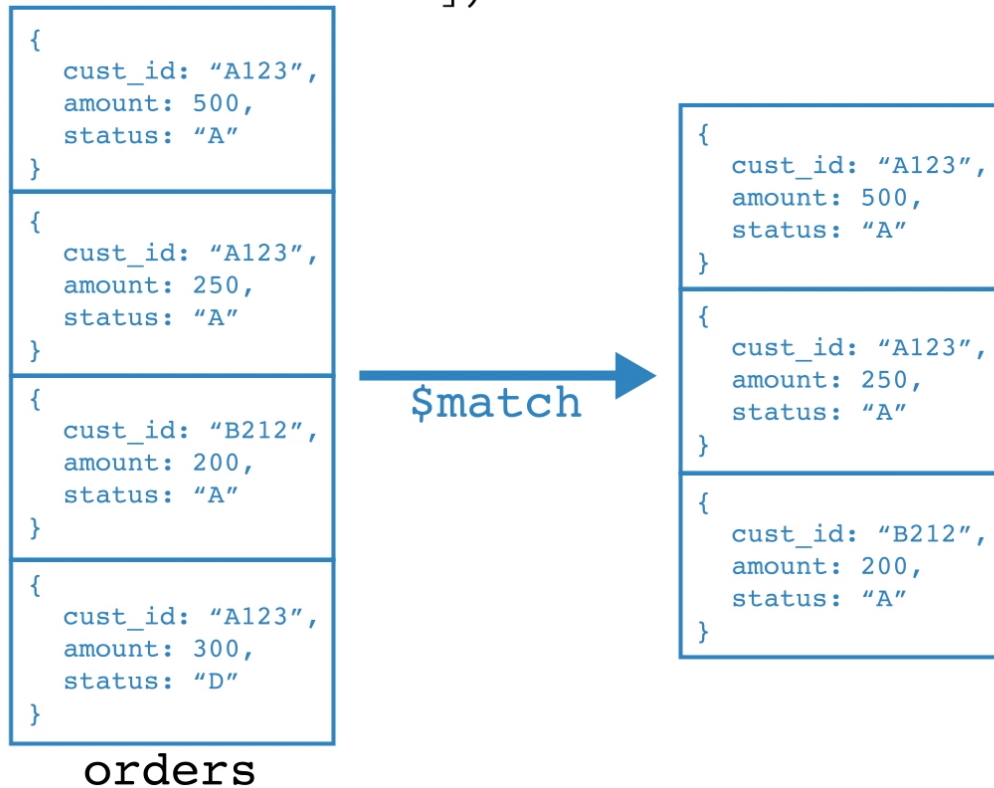
<pre>{ cust_id: "A123", amount: 500, status: "A" }</pre>
<pre>{ cust_id: "A123", amount: 250, status: "A" }</pre>
<pre>{ cust_id: "B212", amount: 200, status: "A" }</pre>
<pre>{ cust_id: "A123", amount: 300, status: "D" }</pre>

orders

AGGREGATE

Collection

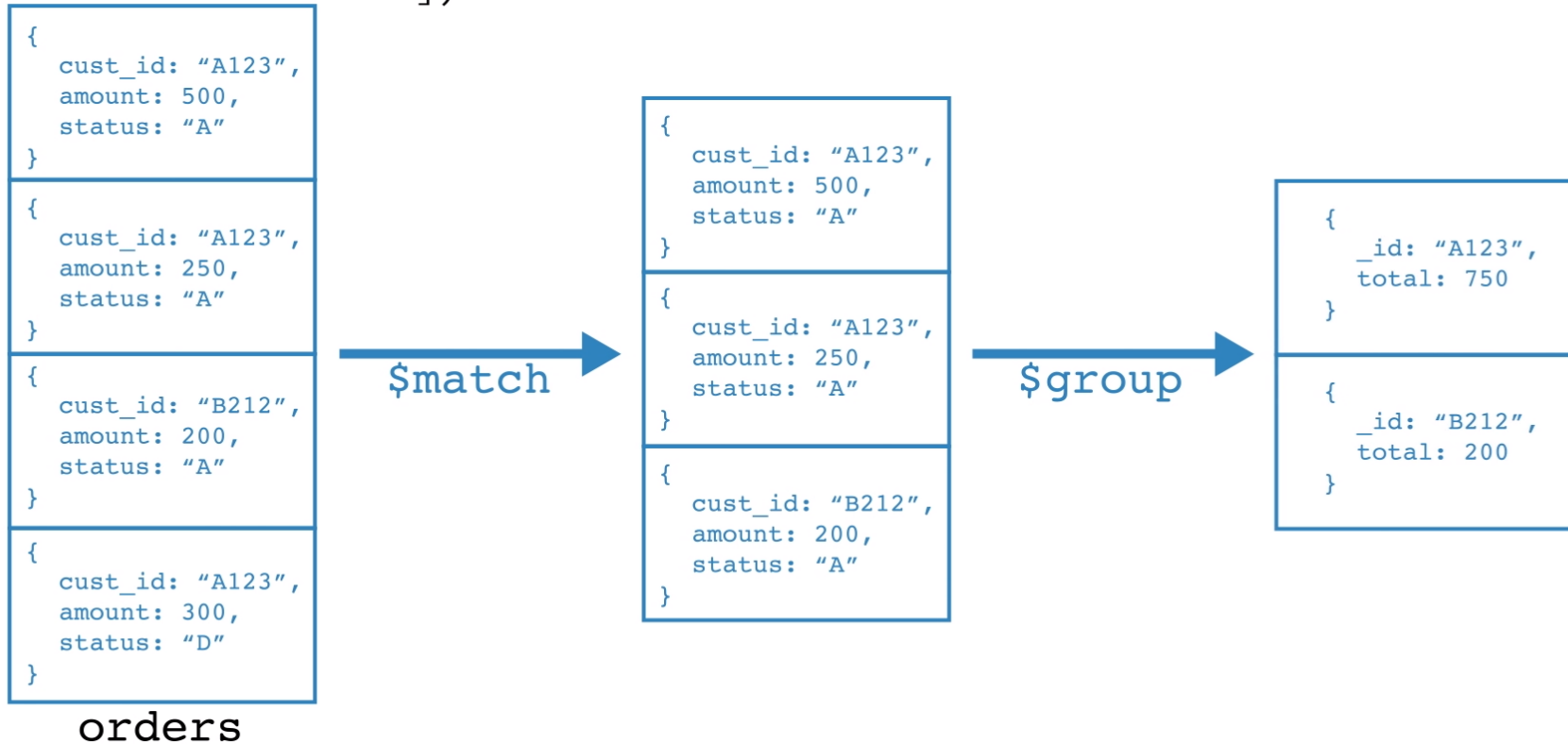
↓
db.orders.aggregate([
 \$match stage → { \$match: { status: "A" } },
 { \$group: { _id: "\$cust_id", total: { \$sum: "\$amount" } } }
)



AGGREGATE

Collection

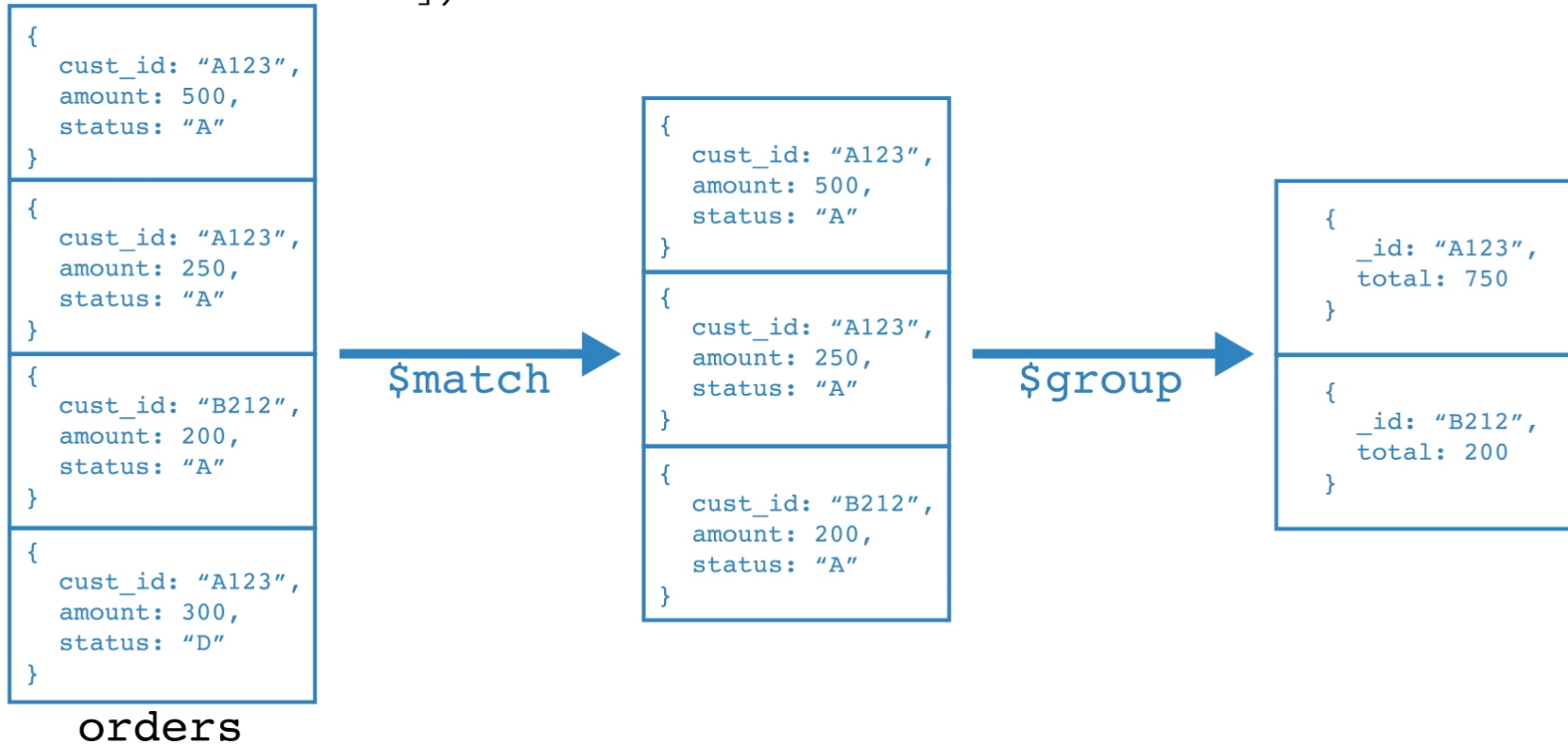
↓
db.orders.aggregate([
 \$match stage → { \$match: { status: "A" } },
 \$group stage → { \$group: { _id: "\$cust_id", total: { \$sum: "\$amount" } } }
])



AGGREGATE

Collection

↓
db.orders.aggregate([
 \$match stage → { \$match: { status: "A" } },
 \$group stage → { \$group: { _id: "\$cust_id", total: { \$sum: "\$amount" } } }
])



AGGREGATE

- Existem diversos operadores de agregação
 - Arithmetic Expression Operators
 - Array Expression Operators
 - Boolean Expression Operators
 - Comparison Expression Operators
 - Conditional Expression Operators
 - Date Expression Operators
 - Literal Expression Operator
 - Object Expression Operators
 - Set Expression Operators
 - String Expression Operators
 - Text Expression Operator
 - Trigonometry Expression Operators
 - Type Expression Operators
 - Accumulators (\$group)
 - Accumulators (in Other Stages)
 - Variable Expression Operators
- Exemplos:
 - Aritméticos : \$avg, \$max, \$min, \$stdDevPop, \$stdDevSamp, \$sum. ...
 - Array: \$filter, \$in, \$arrayToObject, ...
 - Date: \$dateToString, \$dateFromParts. \$dayOfWeek

AGGREGATE

- Usando agregação é em geral a melhor forma de resolver consultas complexas no MongoDB
- Mas o modelo de agregação exige mais conhecimento de todos os operadores...

Usando agregação, procure no nosso grupo de italianos pessoas o mesmo primeiro nome de seu pai (father)

Dica: Use \$match, \$project e \$match



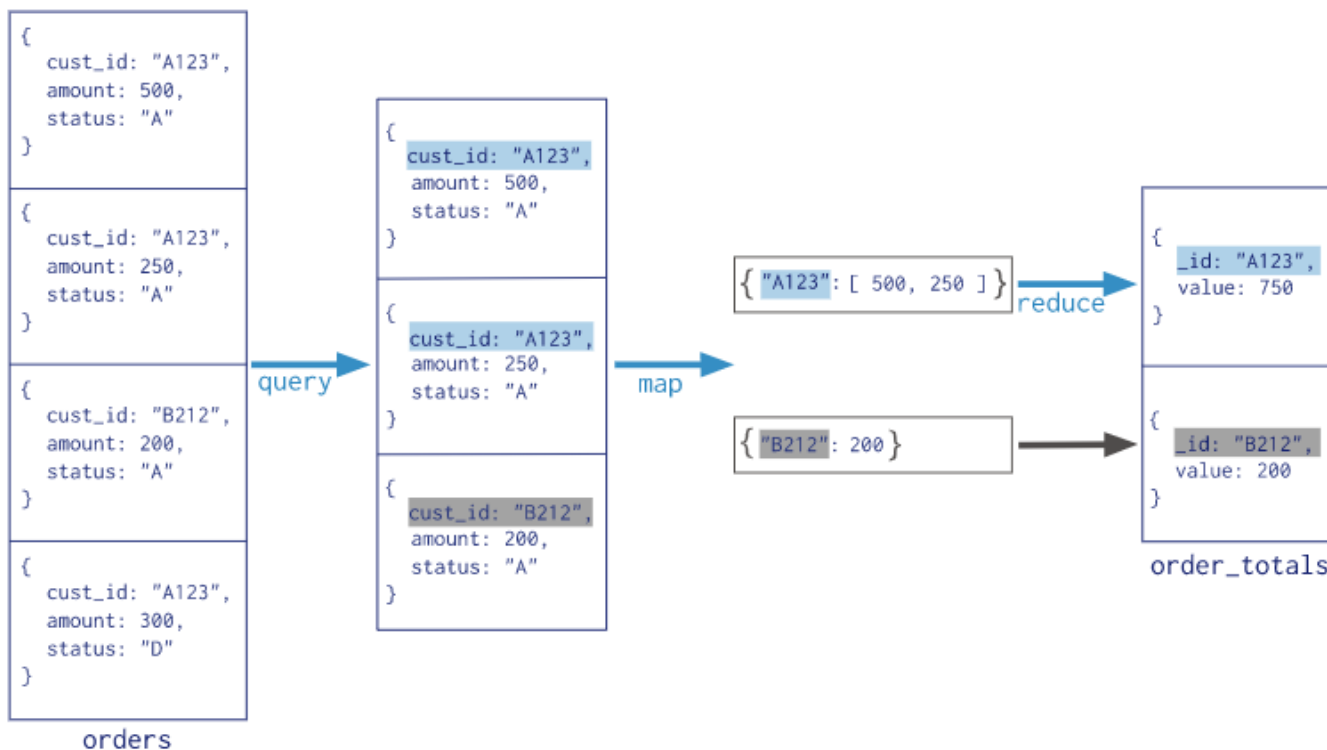
```
db.italians.aggregate([
  {'$match': { father: { $exists: 1} }},
  {'$project': {
    "firstname": 1,
    "father": 1,
    "isEqual": { "$cmp": ["$firstname", "$father.firstname"]}
  }},
  {'$match': {"isEqual": 0}}
])
```

MAP-REDUCE

- Operações de map-reduce ficaram famosas com a demanda de processamento de muitos arquivos
- A ideia do map reduce é executar processos encadeados em duas fases:
 1. map - Processamento de cada documento gerando saídas mapeadas
 2. Reduce – Processo de combinar/reuni múltiplas saídas dos objetos emitidos pelo map
- Em alguns casos, ainda é possível ter um passo adicional que é formatar o resultado
- No MongoDB, Map-Reduce utiliza funções JavaScript para fazer esse processamento de forma mais flexível

MAP-REDUCE

Collection
↓
db.orders.mapReduce(
 map → function() { emit(this.cust_id, this.amount); },
 reduce → function(key, values) { return Array.sum(values) },
 {
 query → { status: "A" },
 output → "order_totals"
 }
)



MAP-REDUCE OU AGGREGATION

- Ambas soluções podem ser usadas como alternativa a queries complexas
- O aggregation framework tem mais operadores
- O map-reduce tem flexibilidade de linguagem (JavaScript)
- Alguns testes indicam mais velocidade para aggregate framework, outros para map-reduce
- É uma questão de escolha

THAT'S ALL



marciogj@gmail.com