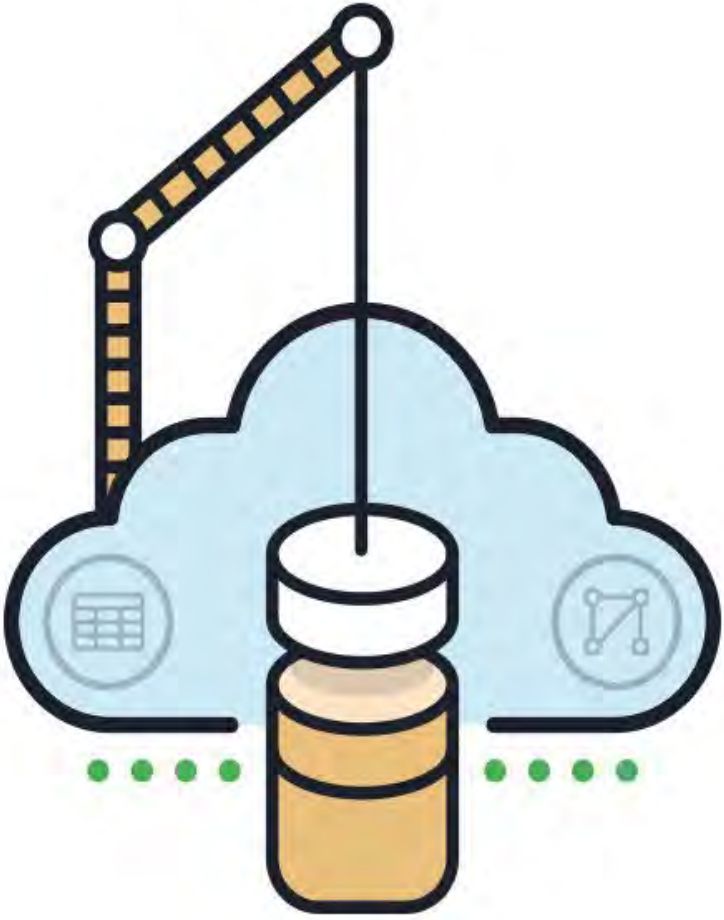
NoSQL

Como armazenar os dados de uma aplicação moderna



Casa do Código

DAVID PANIZ

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

Revisão técnica

Carlos Panato

[2016]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

ISBN

- Impresso e PDF: 978-85-5519-192-3
- Epub: 978-85-5519-193-0
- Mobi: 978-85-5519-194-7

AGRADECIMENTOS

Agradeço a todo o pessoal da Casa do Código e da Caelum, em especial Paulo e Adriano, pela oportunidade que me deram para escrever este livro.

Agradeço também a todos os grandes profissionais com quem trabalhei ao longo dos últimos anos e com quem tive a aportunidade de aprender muito. Principalmente, os líderes e gestores que auxiliaram no meu crescimento e possibilitaram que eu testasse a maior parte das tecnologias que apresentarei neste livro, entre eles Paulo e Guilherme Silveira, Gleicon Morais, Otavio Ferreira e Edward Wible.

Mais do que um agredecimento, dedico esta obra à minha família por todo o apoio e as horas que deixei de passar com eles para investir neste livro: minha esposa Mayra, e meus filhos Luca e Lia.

SOBRE O AUTOR

David Paniz é desenvolvedor de software desde 2005. Já passou por diversos tipos de empresas e projetos, grandes consultorias (prestando serviço para bancos e para BM&F), até pequenas empresas, incluindo times de desenvolvimento notáveis como o da Caelum, e das startups Baby e Nubank.

Ao longo de sua carreira, trabalhou com várias linguagens e plataformas, em especial Java, Ruby e atualmente Clojure, na maior parte do tempo com foco no desenvolvimento back-end e nas práticas ágeis.

Além de manter seu blog pessoal (http://www.davidpaniz.com/), já palestrou em importantes eventos nacionais, como Agile Brazil, Oxente Rails e QCon.

SOBRE O LIVRO

Este livro se destina a desenvolvedores e arquitetos de software que já tenham experiência com algum tipo de banco de dados relacional ou NoSQL, e querem aprender mais sobre os tipos de bancos de dados e entender os impactos que a escolha deles pode trazer para sua arquitetura e seus clientes.

Pré-requisitos

No decorrer do livro, quando um novo conceito ou funcionalidade é apresentado, normalmente existe a comparação com os bancos relacionais e o SQL. Então, ter um conhecimento prévio sobre conceitos básicos de bancos de dados, como tabelas, joins, forma normal (normalização de dados) e um pouco de SQL, ajudará muito o entendimento.

Apesar de o livro abordar exercícios práticos de programação, ele foca totalmente na camada de persistência, com alguns exemplos de código em JavaScript, erlang, bash, e algumas query languages. Por isso, não existe a necessidade de conhecer bem alguma linguagem de programação específica, nem mesmo as utilizadas, mas é importante ter noções de programação para compreender e reproduzir os códigos apresentados.

A maior parte das tecnologias apresentadas pode ser instalada em Windows, Mac ou Linux, e todas elas possuem imagens prontas para rodar em docker containers. Apesar da preferência por ambientes baseados em Unix, é possível reproduzir quase todos os exercícios em qualquer sistema operacional.

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sumário

Capítulo 1

INTRODUÇÃO

1.1 O PROBLEMA DO RELACIONAL

Sempre que alguém ou um time vai começar um novo projeto, algumas decisões importantes são tomadas, como qual ou quais linguagens de programação serão usadas, os principais frameworks e algumas outras decisões sobre a arquitetura. Mas um detalhe importantíssimo e pouco discutido é sobre a forma de persistência de dados.

É comum a discussão sobre qual banco relacional será usado entre MySQL, Postgres, Oracle, ou algum outro. Porém, o uso de um banco de dados relacional é praticamente unânime.

Com o crescimento e popularização da internet, as exigências cobradas de um banco de dados mudaram. Aplicações como portais de notícias exigem um alto volume de leitura por conta da audiência. Aplicações que indexam logs ou geram painéis com indicadores de negócio em tempo real são aplicações com muita escrita para agregar os dados.

Além da questão do número de leituras e escritas. Em muitos casos, a simples questão do volume de dados para armazenar já é um problema. Pense no banco de dados que armazena todos os mais de 1 **bilhão** de usuários ativos no Facebook, além de todo o conteúdo que eles geram diariamente. Ou pense em todos os sites indexados pelo Google e como eles guardam o histórico de buscas

de cada usuário logado.

Além disso, algumas necessidades de negócio são muito complicadas de implementar em cima de um banco relacional. Há anos, áreas de inteligência utilizam Data Warehouse para análise de dados e geração de relatórios, por exemplo.

Baseado nessas novas necessidades e novos modelos de aplicações, um grupo de pessoas começou a ressaltar a importância na forma de armazenar os dados, e daí surgiu o movimento chamado NoSQL. No começo, a sigla era interpretada como 'No SQL' (*Não SQL*, em inglês). Ou seja, era interpretado como um movimento contrário à utilização de um RDBMS (*relational database management system*).

Entretanto, o objetivo principal não era esse. Então, para evitar confusões, assumiu-se que NoSQL é a sigla para 'Not Only SQL' (*Não apenas SQL*, em inglês), ressaltando a importância de se perguntar sobre a melhor ferramenta para a sua necessidade, que pode inclusive ser um banco relacional.

Outra interpretação que 'Não apenas SQL' abre é a utilização de mais de um sistema para armazenamento. Assim como existem aplicações que usam bancos relacionais para as leituras e escritas da aplicação — mas utilizam um Data Warehouse para geração de relatórios —, uma aplicação moderna pode usar um banco relacional para a maior parte das necessidades do sistema — porém usar um outro tipo de banco para uma determinada funcionalidade do sistema, como um sistema de recomendação, cache, execução de tarefas em segundo plano ou alguma outra.

1.2 TIPOS DE BANCOS DE DADOS NÃO RELACIONAIS

Existem diversos bancos não relacionais e, normalmente, eles são rotulados de acordo com a forma como os dados são armazenados. Não se preocupe em entendê-los por enquanto; no decorrer do livro, vamos usar alguns deles. Os principais tipos de bancos são:

- **Chave-valor:** todos os registros fazem parte da mesma coleção de elementos, e a única coisa que todos eles têm em comum é uma chave única;
- **Colunar:** todos os registros fazem parte da mesma tabela, mas cada um deles pode ter colunas diferentes;
- **Documento:** cada registro fica armazenado em uma coleção específica, mas mesmo dentro de uma colação, não existe um esquema fixo para os registros;
- **Grafo:** os registros são nós em um grafo interligados por relacionamentos.

Cada um deles tem suas vantagens e desvantagens, principalmente na hora de consultar e recuperar os registros. Porém, mesmo dentro do mesmo tipo de banco, existem diferentes implementações que podem ter diferenças de performance e escalabilidade.

1.3 O PROBLEMA NA VIDA REAL

Durante o livro, vamos usar um projeto exemplo no qual enfrentaremos alguns dos problemas citados, em que um banco de dados relacional pode não ser exatamente a melhor solução para persistir nossos dados. O projeto que vamos usar é o *ligado*, uma aplicação que contém o perfil de músicos, bandas, álbuns e músicas.

Para começar a explorar um pouco mais do projeto, vamos focar primeiro no cadastro dos álbuns. Na nossa aplicação, além da banda e das músicas, um álbum também possui alguns dados extras, como:

ano de lançamento, ilustrador da capa, produtor ou qualquer outra informação que desejarmos.

Porém, cada disco pode ter mais ou menos informações que os outros. Os usuários devem poder adicionar qualquer informação a um disco, mesmo que seja um tipo de informação que somente um disco terá. Por exemplo, um disco pode ter um campo com o número de semanas que ficou em primeiro lugar na *billboard*, e o número de *singles* que atingiu a posição — mas a maior parte dos discos não consegue nem um *single* sequer na primeira posição e consequentemente nunca terá dados nestes campos.

Outro exemplo interessante é informação sobre o estúdio. Boa parte dos discos mais tradicionais é gravada inteiramente em um único estúdio, mas existem discos gravados em mais de um lugar. Para piorar, também existem discos gravados inteiramente na casa dos músicos, o que acaba com o significado deste campo para alguns discos.

Independente de o campo fazer sentido ou não, quando precisamos obter informações de um determinado assunto, normalmente ou se obtém pouca informação sobre muitos itens, ou muita informação sobre poucos itens. Com o objetivo de catalogar todos os álbuns de um gênero de música específico, é praticamente impossível obter todos os dados, inclusive os menos relevantes de todos os álbuns, incluindo os menos conhecidos. Por isso, mesmo que a informação exista, nós temos de aceitar que não vamos popular todos os dados possíveis para todos os álbuns.

Dado este cenário, o que precisamos é que cada álbum possua uma estrutura diferente, não necessariamente única. Mas enquanto alguns álbuns possuirão apenas nome, artista e músicas, outros possuirão data de lançamento, estúdio e produtor. Porém, poderemos ter álbuns apenas com dados básicos e o estúdio, e outros apenas com os dados básicos e o produtor, e assim por

diante. O que o nosso modelo de dados exige é que cada álbum possa ter todos os dados que conhecemos sobre ele, mas sem afetar os outros álbuns dos quais conhecemos um conjunto de dados diferentes.

Em um banco de dados relacional tradicional, onde existe o conceito de tabela, todos os registros de uma tabela devem possuir a mesma estrutura. Cada tabela possui uma definição de colunas e tipos de dados, colunas obrigatórias, valores padrões e qualquer outra regra que o banco de dados usado suporte. Este conjunto de regras que define uma tabela é chamado de *schema*.

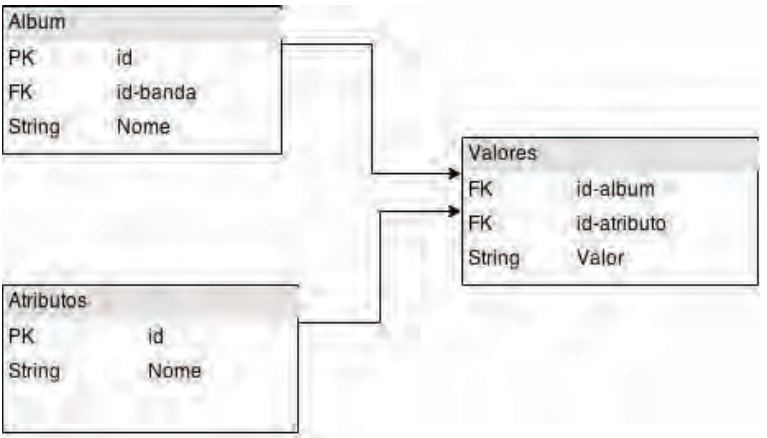
Utilizando um *schema*, temos algumas vantagens sobre como saber exatamente o que esperar de um registro. Se carregarmos um registro aleatório de uma tabela, nós sabemos exatamente quais são as colunas que ele terá e que tipo de valor podemos encontrar em cada uma delas. Mas com o nosso cadastro de álbuns, é exatamente o oposto disso que precisamos.

Reparem que, para suportar este requisito dos álbuns, uma possível solução seria criar várias colunas para qualquer atributo que um álbum possa ter, mesmo que exista apenas um registro que a utilize. A grande desvantagem desta abordagem é que acabaremos com uma tabela com dezenas, talvez centenas, de colunas, e a maior parte das linhas teriam apenas valores vazios.

Embora o banco de dados consiga trabalhar relativamente bem nessas condições, imagine escrever uma cláusula de `INSERT` para uma tabela com, por exemplo, 80 colunas. Imagine como seria a interface exibida para o usuário inserir estes dados. Além desses detalhes, toda vez que precisassem criar um atributo novo para um álbum, seria necessário adicionar uma nova coluna na tabela, o que significa executar uma alteração na estrutura da tabela.

Uma outra solução mais maleável é a criação de uma tabela na

qual definimos os atributos dinâmicos, e outra em que associamos um valor de um tipo predefinido de atributo a um álbum. Existem algumas variações para essa abordagem. Veja como ficaria a modelagem de dados para uma dessas variações mais simples:



Ao inserir os dados, teríamos algo como:

```
Albuns
| id | id-banda |          nome          |
| 1  |     1    | Master of Puppets      |
| 2  |     1    | ...And Justice for All |

Atributos
| id |          nome            |
| 1  | Data de lançamento       |
| 1  | Estudio onde foi gravado |

Valores
| id-album | id-atributo |         valor         |
|     1    |       1     | 03/03/1086            |
|     2    |       1     | 25/08/1088            |
|     1    |       2     | Sweet Silence Studios |
```

Nesse exemplo, vemos que o álbum `Master of Puppets` foi lançado em `03/03/1986` e gravado no estúdio `Sweet Silence Studios`. Já o álbum `...And Justice for All` foi lançado em `25/08/1088` e, mesmo não tendo informações sobre em qual estúdio foi gravado, não adicionamos nenhuma sujeira nas nossas

tabelas.

Embora esta solução resolva nossa necessidade, reparem que para a simples tarefa de exibir os detalhes de um álbum precisaremos de uma `query` um pouco mais complexa, como a seguinte:

```
SELECT atr.nome, val.valor
FROM atributos atr
INNER JOIN valores val
  ON val.id-atributo = atr.id
WHERE val.id-album = 1;
```

Talvez o exemplo dos álbuns possa parecer um pouco exagerado para você, mas pense nas necessidades do catálogo de produtos de um `e-commerce`, no qual existem produtos como roupas que têm tamanhos P, M e G, e sapatos com tamanhos 33, 34 etc., além de TVs com atributos como voltagem e tamanho de tela. Mesmo dentro da categoria de TVs, podemos ter um atributo como número de óculos na caixa, para o caso de TVs 3D. Este tipo de problema está mais próximo do que imaginamos.

Muito do que vamos ver na utilização de bancos de dados NoSQL não é sobre resolver problemas que são impossíveis de serem resolvidos com um banco relacional, mas sim sobre como podemos ter soluções mais elegantes e mais práticas, além de muitas vezes também mais performáticas e escaláveis.

Voltando ao nosso problema do cadastro de álbuns, embora seja possível armazenar os dados em um banco relacional, a estrutura de dados de tabelas com **esquema** predefinido acaba nos trazendo algumas limitações quando precisamos de uma estrutura de dados mais maleável.

Como descrito anteriormente, o tipo de banco NoSQL based em **documento** permite que cada documento de uma coleção tenha um *esquema* único. Ou seja, podemos cadastrar vários álbuns, sendo

que cada um deles pode ter um conjunto de dados diferentes dos outros.

Note que a *query* apresentada há pouco é apenas para exibir os atributos de um álbum específico cujo `id` já conhecemos. Como faríamos para, por exemplo, buscar todos os álbuns lançados em 1986? Usando uma estrutura de dados baseada em documento, podemos facilmente buscar por todos os documentos de uma coleção por um atributo qualquer. Veremos como fazer isso no próximo capítulo.

CAPÍTULO 2

COMEÇANDO COM NOSQL

2.1 CONHECENDO O MONGODB

O primeiro banco NoSQL que usaremos será o MongoDB. Assim como vários outros bancos não relacionais, o MongoDB é um projeto *open source* com distribuição gratuita para Linux, Mac e Windows. Além disso, o MongoDB é provavelmente o banco NoSQL mais usado no mundo atualmente. Embora seja escrito em C++, seu ambiente iterativo e suas buscas são escritas em JavaScript, que também vem ganhando grande força nos últimos anos.

Voltado à principal classificação dos bancos não relacionais, o MongoDB se enquadra na categoria dos bancos com armazenamento baseado em **documentos**. No caso do MongoDB, quando persistimos um **documento** em uma **coleção** — o equivalente a uma **linha** em uma **tabela** —, os dados ficam armazenados em um formato muito semelhante ao JSON, chamado de BSON (http://bsonspec.org/).

Assim como as tabelas possuem **colunas**, um documento possui seus **campos**. Além da terminologia, outro ponto que bancos orientados a documentos diferem dos relacionais é que cada documento de uma coleção pode ter qualquer campo. Ou seja, não existe o que é conhecido nos bancos relacionais por **esquema**.

Termos/conceitos do SQL	Termos/conceitos do MongoDB
Database	Database

Tabela	Coleção
Linha	Documento ou documento BSON
Coluna	Campo
Index	Index
Table join	Documentos aninhado (*embedded*) e vinculados
Chave primária — especifica qualquer coluna única ou uma combinação de colunas como chave primária	Chave primária — No MongoDB, a chave primária é automaticamente definida como campo *_id*
Agregação (*group by*)	Agregação de pipeline

2.2 INSTALANDO O MONGODB

Linux

O MongoDB pode ser instalado no Linux por meio de pacotes `.deb` ou `.rpm`, bastando adicionar em seu gerenciador de pacotes o repositório oficial. Este pode ser encontrado em https://docs.mongodb.org/manual/administration/install-on-linux/.

Por exemplo, para instalar no Ubuntu 14.04, podemos configurar o repositório com os seguintes comandos:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
                --recv 7F0CEB10

$ echo "deb http://repo.mongodb.org/apt/ubuntu \
trusty/mongodb-org/3.0 multiverse" | \
sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

Depois, precisamos atualizar o gerenciador de pacotes e instalar o pacote do Mongo, `mongodb-org`.

```
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
```

Após o término da instalação, usamos `sudo service mongod` para iniciar, parar ou reiniciar o serviço do MongoDB. Para iniciar,

vamos executar o seguinte comando:

```
$ sudo service mongod start
```

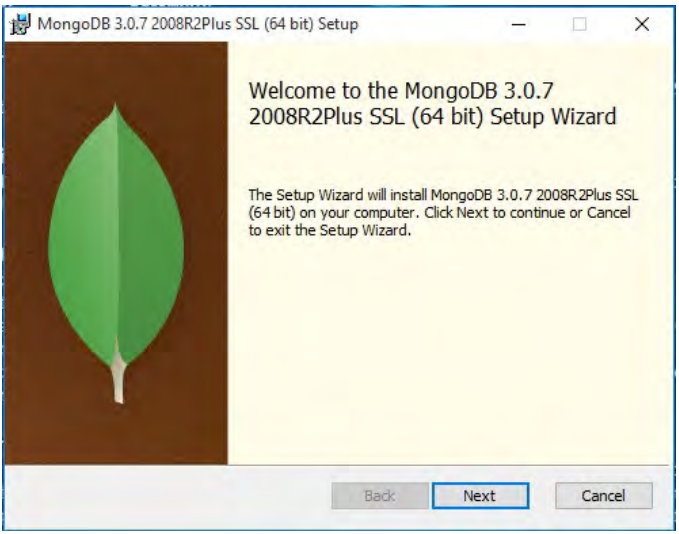
MacOs

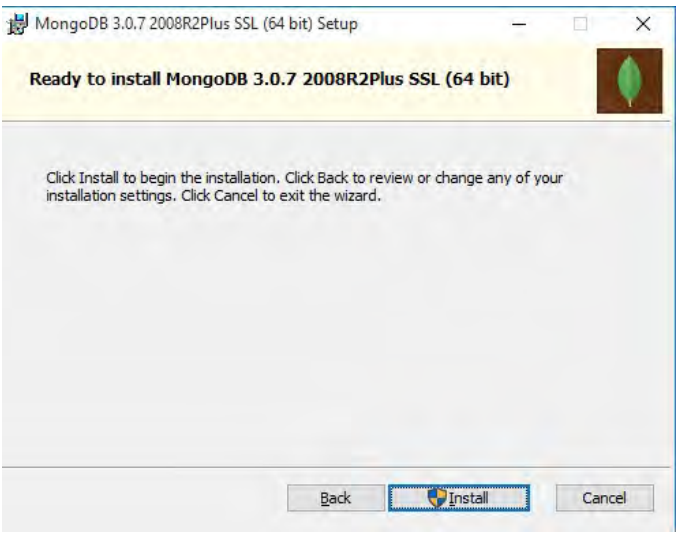
A instalação no MacOS pode ser feita usando o gerenciador *Homebrew*. Basta fazer a atualização dele, e então instalar o pacote `mongodb`.

```
brew update
brew install mongodb
```

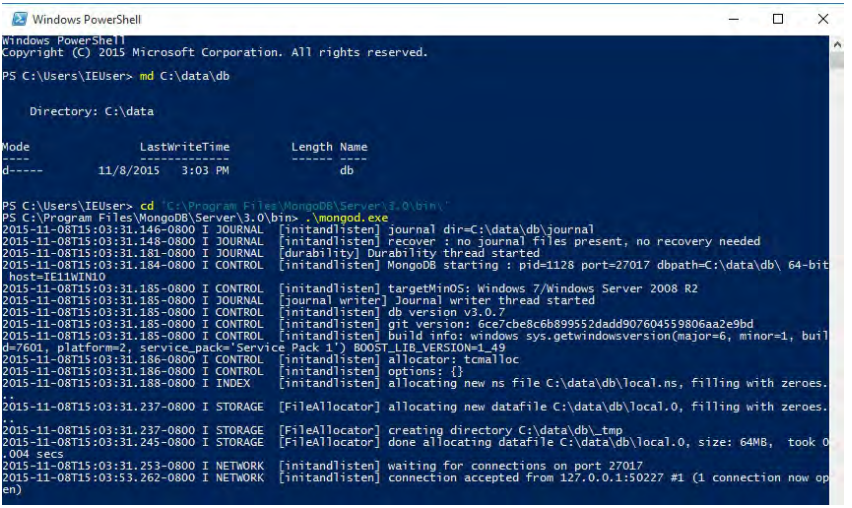
Windows

No Windows, temos a possibilidade de usar um instalador iterativo `.msi`, que pode ser baixado no próprio site, em http://mongodb.org/downloads. Após o download, precisamos executar o instalador e seguir o passo a passo.





Após a instalação, podemos iniciar o servidor do MongoDB usando o arquivo `mongod.exe` no diretório `bin` onde ele foi instalado, por exemplo, `C:\Program Files\MongoDB\Server\3.0\bin\mongod.exe` . Por padrão, ele tentará usar o diretório `C:\data\db\` para armazenar os arquivos, mas este diretório ainda não existe. Logo, devemos criá-lo antes usando o comando `md` no `power shell` .



Outra opção é executar o `mongodb` em um docker container, dispensando a instalação dele no seu sistema operacional. Se preferir esta opção, basta executar `docker run -p 27017:27017 mongo`.

2.3 CRIANDO NOSSO PRIMEIRO DOCUMENTO

Agora que já estamos com o MongoDB instalado, podemos começar a manipular nossos documentos. Assim como nos bancos relacionais, o MongoDB possui o conceito de *banco de dados* (ou *database*, em inglês), que define uma espécie de escopo para coleções, algo como um `namespace`.

Podemos listar todos os bancos de dados que temos na nossa instalação usando o comando `show dbs`. Para escolhermos o banco de dados que vamos utilizar, basta executar o comando `use NOME_DO_BANCO`. Caso o banco ainda não exista, o MongoDB criará um banco vazio com o nome usado.

Para a nossa aplicação *ligado*, criaremos um novo banco de dados homônimo, usando o comando `use ligado`.

Vamos executar todos esses comandos usando o cliente do MongoDB, que é instalado junto com o servidor. Para iniciar o servidor, nós usamos o `mongodb`. Já para iniciar o cliente, vamos usar o `mongo`; ou no Windows o arquivo `mongo.exe` no diretório `bin`, onde o MongoDB foi instalado.

```
$ mongo
MongoDB shell version: 2.6.7
connecting to: test

> show dbs;
admin
test

> use ligado;
switched to db ligado

> show dbs;
admin
test
ligado
```

Para listar todas as coleções que temos no nosso banco de dados, podemos usar o comando `show collections`, equivalente ao `show tables` do MySQL. Como acabamos de criar o banco de dados, por enquanto ele está vazio, sem nenhuma coleção.

Da mesma forma que o MongoDB cria um banco de dados quando tentamos usar um nome que ainda não foi utilizado, para criar uma coleção — o equivalente a uma tabela em um banco relacional —, basta salvar um documento especificando um novo nome de coleção. Como a maneira mais fácil de criar uma coleção é inserindo um documento, é exatamente isso que veremos como fazer agora.

Para inserir um documento, usamos a função chamada `insert`. Como vimos há pouco, para executar comandos no

MongoDB, utilizamos JavaScript; para inserir um novo documento, não será diferente. Semelhante à utilização do JavaScript em um browser — onde temos alguns metaobjetos como `window` e `document` —, no terminal iterativo do MongoDB temos o metaobjeto `db`, que é a raiz dos principais comando que executamos para manipular os dados.

A função `insert` deve ser chamada a partir do nome de uma coleção, que deve ser chamada a partir do nosso objeto base `db`. No final, a chamada que faremos para criar um documento será `db.nome_da_colecao.insert(CORPO_DO_DOCUMENTO)`.

O primeiro documento que criaremos será um álbum vazio na nossa nova coleção chamada de `albuns`.

```
> use ligado;
switched to db ligado

> show collections;

> db.albuns.insert({});
WriteResult({ "nInserted" : 1 })

> show collections;
albuns
system.indexes
```

Agora que nossa coleção `albuns` foi criada, podemos buscar um documento dentro dela. Da mesma maneira que a função `insert` é chamada a partir da coleção, para fazermos uma busca, usaremos a função `find` que deve ser invocada partir da coleção cujos documentos se deseja buscar. No nosso caso, é `albuns`, e faremos da seguinte maneira: `db.albuns.find({})`.

```
> show collections;
albuns
system.indexes

> db.albuns.find({});
{ "_id" : ObjectId("54c023bf09ad726ed094e7db") }
```

Veja que o resultado da execução retornou um **documento** que possui apenas um **campo**, o `_id`. Assim como nos bancos relacionais onde uma tabela deve possuir uma chave primária (que nada mais é do que um identificador único de cada linha da tabela), no MongoDB todas as coleções possuem um campo chamado `_id`, que também funciona como chave primária, mas é gerenciado pelo próprio MongoDB e não temos controle sobre ele.

Toda vez que inserirmos um documento, ele automaticamente gera um `ObjectId`, um tipo especial de BSON com 12 bytes. No caso do meu exemplo, o valor gerado foi `54c023bf09ad726ed094e7db`.

2.4 INSERINDO DOCUMENTOS NO MONGODB

Como visto anteriormente, ao chamar a função `insert`, passamos um parâmetro; no exemplo anterior, utilizamos `{}`. Este argumento que a função espera é o corpo do documento que queremos inserir na coleção. E por se tratar de um ambiente onde usamos JavaScript, a única exigência é que esse corpo seja um objeto JavaScript válido, um JSON.

Para quem não está acostumado com o formato, bem resumidamente, definimos um objeto usando as chaves `{}` e, dentro dela, definimos um grupo de chaves e seus respectivos valores separados pelo caractere `:`. E usamos `,` entre cada um desses pares de chave/valor. Veja um exemplo:

```
{"nome" : "MongoDB",
 "tipo" : "Documento"}
```

Além de Strings, como `"MongoDB"` ou `"Documento"`, os valores também podem ser de outros tipos, como números, datas ou qualquer outro JSON.

```
{"nome"            : "Master of Puppets",
 "dataLancamento"  : new Date(1986, 2, 3),
 "duracao"         : 3286}
```

Trabalhando com datas

Podemos usar `new Date()` para criar um objeto do tipo *Date/Time* com precisão de segundos. Ou seja, um objeto que armazena ano, mês, dia, hora, minuto e segundo.

Se usarmos desta maneira, sem passar nenhum argumento, criaremos um objeto que representa o instante em que a função foi invocada. Outra opção é passar parâmetros para criar um objeto que apresente uma data específica. No exemplo anterior, `new Date(1986, 2, 3)` representa 03/03/1986. Não se esqueça que o segundo argumento, que representa o mês, começa no zero, logo, usamos **2** para representar março.

Agora que já entendemos um pouco melhor o `insert`, vamos criar alguns álbuns para vermos como efetuar buscas no MongoDB.

```
> db.albuns.insert(
        {"nome"            : "Master of Puppets",
         "dataLancamento"  : new Date(1986, 2, 3),
         "duracao"         : 3286})

> db.albuns.insert(
        {"nome"            : "...And Justice for All",
         "dataLancamento"  : new Date(1988, 7, 25),
         "duracao"         : 3929})

> db.albuns.insert(
        {"nome"            : "Peace Sells... but Who's Buying?",
         "duracao"         : 2172,
         "estudioGravacao" : "Music Grinder Studios",
         "dataLancamento"  : new Date(1986, 8, 19)})

> db.albuns.insert(
        {"nome"            : "Reign in Blood",
```

```
         "dataLancamento"  : new Date(1986, 9, 7),
         "artistaCapa"     : "Larry Carroll",
         "duracao"         : 1738})

> db.albuns.insert(
        {"nome"            : "Among the Living",
         "produtor"        : "Eddie Kramer"})
```

2.5 BUSCANDO DOCUMENTOS NO MONGODB

Assim como o `insert`, agora há pouco utilizamos a função `find`, passando como argumento `{}`. Este argumento é chamado de `criteria`, e é uma forma de definir filtros para a busca que desejamos fazer. A primeira busca que faremos é buscar em um campo específico por um valor específico. Podemos, por exemplo, buscar pelo álbum que se chama *Master of Puppets* utilizando `db.albuns.find({"nome" : "Master of Puppets"})`.

```
> db.albuns.find({"nome" : "Master of Puppets"})
{ "_id" : ObjectId("54c6c48d91b5bfb09cb91948"),
  "nome" : "Master of Puppets",
  "dataLancamento" : ISODate("1986-03-03T02:00:00Z"),
  "duracao" : 3286 }
```

Equivalente a:

```
SELECT *
FROM albuns a
WHERE a.nome = "Master of Puppets"
```

Além do `find` que retorna uma lista de documentos que satisfazem os critérios passados como argumento, o MongoDB também disponibiliza a função `findOne`, que retorna apenas um documento: o primeiro que satisfaça as condições.

```
> db.albuns.findOne({"nome" : "Master of Puppets"})
{
    "_id" : ObjectId("54c6c48d91b5bfb09cb91948"),
    "nome" : "Master of Puppets",
    "dataLancamento" : ISODate("1986-03-03T02:00:00Z"),
```

```
    "duracao" : 3286
}
```

Outra diferença entre eles é que o `find` sempre retorna uma lista, mesmo que vazia, caso nenhum documento satisfaça os critérios da busca. Já o `findOne` retorna, ou um documento, ou `null` caso nenhum documento seja encontrado.

```
> db.albuns.find({"nome" : "zzzz"})

> db.albuns.findOne({"nome" : "zzzz"})
null
```

Além da busca por igualdade, é possível fazer buscas mais complexas no MongoDB utilizando os operadores de comparação disponíveis, equivalentes a "maior que", "menor que", "diferente", entre outros.

Nome	Descrição
$gt	Corresponde a valores que são maiores que o valor específico na query.
$gte	Corresponde a valores que são maiores ou iguais ao valor específico na query.
$in	Corresponde a quaisquer valores que existem em um array específico em uma query.
$lt	Corresponde a valores que são menores que o valor específico na query.
$lte	Corresponde a valores que são menores ou iguais que o valor específico na query.
$ne	Corresponde a todos os valores que não são iguais ao valor específico na query.
$nin	Corresponde a valores que **não** existem em um array específico da query.

A sintaxe para utilizar esses operadores é `{"nomeDoCampo" : {"operador" : " valor "}}` . Por exemplo, para buscar os álbuns com duração menor que 30 minutos, procuraremos os documentos cujo campo `duracao` seja *menor que* `1800` . Estamos armazenando a duração em segundos, por isso usamos o valor 1800, que é o equivalente a 60 multiplicado por 30.

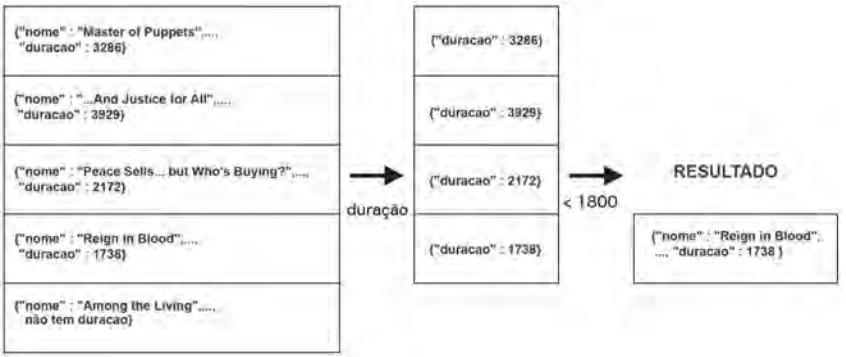
```
> db.albuns.find({"duracao" : {"$lt" : 1800}})
```

```
{ "_id" : ObjectId("54c6c49e91b5bfb09cb9194b"),
  "nome" : "Reign in Blood",
  "dataLancamento" : ISODate("1986-10-07T03:00:00Z"),
  "artistaCapa" : "Larry Carroll", "duracao" : 1738 }
```

Equivalente a:

```
SELECT *
FROM albuns a
WHERE a.duracao < 1800
```

Nesta última busca, criamos um critério para encontrar apenas os álbuns com o campo `duracao` inferior a `1800`. Porém, entre os dados que inserimos, existe o álbum "Among the Living", para o qual não informamos o campo `duracao`. A busca do MongoDB primeiro cria uma espécie de tabela virtual apenas com os documentos que possuem o campo que será filtrado, depois aplica o filtro especificado, removendo os documentos que não satisfazem os critérios da busca, e finalmente retorna os documentos que sobraram.



Agora que estamos começando a nos familiarizar com a interface de buscas do MongoDB, podemos tentar montar a query que retorna todos os álbuns lançados em 1986. A maneira mais simples para fazer esse tipo de busca por intervalos, seja de datas ou de números, é aplicando dois filtros: um onde o campo deve ser

maior ou igual que o início do intervalo **E** *menor que* o fim do intervalo.

Nós já vimos como utilizar os operadores de comparação. Então, a única coisa que falta para conseguirmos efetivamente fazer essa busca é aprendermos a utilizar os operadores lógicos e aplicarmos o **E**.

Nome	Descrição
$and	Junta *query clauses* com uma lógica **E** retorna todos os documentos que combinam com ambas condições.
$nor	Junta *query clauses* com uma lógica **NEM** retorna todos os documentos que falham em combinar ambas as condições.
$not	Inverte o efeito de uma *query expression* e retorna os documentos que não combinam com a condição.
$or	Junta *query clauses* com uma lógica **OU** retorna todos os documentos que combinam com ambas condições.

A utilização dos operadores lógicos é um pouco diferente dos operadores de comparação. Na maior parte dos casos, a sintaxe é *{operador : [expressão 1, expressão 2, expressão n]}*. A chave do objeto é o operador lógico, e o valor é um array de outros critérios.

Os filtros que queremos executar são: data de lançamento *maior ou igual que* 01/01/1986 — que é `{"dataLancamento" : {$gte : new Date(1986, 0, 1)}}` — **E** data de lançamento *menor que* 01/01/1987 — ou `{"dataLancamento" : {$lt : new Date(1987, 0, 1)}}` . No final, teremos:

```
 > db.albuns.find(
   {$and : [{"dataLancamento" : {$gte : new Date(1986, 0, 1)}},
            {"dataLancamento" : {$lt :  new Date(1987, 0, 1)}}]}
)

{ "_id" : ObjectId("54c6c48d91b5bfb09cb91948"),
 "nome" : "Master of Puppets",
 "dataLancamento" : ISODate("1986-03-03T02:00:00Z"),
 "duracao" : 3286 }
{ "_id" : ObjectId("54c6c49891b5bfb09cb9194a"),
```

```
  "nome" : "Peace Sells... but Who's Buying?",
  "duracao" : 2172,
  "estudioGravacao" : "Music Grinder Studios",
  "dataLancamento" : ISODate("1986-09-19T03:00:00Z") }
{ "_id" : ObjectId("54c6c49e91b5bfb09cb9194b"),
  "nome" : "Reign in Blood",
  "dataLancamento" : ISODate("1986-10-07T03:00:00Z"),
  "artistaCapa" : "Larry Carroll",
  "duracao" : 1738 }
```

Equivalente a:

```
SELECT *
FROM albuns a
WHERE a.dataLancamento >= '1986-01-01 00:00:00'
  AND a.dataLancamento < '1987-01-01 00:00:00'
```

Nesta busca, aplicamos os dois filtros no mesmo campo e, por isso, ela poderia ser um pouco simplificada. Lembrem de que a primeira busca que fizemos foi utilizando *{"nomeDoCampo" : {**alguma restrição**}}*.

Notem que neste objeto estamos especificando uma regra para ser aplicada em um campo. A grande questão é que essa regra pode ser composta. Podemos usar uma variação dessa sintaxe da seguinte maneira: *{"nomeDoCampo" : {**comparador1** : "valor1", **comparador2** : "valor2"}}*.

Para a nossa busca de um intervalo de datas, o resultado seria:

```
> db.albuns.find({"dataLancamento" :
       {"$gte" : new Date(1986, 1, 1),
        "$lt"  : new Date(1987, 1, 1)}})
```

Infelizmente, essa sintaxe sempre funciona como o operador **E**. Caso queira utilizar outro operador lógico, mesmo que filtrando o mesmo campo, precisará usar a sintaxe tradicional dos operadores lógicos.

2.6 REMOVENDO DOCUMENTOS NO

MONGODB

No mundo relacional, o grande segredo para apagar registros de uma tabela é saber como filtrá-los. O que realmente importa em um comando SQL para apagar registros é o que vai dentro do `WHERE`. Em resumo, se você sabe como efetuar buscas, é só trocar o `SELECT` pelo `DELETE`, que você não terá problemas.

No MongoDB, não é diferente. As regras e sintaxes para filtrar os documentos são idênticas. Porém, em vez de usar a função `find`, utilizaremos a função `remove`, também invocada a partir da coleção.

O comando para buscar o álbum cujo nome é `"...And Justice for All"` é:

```
> db.albuns.find({"nome": "...And Justice for All"})
{ "_id" : ObjectId("54c6c49391b5bfb09cb91949"),
  "nome" : "...And Justice for All",
  "dataLancamento" : ISODate("1988-08-25T03:00:00Z"),
  "duracao" : 3929 }
```

Equivalente a:

```
SELECT *
FROM albuns a
WHERE a.nome = "Master of Puppets"
```

Para apagá-lo, basta usar o função `remove` no lugar de `find`.

```
> db.albuns.remove({"nome": "...And Justice for All"})
WriteResult({ "nRemoved" : 1 })
```

Equivalente a:

```
DELETE
FROM albuns a
WHERE a.nome = "Master of Puppets"
```

Apagando todos os dados de uma coleção

Sempre que precisar remover documentos, tome muito cuidado. Existe uma boa prática de sempre começar o comando pela busca e, quando tiver certeza de que está retornando apenas os documentos que realmente quer apagar, substitua o `find` pelo `remove`.

Esta dica também vale para o SQL! Comece seu comando com um `SELECT` e, somente depois que sua cláusula `WHERE` estiver completa e testada, substitua pelo `DELETE`.

Vale lembrar de que, assim como `DELETE FROM albuns` apaga **todos** os registros de uma tabela sem especificar nenhuma cláusula `WHERE`, executar `db.albuns.remove({})` sem nenhum critério também removerá todos os documentos de sua coleção.

2.7 ALTERANDO DOCUMENTOS NO MONGODB

Da mesmo forma que para apagar documento no MongoDB, o grande segredo para atualizar um documento é saber como filtrá-los. A diferença é que, ao chamar a função necessária, a `update`, normalmente passamos dois argumentos: uma `query` e os dados que queremos atualizar. Por exemplo, *db.albuns.update(query, **novos valores**)*.

Vamos aproveitar e adicionar a duração no álbum `"Among the Living"` da seguinte maneira:

```
> db.albuns.find({"nome" : "Among the Living"})
{ "_id" : ObjectId("54c828b5342dda43e93bee1e"),
```

```
  "nome" : "Among the Living",
  "produtor" : "Eddie Kramer" }

> db.albuns.update({"nome" : "Among the Living"},
                   {"duracao" : 3013})
WriteResult({ "nMatched" : 1,
              "nUpserted" : 0,
              "nModified" : 1 })
```

O resultado que o MongoDB nos retorna informa que a `query` que passamos retornou um documento e que um documento foi modificado. Porém, vejamos o que aconteceu com nossos álbuns:

```
>  db.albuns.find({})
{ "_id" : ObjectId("54c023bf09ad726ed094e7db") }

{ "_id" : ObjectId("54c6c48d91b5bfb09cb91948"),
  "nome" : "Master of Puppets",
  "dataLancamento" : ISODate("1986-03-03T02:00:00Z"),
  "duracao" : 3286 }

{ "_id" : ObjectId("54c6c49891b5bfb09cb9194a"),
  "nome" : "Peace Sells... but Who's Buying?",
  "duracao" : 2172,
  "estudioGravacao" : "Music Grinder Studios",
  "dataLancamento" : ISODate("1986-09-19T03:00:00Z") }

{ "_id" : ObjectId("54c6c49e91b5bfb09cb9194b"),
  "nome" : "Reign in Blood",
  "dataLancamento" : ISODate("1986-10-07T03:00:00Z"),
  "artistaCapa" : "Larry Carroll",
  "duracao" : 1738 }

{ "_id" : ObjectId("54c828b5342dda43e93bee1e"),
  "duracao" : 3013 }
```

Reparem que o último documento, cujo `"_id"` é `"54c828b5342dda43e93bee1e"`, foi atualizado com a duração de 3013 segundos, mas todos os outros dados sumiram. Da maneira que usamos o `update`, o MongoDB substitui o documento todo com os novos dados, com exceção da `_id`, que nunca muda.

Embora essa utilização possa ser útil às vezes, não era exatamente o que queríamos. Assim como temos algumas palavras-

chaves que usamos na hora das buscas, como os operadores lógicos e de comparação, o MongoDB também disponibiliza uma espécie de operador que faz com que ele altere apenas as chaves que informarmos. Este operador de atualização é o `$set`.

Para adicionar ou alterar apenas o campo `"duracao"` de um documento, podemos usar: *db.albuns.update(query, { **$set** : { "duracao" : 3013}})*.

Como já executamos a atualização anterior, se fizermos o filtro pelo nome `"Among the Living"`, não encontraremos nenhum álbum, por isso farei o filtro pelo campo `_id`. Tome cuidado dobrado ao executar o código a seguir, pois este campo é gerado automaticamente e, muito provavelmente, o valor será diferente nos seus testes. Vamos ver o resultado de algumas buscas com os dados atuais:

```
> db.albuns.find({"nome" : "Among the Living"})

> db.albuns.find({"_id" : "54c828b5342dda43e93bee1e"})
```

Embora fosse esperado que a busca por nome não encontrasse nenhum documento, a nossa busca pelo `_id` também retornou uma lista vazia. Talvez você já tenha notado que o `_id` dos álbuns não são do tipo `String`, e sim do tipo `ObjectId`. Assim como as datas, os dados do campo `_id` são objetos.

Para criar um `ObjectId`, basta usar a sintaxe: *ObjectId(**SEU-ID**)*. Agora que sabemos usar criar um `ObjectId`, fazer a filtro pelo campo `_id` é simples. Apenas lembre-se de usar o valor que foi gerado no seu banco.

Para o valor que estou usando de exemplo, a busca ficaria: `db.albuns.find({"_id" : ObjectId("54c828b5342dda43e93bee1e")})`. Agora que sabemos como encontrar o álbum que queremos atualizar, já podemos fazer

a atualização usando o operador `$set` para adicionarmos os campos que foram removidos acidentalmente na nossa atualização anterior.

```
> db.albuns.find({"_id" : ObjectId("54c828b5342dda43e93bee1e")})
{ "_id" : ObjectId("54c828b5342dda43e93bee1e"),
  "duracao" : 3013 }

> db.albuns.update(
       {"_id" : ObjectId("54c828b5342dda43e93bee1e")},
       {$set : {"nome"     : "Among the Living",
                "produtor" : "Eddie Kramer"}})
WriteResult({ "nMatched" : 1,
              "nUpserted" : 0,
              "nModified" : 1 })

> db.albuns.find({"_id" : ObjectId("54c828b5342dda43e93bee1e")})
{ "_id" : ObjectId("54c828b5342dda43e93bee1e"),
  "duracao" : 3013,
  "nome" : "Among the Living",
  "produtor" : "Eddie Kramer" }
```

Agora sim nossos álbuns possuem nome, produtor e duração. Finalmente temos todos os dados que queríamos, mas ainda não criamos o relacionamento dos nossos álbuns com as bandas/artistas. Agora que temos álbuns persistidos em documentos em uma coleção, como faremos para criar o relacionamento com outra coleção?

Capítulo 3

MAIS SOBRE BANCOS ORIENTADOS A DOCUMENTOS

3.1 CRIANDO RELACIONAMENTOS

Se queremos criar algum tipo de relação entre os álbuns e os artistas, a primeira coisa que faremos é criar a nova coleção na qual armazenaremos os dados destes artistas. Desta vez, embora usando a função `insert` , vamos passar um array de documentos em vez de apenas um para criarmos vários documentos em apenas uma operação.

```
> db.artistas.insert([ {"nome" : "Metallica"},
                       {"nome" : "Megadeath"},
                       {"nome" : "Slayer"},
                       {"nome" : "Anthrax"} ])

BulkWriteResult({
    "writeErrors" : [ ],
    "writeConcernErrors" : [ ],
    "nInserted" : 4,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]

> db.artistas.find({})
{ "_id" : ObjectId("54d1562cf7bb967d7b976d07"),
  "nome" : "Metallica" }
{ "_id" : ObjectId("54d1562cf7bb967d7b976d08"),
```

```
  "nome" : "Megadeath" }
{ "_id" : ObjectId("54d1562cf7bb967d7b976d09"),
  "nome" : "Slayer" }
{ "_id" : ObjectId("54d1562cf7bb967d7b976d0a"),
  "nome" : "Anthrax" }
```

Equivalente a:

```
INSERT
  INTO artistas (nome)
  VALUES ("Metallica"), ("Megadeath"), ("Slayer"), ("Anthrax");
```

Agora que temos alguns artistas cadastrados, podemos voltar ao problema de como criar a ligação entre um álbum e o seu artista. No mundo relacional, existe o conceito de chave estrangeira, em que criamos uma coluna de uma tabela que é, de certa forma, ligada à chave primária de outra. Quando criamos uma chave estrangeira, dependendo do banco de dados que estamos usando, ganhamos vantagens como integridade referencial (que nos proíbe de inserir um valor de `id` que não existe na outra tabela), remoção em cascada (que automaticamente apaga todos os registros de uma tabela que estavam ligados a um registro que foi apagado), entre outros.

Uma das formas de criar o relacionamento entre os álbuns e os artistas é copiar este conceito de chave estrangeira, e introduzir um campo nos documentos dos álbuns, onde o valor será uma cópia do campo `_id` do artista relacionado. Por exemplo, no documento do álbum cujo nome é `"Master of Puppets"`, podemos adicionar um campo `artista_id` com o valor `ObjectId("54d1562cf7bb967d7b976d07")`, que é o `id` gerado para o artista cujo nome é `"Metallica"`.

Vamos fazer isso para todos os álbuns previamente cadastrados.

```
> db.albuns.update(
    {"nome" : "Master of Puppets"},
    {$set :
      {"artista_id" : ObjectId("54d1562cf7bb967d7b976d07")}}
  );
```

```
WriteResult({ "nMatched" : 1,
              "nUpserted" : 0,
              "nModified" : 1 })

> db.albuns.update(
    {"nome" : "Peace Sells... but Who's Buying?"},
    {$set :
      {"artista_id" : ObjectId("54d1562cf7bb967d7b976d08")}}
  );
WriteResult({ "nMatched" : 1,
              "nUpserted" : 0,
              "nModified" : 1 })

> db.albuns.update(
    {"nome" : "Reign in Blood"},
    {$set :
      {"artista_id" : ObjectId("54d1562cf7bb967d7b976d09")}}
  );
WriteResult({ "nMatched" : 1,
              "nUpserted" : 0,
              "nModified" : 1 })


> db.albuns.update(
    {"nome" : "Among the Living"},
    {$set :
      {"artista_id" : ObjectId("54d1562cf7bb967d7b976d0a")}}
  );
WriteResult({ "nMatched" : 1,
              "nUpserted" : 0,
              "nModified" : 1 })
```

Após ligar os álbuns aos artistas relacionados, podemos começar a trabalhar com ambas coleções, como por exemplo, listar todos os álbuns de uma banda. Para evitar adicionar complexidade de outra linguagem de programação e problemas como conexão ao banco, vamos continuar usando o `mongo Shell` .
Mas, a partir de agora, usaremos um pouco mais de JavaScript como declaração de variáveis, condicionais e laços simples.

Basicamente, toda vez que executarmos uma função por meio do `db` , seria o equivalente a uma chamada ao banco dentro da sua aplicação, enquanto a manipulação de variáveis e outras atividades básicas estariam escritas na linguagem principal, fora da camada de

persistência, também dentro da sua aplicação.

Neste instante, temos apenas um álbum por artista, então vamos adicionar mais alguns álbuns antes.

```
> db.albuns.insert(
    {"nome"            : "...And Justice for All",
     "dataLancamento"  : new Date(1988, 7, 25),
     "duracao"         : 3929,
     "artista_id"      : ObjectId("54d1562cf7bb967d7b976d07")}
  );
WriteResult({ "nInserted" : 1 })

> db.albuns.insert(
    {"nome"            : "Metallica (The Black Album)",
     "dataLancamento"  : new Date(1991, 7, 12),
     "duracao"         : 3751,
     "artista_id"      : ObjectId("54d1562cf7bb967d7b976d07")}
  );
WriteResult({ "nInserted" : 1 })

> db.albuns.insert(
    {"nome"            : "So Far, So Good... So What!",
     "dataLancamento"  : new Date(1989, 0, 19),
     "duracao"         : 2066,
     "artista_id"      : ObjectId("54d1562cf7bb967d7b976d08")}
  );
WriteResult({ "nInserted" : 1 })

> db.albuns.insert(
    {"nome"            : "South of Heaven",
     "dataLancamento"  : new Date(1988, 6, 5),
     "duracao"         : 2214,
     "artista_id"      : ObjectId("54d1562cf7bb967d7b976d09")}
  );
WriteResult({ "nInserted" : 1 })

> db.albuns.insert(
    {"nome"            : "State of Euphoria",
     "dataLancamento"  : new Date(1988, 8, 19),
     "duracao"         : 3155,
     "artista_id"      : ObjectId("54d1562cf7bb967d7b976d0a")}
  );
WriteResult({ "nInserted" : 1 })
```

Para a primeira iteração nas coleções, recuperaremos todos os álbuns de um artista, buscando pelo nome dele.

```
> var artista = db.artistas.findOne({"nome" : "Metallica"});

> artista
{ "_id" : ObjectId("54d1562cf7bb967d7b976d07"),
  "nome" : "Metallica" }

> var albuns = db.albuns.find({"artista_id" : artista._id})

> albuns.forEach( function(album) {
                    print(album["nome"]);
                  });
Master of Puppets
...And Justice for All
Metallica (The Black Album)
```

Nesse código, na primeira consulta, armazenamos o resultado em uma variável chamada `artista`, que utilizamos na segunda busca para definir o `_id` de artista que seria usado para filtrar os álbuns. Na segunda busca, armazenamos o resultado em uma segunda variável chamada `albuns`, que contém todos os álbuns do artista que buscamos.

Na última linha, invocamos a função `forEach` que recebe como argumento outra função que será executada para cada álbum contido naquela variável. Neste exemplo, criamos uma função que escreve apenas o campo `"nome"` do álbum, por isso o resultado exibido foi apenas o nome dos álbuns cujo artista é o Metallica.

3.2 LISTANDO ÁLBUNS E O PROBLEMA "N+1 QUERY"

A partir de um artista, foi fácil encontrar os seus álbuns e exibir informações relevantes deles. Agora tentaremos fazer o contrário. Vamos novamente executar a busca pelos álbuns lançados em 1986, mas, desta vez, além de exibir os dados do álbum, vamos exibir também o nome do artista responsável por ele.

```
> var albuns = db.albuns.find(
    {"dataLancamento" : {"$gte" : new Date(1986, 1, 1),
```

```
                        "$lt"  : new Date(1987, 1, 1)}}
  );

> albuns.forEach( function(album) {
                    print(album["nome"]);
                    print(album["artista_id"]);
                  })
Peace Sells... but Who's Buying?
ObjectId("54d1562cf7bb967d7b976d08")
Reign in Blood
ObjectId("54d1562cf7bb967d7b976d09")
Master of Puppets
ObjectId("54d1562cf7bb967d7b976d07")
```

Notem que o único local no qual temos o nome do artista armazenado é na coleção de artistas. Neste último exemplo, a partir do documento de um álbum, conseguimos encontrar o `_id` do artista relacionado. Porém, para efetivamente exibir o nome dele, precisaremos de mais uma consulta.

```
> var albuns = db.albuns.find(
    {"dataLancamento" : {"$gte" : new Date(1986, 1, 1),
                         "$lt"  : new Date(1987, 1, 1)}}
  );

> albuns.forEach( function(album) {
                    var artista = db.artistas.findOne(
                      { "_id" : album["artista_id"]}
                    );
                    print(album["nome"], artista["nome"]);
                  });
Peace Sells... but Who's Buying? Megadeath
Reign in Blood Slayer
Master of Puppets Metallica
```

Embora tenhamos atingido nosso objetivo e conseguido exibir o nome do álbum e do artista junto, algo muito sutil e perigoso pode ter passado despercebido. Reparem que a busca pelo artista acontece dentro do nosso laço, o que significa que, na verdade, não fizemos apenas uma busca, mas sim três. Para piorar ainda mais, se nossa busca inicial por álbuns lançado em 1986 tivesse retornado centenas de registros, faríamos uma busca na coleção de artista para cada um deles, o que tende a levar a grandes problemas de performance.

O problema que enfrentamos na nossa última busca é tão comum que tem até um nome comum na literatura: **n+1 query**. O **n+1** significa que o total de queries a ser executada será 1 da consulta inicial *mais* **n**, onde **n** é o número de registros/documentos encontrados. Ou seja, se formos buscar todos os álbuns lançados em 1986 e encontrarmos três documentos, no total faremos quatro operações no banco de dados.

Usando o paradigma relacional, podemos resolver este problema facilmente usando o `JOIN`. Para o exemplo anterior, podemos limitar o **n+1** e retornar tanto os dados dos álbuns quanto dos artistas, com a seguinte query:

```
SELECT al.nome, ar.nome
FROM albuns al
  INNER JOIN artistas ar
    ON al.artista_id = ar.id
WHERE al.dataLancamento >= '1986-01-01 00:00:00'
  AND al.dataLancamento < '1987-01-01 00:00:00'
```

Infelizmente, no MongoDB, não temos uma operação equivalente ao `JOIN` ou parecido. Enquanto no mundo relacional é comum ouvir falar sobre *formas normais* e evitar ao máximo dados duplicados, se seguirmos à risca essas técnicas de modelagem de dados, podemos acabar com a performance da nossa aplicação quando usarmos um banco orientado a documentos.

Uma forma diferente de modelar este relacionamento, que é bem comum em bancos orientados a documentos, é **aninhando** os dados do artista dentro do documento do álbum. Por exemplo, o primeiro álbum que cadastramos, `Master of Puppets`, poderia ter o seguinte conteúdo:

```
{"nome"            : "Master of Puppets",
 "dataLancamento"  : new Date(1986, 2, 3),
 "duracao"         : 3286
 "artista"         : {"nome" : "Metallica"}}
```

3.3 TRABALHANDO COM DOCUMENTOS EMBUTIDOS

Além de `Date` e `ObjectId`, um **subdocumento** também é um valor válido para um campo de um documento. Na documentação do MongoDB, este padrão é chamado de **Embedded Document**, comumente traduzido como *Documento aninhado* em vez de *Documento embutido*. Embora o termo *embedado* não exista em português, não é difícil encontrar referencias ao padrão como *Documento Embedado*, ou com os idiomas misturados como *Documento Embedded*.

Vamos cadastrar alguns álbuns usando este padrão para entendermos como a iteração com os dados muda comparado com o padrão de *Referência de documento*, usado anteriormente.

```
> db.albuns.insert(
    {"nome"            : "Somewhere Far Beyond",
     "dataLancamento"  : new Date(1992, 5, 30),
     "duracao"         : 3328,
     "artista"         : {"nome" : "Blind Guardian"}});
WriteResult({ "nInserted" : 1 })

> db.albuns.insert(
    {"nome"            : "Imaginations from the Other Side",
     "dataLancamento"  : new Date(1995, 3, 4),
     "duracao"         : 2958,
     "artista"         : {"nome" : "Blind Guardian"}});
WriteResult({ "nInserted" : 1 })

> db.albuns.insert(
    {"nome"            : "Nightfall in Middle-Earth",
     "dataLancamento"  : new Date(1998, 3, 28),
     "duracao"         : 3929,
     "artista"         : {"nome" : "Blind Guardian"}});
WriteResult({ "nInserted" : 1 })

> db.albuns.insert(
    {"nome"            : "Tunes of War",
     "dataLancamento"  : new Date(1996, 7, 25),
     "duracao"         : 3165,
     "artista"         : {"nome" : "Grave Digger"}});
```

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

> db.albuns.insert(
    {"nome"            : "Knights of the Cross",
     "dataLancamento"  : new Date(1998, 4, 18),
     "duracao"         : 3150,
     "artista"         : {"nome" : "Grave Digger"}});
WriteResult({ "nInserted" : 1 })
```

A primeira coisa que pode parecer estranho apenas olhando para esse código é a quantidade de vezes que o nome da banda foi copiado, mas a ideia é exatamente essa. Vejamos por que isso pode ser melhor, buscando os álbuns lançados em 1998 e exibindo o nome do álbum e o nome do artista relacionado.

```
> var albuns = db.albuns.find(
    {"dataLancamento" : {"$gte" : new Date(1998, 1, 1),
                         "$lt"  : new Date(1999, 1, 1)}}
  );

> albuns.forEach( function(album) {
                    print(album["nome"],
                          album["artista"]["nome"]);
                  });
Nightfall in Middle-Earth Blind Guardian
Knights of the Cross Grave Digger
```

Agora que não temos um `ObjectId` para cada banda, como podemos buscar todos os álbuns de uma determinada banda? Assim como podemos buscar pelo campo `nome` do álbum, também podemos buscar por campos específicos de um subdocumento. Para isso, usamos a sintaxe `{ "campo_doc_principal" : {"campo_sub_doc" : valor }}`.

Além da busca por igualdade, todos os outros operadores também são suportados. Vejamos como fica a busca para exibir todos os álbuns da banda `"Blind Guardian"`.

```
> db.albuns.find({"artista" : {"nome" : "Blind Guardian"}});
{ "_id" : ObjectId("54d941a8f7bb967d7b976d10"),
  "nome" : "Somewhere Far Beyond",
  "dataLancamento" : ISODate("1992-06-30T03:00:00Z"),
  "duracao" : 3328,
```

```
  "artista" : { "nome" : "Blind Guardian" } }
{ "_id" : ObjectId("54d94273f7bb967d7b976d11"),
  "nome" : "Imaginations from the Other Side",
  "dataLancamento" : ISODate("1995-04-04T03:00:00Z"),
  "duracao" : 2958,
  "artista" : { "nome" : "Blind Guardian" } }
{ "_id" : ObjectId("54d9427af7bb967d7b976d12"),
  "nome" : "Nightfall in Middle-Earth",
  "dataLancamento" : ISODate("1998-04-28T03:00:00Z"),
  "duracao" : 3929,
  "artista" : { "nome" : "Blind Guardian" } }
```

Persistindo o documento do artista dentro do documento do álbum, temos acesso a todos os dados dele a partir do documento pai, o do álbum. Resolvemos o problema do **n+1**, mas, em contra partida, temos dados duplicados. A cada dia, o preço e a qualidade do armazenamento são melhores. Hoje podemos comprar uma unidade de estado sólido (SSD) mais barato do que um disco rígido (HD) de 5600 rpm há alguns anos. Ter alguns dados duplicados não costuma ser um grande problema, mas sempre com bom senso.

3.4 DESNORMALIZANDO OS DADOS

Dada a situação atual, poderíamos dizer que os dados duplicados não são um problema, e que o padrão de subdocumentos é melhor do que o padrão de referência dos documentos para o nosso cenário. Mas, e se além do nome também precisássemos armazenar mais informações sobre o artista, como biografia, integrantes, local de origem e site?

O que antes eram apenas alguns bytes duplicados passariam a ser muitos dados duplicados. Além disso, embora o nome e o local de origem não mudem com frequência, a biografia e os integrantes tendem a ser atualizados mais frequentemente; e, para tal, seria preciso atualizar os documentos de todos os álbuns do artista em questão.

Vimos há pouco que nos bancos relacionais, o `join` ajuda

muito quando precisamos de dados de mais de uma tabela. O problema é que, mesmo com todas as otimizações que esse tipo de estrutura de dados tem, quando precisamos unir dados de várias tabelas, é comum cairmos em uma operação bem custosa.

Uma técnica comum para evitar muitos `joins` é copiar os dados que normalmente vamos precisar de uma tabela para outra. Outra situação em que os dados são duplicados em bancos relacionais é para manter históricos.

Imagine um sistema de vendas no qual temos uma tabela de produtos, uma de vendas e uma de itens da venda. O preço dos produtos fica na tabela de produtos, mas se não copiarmos este valor para a tabela itens da venda, no momento da venda, ao atualizar o preço de um produto, estaríamos mudando o valor de todas as vendas dele.

Podemos usar uma abordagem semelhante a esta para resolver nosso problema de modelagem. Se tivermos muitos dados no documento, embuti-lo em outro documento vai introduzir muitos dados duplicados e, além disso, gerará complexidades para atualizar seus dados e garantir a integridade. Entretanto, se usarmos o padrão de referência de documento, cairemos no problema de **n+1** query quando os dados necessários não estiverem acessíveis.

A sugestão nesta situação é usar a referência de documentos para definir o relacionamento, mas duplicar apenas os dados que realmente serão necessários no outro documento para evitar o **n+1**. No nosso exemplo, significa que vamos cadastrar todos os artistas na coleção `artistas`, onde teremos várias informações sobre eles além do nome; e na coleção `albuns` nossos documentos terão um campo `artista_id` e um campo `nome_artista`, onde duplicaremos apenas o nome do artista.

Neste instante, nosso banco possui álbuns cadastrados usando

os dois padrões. Agora que já decidimos como ficará nossa modelagem, devemos padronizar os documentos. Para os primeiros álbuns que cadastramos, usando o padrão de referência de documento, precisaremos adicionar o campo `nome_artista` em todos eles. Dado que temos apenas 4 artistas por enquanto, podemos iniciar a operação pelo artista. Ou seja, carrego um artista e atualizo todos os seus álbuns, adicionando o nome nele. Veja o exemplo:

```
> var metallica =
      db.artistas.findOne({"nome" : "Metallica"});
> metallica;
{ "_id" : ObjectId("54d1562cf7bb967d7b976d07"),
  "nome" : "Metallica" }

> db.albuns.count({"artista_id" : metallica["_id"]});
3

> db.albuns.update(
    {"artista_id" : metallica["_id"]},
    {$set : {"nome_artista" : metallica["nome"]}})
WriteResult({ "nMatched" : 1,
              "nUpserted" : 0,
              "nModified" : 1 });
```

Embora a nossa busca retorne três álbuns que pertencem ao artista `"Metallica"` , o resultado do update diz que apenas um álbum foi atualizado. Isso aconteceu porque a função update, por padrão, atualiza apenas um documento. Isso é uma espécie de trava de segurança para evitar que alguém destrua o banco todo por engano.

Para atualizar todos os documentos que atenderem a busca, é necessário passar um terceiro argumento com a chave `multi` , com valor `true` : *db.colecao.update({BUSCA}, {ATUALIZACAO}, **{multi : true}**)*.

Para atualizar todos os álbuns, vamos executar um código como o anterior, mas usando o `multi` para atualizar todos os álbuns do

artista. Desta vez, em vez de carregar apenas um artista, vamos executar o update dentro de um laço a partir de todos os artistas.

```
> var artistas = db.artistas.find({});
> artistas.forEach( function(artista) {
    print("Atualizando albuns de ", artista["nome"]);
    var result = db.albuns.update(
                    {"artista_id" : artista["_id"]},
                    {$set : {"nome_artista" : artista["nome"]}},
                    {multi : true});
    print(result);
});
Atualizando albuns de  Metallica
WriteResult({ "nMatched" : 3,
              "nUpserted" : 0,
              "nModified" : 2 })
Atualizando albuns de  Megadeath
WriteResult({ "nMatched" : 2,
              "nUpserted" : 0,
              "nModified" : 2 })
Atualizando albuns de  Slayer
WriteResult({ "nMatched" : 2,
              "nUpserted" : 0,
               "nModified" : 2 })
Atualizando albuns de  Anthrax
WriteResult({ "nMatched" : 2,
              "nUpserted" : 0,
               "nModified" : 2 })
```

A FUNÇÃO COUNT

A função `count` faz o mesmo que o `COUNT` no SQL, que retorna o número de registros/documentos que satisfazem a busca.

```
> db.albuns.count({"artista_id" : UM_ID})
```

Isso é equivalente a:

```
SELECT COUNT(*)
FROM albuns
WHERE artista_id = UM_ID
```

Agora que nossos álbuns que usam referência de documento já estão como gostaríamos, precisamos atualizar os outros. Para aqueles que usamos o padrão de documentos embutidos, precisaremos remover o campo `artista`, onde estávamos armazenando o outro documento além de criar um novo na coleção de artistas para aqueles que estavam embutidos nos álbuns.

Para remover um campo de um documento temos algumas opções. Uma delas é usar a função `update` passando o documento inteiro, exatamente como estava, mas sem o campo que não queremos. Outra opção é usar o operador `$unset`, que serve justamente para remover um ou mais campos de um documento. A sintaxe é a seguinte: *db.albuns.update({BUSCA}, {**$unset** : {CAMPO_PARA_REMOVER : true, OUTRO_CAMPO_PARA_REMOVER : true}})*

Além de remover o documento embutido, também vamos precisar adicionar os campos `artista_id` e `nome_artista` referentes ao artista. Para isso, precisaremos ter o documento do artista já criado. O que faremos então é criar o documento do artista e atualizar o documentos dos álbuns, adicionando os novos campos e removendo o antigo documento embutido.

```
> db.artistas.insert({"nome" : "Blind Guardian"});
WriteResult({ "nInserted" : 1 })
> db.artistas.insert({"nome" : "Grave Digger"});
WriteResult({ "nInserted" : 1 })
> var artistas =  db.artistas.find(
    { $or : [{"nome" : "Blind Guardian"},
             {"nome" : "Grave Digger"}]});
> artistas.forEach( function (artista) {
    print("Atualizando albuns de ", artista["nome"]);

    var result = db.albuns.update(
                  {"artista.nome" : artista["nome"]},

                  {$unset : {"artista" : true},
                   $set   : {"artista_id"   : artista["_id"],
                             "nome_artista" : artista["nome"]}},
```

```
                 {multi : true}
                );
    print(result);
});
Atualizando albuns de  Blind Guardian
WriteResult({ "nMatched" : 3,
              "nUpserted" : 0,
               "nModified" : 3 })
Atualizando albuns de  Grave Digger
WriteResult({ "nMatched" : 2,
              "nUpserted" : 0,
              "nModified" : 2 })
```

Neste ponto, já vimos algumas das principais vantagens e desvantagens em relação aos bancos de dados orientados a documentos, como também algumas dicas úteis sobre modelagem, e as funções e operadores mais importantes para o dia a dia usando MondoDB. Além disso, nossos aplicativo `ligado` já está com os dados de álbuns e artistas cadastrados de maneira efetiva, o que nos deixa livres para o próximo desafio, que será uma área de votações.

A modelagem de dados para este serviço de votação é bem simples. Teremos uma entidade com o título da eleição, uma outra com os candidatos, e uma última com os votos. Provavelmente, é possível criar uma modelagem eficiente para este serviço tanto em um banco relacional quanto em uma banco orientado a documentos. O grande desafio deste serviço será disponibilidade, performance e consistência.

Imagine que este serviço estará na capa de grandes portais da internet e servirá para eliminação de candidatos de um reality show musical muito popular. Definitivamente nós não queremos que o serviço fique indisponível, e também não queremos perder os votos. Será que o MongoDB é o melhor candidato para este problema? Quais as vantagens e desvantagens dele? Quais são as outras opções que nós temos?

Capítulo 4

CONSISTÊNCIA VERSUS DISPONIBILIDADE E OS BANCOS CHAVE VALOR

Para o nosso serviço de votação, um dos pontos mais complexos para atingir é a alta disponibilidade (*High Availability*, ou simplesmente *HA*, em inglês). Justamente por conta desta complexidade, vamos focar neste tópico primeiro.

Alta disponibilidade significa manter o serviço funcionando pelo maior tempo possível, ou seja, sem quedas ou indisponibilidades. Quando temos essa necessidade, a primeira coisa a trabalhar é aumentar o número de servidores.

Todos os servidores podem parar de responder por diversos motivos, seja uma falha de software ou hardware, queda de rede, energia ou mesmo falha humana, como alguém desligando ou reiniciando o servidor acidentalmente. Backups geralmente só servem para *disaster recovery*, ou seja, para criarmos um ambiente novo ou recuperarmos os dados caso um desastre aconteça.

Para termos alta disponibilidade de verdade, devemos ter, no mínimo, dois servidores e, de preferência, em lugares diferentes. Mesmo usando *cloud computing* (computação na nuvem), na qual só usamos servidores virtuais, boa parte dos provedores do serviço costumam oferecer opções de zonas e regiões, para que você

distribua seus servidores virtuais entre *data centers* diferentes. Dessa maneira, se alguma coisa acontecer em um data center e perdermos um dos nossos servidores, nosso serviço deve continuar rodando apenas com o outro.

Partindo do pressuposto de que temos mais de um servidor de banco de dados, existem várias estratégias diferentes para trabalhar com eles, incluindo quantos e quais servidores vamos ler e escrever, como eles vão compartilhar os dados entre si e, o mais importante, o que vai acontecer se eles perderem a comunicação ou um deles parar de responder.

4.1 CAP THEOREM

O *CAP Theorem*, também conhecido como *Brewer's Theorem*, é um dos mais famosos trabalhos da ciência da computação, em que é discutida a relação entre **Consistência** (*Consistency*), **Disponibilidade** (*Availability*) e **Tolerância a particionamento** (*Partition-tolerance*) (BROWNE, 2009). Segundo o autor, todos estes três itens são desejados, mas é impossível ter todos simultaneamente.

Antes de entender como cada item afeta o outro, vamos primeiro ver o significado de cada um deles neste contexto.

- **Consistência** (*Consistency*) — Significa que a partir do momento que uma operação é executada, o estado dos dados é alterado e qualquer operação posterior usará o mesmo estado até que ele seja modificado novamente. Ou seja, se um registro foi modificado, qualquer busca futura deverá retornar o valor atualizado.
- **Disponibilidade** (*Availability*) — Significa que o banco tem de estar disponível tanto para leitura quanto para escrita, mesmo que um ou mais servidores estejam

enfrentando problemas, ou mesmo desligados.

- **Tolerância a particionamento** (*Partition-tolerance*) — Significa que os servidores podem perder a comunicação entre eles por tempo indeterminado.

A abordagem mais comum para aumentar a disponibilidade é a famosa replicação *master slave*. Nós criamos um servidor que será o principal (*master*), e um outro secundário com uma cópia dos dados chamado de *slave*. Nesta abordagem, as escritas obrigatoriamente só podem ser feitas no *master*, que automaticamente replica os dados para o *slave*. Já as leituras podem ir para ambos.

Mesmo dentro desta arquitetura, temos decisões que impactam o quanto de consistência, disponibilidade e tolerância a particionamento temos. Por exemplo, se o banco *master* confirmar a escrita antes de propagar para o *slave*, uma consulta neste banco secundário pode retornar dados desatualizados (*stale*). Isso significa que nossa consistência já foi comprometida, mesmo que por apenas alguns milissegundos. Se por acaso tivermos uma falha no servidor *master*, antes de ele propagar as mudanças para o *slave*, podemos literalmente perder dados! Se ambos estiverem rodando perfeitamente, mas a conexão entre eles for perdida, você consegue continuar escrevendo no *master* e continuar lendo dados, mesmo

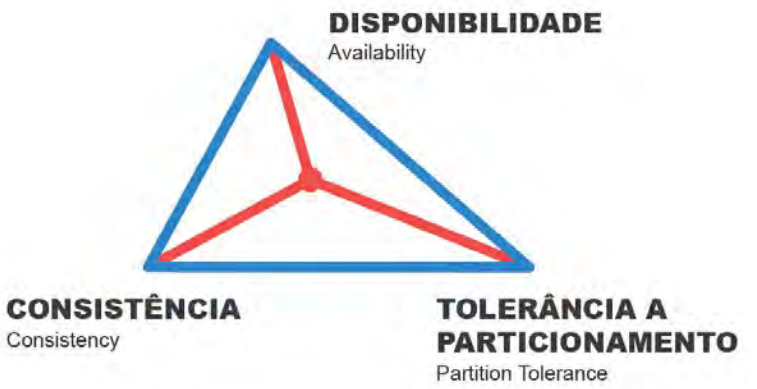
que desatualizados, no *slave*.

Resumindo, nesta abordagem, em caso de falha de comunicação (**Tolerância a particionamento**), podemos manter o sistema respondendo (**Disponibilidade**), mas com dados desatualizados, ou seja, sem **Consistência**. Esta abordagem é classificada como **cAP** ou **AP**, pois foca nos pontos *A* e *P* do CAP.

Ainda usando replicação *master slave*, podemos mudar um pouco a estratégia para aumentar a consistência. Usando o conceito de transação distríbuida, só vamos considerar o dado salvo de verdade após todas as réplicas confirmarem a escrita. Desta forma, qualquer leitura em qualquer banco sempre retornará o dado correto, ou seja, 100% de *Consistência*.

Em contra partida, com essa estratégia, aumentamos a latência. Como é preciso confirmar em todos os servidores, a operação de escrita fica bem mais lenta. Além disso, caso ocorra a falha de comunicação entre os servidores, nenhuma escrita poderá ser confirmada, consequentemente nenhuma escrita funcionará. Ou seja, durante uma falha de comunicação (**Tolerância a particionamento**), não teremos dados desatualizados (**Consistência**), mas perderemos as operações de escrever, consequentemente perdemos **Disponibilidade**. Por focar nos pontos *C* e *P* do CAP, esta solução pode ser classificada como **CaP** ou **CP**.

Até pouco tempo atrás, era comum simplificarem o CAP Theorem como *"escolha 2 entre os 3"*. Mas após uma atualização recente (BREWER, 2012), é mais aceito que cada um dos itens não é uma escolha binária, na qual ou você tem ou não tem, mas sim uma escolha do quanto você quer de cada item. Para garantir na totalidade um dos itens, é preciso abrir mão de outro. Entretanto, existem arquiteturas nas quais conseguimos ter um pouco de cada.



Vale lembrar de que as falhar de rede (particionamento) devem ser eventos raros, mas que com certeza vão ocorrer mais cedo ou tarde. Como são eventos esporádicos, além de entender como o seu banco de dados se comporta durante as falhas, também é muito importante entender como sua replicação se comporta enquanto a comunicação está funcionando.

Em um dos exemplos de implementação *master slave*, temos a confirmação da escrita mais rápida, com baixa *latência*, mas abrimos mão de consistência. Já na outra implantação, para garantir a consistência, aceitamos aumentar a *latência*. Além da relação entre consistência e disponibilidade durante uma falha de rede, que é o foco do CAP, também é importante avaliarmos a relação entre consistência e performance (latência) enquanto os servidores não enfrentam problemas de comunicação.

Existem estudos do Google e da Amazon mostrando quedas nas buscas e vendas, respectivamente, com o aumento do tempo de resposta (KOHAVI; LONGBOTHAM, 2007). Para a internet moderna, entender e minimizar a latência podem ser encarados como necessidades de negócio. Por causa dessas necessidades especiais, algumas empresas criaram seus próprios banco de dados,

como Google com o Bigtable (GHEMAWAT; HSIEH; WALLACH; BURROWS et al., 2006), Amazon com DynamoDB (JAMPANI; KAKULAPATI; LAKSHMAN; PILCHIN et al., 2007) e Facebook com Cassandra (LAKSHAMN; MALIK, 2009).

Embora a primeira configuração de replicação *master slave* citada garanta disponibilidade mesmo com falha de rede, o fato de todas as escritas obrigatoriamente irem para o mesmo servidor pode gerar problemas de performance em caso de muitas escritas. Além disso, estamos dependendo de um único servidor, o que é conhecido com *single point of failure (SPOF)*, ou *único ponto de falha*, em português. Isso significa que basta um servidor parar para que a aplicação esteja comprometida.

Como temos uma réplica, basta eleger aquele servidor como o novo *master* que já teremos as escritas reestabelecidas, mas ainda teremos um *downtime* mesmo que de apenas alguns segundos. Segundo nossa especificação, precisamos garantir as escritas, então precisamos de uma outra forma de replicação de dados.

Alguns bancos NoSQL possibilitam HA usando uma arquitetura de replicação *masterless*. Ou seja, uma arquitetura na qual qualquer server (normalmente chamado de *node* nesta arquitetura) pode receber requisições tanto de leitura quanto de escrita. Consequentemente, mesmo com falha de rede ou em problema em algum servidor, o *cluster* (como é chamado o grupo de servidores) consegue manter leituras e escritas disponíveis.

Como é de se imaginar, a consistência é comprometida. Além disso, podemos ter um problema que a escrita concorrente introduz, que é o conflito. Conflitos acontecem quando temos escritas divergentes em dois nodes diferentes. Quando estão sincronizando os dados dentro do cluster, é preciso tratar qual dos dados é o correto. Existem algumas formas de tratar essas divergências, e cada banco de dados vai oferecer opção diferentes.

Para atender nossa necessidade de alta disponibilidade de escrita, vamos utilizar um banco de dados que já suporte este modelo de replicação sem um servidor master. Agora que já escolhemos a forma de replicação dos dados, precisamos escolher o tipo de banco que desejamos usar.

4.2 O MODELO CHAVE/VALOR E O RIAK

O serviço de votação não exige um modelo de dados dinâmico, nem buscas complexas. Precisaremos apenas armazenar alguns dados do usuário e em qual artista ele votou. Os bancos de dados do tipo **chave/valor** (*key/value*, em inglês) são extremamente simples. Fazendo uma comparação com os bancos relacionais, é como se o banco de dados inteiro fosse uma única tabela que possui apenas duas colunas: uma `chave`, que é a chave primária da tabela, e uma outra `valor`, de um tipo binário como `BLOB` ou similar, no qual o banco de dados não tem nenhum tipo de controle sobre o tipo de dado.

Estes tipos de bancos, embora muito simples, costumam ser bem rápidos e são extremamente úteis. Existem alguns problemas comuns em que a solução mais prática é usar um banco *key/value*. Um exemplo é compartilhar os dados de sessão de usuário entre vários servidores de aplicação, conhecido como *session storage*. Outra utilização muito comum em aplicação web é para armazenar pedaços de HTML dinâmico pré-processados, evitando acesso ao disco e processamento de templates em todas as requisições, conhecido como *fragment cache* ou *page cache*.

Entre os bancos de dados do tipo **chave/valor**, provavelmente os mais usados são *Redis* e *Memcached*. Mas devido à nossa necessidade de alta disponibilidade de escrita, usaremos o **Riak**. Além de um modelo de replicação de dados muito avançado que possibilita uma configuração muito flexível entre disponibilidade,

consistência e latência, o Riak é um projeto de código aberto, escrito em Erlang, com distribuição para Linux e MacOS.

4.3 INSTALANDO O RIAK

A instalação do Riak em Linux e Mac é muito semelhante a do MongoDB. Existem pacotes oficiais para *Debian*, *Ubuntu*, *Red Hat*, *Fedora*, *Mac* e outros disponíveis no site da *Basho*, a empresa mantenedora do Riak (http://docs.basho.com/riak/latest/downloads/).

Infelizmente, o Riak depende de algumas funções `bash` e, por isso, não é compatível com Windows. Apesar de isso ser impedimento para produção, felizmente hoje em dia é muito fácil trabalhar com virtualização. Se você usa Windows, pode instalar uma ferramenta de virtualização, como *VirtualBox*, *VMWare*, ou qualquer outro de sua preferência, e rodar uma máquina virtual com Linux. Ou ainda, pode instalar o **Docker** e rodar o Riak em um contêiner.

Diferente dos outros bancos de dados deste livro, o Riak não possui uma imagem de docker padrão, mas várias opções podem ser encontradas no http://dockerhub.com. Inclusive, criei uma para acompanhar este livro, a *davidpaniz/riak*. Para subir um docker container com o Riak, basta executar:

```
docker run -p 8087:8087 -p 8098:8098 davidpaniz/riak
```

A instalação no Mac é muito semelhante à do MongoDB, basta usar o *Homebrew* e instalar com `brew install riak` . Já no Linux, comparado com o MongoDB, a instalação local é ainda mais fácil. Existe um script que detecta automaticamente a distribuição e já configura o gerenciador de pacotes. Por exemplo, para instalar no Ubuntu, basta executar os seguintes comando:

```
$ curl -s https://packagecloud.io/\
```

```
install/repositories/basho/riak/script.deb.sh | sudo bash
$ sudo apt-get install riak=2.1.1-1
```

4.4 MANIPULANDO OBJETOS

Assim como MongoDB, o Riak também disponibiliza um console para manipularmos nossos dados. A grande diferença é que o console do Riak utiliza Erlang, uma linguagem que não é tão popular quanto o JavaScript. Embora exista essa barreira inicial de usar uma linguagem nova, a estrutura de dados é tão simples que provavelmente não teremos problemas manipulando nossos dados pelos console.

Para abrir o console, vamos usar o comando `riak console`. Dentro do console, para podermos manipular os dados, vamos precisar configurar em qual servidor nos conectaremos, e depois criar uma conexão com ele. Para isso, vamos usar a função `riak:client_connect`, que recebe como argumento o endereço do servidor a que vamos nos conectar.

Se o servidor já estiver rodando, é possível que você veja uma mensagem de erro parecida com: `Node is already running - use 'riak attach' instead`. Se isso acontecer, basta usar `riak attach`.

```
$ riak console

(riak@127.0.0.1)1> Servidor = 'riak@127.0.0.1'.
'riak@127.0.0.1'

(riak@127.0.0.1)2> net_adm:ping(Servidor).
pong

(riak@127.0.0.1)3> {ok,Conexao} = riak:client_connect(Servidor).
{ok,{riak_client,['riak@127.0.0.1',undefined]}}
```

Nesse código, a primeira linha cria uma variável chamada `Servidor`, e atribui o valor `'riak@127.0.0.1`, que é o usuário

padrão e o endereço da sua instalação local. Na segunda linha, a função `net_adm:ping` não é necessária, mas assim como o comando `ping`, ela testa a conectividade com o host. Na terceira linha, finalmente usamos a função `riak:client_connect`, que efetivamente cria a conexão com o servidor, e caso o resultado seja ok, vamos atribuir a conexão à variável `Conexao`. A partir do momento em que já temos a conexão estabelecida, podemos executar nossas operações no banco de dados.

Para criarmos um objeto no Riak, precisamos de três valores: `chave`, `valor` e `bucket`. A maior parte dos bancos chave/valor usa apenas dois parâmetros, mas o Riak possibilita uma separação entre as chaves, algo como um namespace, que é chamado internamente de *bucket*. Desta forma, podemos usar o mesmo cluster de servidores para mais de uma finalidade, sem precisar ficar misturando as chaves entre as aplicações.

No nosso serviço de votação, vamos utilizar o nome `votacao` para o bucket e, nos pares de *chave/valor*, usaremos o login do usuário como `chave` e o seu voto como `valor`.

A função que cria um objeto no Riak pelo console espera como argumento um objeto que pode ser criado usando a função `riak_object:new`. Esta recebe os três parâmetros que acabamos de definir: `bucket`, `chave` e `valor`.

```
(riak@127.0.0.1)4> Voto = riak_object:new(<<"votacao">>,
                                          <<"davidpaniz">>,
                                          <<"Fulano">>).
{r_object,<<"votacao">>,<<"davidpaniz">>,
          [{r_content,{dict,0,16,16,8,80,48,
              {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],...},
              {{[],[],[],[],[],[],[],[],[],[],[],[],[],...}}},
              <<"Fulano">>}],
          [],
          {dict,1,16,16,8,80,48,
            {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],...},
            {{[],[],[],[],[],[],[],[],[],[],[],[],[],...}}},
          undefined}
```

No Erlang, a sintaxe `<<valor>>` é usada para definir valores binários. Para nós, esses dados funcionarão como se fossem strings, mas a biblioteca nos obriga a usar dessa maneira.

No código anterior, criamos um objeto que representa o *bucket* `votacao`, *chave* `davidpaniz` e como **valor** o nome do candidato, `Fulano`. Por enquanto, criamos uma variável com um objeto que representa um objeto do Riak, mas ainda não inserimos nada no banco de dados!

Para registar o voto no servidor, precisaremos invocar a função `put` na nossa conexão que criamos anteriormente.

```
(riak@127.0.0.1)5> Conexao:put(Voto, 1).
ok
```

Essa operação seria comparável à seguinte operação em um banco relacional:

```
INSERT
  INTO votacao (chave, valor)
  VALUES ("davidpaniz", "Fulano");
```

Outra funcionalidade bem interessante do Riak é que ele disponibiliza uma interface HTTP para manipularmos os dados. Para exibir o objeto que acabamos de inserir usando o console, podemos fazer uma requisição para *http://localhost:8098/types/default/buckets/votacao/keys/davidpaniz*.

```
$ curl http://localhost:8098/\
types/default/buckets/votacao/keys/davidpaniz
Fulano
```

Esta interface HTTP aceita praticamente todas as operações suportadas pelo Riak. Por isso, qualquer linguagem de programação com comunicação HTTP pode utilizar o Riak por meio deste protocolo, caso não exista biblioteca nativa para isso.

Assim como podemos ler o valor de uma chave, também

podemos criar uma chave nova. Para isso, basta fazer a requisição usando o método (*method*) `POST` ou `PUT`, seguindo o mesmo esquema de URL: `/_types_/Tipo/_buckets_/NomeDoBucket/_keys_/Chave`.

```
$ curl -XPUT -d "Ciclano" \
        http://localhost:8098/\
types/default/buckets/votacao/keys/adriano

$ curl http://localhost:8098/\
types/default/buckets/votacao/keys/adriano
Ciclano
```

Voltando ao console, agora vamos tentar recuperar o valor a partir do nome do bucket e da chave.

```
(riak@127.0.0.1)6> {ok, Resultado} =
        Conexao:get(<<"votacao">>,
                    <<"davidpaniz">>,
                    1).
{ok,{r_object,<<"votacao">>,<<"davidpaniz">>,
              [{r_content,{dict,3,16,16,8,80,48,
                  {[],[],[],[],[],[],[],[],[],[],[],[],[],...},
                  {{[],[],[],[],[],[],[],[],[],[],...}}},
                  <<"Fulano">>}],
              [{<<35,9,254,249,2,117,42,222>>,{1,63592908814}}],
              {dict,1,16,16,8,80,48,
                  {[],[],[],[],[],[],[],[],[],[],[],[],[],...},
                  {{[],[],[],[],[],[],[],[],[],[],[],...}}},
              undefined}}

(riak@127.0.0.1)7> riak_object:get_value(Resultado).
<<"Fulano">>
```

Na primeira linha, usamos a função `get` da conexão que recebe dois valores binários, **bucket** e **valor**, e retorna o objeto armazenado. Caso ele seja encontrado, o valor será atribuído à variável `Resultado`. Para extrair o valor real deste objeto, usamos a função `riak_object:get_value` na segunda linha.

Isso é equivalente a:

```
SELECT valor
FROM votacao
```

```
WHERE chave = "davidpaniz"
```

Além disso, também podemos recuperar a chave que foi criada usando a interface HTTP.

```
(riak@127.0.0.1)8> {ok, OutroVoto} =
      Conexao:get(<<"votacao">>,
                  <<"adriano">>,
                  1).
{ok,{r_object,<<"votacao">>,<<"adriano">>,
      [{r_content,{dict,6,16,16,8,80,48,
        {[],[],[],[],[],[],[],[],[],[],[],[],[],...},
        {{[],[],[[<<"Links">>]],[],[],[],[],[],[],[],...}}},
        <<"Ciclano">>}],
      [{<<35,9,254,249,2,111,105,242>>,{1,63592909253}}],
        {dict,1,16,16,8,80,48,
        {[],[],[],[],[],[],[],[],[],[],[],[],[],...},
        {{[],[],[],[],[],[],[],[],[],[],[],...}}},
        undefined}}

(riak@127.0.0.1)9> riak_object:get_value(OutroVoto).
<<"Ciclano">>
```

Outra operação muito utilizada nos bancos *chave/valor* é listar todas as chaves cadastradas. No Riak, temos a função `list_keys` na conexão que recebe como argumento o nome do bucket e retorna uma lista com todas as chaves que existem nele.

```
(riak@127.0.0.1)10> Conexao:list_keys(<<"votacao">>).
{ok,[<<"davidpaniz">>,<<"adriano">>]}
```

Ou usando a interface HTTP:

```
$ curl http://localhost:8098/\
types/default/buckets/votacao/keys?keys=true
{"keys":["davidpaniz","adriano"]}
```

Ambos equivalentes ao SQL:

```
SELECT chave
FROM votacao
```

Em bancos de dados do tipo chave/valor, é comum não diferenciar operações de inserção e de atualização, e o que vale é o último valor inserido/atualizado. Portanto, das operações básicas, só

falta aprendermos a remover um registro.

Usando a interface HTTP, basta usar a mesma URI que usamos para recuperar o valor. Mas, em vez da usar o método `GET`, usamos o método `DELETE`. Já usando o console, utilizaremos a função `delete` da conexão passando como argumento o bucket e a chave.

```
(riak@127.0.0.1)11> Conexao:delete(<<"votacao">>,
                                   <<"davidpaniz">>).
ok

(riak@127.0.0.1)12> Conexao:list_keys(<<"votacao">>).
{ok,[<<"adriano">>]}
```

Ou usando a interface HTTP:

```
$ curl -XDELETE \
       http://localhost:8098/\
types/default/buckets/votacao/keys/adriano

$ curl http://localhost:8098/\
types/default/buckets/votacao/keys?keys=true
{"keys":[]}
```

Equivalentes ao SQL:

```
DELETE
FROM votacao
WHERE chave = "adriano"
```

E se, em vez de armazenarmos o candidato escolhido dentro de uma chave com o nome do usuário que votou, nós simplesmente criássemos uma chave para cada candidato com um número inteiro e, então, incrementássemos este número toda vez que alguém votar nele?

O resultado da primeira solução seria algo como:

```
|   chave   |    valor    |
| usuario 1 | candidato X |
| usuario 2 | candidato Y |
| usuario 3 | candidato Y |
| usuario 4 | candidato X |
| usuario 5 | candidato Y |
```

Já no resultado da segunda solução teremos algo como:

```
|    chave    | valor |
| candidato X |   2   |
| candidato Y |   3   |
```

4.5 O PROBLEMA DAS ESCRITAS CONCORRENTES

Vamos tentar implementar um contador com o que já conhecemos. Vamos criar uma chave nova com valor 0. Sempre que quisermos registrar um novo voto, nós recuperamos o valor atual, somamos 1 voto nele, e depois atualizamos com o valor novo.

Criando a chave para o candidato X com total de votos 0:

```
$ curl -XPOST \
       -d "0" \
       http://localhost:8098/\
types/default/buckets/votacao/keys/candidatox
```

Recuperando o número atual de votos do candidato:

```
$ curl http://localhost:8098/\
types/default/buckets/votacao/keys/candidatox
0
```

Agora com tudo junto, vamos criar uma função que recupera o valor atual, soma 1 e atualiza o total de votos com o valor novo:

```
$ function votar {
>   URL=http://localhost:8098/\
> types/default/buckets/votacao/keys/candidatox
>   TOTAL_VOTOS=`curl $URL`
>   NOVO_VALOR=$(($TOTAL_VOTOS+1))
>   curl -XPOST -d $NOVO_VALOR  $URL
> }

$ votar
  % Total    % Received % Xferd  ...
100     1  100     1    0     0  ...

$ curl http://localhost:8098/\
```

```
types/default/buckets/votacao/keys/candidatox
1

$ votar
  % Total    % Received % Xferd  ...
100     1  100     1    0     0  ...

$ curl http://localhost:8098/\
types/default/buckets/votacao/keys/candidatox
2
```

Agora que criamos a função `votar`, toda vez que ela é executada, atualizamos o valor com o valor antigo mais um. Parece funcionar muito bem. Mas o que vai acontecer em caso de leituras e escritas concorrentes? Ou seja, qual será o resultado se duas pessoas votarem ao mesmo tempo?

```
$ function dezvotos {
>  for i in `seq 1 10`;
>  do
>    votar
>  done
> }
```

Nesse código, criamos uma nova função `dezvotos`, que executa a função `votar` dez vezes sequencialmente. Agora, vamos criar um laço e executar a função `dezvotos` em 10 processos paralelos, usando `&` no final da linha. O objetivo é criar 10 processos que devem criar 10 votos cada, resultando em um total de 100 votos. Antes de começar o laço, vamos limpar os votos para facilitar a visualização do resultado.

```
$ curl -XPOST -d "0" \
       http://localhost:8098/\
types/default/buckets/votacao/keys/candidatox

$ curl http://localhost:8098/\
types/default/buckets/votacao/keys/candidatox
0

$ for i in `seq 1 10`;
> do
>  dezvotos &
```

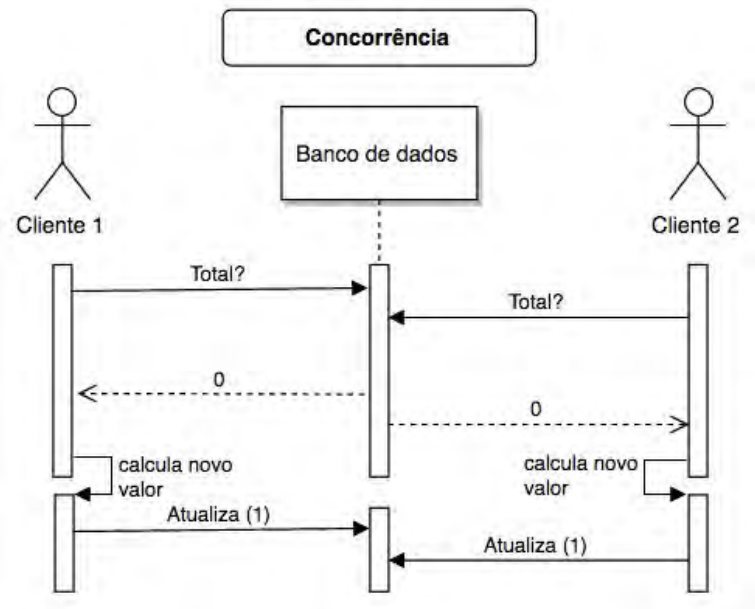
```
> done

# Após finalizar o laço os processos serão inicializados e você
# deve ver um monte de log das requisições sendo escrito nesta
# área. Até que os processos terminem e você verá algo parecido
# com o log abaixo

[1]   Done                    dezvotos
[2]   Done                    dezvotos
[3]   Done                    dezvotos
[4]   Done                    dezvotos
[5]   Done                    dezvotos
[6]   Done                    dezvotos
[7]   Done                    dezvotos
[8]   Done                    dezvotos
[9]-  Done                    dezvotos
[10]+  Done                    dezvotos
```

Agora que inserimos 100 votos, vejamos qual é o total de votos que temos armazenado.

```
$ curl http://localhost:8098/\
types/default/buckets/votacao/keys/candidatox
17
```

Este valor possivelmente será diferente de 17 quando você executar o código. Inclusive, é provável que obtenha resultados diferentes a cada execução. O que realmente importa é que, se você tem leituras e escritas paralelas, é praticamente impossível obter o resultado que gostaríamos usando esta abordagem.

O problema aqui é que, como a soma acontece do lado do cliente, é possível que o cálculo seja feito com um valor desatualizado. Para exemplificar, o usuário A carrega o total de votos, que por enquanto é 0. Logo depois, o usuário B faz a mesma consulta e encontra o mesmo valor 0. O usuário A efetua o cálculo e atualiza o total de votos com o novo valor 1. O usuário B acredita que o total de votos ainda é 0, e atualiza com o valor calculado por ele que também será 1.



Na verdade, este problema não está diretamente relacionado ao Riak ou aos bancos chave/valor. Todos os bancos de dados, inclusive os relacionais, também sofrem com este problema de concorrência. A forma mais comum de enfrentar este problema é utilizando *locks*.

Nos bancos de dados relacionais, normalmente *lock* está relacionado com literalmente travar (tradução de *lock*) o registro que será atualizado. E se algum outro processo tentar fazer o lock no mesmo registro, deverá esperar o primeiro processo liberar o registro. Esta técnica é conhecida como *lock pessimista*.

Nós partimos do pressuposto que fatalmente teremos as escritas concorrentes no mesmo registro, e nos preocupamos com isso sempre. A sintaxe de lock varia bastante de banco para banco. No MySQL, usamos as palavras chave `FOR UPDATE` no final de um `SELECT` para travar o registro.

```
BEGIN
SELECT value
FROM votos
WHERE chave='candidatoX'
FOR UPDATE
// As próximas linhas só serão executadas após
// travar o registro. Se algum outro processo
// já o travou, o nosso fica esperando.

// Após conseguirmos o lock do registro nenhum outro
// processo pode fazer o SELECT ... FOR UPDATE
// até terminarmos a transação.

UPDATE votos
SET value=1
WHERE chave='candidatoX'
COMMIT
```

A outra abordagem para os locks, mesmo em bancos relacionais, é chamada de *lock otimista*. Diferente do pessimista, no lock otimista nós acreditamos que as escritas concorrentes são muito raras, e não adicionamos o lock no banco para evitar que as operações de escritas fiquem mais lentas. Em vez disso, normalmente o lock otimista funciona adicionando um campo extra no registro com a versão dele, que pode ser um número sequencial ou uma data, e quando vamos fazer a atualização, verificamos se a versão do registro ainda é a mesma.

Em um exemplo simples de lock otimista, imaginando que ao carregar os dados do `candidatoX` encontramos o *valor* `0` e a *versão* `1`, podemos usar o seguinte SQL para fazer o update:

```
UPDATE votos
SET value=1, versao=2
WHERE chave='candidatoX'
AND versao=1
```

Normalmente, os bancos relacionais retornam o número de registros que foram atualizados quando executamos um *update*. Usando a versão como parte da cláusula `where`, caso o registro tenha sido previamente atualizado, a versão do registro estará

diferente e, por isso, a atualização não vai funcionar. Se isso realmente acontecer, o número de registros atualizados será 0, e podemos tratar na aplicação o que deve ser feito.

Ainda no mundo do SQL, para implementar um contador, temos a vantagem de ter uma referência para o valor que está no banco de dados no instante da atualização. Podemos executar a soma dentro do banco, como neste exemplo.

```
UPDATE votos
SET value=value + 1
WHERE chave='candidatoX'
```

Para efetuar esta atualização, não precisamos saber qual é o valor que será atualizado, nem nos preocupar com *lock*, pois o próprio banco de dados vai efetuar a soma usando o valor mais recente. Infelizmente, os bancos do tipo chave/valor não suportam este tipo de atualização na qual nós podemos aproveitar o valor atual do registro. Por isso, existem os tipos especiais, que alguns bancos suportam.

4.6 OUTROS TIPOS DE DADOS DO RIAK

Normalmente, os valores armazenados em bancos do tipo chave/valor são dados pré-processados, e funcionam como grandes strings ou objetos binários. Mas alguns bancos suportam outros tipos de dados especiais, que auxiliam em problemas um pouco mais específicos do que simplesmente indexar dados por uma chave, como nos problemas de *cache* ou *session storage*.

O *Redis*, por exemplo, suporta estruturas de dados como listas (*List*), conjuntos (*Set*) e operações (como *push* e *pop*) que permitem a fácil implementação de filas e pilhas. O Riak, por outro lado, não possui suporte a listas, mas suporta outros tipos de dados interessantes, como:

- *Flag*, para armazenar valores booleanos / binários;
- *Set*, para armazenar os já citado conjuntos;
- *Map*, para armazenar mapas, possibilitando atualizar apenas parte do valor, parecido com um banco orientado a documento;
- *Counter*, que armazena números inteiros, mas com a vantagem de utilizar operação de incremento, que resolve o problema dos votos. Este tipo de dado é muito útil para armazenar números de visualização, números de compartilhamentos, total de cliques ou qualquer outra informação que seja um contador inteiro.

Embora o Riak suporte esses tipos especiais, nenhum deles vem habilitado por padrão. Por enquanto, tínhamos trabalhado com os *buckets* que encapsulam as chaves. Agora vamos precisar criar um *bucket type*, que habilita algum dos tipos especiais. Somente os buckets dentro deste *bucket type* são que usarão o tipo configurado nele.

Para manipular os *bucket types*, utilizaremos a API administrativa do Riak. No terminal, vamos usar o comando `riak-admin bucket-type` sempre que queremos gerenciar os bucket types. Usando a operação `list`, nós listamos todos os bucket types existentes. Já com a operação `status`, conseguimos ver as configurações dele.

```
$ riak-admin bucket-type list
default (active)

$ riak-admin bucket-type status default
default is active

allow_mult: false
basic_quorum: false
big_vclock: 50
chash_keyfun: {riak_core_util,chash_std_keyfun}
dvv_enabled: false
dw: quorum
```

```
last_write_wins: false
linkfun: {modfun,riak_kv_wm_link_walker,mapreduce_linkfun}
n_val: 3
notfound_ok: true
old_vclock: 86400
postcommit: []
pr: 0
precommit: []
pw: 0
r: quorum
rw: quorum
small_vclock: 50
w: quorum
young_vclock: 20
```

Para utilizar os contadores, vamos precisar criar um bucket type com a propriedade `datatype` configurada para `counter`. Para isso, usamos a operação `create`, passando o nome do bucket type e as suas propriedades.

```
$ riak-admin bucket-type \
create contador '{"props":{"datatype":"counter"}}'
contador created

$ riak-admin bucket-type list
default (active)
contador (not active)
```

Como pode ser visto no resultado da listagem, assim que um bucket type é criado, ele está desativado. Para ativá-lo, vamos usar a operação `activate`.

```
$ riak-admin bucket-type activate contador
contador has been activated

$ riak-admin bucket-type list
default (active)
contador (active)

$ riak-admin bucket-type status contador
contador is active

young_vclock: 20
w: quorum
small_vclock: 50
rw: quorum
```

```
r: quorum
pw: 0
precommit: []
pr: 0
postcommit: []
old_vclock: 86400
notfound_ok: true
n_val: 3
linkfun: {modfun,riak_kv_wm_link_walker,mapreduce_linkfun}
last_write_wins: false
dw: quorum
dvv_enabled: true
chash_keyfun: {riak_core_util,chash_std_keyfun}
big_vclock: 50
basic_quorum: false
allow_mult: true
datatype: counter
active: true
claimant: 'riak@127.0.0.1'
```

Agora que já temos um *bucket type* com o *datatype* `counter` ativado, já podemos começar a usar o contador. Usando o tipo `counter`, ganhamos a operação de incremento dentro do banco, algo semelhante ao `valor = valor + 1`, que vimos no SQL. Diferente do tipo padrão do Riak, no qual passamos o novo valor que substituirá o antigo da chave, nós usamos o incremento dizendo quanto queremos adicionar ao valor atual.

Usando a interface HTTP, passamos como argumento `{"increment" : 1}` para somar 1 ao valor atual do contador. Assim como no tipo padrão, não é preciso nenhuma operação especial para criar o bucket ou a chave. A primeira escrita se encarregará de criá-los caso ainda não existam. Para o caso do contador, se ele ainda não existir, será inicializado com o valor 0, que será incrementado com o valor passado.

Além do parâmetro diferente, um outra diferença na interface HTTP quando usamos os tipos especiais é o caminho na URL. No padrão, estávamos usando `types/DATA_TYPE/buckets/BUCKET/keys/CHAVE`. E nos tipos

especiais usaremos `types/DATA_TYPE/buckets/BUCKET/datatypes/CHAVE`. Por enquanto, vamos utilizar um bucket de teste para entender melhor como funciona o contador.

```
$ curl -XPOST http://localhost:8098/\
types/contador/buckets/teste/datatypes/foo \
       -H "Content-Type: application/json" \
       -d '{"increment" : 1}'

$ curl http://localhost:8098/\
types/contador/buckets/teste/datatypes/foo
{"type":"counter","value":1}

$ curl -XPOST http://localhost:8098/\
types/contador/buckets/teste/datatypes/foo \
       -H "Content-Type: application/json" \
       -d '{"increment" : 5}'

$ curl http://localhost:8098/\
types/contador/buckets/teste/datatypes/foo
{"type":"counter","value":6}
```

CURL USANDO DOCKER

Se você decidiu usar o docker container em vez de instalar o Riak localmente, não se esqueça que, dependendo do seu sistema operação, os contêineres rodam dentro de uma máquina virtual, e não dentro do seu SO. Consequentemente, o Riak não está respondendo em `localhost`, mas sim no IP da máquina virtual. Normalmente, você pode descobrir o IP desta máquina usando o comando `docker-machine ip`.

Agora que estamos com o `_Counter_` configurado, podemos fazer o teste dos votos simultâneos novamente. Vamos criar novas funções `votarcontador` e `dezvotoscontador` executando as chamadas HTTP para um novo *bucket* `votacao`, dentro do *bucket*

type `contador` , armazenando os votos na *key* `candidatoX` .

```
$ function votarcontador {
>   curl -XPOST http://localhost:8098/\
> types/contador/buckets/votos/datatypes/candidatoX \
>   -H "Content-Type: application/json" \
>   -d '{"increment": 1}'
> }

$ function dezvotoscontador {
>   for i in `seq 1 10`;
>   do
>     votarcontador
>   done
> }
```

Agora que temos tudo preparado, vamos executar a função `dezvotoscontador` em 10 processos paralelos, e depois verificar o resultado.

```
$ for i in `seq 1 10`;
> do
>  dezvotoscontador &
> done

[1] 31690
[2] 31691
[3] 31692
[4] 31693
[5] 31695
[6] 31697
[7] 31700
[8] 31703
[9] 31706
[10] 31711

# As mensagens acima aparecem assim que o laço começa, agora
# é preciso aguardar que a execução termine. Somente após a
# mensagem abaixo é que devemos verificar o valor total dos
# votos

[1]   Done                    dezvotoscontador
[2]   Done                    dezvotoscontador
[3]   Done                    dezvotoscontador
[4]   Done                    dezvotoscontador
[5]   Done                    dezvotoscontador
[6]   Done                    dezvotoscontador
```

```
[7]   Done                    dezvotoscontador
[8]   Done                    dezvotoscontador
[9]-  Done                    dezvotoscontador
[10]+  Done                    dezvotoscontador

$ curl http://localhost:8098/\
types/contador/buckets/votos/datatypes/candidatoX
{"type":"counter","value":100}
```

Desta vez, usando o tipo `Counter`, chegamos ao resultado esperado de 100 votos. Essa solução é muito boa para a contagem. Na nossa primeira ideia, estávamos armazenando também o usuário que efetuou o voto, e não só o total de votos de cada candidato. O que pode acontecer se unirmos as duas ideias, e fizermos sempre duas atualização no banco a cada voto?

Capítulo 5

MAIS SOBRE BANCOS CHAVE VALOR E PERSISTÊNCIA POLIGLOTA

No final do capítulo anterior, logo após resolvermos o problema do contador, começamos a questionar a possibilidade de juntar as duas abordagens trabalhadas para o problema da votação. Na primeira solução, estávamos armazenando o candidato votado dentro do usuário. Já na segunda, tínhamos apenas um contador de votos para cada candidato.

Apenas com a primeira é muito custoso calcular o total de votos de cada candidato, porque precisamos percorrer todas as chaves do bucket e separar os votos por candidato. Na segunda, embora o total de votos esteja sempre atualizado, nós perdemos a rastreabilidade dos votos e não poderíamos auditar a eleição, se necessário.

Provavelmente, em uma votação de eliminação de reality show, não precisaríamos de uma auditoria. Entretanto, saber em qual candidato um usuário votou nos ajuda muito para segmentação de marketing, então acaba sendo uma funcionalidade relevante para ser implementada. A ideia então é, quando o usuário estiver logado no nosso site e efetuar um voto, nós fazemos duas operações, salvamos o candidato que ele escolheu na chave dele, e também incrementamos 1 voto no total do candidato escolhido.

5.1 CONSISTÊNCIA X CONSISTÊNCIA EVENTUAL

Seguindo os exemplos anteriores, podemos implementar o novo processo de votação com a junção dos dois exemplos. Se o usuário `david` votar no `fulano`, faremos as seguintes operações:

```
$ curl -XPOST http://localhost:8098/\
types/contador/buckets/votos/datatypes/fulano
$ curl -XPUT -d "fulano" \
       http://localhost:8098/\
types/default/buckets/votacao/keys/david
```

Até agora, todas as operações que realizamos alteravam apenas uma chave, mas a partir de agora vamos precisar alterar duas chaves para efetivar um voto. O que vai acontecer se, por acaso, uma delas falhar? Deveríamos salvar a preferência do usuário sem computar o voto? Deveríamos incrementar o total de votos do candidato sem persistir qual foi a escolha do usuário?

Quando se fala em bancos de dados, uma das propriedades que devemos considerar é a atomicidade, uma das propriedades do acrônimo **ACID** (HAERDER; REUTER, 1983).

ACID define um conjunto de princípios para bancos de dados orientados a transações, que é formado por:

- **Atomicidade** (*Atomicity*) — Todo o conjunto de modificações deve ser tratado como uma unidade. Deve ser materializado, ou tudo, ou nada.
- **Consistência** (*Consistency*) — Os dados devem se manter íntegros. Apenas dados válidos devem ser inseridos em uma transação.
- **Isolamento** (*Isolation*) — As modificações temporárias de uma transação não devem refletir dentro de outras

transações.

- **Durabilidade** (*Durability*) — Após o final da transação, o banco de dados deve garantir que as alterações sobrevivam a falhas.

Consistência ACID x Consistência CAP

Infelizmente, esses dois artigos importantíssimos para a história dos bancos de dados utilizam a mesma palavra para definir dois conceitos diferentes.

No **ACID**, a consistência diz respeito à integridade, tanto integridade referencial quanto a não violação de restrições (`constraint`) no banco de dados, ou seja, ausência de dados inválidos. Por exemplo, em uma coluna de tipo numérico com uma restrição de dado obrigatório (`constraint NOT NULL`), não vai existir nenhum registro sem valor ou com valor que não seja um número; ou, em uma coluna que seja uma chave estrangeira, nunca vai existir um valor que não seja o identificador de um registro existente.

Já no **CAP**, a consistência é definida pelo modelo de consistência atômico, também conhecido como linearizabilidade (HERLIHY; WING, 1989) — *atomic consistency* ou *linearizability*, em inglês. Neste modelo de consistência, todas as operações são enfileiradas e sempre executadas na ordem como em uma fila FIFO (*first in, first out*), e sempre se baseando no estado atual considerando as operações anteriores.

Neste modelo, cada operação possui um intervalo de execução, e suas alterações estão disponíveis globalmente até o término deste intervalo. Em outras palavras, ao término da execução de uma operação, o seu resultado deve estar disponível globalmente para qualquer outra operação.

Os bancos de dados relacionais normalmente têm transações e

seguem o modelo ACID. Por isso, para resolver o problema das duas operações para registrar um voto em um banco relacional, bastaria criar uma transação que englobasse as duas operações como no exemplo a seguir.

```
BEGIN

  INSERT
    INTO votacao (chave, valor)
    VALUES ("davidpaniz", "fulano");


  UPDATE votos
    SET value=value + 1
    WHERE chave='fulano';

COMMIT;
```

No Riak não temos transações e, por isso, vamos precisar tratar o problema da votação na nossa aplicação fora da camada de banco de dados. Boa parte dos dos bancos NoSQL não se enquadra no modelo ACID; ao contrário disso, segue o modelo BASE.

- **Basicamente disponível** (*Basically Available*) — Até em casos de falha, o banco de dados deve continuar respondendo mesmo que retorne dados inconsistentes ou desatualizados.
- **Estado leve** (*Soft-state*) — Como se o estado do banco fosse volátil. Devido à consistência eventual, é possível que duas consultas iguais retornem dados diferentes, mesmo sem nenhuma atualização entre elas.
- **Consistência eventual** (*Eventual consistency*) — A consistência não é garantida, mas toda atualização é propagada. Mesmo após um problema de rede, quando a comunicação normalizar, as atualizações chegarão até ele se tornar consistente novamente.

ACID E BASE

Apenas uma curiosidade que talvez tenha passada despercebida: embora ACID e BASE sejam acrônimos, as palavras em inglês significam ácido e base, que na química são substâncias com propriedades opostas.

Embora existam alguns bancos NoSQL que seguem o modelo ACID, normalmente eles acabam abrindo mão da consistência em prol da disponibilidade, resultando em consistência eventual e estado leve, e consequentemente seguem o modelo BASE.

Em alguns tipos de literatura, é comum encontrarmos exemplos de contas bancárias para demonstrar transações e a importância da consistência, mas na vida real não é bem assim. O exemplo clássico é quando fazemos uma transferência entre contas. Se o dinheiro sair da conta "A", ele tem de ir para a conta "B". E se não conseguirmos somar o valor na conta "B", não podemos subtraí-lo da conta "A". Se o sistema bancário fosse realmente consistente, não existiriam estornos.

Na vida real, quando fazemos uma transferência entre bancos usando DOC, o dinheiro literalmente sai da conta "A" antes de ir para a conta "B". Se a transferência falhar, devolvemos o dinheiro à conta "A" com um estorno. Entretanto, entre a transferência e o estorno, o dinheiro não estava nem na conta "A" nem na conta "B".

É comum acharmos que milissegundos de latência é demais, mas existem muitos casos na vida real nos quais temos até minutos com dados desatualizados sem problemas. Da próxima vez que fizer uma compra online, tente entrar na página de meus pedidos instantaneamente após finalizar o pedido. Provavelmente, ele ainda

não estará lá. Além disso, de vez em quando, ao finalizar um pedido, mesmo que com sucesso, o pedido não será concretizado de verdade, porque alguém comprou a última unidade praticamente ao mesmo tempo.

Voltando ao problema da votação, nós chegamos à conclusão de que o que realmente importa é somar o voto na chave do candidato, e que seria bom armazenarmos o voto do usuário na chave dele. Assim como em alguns casos, é aceitável ter grandes períodos de inconsistência, em alguns casos também é aceitável simplesmente perder dados.

Nós escolhemos o Riak justamente pela sua característica de alta disponibilidade, porque não queremos correr o risco de perder um voto. Mas se por algum motivo registrarmos o voto e não conseguirmos salvar a opção do usuário, tudo bem.

Então, a solução para o nosso problema será a mais simples. Vamos executar as duas operações, uma em sequência da outra, dando preferência para a soma do voto que é a nossa operação principal, exatamente como fizemos no começo do capítulo.

```
$ curl -XPOST http://localhost:8098/\
types/contador/buckets/votos/datatypes/fulano
$ curl -XPUT -d "fulano" http://localhost:8098/\
types/default/buckets/votacao/keys/david
```

5.2 MONGODB, RIAK OU OUTRO BANCO?

Nos primeiros capítulos, estávamos enfrentando o problema do cadastro de álbuns que precisam de um modelo de persistência maleável, *schemaless*. Por conta disso, optamos por um banco de dados do tipo `Documento`. Nos últimos capítulos, para poder garantir a disponibilidade do sistema de votação, optamos por um banco de dados que tivesse alta disponibilidade de escrita.

Nestes capítulos, exploramos algumas vantagens e desvantagens dos modelos de dado orientado a documentos e chave valor, além de peculiaridades nas implementações do MongoDB e do Riak. Mas após terminar de ler o livro, quando estiver começando o um novo projeto, quantos e quais bancos você pode / deve utilizar?

Um pouco após a origem do termo NoSQL, muito se discutiu sobre como e quando usar cada tipo de modelo e a melhor implementação para as necessidades de disponibilidade e consistência de cada projeto. O que muitos notaram é que, mesmo em aplicações não tão grandes, é comum existirem necessidades diferentes, e um banco de dados apenas pode não ser perfeito para todas as situações. Mesmo pensando apenas em bancos relacionais, é comum existirem diferentes bancos de dados para operações `OLAP` e `OLTP`.

- **OLTP** (*On-line Transaction Processing*): são as operações comuns feitas pelo núcleo de negócio do seu sistema, ou seja, efetuar um agendamento, confirmar uma compra, criar um usuário etc. São as operações que criam, leem e alteram os dados gerados pelos usuários ou para os usuários. Normalmente, tem um alto volume de pequenas transações.

- **OLAP** (*On-line Analitical Processing*): como o nome sugere, são as operações para análise. Em algumas empresas, pode ser apenas uma planilha com informações de vários lugares usada para análise estratégica e tomada de decisão. Mas quando precisamos de uma análise mais organizada, criamos um outro banco de dados. Normalmente, tem um volume baixo de operações, porém bem mais pesadas e lentas, usualmente com muitos relacionamentos e agregações. É comum termos bancos de dados

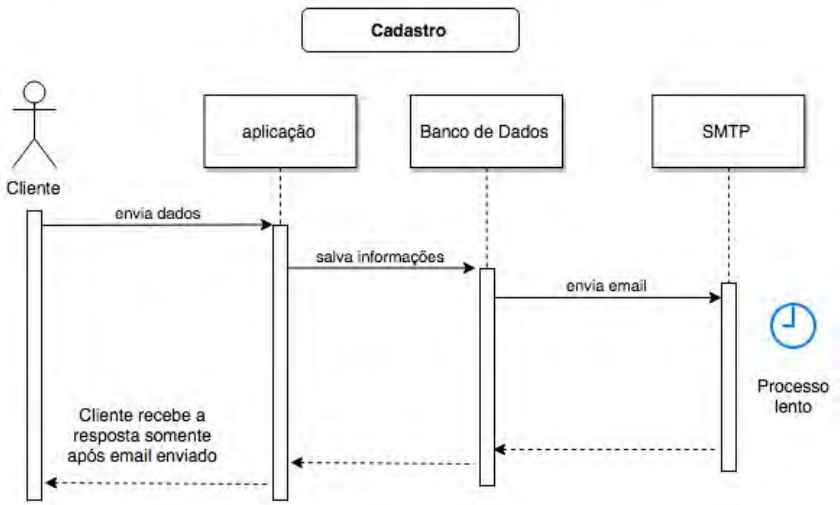
exclusivos para análise, para evitar problemas de performance na aplicação, especialmente durante a geração de relatórios que consolidam dados de grandes períodos como meses, semestres ou mesmo anos. Além disso, é comum os bancos de OLAP terem esquema (schema) diferente, já como várias desnormalizações para ajudar no processamento das informações. Em algumas arquiteturas, existe um *Data Warehouse*, um repositório de dados que centraliza informações de vários lugares para facilitar essas operações analíticas.

Além das necessidades diferenciadas para operações de `OLAP`, no mundo atual é comum aplicações relativamente pequenas terem um alto volume de visualizações ou um alto volume de escritas que podem prejudicar a performance do seu banco de dados, independente do tipo dele. Para evitar grandes quantidades de acessos ao banco principal de `OLTP`, é comum utilizar outros bancos como cache de leitura ou como buffer de escrita.

Quando um usuário faz uma requisição usando um browser como Chrome, Firefox ou qualquer outro, o browser fica esperando uma resposta do servidor com uma espécie de canal aberto entre os dois, cliente e servidor, que só será fechado quando o servidor enviar uma resposta, ou o cliente decidir fechar a conexão, por exemplo, porque o servidor demorou demais para responder (*time out*).

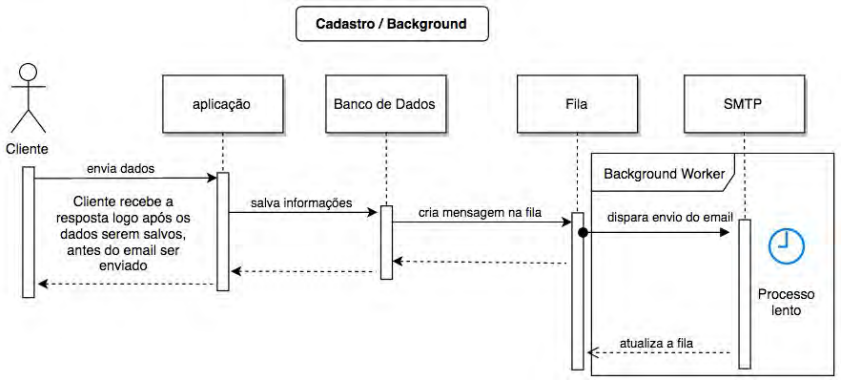
Independente da tecnologia usada, todo servidor tem um número máximo de conexões que ele consegue manter aberto, então quanto mais rápido for o tempo médio de reposta, maior o número de requisições que o servidor consegue atender por minuto. Por isso um dos segredos para escalar uma aplicação web é manter o tempo de resposta do seu servidor o mais baixo possível.

Para exemplificar com um problema do mundo real, digamos que quando um usuário se cadastra na sua aplicação, precisamos salvar os dados dele no nosso banco de dados e enviar um e-mail de boas-vindas. A escrita no banco de dados é uma operação relativamente lenta, mas normalmente na casa dos poucos milissegundos, o que é aceitável, porém o envio do e-mail costuma beirar a marca de 1 segundo ou mais, dependendo da forma utilizada. Se seu servidor suporta 10 conexões abertas, e cada requisição demora aproximadamente 1 segundo para ser respondida, sua aplicação vai conseguir responder aproximadamente 10 requisições por segundo.

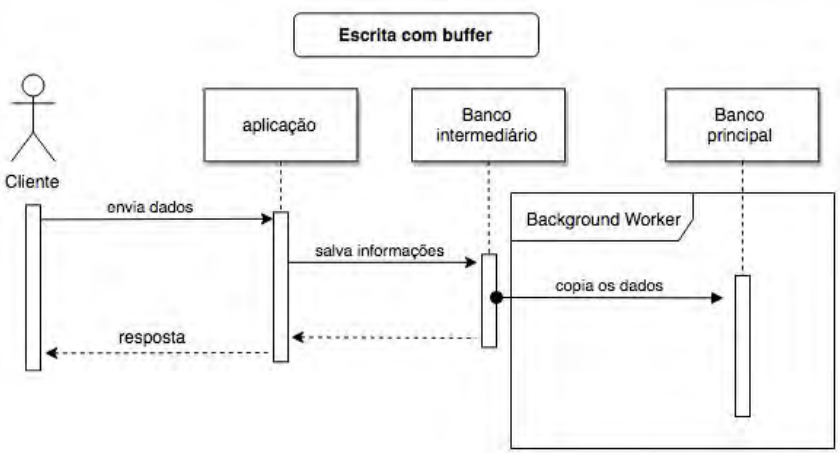


Se em vez de enviar o e-mail sincronicamente, que é uma operação lenta, nós salvarmos essa informação em um sistema de fila, que tende a ser mais rápido, podemos responder as requisições mais rapidamente para o cliente e ainda liberar a conexão do servidor. A partir do momento em que os dados estão salvos no sistema de fila, podemos ter outro servidor, de preferência que não esteja atendendo requisições de clientes, lendo as informações dessa fila e enviando os e-mails.

Esta arquitetura é extremamente comum. Por isso quando usamos funcionalidades como "Esqueci minha senha" ou finalizamos um pedido em uma loja virtual, os e-mails sempre demoram alguns segundos ou minutos para chegar, embora o site diga que sua solicitação foi confirmada quase instantaneamente.



Embora existam sistemas de mensageria robustos, é fácil encontrar aplicações utilizando bancos do tipo chave/valor para implementar filas deste tipo de trabalho. Além disso, quando precisamos de grandes volumes de escritas, também podemos usar uma banco de dados auxiliar com uma performance de escrita melhor do que o do seu banco de dados principal; e de tempos em tempos ir migrando os dados de um banco para o outro, algo parecido com um buffer em memória.



Nos últimos capítulos, tomamos decisões arquiteturais que aumentavam a latência, mas que aumentam a disponibilidade. Quando adicionamos a assincronicidade, também aumentamos a latência, mas com isso melhoramos a sensação de performance que o usuário final tem e impactamos diretamente na escalabilidade da aplicação.

Como vocês podem notar, existem diversos motivos para usar um banco de dados NoSQL. Nos primeiros capítulos, usamos um banco que atende melhor o modelo de negócio, depois escolhemos um banco por motivos de disponibilidade, e agora acabamos de ver a possibilidade de usar um outro banco para melhorar a performance, a escalabilidade, ou simplesmente para melhorar as operações analíticas.

Assim como existem alguns tipos de bancos de dados, existem vários motivos para querer utilizar algum deles. E quando temos motivações diferentes, podemos sim ter bancos diferentes também! Nada nos impede de usar na mesma aplicação um banco relacional como banco principal, um banco de documento para um tipo de cache desnormalizado com os dados que queremos exibir em uma

página, usar um banco chave/valor como um buffer de escrita, um banco colunar para OLAP e um banco orientado a grafo para um sistema antifraude ou de recomendações.

Uma coisa que precisamos sempre manter em mente é que, embora cada banco tenha suas vantagens e agregue valor à aplicação, cada banco novo adiciona complexidade e são novos pontos de possível falha. Se sua aplicação possui três bancos de dados, o que vai acontecer com ela quando o servidor de um deles tiver problemas?

Quando pensamos em uma arquitetura de microsserviços, a escolha de vários tipos de bancos de dados pode fazer mais sentido. Nessa abordagem, uma aplicação é formada por várias "aplicações" pequenas, chamadas de serviços. Esses serviços não devem compartilhar o mesmo banco de dados, eles devem comunicar entre si, por trocas de mensagens — que podem ser síncronas como uma requisição HTTP ou assíncronas como uma publicação em um tópico que outros serviços escutam.

Já que, nessa arquitetura, idealmente cada serviço deve possuir o próprio banco, e cada serviço tem regras de negócio específicas, logo, é uma boa ideia cada um usar o melhor tipo de banco de dados.

Independentemente da arquitetura adotada, chamamos a adoção de mais um tipo de banco de dados na mesma aplicação de *persistência poliglota*.

5.3 ARMAZENANDO PLAYLISTS

Agora que já discutimos alguns motivos para usar mais de um banco de dados e entendemos o que é persistência poliglota, podemos voltar aos problemas da nossa aplicação. O próximo passo

será o gerenciamento de playlists, também chamadas de listas de reprodução.

A funcionalidade de playlist que vamos precisar implementar tem um conceito básico bem simples: uma playlist é uma lista de músicas ordenadas. Embora a estrutura das listas seja bem fácil, temos um requisito que complica bastante as coisas: as listas podem ser editadas por mais de uma pessoa ao mesmo tempo, e podem ser editadas por cliente offline.

O grande problema dessa funcionalidade é como tratar as mudanças concorrentes e as mudanças efetuadas por clientes offline que podem estar editando versões já desatualizadas das listas, mas vamos detalhar melhor esse problema em breve. Na verdade, a funcionalidade que vamos precisar criar é uma ferramenta de controle de versão, algo semelhante a um controle de versão de código como o *git* ou *mercurial*. Porém, em vez de gerenciar arquivos de códigos, vamos controlar uma lista de músicas.

Um modelo básico de playlist seria algo como:

O grande problema dessa implementação é para os casos de modificação assíncrona. Vamos criar uma playlist inicial e exibir as modificações concorrentes. Para facilitar o entendimento do

exemplo, usaremos o nome da música no lugar do seu `id`.

```
Playlists EXEMPLO1
| idPlaylist |   idMusica  | posicao |
|     1      |     Help!   |    1    |
|     1      |  Yesterday  |    2    |
|     1      |  Blackbird  |    3    |
```

Existem dois clientes que estão com essa versão da lista, e ambos estão offline. Um deles muda a ordem das músicas 2 e 3, e fica com a seguinte versão em seu dispositivo:

```
Playlists EXEMPLO2
| idPlaylist |   idMusica  | posicao |
|     1      |     Help!   |    1    |
|     1      |  Blackbird  |    2    |
|     1      |  Yesterday  |    3    |
```

O outro cliente tenta adicionar uma música entre as músicas 2 e 3, e seu dispositivo fica com a seguinte versão da playlist:

```
Playlists EXEMPLO3
| idPlaylist |   idMusica  | posicao |
|     1      |     Help!   |    1    |
|     1      |  Yesterday  |    2    |
|     1      |  Blackbird  |    3    |
|     1      |  Something  |    4    |
```

Se o servidor armazenar apenas a versão "atual" da playlist, quando o cliente sincronizar a playlist `EXEMPLO3`, vamos adicionar a nova música na lista. Mas quando o outro cliente tentar atualizar as suas modificações, resultando na playlist `EXEMPLO2`, provavelmente vamos interpretar que, além de modificar a ordem das músicas, ele também removeu a música que foi adicionada pelo outro cliente.

Para a funcionalidade de controle de versão, precisaremos de um modelo parecido com o seguinte:

Usando o mesmo exemplo anterior, começamos com uma lista inicial com as mesmas três músicas iniciais, porém usando o outro modelo.

```
PlaylistVersionada EXEMPLO1
| idPlaylist |    acao  | posicao | tamanho | para |  idMusica |
|     1      | adiciona |    1    |         |      |    Help!  |
|     1      | adiciona |    2    |         |      | Yesterday |
|     1      | adiciona |    3    |         |      | Blackbird |

### RESULTADO  ####
Help!
Yesterday
Blackbird
```

Como no exemplo anterior, um dos clientes altera a ordem de duas músicas.

```
PlaylistVersionada EXEMPLO2
|idPlaylist|versao|   acao |posicao|tamanho|para| idMusica|
|    1     |   1  |adiciona|   1   |       |    |   Help! |
|    1     |   2  |adiciona|   2   |       |    |Yesterday|
|    1     |   3  |adiciona|   3   |       |    |Blackbird|
|    1     |   4  |  troca |   3   |   1   |  2 |         |

### RESULTADO LOCAL ####
Help!
Blackbird
Yesterday
```

E o outro adiciona uma nova música:

```
PlaylistVersionada EXEMPLO3
|idPlaylist|versao|  acao  |posicao|tamanho|para| idMusica|
|    1     |   1  |adiciona|   1   |       |     |   Help! |
|    1     |   2  |adiciona|   2   |       |     |Yesterday|
|    1     |   3  |adiciona|   3   |       |     |Blackbird|
|    1     |   4  |adiciona|   4   |       |     |Something|

### RESULTADO  LOCAL####
Help!
Yesterday
Blackbird
Something
```

Agora que temos a tabela como uma lista de ações em cima da playlist, ao sincronizar as modificações dos dois clientes, teremos uma playlist com as quatro músicas, mas com a ordem das músicas também modificada.

```
PlaylistVersionada EXEMPLO4 - Resultado FINAL
|idPlaylist|versao|  acao  |posicao|tamanho|para| idMusica|
|    1     |   1  |adiciona|   1   |       |     |   Help! |
|    1     |   2  |adiciona|   2   |       |     |Yesterday|
|    1     |   3  |adiciona|   3   |       |     |Blackbird|
|    1     |   4  |adiciona|   4   |       |     |Something|
|    1     |   5  |  troca |   3   |   1   |  2  |         |

### RESULTADO  LOCAL####
Help!
Blackbird
Yesterday
Something
```

Agora que nós já sabemos o que vamos armazenar, precisamos pensar no melhor tipo de banco de dados para armazenar os dados. Também devemos pensar sobre os requisitos não funcionais, como necessidade de escalabilidade, disponibilidade etc. para escolhermos, além da melhor estrutura, também a implementação que mais se adequa às nossas necessidades.

Um dos grandes problemas dessa abordagem de controle de versão para gerenciar as playlists é que, em vez de armazenar um

registro por música, nós precisamos armazenar um registro por mudança da lista. Então, é possível que uma lista pequena com apenas cinco músicas possa ter dezenas, ou mesmo centenas, de registros para representar o seu histórico.

Além disso, uma única playlist pode ter vários usuários editando, e vários usuários seguindo e escutando a lista. Por causa desses detalhes, novamente vamos precisar nos preocupar com disponibilidade, escalabilidade e performance.

Embora o **Riak** seja bem performático e sua arquitetura nos garanta alta disponibilidade, a estrutura de dados que precisamos armazenar é complexa demais para usar um banco do tipo chave/valor. O **MongoDB** e seu banco do tipo Documento até conseguiria armazenar a estrutura de dados que temos com poucas mudanças. Entretanto, para suportar grandes volumes de dados, precisaríamos de técnicas similares às usadas por bancos relacionais como `shardes` , o que adiciona uma certa complexidade na arquitetura. Mas para garantir uma boa performance de escrita, ele fica sensível na tolerância à falha, podendo até perder dados caso um servidor passe por um problema.

Existe um banco de dados que é famoso por sua *performance*, principalmente de escrita, *durabilidade* (uma vez que foi confirmada a escrita de um dado, esta informação não vai ser perdida, mesmo em caso de falhas), *disponibilidade* e *tolerância à falha*. Esse banco é o **Cassandra**, que, diferente dos bancos vistos anteriormente, pode ser um banco *orientado a colunas*, ou *Colunar*.

Capítulo 6

IMPLEMENTANDO PLAYLISTS EM UM BANCO COLUNAR

O nosso objetivo agora é implementar a funcionalidade das playlists. Por conta das necessidades de disponibilidade, performance, tolerância a falhas e escalabilidade, vimos que o **Cassandra** parece ser uma boa opção para resolver nossos problemas. Entretanto, ele é um banco colunar e precisamos ter certeza de que esta opção será adequada para o nosso contexto.

Como vimos nos primeiros capítulos, a estrutura principal dos bancos colunares é basicamente uma grande tabela. Mas, assim como nos bancos orientados a documentos, cada registro pode ter quantas e quais colunas precisar (*schemaless*). A grosso modo, os bancos colunares são os que mais se assemelham aos bancos relacionais por terem uma "tabela", mesmo que, na verdade eles, sejam muito diferentes.

Os bancos colunares surgiram com um trabalho do Google publicado sob nome *Bigtable: A Distributed Storage System for Structured Data* (GHEMAWAT; HSIEH; WALLACH; BURROWS et al., 2006), em uma tradução livre algo como: "Um sistema de armazenamento distribuído para dados estruturados".

A essência por trás dessa família de bancos de dados é trabalhar

como sistemas distribuídos, ou seja, vários servidores coordenados entre si, principalmente para suportar grandes volumes de dados com alta disponibilidade. Esses dois requisitos são o principal motivo pelo qual esse banco surgiu, e normalmente são as justificativas para se escolher um deles.

Aqui no exemplo do livro, vamos usá-lo para entendermos um pouco da estrutura e de sua utilização. Mas boa parte das empresas que estão usando bancos colunares possui muitos gigabytes, terabytes e, em alguns casos, até mesmo petabytes.

Para se ter uma ideia, o **Bigtable** foi criado e ainda é usado pelo Google em produtos como o **Gmail**, **Google Analytics** e por parte do motor de buscas deles. Baseado no *paper* do Google, a equipe de engenharia do Facebook, que estava passando por problemas similares, lançou em 2008 o **Cassandra** que, além do **Facebook**, empresas como **Ebay** e **Netflix** utilizam.

Para se ter uma pequena noção do tamanho de dados que estamos falando, na página inicial do **HBase**, há uma implementação de código aberto fiel ao paper publicado pelo Google. Nela diz que devemos usar HBase quando temos **bilhões** de linhas X e **milhões** de colunas. No blog técnico do Netflix, é possível encontrar posts sobre como conseguir mais de 1 milhão de operações de escritas por segundo, além de outras dicas sobre o cluster de Cassandra deles, que hoje conta com mais de 2.700 servidores.

Embora nosso exemplo possa parecer muito pequeno perto de todos esses gigantes da internet, ele foi baseado no sistema de playlists do **Spotify**, que também usa o Cassandra.

6.1 INSTALANDO O CASSANDRA

Assim como o MongoDB, o Cassandra possui pacotes para distribuições **Linux** e **Mac**, e uma versão com instalador para **Windows**. O Cassandra depende apenas da *Java Virtual Machine (JVM)* para rodar, então é preciso ter o Java instalado e configurado antes de instalá-lo.

Além disso, tendo a JVM instalada, é possível baixar a versão compilada do Cassandra e executá-la sem a necessidade de instalar via pacote.

Para instalar no Ubuntu 14.04, podemos configurar o repositório com os seguintes comandos:

```
$ curl -L http://debian.datastax.com/debian/repo_key \
| sudo apt-key add -

$ echo "deb http://debian.datastax.com/community stable main" \
| sudo tee -a /etc/apt/sources.list.d/cassandra.sources.list
```

Depois, precisamos atualizar o gerenciador de pacotes e instalar o pacote do Cassandra, `dsc20` .

```
$ sudo apt-get update
$ sudo apt-get install -y dsc20
```

Para iniciar o Cassadra após o término da instalação, basta executar:

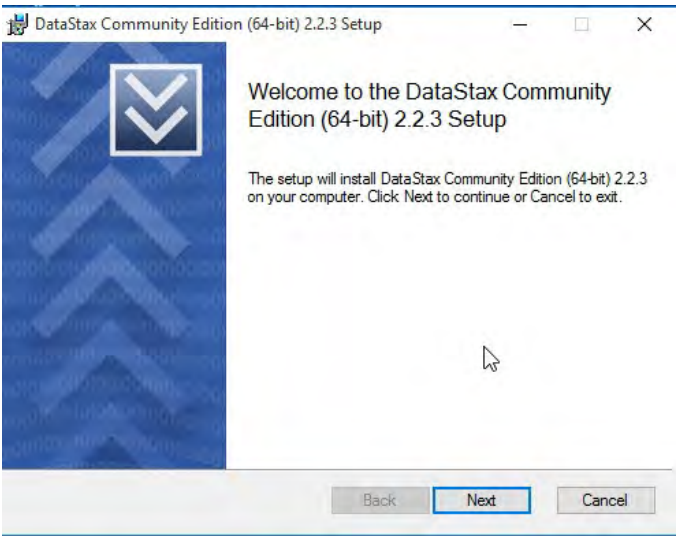
```
$ cassandra -f
```

A instalação no **MacOS** pode ser feita usando o gerenciador `Homebrew` , bastando fazer a atualização dele e, então, instalar o pacote `cassandra` .

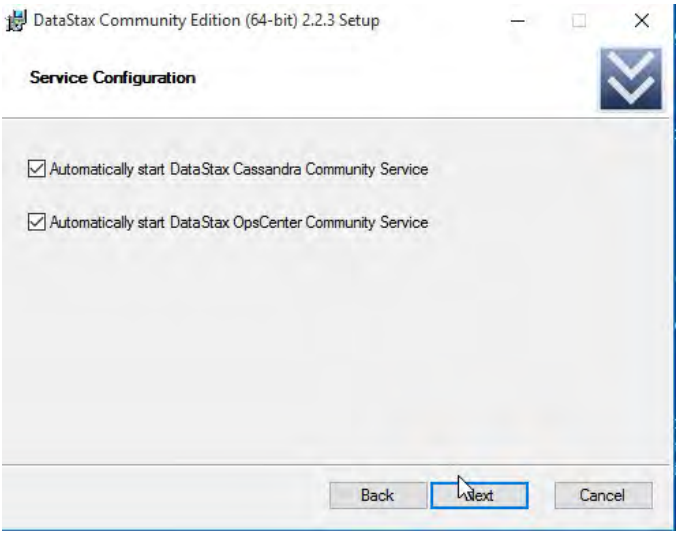
```
$ brew update
$ brew install cassandra
```

No **Windows**, a maneira mais fácil de instalar é usando o instalador `.msi` , que pode ser baixado em: http://downloads.datastax.com/community/. Após o download,

basta executar o instalador e seguir o passo a passo.

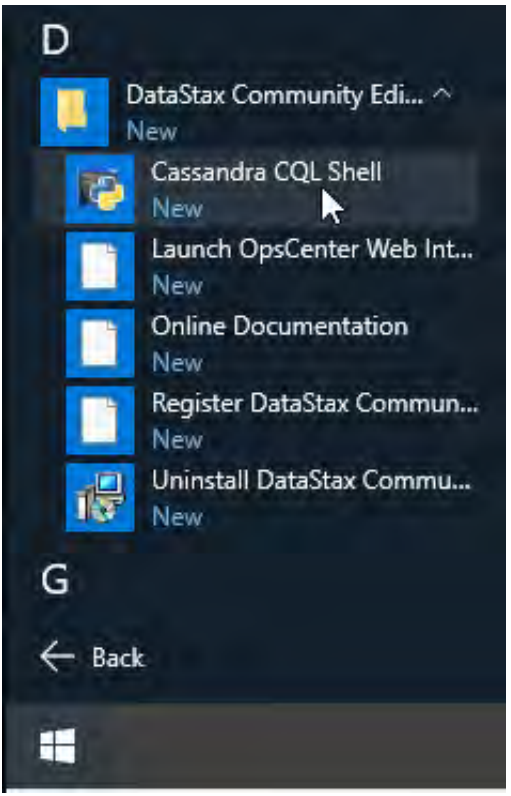


Durante a instalação, é possível deixar marcada a opção para iniciar o Cassandra automaticamente após o término.



Junto com o servidor, o instalador também adiciona o

Cassandra CQL Shell, equivalente ao *cqlsh*, que usaremos em breve.



Assim como nos outros bancos, uma outra alternativa é rodar o Cassandra usando Docker. Para isso, basta executar `docker run cassandra`.

6.2 CRIANDO TABELAS NO CASSANDRA

Em alguns bancos relacionais, antes de criarmos uma tabela, precisamos criar um banco de dados no qual a tabela existirá. No Cassandra, temos uma estrutura semelhante, que é o **Keyspace**. Dentro de um Keyspace, criamos as **Column Families**, o equivalente às tabelas.

Modelo relacional	Modelo Cassandra
Database	Keyspace
Table	Column Family (CF)
Primary key	Row key
Column name	Column name/key
Column value	Column value

O primeiro passo então será criar um keyspace. Para tal, precisamos abrir o console com o Cassandra. O Cassandra possuía duas formar de manipular seus dados: o antigo *Thrift*, que você podia utilizar com o comando `cassandra-cli` , que foi removido a partir do Cassandra 2.2; e novo *Cassandra Query Language (CQL)*, que pode ser usado pelo comando `cqlsh` .

Para começar nosso exemplo, vamos abrir o `cqlsh` .

```
$ cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.7 | CQL spec 3.2.0 | Native protocol
v3]
Use HELP for help.
cqlsh>
```

Dentro do `cqlsh` , podemos listar os keyspaces utilizando o comando `DESCRIBE KEYSPACES` .

```
cqlsh> DESCRIBE KEYSPACES;

system_traces  system
```

Para criar um novo keyspace, é possível usar o comando `CREATE KEYSPACE [nome] WITH REPLICATION = [estratégia de replicação]` . Além do nome, um keyspace precisa de uma estratégia de replicação, que normalmente é *SimpleStrategy* (replica os dados nos próximos servidores sem levar em conta a topologia da rede), ou *NetworkTopologyStrategy* (muito mais complexo, pois permite configurar réplicas por Rack, Data Center etc.).

Para termos durabilidade e alta disponibilidade de verdade, é praticamente indispensável o uso da estratégia *NetworkTopologyStrategy* em produção. Mas em desenvolvimento, podemos usar o *SimpleStrategy* sem problemas. Assim como nos bancos anteriores, vamos criar um namespace chamado `ligado`, dessa vez usando o comando: `CREATE KEYSPACE ligado WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '3'};`

```
cqlsh> CREATE KEYSPACE ligado
WITH replication = {'class': 'SimpleStrategy', 'replication_factor
': '3'};

cqlsh> DESCRIBE KEYSPACES;

system_traces  system ligado
```

Agora que já temos um keyspace criado, criaremos nossa primeira tabela. Antigamente no Cassandra só existiam os *Column Families*, mas nas versões mais novas do banco, eles estão abrindo mão dessa nomenclatura e usando o nome tabela. Consequentemente, o comando para criar uma nova *column family* é `CREATE TABLE`.

Semelhante a um banco relacional, além do nome, precisamos passar uma lista de colunas usando o padrão `_NOME_DA_COLUNA TIPO_DE_DADO_DA_COLUNA_`. Além de nome e das colunas, uma tabela também precisa definir qual coluna será usada como *Row key*, ou chave primária.

A primeira tabela que vamos criar é a de músicas, que conterá o nome da música, do artista e do álbum. A única coisa que precisaremos antes de criar a tabela é dizer ao Cassandra que queremos executar os próximos comandos no keyspace `ligado`.

```
cqlsh> use ligado;
cqlsh:ligado>
```

Agora vamos criar a tabela de músicas:

```
cqlsh:ligado> CREATE TABLE musicas (
  id uuid PRIMARY KEY,
  nome text,
  album text,
  artista text
 );
```

Podemos ver os detalhes da tabela criada usando o comando `DESCRIBE TABLE`.

```
cqlsh:ligado> DESCRIBE TABLE musicas;

CREATE TABLE ligado.musicas (
    id uuid PRIMARY KEY,
    album text,
    artista text,
    nome text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = '{"keys":"ALL", "rows_per_partition":"NONE"}'
    AND comment = ''
    AND compaction = {
'min_threshold': '4',
'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionS
trategy',
'max_threshold': '32'
}
    AND compression = {
'sstable_compression': 'org.apache.cassandra.io.compress.LZ4Compre
ssor'
}
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99.0PERCENTILE';
```

Reparem que, apesar de não termos passado nenhum parâmetro além dos dados básicos da tabela, o Cassandra usou opções padrões para várias configurações avançadas da nossa tabela.

6.3 MANIPULANDO REGISTROS NO CASSANDRA

Assim como a sintaxe para criar keyspace e tabelas se parece muito com o SQL dos bancos relacionais, os comandos básicos para manipular os dados também são muito semelhantes. Para inserir dados, usamos `INSERT INTO nome_da_tabela(col1, col2, col3) VALUES (val1, val2, val3);` . Para visualizar os dados, podemos utilizar `SELECT colunas FROM nome_da_tabela;` .

```
cqlsh:ligado>
INSERT INTO
  musicas (id, nome, album, artista)
VALUES
  (a70ca7ff-6d57-4f89-be89-08421c432bb7, 'Help', 'Help', 'Beatles'
);

cqlsh:ligado> SELECT * FROM musicas;

 id                                   | album | artista | nome
--------------------------------------+-------+---------+------
 a70ca7ff-6d57-4f89-be89-08421c432bb7 |  Help | Beatles | Help

(1 rows)
```

Para atualizar um registro, basta utilizar o `UPDATE` , também muito semelhante ao SQL.

```
cqlsh:ligado>
UPDATE musicas SET
  nome='Help!',
  album='Help!'
WHERE id = a70ca7ff-6d57-4f89-be89-08421c432bb7;

cqlsh:ligado> SELECT * FROM musicas;

 id                                   | album | artista | nome
--------------------------------------+-------+---------+-------
 a70ca7ff-6d57-4f89-be89-08421c432bb7 | Help! | Beatles | Help!

(1 rows)
```

Finalmente, para acabar as operações básicas, usaremos o

`DELETE` para apagar um registro.

```
cqlsh:ligado>
DELETE from musicas
WHERE id = a70ca7ff-6d57-4f89-be89-08421c432bb7;

cqlsh:ligado> SELECT * FROM musicas;

id | album | artista | nome
----+-------+---------+------

(0 rows)
```

6.4 BUSCANDO DADOS NO CASANDRA

Antes de começarmos a fazer as buscas, vamos adicionar as músicas utilizadas no exemplo:

```
cqlsh:ligado> INSERT INTO musicas (id, nome, album, artista)
VALUES (04b57c98-33df-11e5-a151-feff819cdc9f,
'Help!', 'Help!', 'Beatles');

cqlsh:ligado> INSERT INTO musicas (id, nome, album, artista)
VALUES (1a8d6a80-33df-11e5-a151-feff819cdc9f,
'Yestarday', 'Help!', 'Beatles');

cqlsh:ligado> INSERT INTO musicas (id, nome, album, artista)
VALUES (04b57f0e-33df-11e5-a151-feff819cdc9f,
'Something', 'Abbey Road', 'Beatles');

cqlsh:ligado> INSERT INTO musicas (id, nome, album, artista)
VALUES (1a8d649a-33df-11e5-a151-feff819cdc9f,
'Blackbird', 'The Beatles', 'Beatles');

cqlsh:ligado> SELECT * FROM musicas;

id                                   | album     |artista| nome
-------------------------------------+-----------+-------+--------
04b57c98-33df-11e5-a151-feff819cdc9f|      Help!|Beatles|   Help!
1a8d6a80-33df-11e5-a151-feff819cdc9f|      Help!|Beatles|Yesterday
1a8d649a-33df-11e5-a151-feff819cdc9f|The Beatles|Beatles|Blackbird
04b57f0e-33df-11e5-a151-feff819cdc9f| Abbey Road|Beatles|Something

(4 rows)
```

Se quisermos, por exemplo, buscar todas as músicas do artista

`Beatles`, já é possível imaginar como usar a cláusula `WHERE` para fazer o filtro, e provavelmente você não está errado. Para filtrarmos os registros pelo valor de uma das colunas, basta usar o `SELECT colunas FROM tabela WHERE campo=valor`. Vamos tentar fazer a busca e verificar o resultado que o banco retorna.

```
cqlsh:ligado> SELECT *
              FROM musicas
              WHERE artista='Beatles';

 InvalidRequest: code=2200 [Invalid query]
 message="No secondary indexes on the restricted columns support t
he provided operators: "
```

Apesar de a sintaxe da busca ser idêntica ao SQL dos bancos relacionais, diferente deles, o Cassandra não faz buscas em campos que não possuem índices. Nos bancos relacionais, apesar de não ser performático, esse tipo de busca funciona. Porém, por causa da forma com que o Cassandra distribui os dados, uma busca dessas seria tão ineficiente que eles preferiram não suportar.

Para podermos filtrar as músicas por artista, vamos precisar criar um índice nessa coluna, usando a sintaxe `CREATE INDEX ON nome_da_tabela (nome_do_campo)`.

```
cqlsh:ligado> CREATE INDEX ON musicas (artista);
cqlsh:ligado> SELECT *
              FROM musicas
              WHERE artista='Beatles';

id                                   |album      |artista| nome
-------------------------------------+-----------+-------+--------
04b57c98-33df-11e5-a151-feff819cdc9f|      Help!|Beatles|    Help!
1a8d6a80-33df-11e5-a151-feff819cdc9f|      Help!|Beatles|Yestarday
1a8d649a-33df-11e5-a151-feff819cdc9f|The Beatles|Beatles|Blackbird
04b57f0e-33df-11e5-a151-feff819cdc9f| Abbey Road|Beatles|Something

(4 rows)
```

6.5 ARMAZENANDO AS PLAYLISTS

Apesar de precisarmos suportar o versionamento das listas, ter de calcular quais as músicas e a ordem em que elas devem ser tocadas toda vez que alguém quiser ver, ou escutar uma playlist, não é nada performático. Então, também teremos uma tabela auxiliar com uma espécie de "fotografia" de como a playlist está nesse momento, que nada mais é do que um cache da versão calculada.

Toda vez que um cliente fizer uma alteração na playlist, vamos inserir um novo registro do controle de versão da playlist. Após interpretar a mudança e calcular o seu conteúdo, armazenaremos esses dados nessa nova tabela.

Como os clientes não escreverão direto nesta tabela, é mais fácil controlar os problemas de concorrência. Por isso, vamos usar nessa tabela a forma mais simples de representar a lista, apenas a ligação entre o `id` dela, a música e qual a posição da música naquela lista.

```
PlaylistFinal
| id_playlist | id_musica | posicao |
|      1      |      1    |    1    |
|      1      |      2    |    2    |
|      1      |      4    |    3    |
|      1      |      3    |    4    |
```

Usando um banco relacional, é muito simples pegar as informações das músicas quando buscarmos uma playlist usando `JOIN` no nosso SQL.

```
SELECT m.nome, m.artista, p.posicao
FROM playlistFinal p
  JOIN musicas m
    ON p.idMusica = m.id
WHERE p.idPlaylist = 1;
```

Infelizmente, assim como os bancos chave/valor e orientados a documentos, os bancos de dados colunares também não suportam a operação de juntar (`JOIN`) tabelas. Por causa disso, novamente vamos cair no dilema entre *N+1 queries* ou *desnormalização* dos dados.

Para o nosso problema da playlists, imagine ter de executar 31 operações no banco de dados para exibir uma simples lista de músicas. Nós podemos fazer uma chamada extra na hora de tocar uma música, mas quando carregar uma playlist, gostaria de poder ver, pelo menos, o nome da música e do artista de todas elas. Então, não teremos muitas opções se não partir para a desnormalização.

Isso não é uma regra, mas é muito comum encontrar a desnormalização como boa prática para bancos colunares. Especialmente no Cassandra, que possui ótimos registros de performance de escrita, o comum é tentar manter a complexidade na escrita para facilitar ao máximo a leitura.

Como resultado final, nossa tabela com o estado atual das listas terá todos os dados relevantes da música, que no nosso modelo serão todos os atributos (nome, álbum e artista).

```
cqlsh:ligado> CREATE TABLE playlist_atual (
  id_playlist uuid PRIMARY KEY,
  posicao int,
  id_musica uuid,
  nome text,
  album text,
  artista text
 );
```

No nosso exemplo, a tabela de música é um pouco simplificada. Por isso, o resultado final da nossa tabela `playlist_atual` parece apenas uma cópia da tabela `musicas`. Se de fato as tabelas fossem tão parecidas, não existiria motivo para criar a tabela `musicas`. Porém, considerando que nesta tabela poderia haver mais informações sobre a música, ou até mesmo a própria música em um formato binário, vamos continuar nosso exemplo com as duas tabelas. Com isso em mente, podemos seguir em frente e adicionar os dados da nosso primeira playlist.

```
cqlsh:ligado> INSERT INTO
playlist_atual (id_playlist, posicao, id_musica, nome, album, arti
```

```
sta)
VALUES (c4f408dd-00f3-488e-8800-050d2775bbc7, 1,
04b57c98-33df-11e5-a151-feff819cdc9f, 'Help!', 'Help!', 'Beatles')
;

cqlsh:ligado> INSERT INTO playlist_atual
(id_playlist, posicao, id_musica, nome, album, artista)
VALUES (c4f408dd-00f3-488e-8800-050d2775bbc7, 2,
1a8d6a80-33df-11e5-a151-feff819cdc9f, 'Yestarday', 'Help!', 'Beatl
es');
```

Vejamos a situação atual da nossa playlist:

```
cqlsh:ligado> SELECT * FROM playlist_atual ;

 id| album | artista | id_musica  | nome      | posicao
---+-------+---------+------------+-----------+---------
 1 | Help! | Beatles |      2     | Yesterday |       2

(1 rows)
```

Curiosamente, em vez de dois registros, temos apenas um. Durante a criação da tabela, cometemos um erro e deixamos o `id` da playlist como chave primária dela. O que não podemos deixar de notar é que, diferente de um banco relacional, ao tentar adicionar um novo registro usando a mesma chave primária, em vez de o Cassandra lançar um erro e recusar a escrita, ele simplesmente altera o registro com os novos valores.

No Cassandra, tanto o `INSERT` quanto o `UPDATE` executam a operação `upsert`. Se já existe o registro, ele altera; caso contrário, ele cria.

Como nós podemos ter várias músicas na mesma playlist, vamos precisar criar uma chave composta usando algum outro campo junto. Uma possibilidade é utilizar o `id` da música, mas isso nos impediria de ter a mesma música na mesma playlist.

Não é muito comum, mas se usarmos a posição como parte da chave, teremos um modelo mais flexível. A solução será utilizar como chave primária da nossa tabela uma chave composta, com o

`id` da playlist e a posição da música. Infelizmente, o Cassandra não suporta alteração na chave primária, então precisaremos apagar e criar a tabela novamente.

```
cqlsh:ligado> DROP TABLE playlist_atual;

cqlsh:ligado> CREATE TABLE playlist_atual (
  id_playlist uuid,
  posicao int,
  id_musica uuid,
  nome text,
  album text,
  artista text,
  PRIMARY KEY (id_playlist, posicao)
 );

cqlsh:ligado> INSERT INTO playlist_atual
(id_playlist, posicao, id_musica, nome, album, artista)
VALUES (c4f408dd-00f3-488e-8800-050d2775bbc7, 1,
04b57c98-33df-11e5-a151-feff819cdc9f,
'Help!', 'Help!', 'Beatles');

cqlsh:ligado> INSERT INTO playlist_atual
(id_playlist, posicao, id_musica, nome, album, artista)
VALUES (c4f408dd-00f3-488e-8800-050d2775bbc7, 2,
1a8d6a80-33df-11e5-a151-feff819cdc9f,
'Yestarday', 'Help!', 'Beatles');

cqlsh:ligado> INSERT INTO playlist_atual
(id_playlist, posicao, id_musica, nome, album, artista)
VALUES (c4f408dd-00f3-488e-8800-050d2775bbc7, 3,
04b57f0e-33df-11e5-a151-feff819cdc9f,
'Something', 'Abbey Road', 'Beatles');

cqlsh:ligado> INSERT INTO playlist_atual
(id_playlist, posicao, id_musica, nome, album, artista)
VALUES (c4f408dd-00f3-488e-8800-050d2775bbc7, 4,
1a8d649a-33df-11e5-a151-feff819cdc9f,
'Blackbird', 'The Beatles', 'Beatles');
```

6.6 MIGRANDO O MODELO DE VERSIONAMENTO PARA UM BANCO COLUNAR

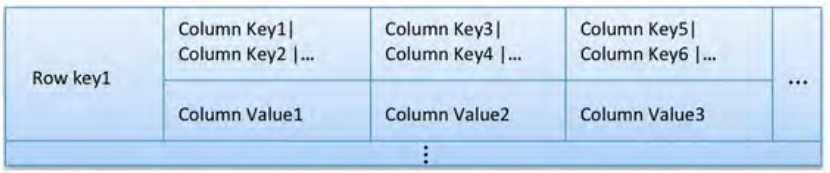
Pensando em estrutura de dados, embora os bancos colunares também utilizem tabelas, os registros são orientados a colunas, e não linhas. Apesar de as tabelas que criamos anteriormente se parecerem muito com tabelas de um banco relacional, não é esse o principal modelo de tabela usado em um banco colunar.

Um modelo de dado amplamente utilizado em bancos colunares são as *wide rows*, ou *dynamic columns*. Essas tabelas, diferentes das que usamos anteriormente, entram na categoria de tabelas *schemaless*, ou seja, elas não possuem colunas predefinidas. Na verdade, nestas tabelas o que nós estamos acostumados a chamar de linha (*row*) é, na verdade, chamado de partição (*partition*).

Cada partição possui sua chave única (*partition key*) e, a partir desta chave, podemos adicionar vários pares chave/valor, ou na terminologia do `cql` : coluna (*columm*) para a chave, e linha (*row*) para o valor. Cada par de chave/valor (coluna/linha) em uma partição é chamado de célula (*cell*).

Termo do Thrift	Termo do CQL
Linha	Partição
Coluna	Célula
[Nome ou valor do componente da célula]	Coluna
[Grupo de células com um componente de prefixos compartilhado]	Linha

Um exemplo de uma tabela *wide row*:



No nosso problema do controle de versão de playlists, usaremos

esse modelo de tabela com colunas dinâmicas. Neste modelo, quando queremos adicionar uma informação sobre um determinado objeto, em vez de adicionarmos novas linhas usando um identificador único do elemento, nós adicionamos uma nova célula, composta por um nome e um valor, que serão a coluna e a linha.

Voltando ao exemplo da playlist, quando montamos a tabela `playlist_versionada` , estávamos adicionando uma nova linha para cada modificação realizada nela. Em um banco colunar, a solução mais comum seria ter uma partição para uma playlist, e adicionar uma nova célula para cada modificação efetuada nela.

Exemplo utilizado anteriormente:

```
|id_playlist|versao|  acao  |posicao|tamanho|para|  idMusica |
|     1     |   1  |adiciona|   1   |       |    |    Help!  |
|     1     |   2  |adiciona|   2   |       |    | Yesterday |
|     1     |   3  |adiciona|   3   |       |    | Blackbird |
|     1     |   4  |adiciona|   4   |       |    | Something |
|     1     |   5  |  troca |   3   |   1   | 2  |           |
```

Esta tabela em um banco colunar poderia ser transformada em:

id_playlist	1	2	3	4	5
1	ADICIONA(Help!)	ADICIONA(Yesterday)	ADICIONA(Blackbird)	ADICIONA(Something)	TROCA(3,1,2)

Reparem como transformamos as linhas em colunas. Passamos a utilizar a `versao` como nome da coluna, e os outros campos foram para o valor da célula. Usando o *Thrift*, o cliente do Cassandra que está deprecado, podemos adicionar dados em uma tabela como se estivéssemos manipulando um `hashmap` . Para inserir os dados listados, teríamos um código parecido com:

```
set playlist_versionada[1][1] = 'ADI(Help!)';
set playlist_versionada[1][2] = 'ADI(Yesterday)';
set playlist_versionada[1][3] = 'ADI(Blackbird)';
set playlist_versionada[1][4] = 'ADI(Something)';
set playlist_versionada[1][5] = 'TROCA(3,1,2)';
```

Usando essa sintaxe antiga, podíamos inserir os dados com o comando `set`, passando como argumento o nome da tabela, o identificador da *partition*, o identificador da *coluna* e, por último, o valor da *linha*, resultando em `set TABELA[coluna] = valor;`. Assim como o `insert` e o `update` do CQL, usando essa sintaxe, nós criaríamos uma partição nova caso ela não existisse, ou a modificaríamos e adicionaríamos uma nova célula caso não existisse aquela coluna na partição, ou alteraríamos o valor da célula para aquela combinação de partição e coluna.

No SQL, a sintaxe e a visualização dos dados são muito diferentes, mas a forma como os dados estão armazenados é exatamente a mesma. Para criar uma tabela com colunas dinâmicas usando SQL, precisamos usar o parâmetro `WITH COMPACT STORAGE` na criação da tabela. Além disso, existem algumas regras na criação dessas tabelas.

Obrigatoriamente, elas precisam de três "colunas": uma que será a chave da partição, uma que será o nome das colunas dinâmicas e a última que será o valor da linha. A outra regra é que a chave primária deve ser uma chave composta entre a chave da partição e o nome da coluna.

A nossa tabela `playlist_versionada` terá as colunas `id_playlist`, `versao` e `modificacao`, e o comando que vamos usar para criá-la é o seguinte:

```
cqlsh:ligado> CREATE TABLE playlist_versionada (
                id_playlist uuid,
                versao int,
                modificacao text,
                PRIMARY KEY (id_playlist, versao)
              ) WITH COMPACT STORAGE;
```

E agora podemos inserir os dados:

```
cqlsh:ligado> INSERT INTO playlist_versionada
    (id_playlist, versao, modificacao)
```

```
VALUES
    (c4f408dd-00f3-488e-8800-050d2775bbc7, 1, 'ADI(Help!)');

cqlsh:ligado> INSERT INTO playlist_versionada
    (id_playlist, versao, modificacao)
VALUES
    (c4f408dd-00f3-488e-8800-050d2775bbc7, 2, 'ADI(Yesterday)');

cqlsh:ligado> INSERT INTO playlist_versionada
    (id_playlist, versao, modificacao)
VALUES
    (c4f408dd-00f3-488e-8800-050d2775bbc7, 3, 'ADI(Blackbird)');

cqlsh:ligado> INSERT INTO playlist_versionada
    (id_playlist, versao, modificacao)
VALUES
    (c4f408dd-00f3-488e-8800-050d2775bbc7, 4, 'ADI(Something)');

cqlsh:ligado> INSERT INTO playlist_versionada
    (id_playlist, versao, modificacao)
VALUES
    (c4f408dd-00f3-488e-8800-050d2775bbc7, 5, 'TROCA(3,1,2)');


cqlsh:ligado> SELECT * FROM playlist_versionada ;

 id_playlist                          | versao | modificacao
--------------------------------------+--------+----------------
 c4f408dd-00f3-488e-8800-050d2775bbc7 |      1 |     ADI(Help!)
 c4f408dd-00f3-488e-8800-050d2775bbc7 |      2 | ADI(Yesterday)
 c4f408dd-00f3-488e-8800-050d2775bbc7 |      3 | ADI(Blackbird)
 c4f408dd-00f3-488e-8800-050d2775bbc7 |      4 | ADI(Something)
 c4f408dd-00f3-488e-8800-050d2775bbc7 |      5 |   TROCA(3,1,2)

(5 rows)
```

Além da facilidade para lidar com escalabilidade e disponibilidade, já vimos algumas vantagens e desvantagens na forma de armazenar e manipular os dados no Cassandra, como a utilização de tabelas do tipo *wild rows*. Como os dados realmente estão armazenados? E quais outras abordagens podemos usar para armazenar nossos dados em um banco de dados colunar?

CAPÍTULO 7

MAIS SOBRE BANCOS COLUNARES

Embora o resultado da consulta seja muito parecido com uma tabela de um banco relacional, os dados estão orientados a colunas internamente. Nas versões do Cassandra inferiores a 2.2, existia um outro cliente além do `cqlsh` , o `cassando-cli` . Se você instalar uma versão antiga, podemos consultar os dados e ter uma visão diferente deles.

7.1 COMO O CASSANDRA ARMAZENA OS DADOS INTERNAMENTE

Como explicado anteriormente, no nosso exemplo existe apenas uma partição, que no cliente antigo do Cassandra era chamada de *Row* (linha). Por isso, no resultado da busca existe apenas uma linha. O valor que usamos em `versao` é representado como o nome da coluna (`name=1`), e o valor inserido em `modificacao` fica como valor (`value=ADI(Help!)`).

```
$ cassandra-cli
[default@unknown] use ligado;
Authenticated to keyspace: ligado

[default@ligado] list playlist_versionada;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: c4f408dd-00f3-488e-8800-050d2775bbc7
```

```
=> (name=1, value=ADI(Help!), timestamp=1438818548125646)
=> (name=2, value=ADI(Yesterday), timestamp=1438818553747085)
=> (name=3, value=ADI(Blackbird), timestamp=1438818571767466)
=> (name=4, value=ADI(Something), timestamp=1438818576933964)
=> (name=5, value=TROCA(3,1,2), timestamp=1438818586229982)

1 Row Returned.
```

Este cliente antigo do Cassandra foi totalmente substituído pelo novo `cql`, que tenta deixar a utilização do Cassandra mais fácil para quem está acostumado com os bancos relacionais. Muitas vezes a sua sintaxe se assemelha bastante ao SQL, mas, apesar disso, é importante lembrarmos que a estrutura real dos dados é bem diferente. Por exemplo, podemos visualizar os dados que inserimos na tabela `playlist_atual` por este cliente para termos uma ideia melhor de como os dados realmente estão.

RODANDO UMA VERSÃO ANTIGA DO CASSANDRA

Apesar de todos os outros exemplos do livro dependerem apenas do `cqlsh`, e ser recomendado sempre utilizar as versões mais novas, se você desejar executar o `cassandra-cli`, a maneira mais simples é utilizando o docker. Com o comando `docker run --name cassandra-antigo cassandra:2.1`, você terá um contêiner rodando uma versão antiga do Cassandra. Então, pode executar o `cassandra-cli` dentro deste contêiner com `docker exec -it cassandra-antigo cassandra-cli`.

```
[default@ligado] list playlist_atual;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: c4f408dd-00f3-488e-8800-050d2775bbc7
=> (name=1:, value=, timestamp=1438305485466704)
=> (name=1:album, value=48656c7021, timestamp=1438305485466704)
```

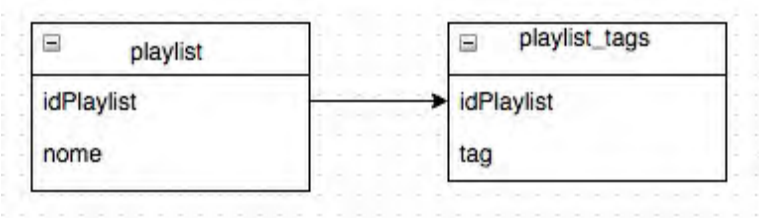
```
=> (name=1:artista, value=426561746c6573, timestamp=14383054854667
04)
=> (name=1:id_musica, value=04b57c9833df11e5a151feff819cdc9f, time
stamp=1438305485466704)
=> (name=1:nome, value=48656c7021, timestamp=1438305485466704)
=> (name=2:, value=, timestamp=1438305493289750)
=> (name=2:album, value=48656c7021, timestamp=1438305493289750)
=> (name=2:artista, value=426561746c6573, timestamp=14383054932897
50)
=> (name=2:id_musica, value=1a8d6a8033df11e5a151feff819cdc9f, time
stamp=1438305493289750)
=> (name=2:nome, value=596573746172646179, timestamp=1438305493289
750)
=> (name=3:, value=, timestamp=1438305505812387)
=> (name=3:album, value=416262657920526f6164, timestamp=1438305505
812387)
=> (name=3:artista, value=426561746c6573, timestamp=14383055058123
87)
=> (name=3:id_musica, value=04b57f0e33df11e5a151feff819cdc9f, time
stamp=1438305505812387)
=> (name=3:nome, value=536f6d657468696e67, timestamp=1438305505812
387)
=> (name=4:, value=, timestamp=1438305511333881)
=> (name=4:album, value=54686520426561746c6573, timestamp=14383055
11333881)
=> (name=4:artista, value=426561746c6573, timestamp=14383055113338
81)
=> (name=4:id_musica, value=1a8d649a33df11e5a151feff819cdc9f, time
stamp=1438305511333881)
=> (name=4:nome, value=426c61636b62697264, timestamp=1438305511333
881)

1 Row Returned.
Elapsed time: 23 msec(s).
```

Com essa visualização, podemos identificar que, embora o `cql` exiba os dados como tabelas com colunas fixas, na verdade internamente os dados são armazenados de uma maneira muito semelhante a uma tabela com colunas dinâmicas, na qual temos uma parte da chave composta mais o nome da coluna com o real nome de coluna. Além disso, podemos ver que não existe uma coluna chamada `posicao`, que, por ser parte da *partition key*, é armazenada como sem valor. O que no `cql` é apresentado como valor internamente é armazenado como nome de coluna.

7.2 ADICIONANDO TAGS ÀS LISTAS

Agora que já criamos algumas tabelas e visualizamos suas representações internas, podemos partir para estruturas de dados um pouco mais complexas. Uma outra característica que uma playlist pode ter são tags. Uma tag é apenas um rótulo, uma palavra que destaca alguma característica especial de uma lista.

Por exemplo, podemos ter uma tag `"80's"` que poderia ser adicionada em uma playlist com músicas dos anos 80, ou uma tag baseada em gênero, uma playlist que foi tocada ao vivo pela banda ou qualquer outra informação adicional. Em um banco relacional, as duas maneiras mais comuns de implementar a funcionalidade são: ou criar uma tabela `tags` e outra `playlist_tags` que terá uma chave estrangeira para a tabela `tags` ; ou apenas uma tabela `playlist_tags` que armazenará os valores das tags desnormalizadas. Em ambos os casos, a chave primária da tabela `playlist_tags` será composta pelo `id` da playlist e o `id` da tag, ou a tag em si dependendo da opção.

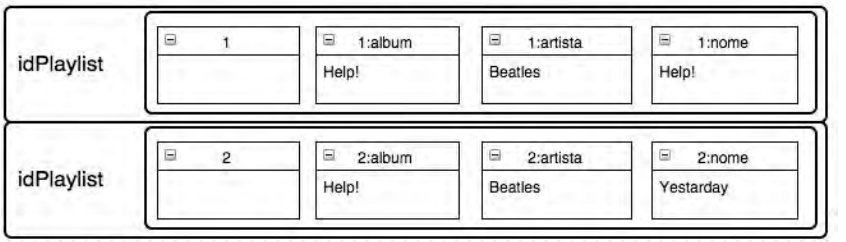


Como vamos implementar usando um banco colunar que não possui `join` , a primeira opção não parece uma boa ideia. Então,

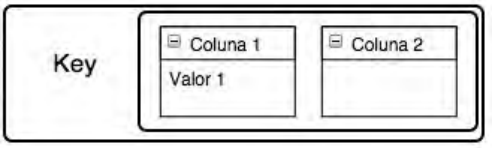
vamos modelar nossa tabela baseada na segunda opção.

Quando vamos migrar de um banco relacional para qualquer outro tipo de banco NoSQL, temos de tomar muito cuidado para não fazer uma "tradução ao pé da letra" e acabar com uma estrutura quebrada ou ineficiente. Além da questão do `join` que eliminou a primeira opção, precisamos lembrar da representação interna dos dados e de como um banco colunar funciona para modelarmos nossos dados no Cassandra.

Para migrar essa estrutura de dados para um banco colunar, temos algumas opções que vamos comparar. Mas antes vamos relembrar como ficou a tabela `playlist_atual` .

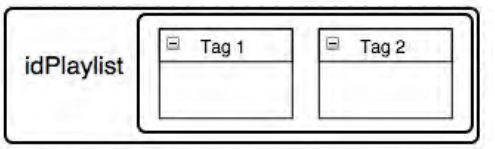


Quando criamos uma chave composta na nossa tabela, internamente ela foi armazenada como um nome de coluna sem valor. Em um banco colunar os dados sempre estarão em células, com nome de coluna e valor, mas muitas vezes só o nome da coluna já é suficiente para armazenar as informações que precisamos.



Como queremos poder adicionar várias tags por lista e queremos garantir a unicidade delas (não queremos tags repetidas

na mesma lista), vamos nos aproveitar desta técnica e criar uma coluna vazia para a tag adicionada à lista. Essa modelagem é uma das opções mais comuns para esse tipo de dado em um banco colunar.



Assim como no exemplo anterior, o Cassandra criou uma coluna vazia para o campo `posicao`, que fazia parte da *partition key*. Criaremos a tabela `playlist_tags` com dois campos, ambos formando a chave primária.

```
cqlsh:ligado> CREATE TABLE playlist_tags (
  id_playlist uuid,
  tag text,
  PRIMARY KEY (id_playlist, tag)
 );

cqlsh:ligado> INSERT INTO playlist_tags (id_playlist, tag)
VALUES
(c4f408dd-00f3-488e-8800-050d2775bbc7, 'Beatles');

cqlsh:ligado> INSERT INTO playlist_tags (id_playlist, tag)
VALUES
(c4f408dd-00f3-488e-8800-050d2775bbc7, '60s');

cqlsh:ligado> SELECT * FROM playlist_tags
WHERE id_playlist = c4f408dd-00f3-488e-8800-050d2775bbc7;


 id_playlist                          | tag
--------------------------------------+---------
 c4f408dd-00f3-488e-8800-050d2775bbc7 |     60s
 c4f408dd-00f3-488e-8800-050d2775bbc7 | Beatles
```

Usando o antigo `cassandra_cli`, como esperado, temos uma coluna para cada tag, ambas sem valor.

```
[default@ligado] list playlist_tags;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: c4f408dd-00f3-488e-8800-050d2775bbc7
=> (name=60s:, value=, timestamp=1440297250921695)
=> (name=Beatles:, value=, timestamp=1440297246620578)

1 Row Returned.
```

Desde o início do capítulo, estamos nos referenciando às playlists, mas ainda não criamos a tabela principal que as armazena. Agora que já conhecemos bem o Cassandra e como funciona um banco colunar, já podemos criar a tabela `playlist`.

```
cqlsh:ligado> CREATE TABLE playlists (
  id uuid PRIMARY KEY,
  nome text
 );

cqlsh:ligado> INSERT INTO playlists (id, nome)
VALUES
(c4f408dd-00f3-488e-8800-050d2775bbc7, 'Beatles forever');

cqlsh:ligado> SELECT * FROM playlists;

 id                                   | nome
--------------------------------------+-----------------
 c4f408dd-00f3-488e-8800-050d2775bbc7 | Beatles forever
```

7.3 USANDO TIPOS ESPECIAIS DO CASSANDRA

Apesar de nossa implementação atual funcionar, nas últimas versões do Cassandra ganhamos a possibilidade de trabalhar com outros tipos de dados especiais, como listas, conjuntos e mapas. Esse exemplo das tags poderia facilmente deixar de ser uma tabela e se tornar um conjunto de palavras dentro da tabela `playlist`.

O tipo especial para conjuntos é o `set` e, quando vamos usá-lo, precisamos definir o que podemos armazenar neste conjunto, algo

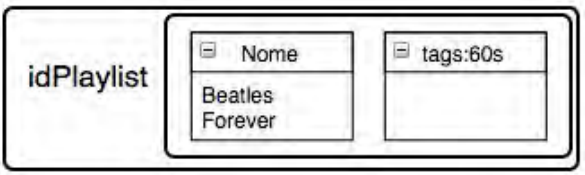
semelhante à funcionalidade `Generics` do Java ou C#. No nosso caso, queremos um `Set` de palavras, então usaremos `set<text>`. Vamos modificar a tabela `playlist` para adicionar uma nova coluna `tags` do tipo `set<test>`.

```
cqlsh:ligado> ALTER TABLE playlists ADD tags set<text>;

cqlsh:ligado> UPDATE playlists
SET tags = {'Beatles', '60s'}
WHERE id = c4f408dd-00f3-488e-8800-050d2775bbc7;

cqlsh:ligado> SELECT * FROM playlists;

id                                   |nome           |tags
-------------------------------------+---------------+-------------
-----
c4f408dd-00f3-488e-8800-050d2775bbc7|Beatles forever|{'Beatles', '
60s'}
```

Para não perder o costume, vamos dar uma olhada em como os dados estão armazenados usando o `cassandra-cli`:

```
[default@ligado] list playlists;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: c4f408dd-00f3-488e-8800-050d2775bbc7
=> (name=, value=, timestamp=1440297241398249)
=> (name=nome, value=426561746c657320666f7265766572,
timestamp=1440297241398249)
=> (name=tags:363073, value=, timestamp=1440342382327002)
=> (name=tags:426561746c6573, value=, timestamp=1440342382327002)

1 Row Returned.
```

Assim como nossa implementação, o Cassandra internamente armazena uma coluna para cada tag, usando o conteúdo dela como parte do nome da coluna. A diferença é que ele usa uma espécie de prefixo no nome.



Agora que temos as tags em um `set` dentro da tabela da playlist, podemos facilmente exibir as tags de uma lista usando apenas uma operação no banco de dados. Mas o que vamos precisar fazer para exibir todas as playlist de uma determinada tag? Vamos ver como funciona o filtro por tags na nossa abordagem antiga, usando a tabela `playlist_tags` .

```
cqlsh:ligado> SELECT * FROM playlist_tags WHERE tag = '60s';
InvalidRequest: code=2200 [Invalid query] message="Cannot execute
this query as
it might involve data filtering and thus may have unpredictable pe
rformance. If
you want to execute this query despite the performance unpredictab
ility,
use ALLOW FILTERING"
```

Praticamente todas as queries que podem ter problemas de performance são desabilitadas no Cassandra, por padrão. Para executar esse tipo de busca, o Cassandra precisa recuperar a tabela inteira e filtrar em memória pelo valor buscado.

Se a sua tabela é pequena ou a busca em questão retornar a grande maioria dos registros (mais de 90% dos registros, por exemplo), habilitar o filtro pode ser uma boa ideia. Para utilizar o filtro, basta usar `ALLOW FILTERING` no final da sua query.

```
cqlsh:ligado> SELECT * FROM playlist_tags
WHERE tag = '60s' ALLOW FILTERING;

 id_playlist                          | tag
--------------------------------------+-----
 c4f408dd-00f3-488e-8800-050d2775bbc7 | 60s
```

No nosso exemplo, muito provavelmente não teremos tags

repetidas em 90% das listas, então usar o `ALLOW FILTERING` com certeza não será uma boa opção para nós. Ainda podemos adicionar um index secundário como fizemos na tabela `musicas` no capítulo anterior, ou ainda podemos criar uma nova tabela fazendo o caminho contrário do relacionamento.

Como em bancos colunares é muito comum utilizar desnormalização, a abordagem de criar a segunda tabela com o caminho contrário é muito usada. Agora que conhecemos o tipo especial `Set`, podemos usar a mesma modelagem da tabela `playlist`, e criar uma nova tabela `tags` com o `set` de playlists.

```
cqlsh:ligado> CREATE TABLE tags (
  nome text PRIMARY KEY,
  playlists set<uuid>
 );

 cqlsh:ligado> INSERT INTO tags (nome, playlists)
 VALUES ( '60s', {c4f408dd-00f3-488e-8800-050d2775bbc7});

 cqlsh:ligado> INSERT INTO tags (nome, playlists)
 VALUES ( 'Beatles', {c4f408dd-00f3-488e-8800-050d2775bbc7});

cqlsh:ligado> SELECT * FROM tags;

 nome    | playlists
---------+--------------------------------------
 Beatles | {c4f408dd-00f3-488e-8800-050d2775bbc7}
     60s | {c4f408dd-00f3-488e-8800-050d2775bbc7}

(2 rows)

cqlsh:ligado> SELECT * FROM tags WHERE nome = '60s';

 nome | playlists
------+--------------------------------------
  60s | {c4f408dd-00f3-488e-8800-050d2775bbc7}

(1 rows)
```

Agora podemos facilmente listar todas as tags e todas as playlists, além de poder listar as tags de uma determinada playlist ou todas as playlists de uma determinada tag. Usando essa modelagem

com `set` s nas tabelas `tags` e `playlists`, resolvemos por completo o problema das tags nas lista. Por isso, já podemos apagar a tabela `playlist_tags`.

```
cqlsh:ligado> DROP TABLE playlist_tags;
```

Uma outra abordagem muito prática para resolver nosso filtro de playlists por tags é a funcionalidade recém-adicionada de indexação de coleções. A partir do Cassandra 2.1, podemos adicionar um índice no nosso `set`, e então filtrar as listas com apenas uma query. A criação do índice funciona da mesma forma que vimos anteriormente: `CREATE INDEX ON NOME_DA_TABELA (NOME_DAS_COLUNAS)`.

```
cqlsh:ligado> CREATE INDEX ON playlists (tags);
```

Agora que temos o índice criado, podemos fazer nossa busca. A única diferença é que, para filtrar dentro da coleção, temos de usar a palavra-chave `CONTAINS`. A sintaxe para buscar por uma coleção é: `SELECT CAMPOS FROM NOME_DA_TABELA WHERE COLUNA_DA_COLEÇÃO CONTAINS VALOR`.

```
cqlsh:ligado> SELECT * FROM playlists
WHERE tags CONTAINS '60s';

id                                   |nome           |tags
-------------------------------------+---------------+-------------
-----
c4f408dd-00f3-488e-8800-050d2775bbc7|Beatles forever|{'60s', 'Beat
les'}
```

O último detalhe importante de como trabalhar com as coleções é como adicionar um tag a uma lista quando não sabemos se esta já possui tags. Vamos trabalhar com a ideia de adicionar uma nova tag `classic rock` na nossa playlist `Beatles forever`. Se usarmos `UPDATE playlists SET tags = {'classic rock'} WHERE id = c4f408dd-00f3-488e-8800-050d2775bbc7`, vamos sobrescrever a coleção toda.

```
cqlsh:ligado> UPDATE playlists
SET tags = {'classic rock'}
WHERE id = c4f408dd-00f3-488e-8800-050d2775bbc7;

cqlsh:ligado> SELECT * FROM playlists;

id                                   |nome           |tags
-------------------------------------+---------------+-------------
---
c4f408dd-00f3-488e-8800-050d2775bbc7|Beatles forever|{'classic roc
k'}
```

Quando queremos modificar uma coleção, devemos usar o estado atual e somar os valores que vamos adicionar, resultando no seguinte comando: `UPDATE NOME_DA_TABELA set COLUNA_DA_COLECAO = COLUNA_DA_COLECAO + VALOR .`

```
cqlsh:ligado> UPDATE playlists
SET tags = {'Beatles', '60s'}
WHERE id = c4f408dd-00f3-488e-8800-050d2775bbc7;

cqlsh:ligado> SELECT * FROM playlists;

id                                   |nome           |tags
-------------------------------------+---------------+-------------
----
c4f408dd-00f3-488e-8800-050d2775bbc7|Beatles forever|{'Beatles', '
60s'}

cqlsh:ligado> UPDATE playlists
SET tags = tags + {'classic rock'}
WHERE id = c4f408dd-00f3-488e-8800-050d2775bbc7;

cqlsh:ligado> SELECT * FROM playlists;

id          |nome           |tags
------------+---------------+---------------------------------
c4f408dd-...|Beatles forever|{'60s', 'Beatles', 'classic rock'}
```

7.4 RELACIONANDO ARTISTAS

Agora nossa implementação de playlist está pronta. Nós demos algumas voltas até chegar à implementação final, mas chegamos à seguinte solução:

1. Quando um usuário cria uma playlist, inserimos um registro na tabela `playlist`.

2. Quando o usuário altera a playlist (insere, remove ou troca a ordem de músicas), inserimos uma nova célula na tabela `playlist_versionada`.

3. Após processar todos os registros da lista na `playlist_versionada`, calculamos o estado atual da playlist e salvamos na tabela `playlist_atual`.

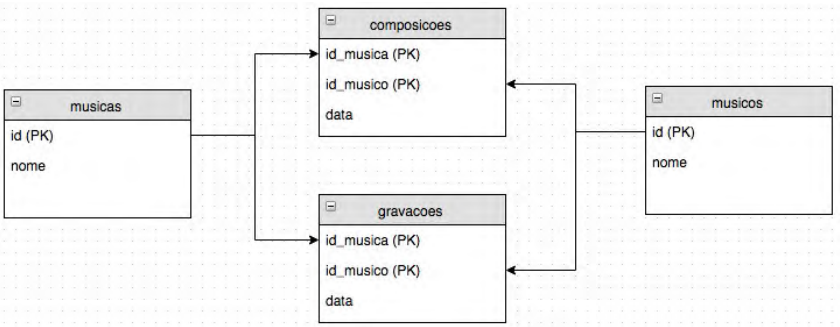
4. Sempre que o cliente adiciona uma tag em uma lista, adicionamos o valor dela no set de tags dentro da tabela `playlist`.

5. Opcionalmente, também adicionamos a tag na tabela `tags`, se for necessário listar todas as tags utilizadas ou calcular tags mais populares.

No nosso sistema `ligado`, já possuímos um cadastro de álbuns e bandas usando **MongoDB**, um sistema de votação usando **Riak** e, finalmente, um sistema de playlists versionadas usando **Cassandra**. A próxima funcionalidade que
trabalharemos será um cadastro de músicos e bandas com o objetivo de mapear as parcerias entre eles e calcular a "distância" entre artistas.

O nosso novo desafio será armazenar as músicas e os músicos responsáveis por compor, gravar e executar essas músicas. Então, depois podemos encontrar relações entre os músicos.

Vamos começar a modelagem do nosso problema pensando em um banco relacional. Vamos precisar de uma tabela `musicas`, uma `musicos`, e também as tabelas `composicoes` e `gravacoes`, que funcionarão como tabelas intermediárias para os relacionamentos *muitos para muitos* entre músicas e músicos.



Talvez vocês não saibam, mas existem alguns compositores que tiveram suas músicas gravadas por muitos artistas, como é o caso de Bob Dylan, que teve suas composições gravadas por Ramones, George Harrison, Jimi Hendrix e vários outros, ou o grande produtor Desmond Child, que escreveu e produziu grandes hits para Ricky Martin, Aerosmith, Kiss e também muitos outros artistas. Logo, vamos usá-los como figuras centrais neste exemplo.

```
mysql> SELECT mo.nome as compositor, ma.nome as musica
       FROM composicoes c
       JOIN musicas ma on ma.id = c.musica_id
       JOIN musicos mo on mo.id = c.musico_id;
+--------------+-------------------------+
| compositor   | musica                  |
+--------------+-------------------------+
| Bob Dylan    | All Along the Watchtower |
| Bob Dylan    | It Ain't Me, Babe       |
| Bob Dylan    | One More Cup of Coffee  |
| Bob Dylan    | If Not For You          |
| Bob Dylan    | My Back Pages           |
| Bob Dylan    | Knockin' on Heavens     |
| Desmond Child | Crazy                  |
| Desmond Child | Livin' la Vida Loca    |
| Desmond Child | Livin' on a Prayer     |
| Desmond Child | You Give Love a Bad Name |
+--------------+-------------------------+

mysql> SELECT mo.nome as interprete, ma.nome as musica
       FROM gravacoes g
       JOIN musicas ma ON ma.id = g.musica_id
       JOIN musicos mo ON mo.id = g.musico_id;
+----------------+-------------------------+
```

```
|   interprete    | musica                   |
+-----------------+--------------------------+
| Jimi Hendrix    | All Along the Watchtower |
| Jack White      | One More Cup of Coffee   |
| Joey Ramone     | My Back Pages            |
| Johnny Cash     | It Ain't Me, Babe        |
| George Harrison | If Not For You           |
| Steve Tyler     | Crazy                    |
| Ricky Martin    | Livin' la Vida Loca      |
| Jon Bon Jovi    | Livin' on a Prayer       |
| Jon Bon Jovi    | You Give Love a Bad Name |
| Jon Bon Jovi    | Knockin' on Heavens      |
+-----------------+--------------------------+
```

Até aqui, nenhum grande desafio. Mas vamos tentar contar a quantidade de músicas que um músico gravou do mesmo compositor.

```
mysql> SELECT mo.nome as interprete,
              com.nome as compositor,
              COUNT(com.id) as total_musicas
       FROM musicos mo
       JOIN gravacoes g ON mo.id = g.musico_id
       JOIN musicas ma ON ma.id = g.musica_id
       JOIN composicoes c ON ma.id = c.musica_id
       JOIN musicos com ON com.id = c.musico_id
       WHERE com.id <> mo.id
       GROUP BY com.id, mo.id
       ORDER BY mo.nome, com.nome;

+-----------------+---------------+---------------+
|   interprete    | compositor    | total_musicas |
+-----------------+---------------+---------------+
| George Harrison | Bob Dylan     |             1 |
| Jack White      | Bob Dylan     |             1 |
| Jimi Hendrix    | Bob Dylan     |             1 |
| Joey Ramone     | Bob Dylan     |             1 |
| Johnny Cash     | Bob Dylan     |             1 |
| Jon Bon Jovi    | Bob Dylan     |             1 |
| Jon Bon Jovi    | Desmond Child |             2 |
| Ricky Martin    | Desmond Child |             1 |
| Steve Tyler     | Desmond Child |             1 |
+-----------------+---------------+---------------+
```

Na última query, podemos encontrar o músico `Jon Bon Jovi` com uma música do `Bob Dylan` e duas do `Desmond Child`.

Podemos ir além e tentar encontrar quem gravou músicas de compositores que um dado músico já gravou. Por exemplo, vamos encontrar quem gravou músicas escritas por compositores que escreveram músicas que o `Jimi Hendrix` gravou.

```
mysql>
SELECT mo.nome as interprete,
       com.nome as compositor,
       COUNT(com.id) as total_musicas
FROM musicos mo
JOIN gravacoes g ON mo.id = g.musico_id
JOIN musicas ma ON ma.id = g.musica_id
JOIN composicoes c ON ma.id = c.musica_id
JOIN musicos com ON com.id = c.musico_id
WHERE mo.id <> 2
AND com.id IN (SELECT DISTINCT in_c.musico_id
               FROM musicos in_mo
               JOIN gravacoes in_g
                           ON in_mo.id = in_g.musico_id
               JOIN musicas in_ma
                           ON in_ma.id = in_g.musica_id
               JOIN composicoes in_c
                           ON in_ma.id = in_c.musica_id
               WHERE in_g.musico_id = 2)
       GROUP BY com.id, mo.id
       ORDER BY total_musicas DESC, mo.nome, com.nome;
+-----------------+------------+---------------+
|   interprete    | compositor | total_musicas |
+-----------------+------------+---------------+
| George Harrison | Bob Dylan  |             1 |
| Jack White      | Bob Dylan  |             1 |
| Joey Ramone     | Bob Dylan  |             1 |
| Johnny Cash     | Bob Dylan  |             1 |
| Jon Bon Jovi    | Bob Dylan  |             1 |
+-----------------+------------+---------------+
```

Executando a mesma query, mas com o id do músicos `Ricky Martin`:

```
+--------------+--------------+---------------+
|  interprete  | compositor   | total_musicas |
+--------------+--------------+---------------+
| Jon Bon Jovi | Desmond Child |             2 |
| Steve Tyler  | Desmond Child |             1 |
+--------------+--------------+---------------+
```

Ou com o id do `Jon Bon Jovi`, que possui músicas de ambos compositores:

```
+-----------------+---------------+---------------+
|   interprete    | compositor    | total_musicas |
+-----------------+---------------+---------------+
| George Harrison | Bob Dylan     |             1 |
| Jack White      | Bob Dylan     |             1 |
| Jimi Hendrix    | Bob Dylan     |             1 |
| Joey Ramone     | Bob Dylan     |             1 |
| Johnny Cash     | Bob Dylan     |             1 |
| Ricky Martin    | Desmond Child |             1 |
| Steve Tyler     | Desmond Child |             1 |
+-----------------+---------------+---------------+
```

Novamente pode parecer um exemplo exagerado, mas esse tipo de query pode ser utilizada para criar as seções de *"Quem comprou este produto também comprou"* em lojas virtuais, além de outras buscas para recomendações básicas ou análise de risco.

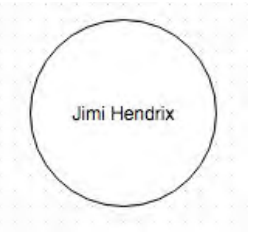
Embora seja possível encontrar o que estamos procurando usando um banco relacional, a query usada é bem complexa. Em um banco de dados com muitos músicos e músicas, ela tende a ficar muito lenta. Qual seria a melhor estrutura de dados para armazenar nossos músicos e músicas de forma que seja fácil buscar e entender o relacionamento entre eles?

Capítulo 8

RELACIONAMENTOS COMPLEXOS COM BANCOS ORIENTADOS A GRAFOS

Quando nosso maior interesse é em como os objetos se relacionam, poucas estruturas de dados são tão efetivas quanto um grafo. Se pedirmos para alguém desenhar em um quadro branco como nossos artistas e músicas se relacionam, é bem provável que acabe com alguns tipos de figuras geométricas, como quadrados ou círculos, interligadas por linhas, que podem ser facilmente traduzidas para um grafo.

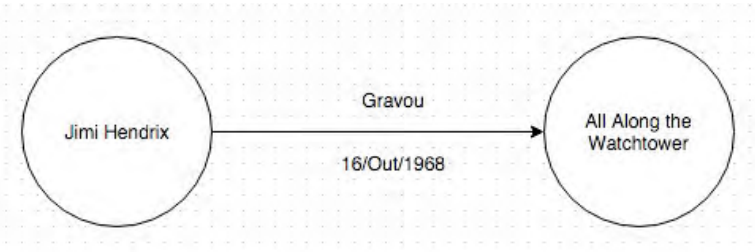
Por exemplo, vamos começar com um círculo com o nome de um artista dentro, Jimi Hendrix.



Por enquanto, nosso grafo possui apenas um elemento representando o artista Jimi Hendrix. Normalmente, cada elemento de um grafo é chamado de nó, ou vértice. Vamos adicionar outro nó no nosso grafo para uma música, *"All Along the Watchtower"*.

Agora podemos desenhar as linhas que unem nós, chamadas de arestas. Quando desenhamos uma aresta, podemos adicionar informações, como qual é o tipo de relacionamento entre os nós, ou alguma outra propriedade específica.

Para finalizar nosso primeiro exemplo então, adicionaremos ao nosso grafo a informação de que o Hendrix **gravou** essa música em **16/Out/1968**.



Dentro da teoria de ciência da computação, existem diversos algoritmos e aplicações para grafos, como árvores de decisão, encontrar o caminho mais curto, centralidade e muitos outros. A ideia por trás dos bancos de dados orientados a grafos é abstrair a complexidade desta estrutura de dados, e nos prover ferramentas para explorar todo o poder dela.

Em um banco orientado a grafos, não existem tabelas, documentos ou qualquer outra estrutura que seja comparável a uma tabela. Neste tipo de banco, tudo são **nós** (vértices) ou **relacionamentos** (arestas). Apesar dos bancos orientados a grafos

serem muito queridos e comentados sob o tópico NoSQL, poucas implementações se destacam. Sem sombra de dúvidas, o mais popular deles é o **Neo4j**, que será o banco que vamos usar.

8.1 INSTALANDO O NEO4J

Apesar de o Neo4j, assim como o Cassandra, também ser executado na máquina virtual do Java, eles disponibilizam instalador para Mac (`.dmg`) e para Windows (`.exe`). Além disso, existe uma versão compactada (`.tar.gz`) para Linux. Todas as versões podem ser baixadas em http://neo4j.com/download.

No Linux, após baixar o arquivo `.tar.gz` , basta descompactar com o seguinte comando:

```
$ tar -xf ARQUIVO-NEO4J.tar.gz
```

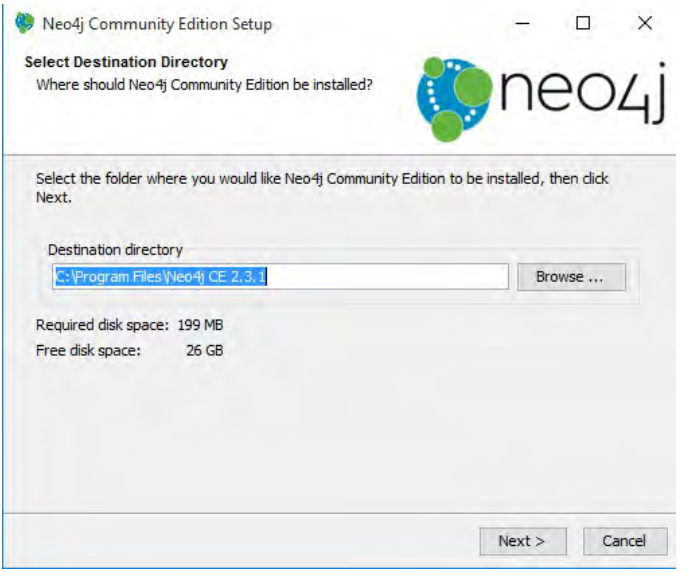
Então, a partir do diretório onde o arquivo foi descompactado, inicie o servidor com `bin/neo4j start` .

```
$ DIRETORIO/bin/neo4j start
```

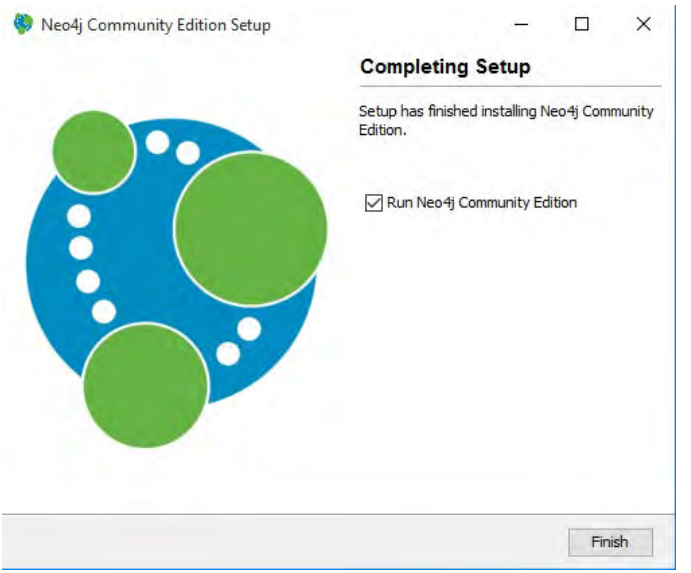
No Mac, apesar da opção de instalação pelo arquivo `.dmg` , a maneira mais rápida e fácil é pelo `homebrew` , bastando instalar o pacote `neo4j` .

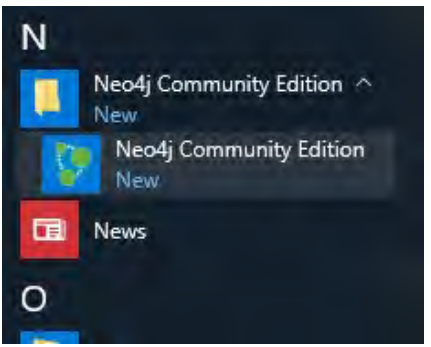
```
$ brew update
$ brew install neo4j
```

No Windows, vamos baixar e executar o instalador `.exe` .

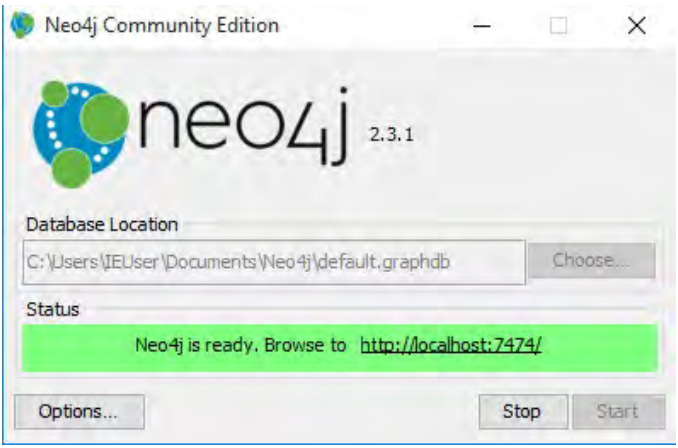


Ao término da instalação, podemos deixar a opção de iniciar o Neo4j, ou se preferir, executá-lo pelo menu *Iniciar*.





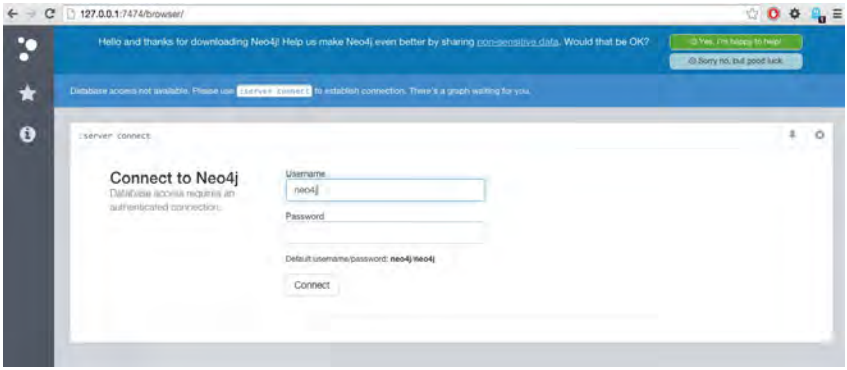
Agora que iniciamos o executável do Neo4j, basta escolher o diretório onde ficarão os dados, e iniciar o servidor usando o botão `start`.



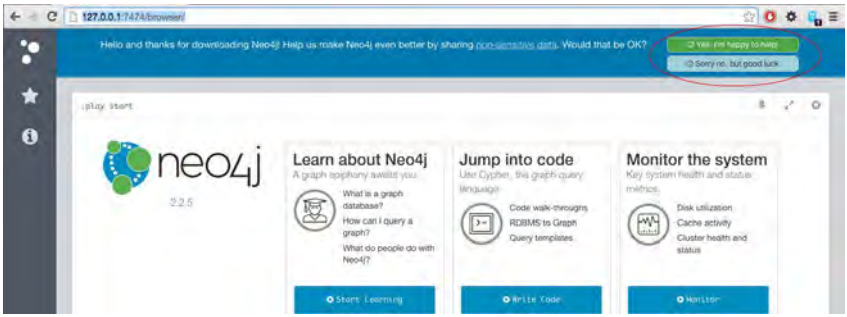
Outra forma de testar o Neo4j é por meio de um docker container. Assim como o Cassandra e o MongoDB, existe uma imagem docker oficial do Neo4j. Para executar um container a partir desta imagem, basta usar o comando `docker run -p 7473:7473 -p 7474:7474 neo4j` .

8.2 CRIANDO NOSSO PRIMEIRO NÓ NO NEO4J

Primeiramente, vamos acessar a interface web do Neo4j pelo seu browser, usando a URL `http://127.0.0.1:7474/` .



Se este for o seu primeiro acesso, tanto o usuário quanto a senha deverão ser `neo4j` . Logo após o primeiro login, você precisará definir uma nova senha. Além disso, você deve escolher se aceita compartilhar seus dados não sensíveis para fazer o Neo4j melhor, clicando em um dos botões do canto superior direito.

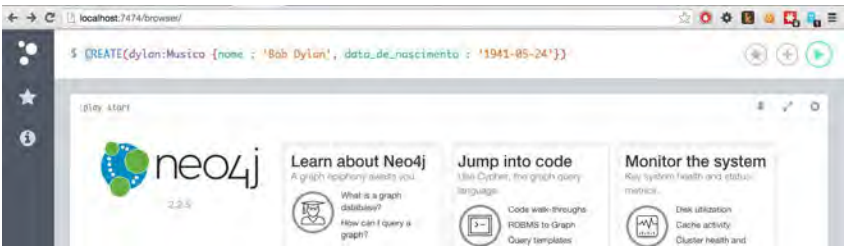


Como vimos há pouco, só existem dois tipos de dados no Neo4j, o nó e o relacionamento, e normalmente armazenamos nossas "entidades" nos nós. Cada nó individualmente pode ser comparado com um documento do MongoDB. Eles podem ter quantos e quais atributos desejarmos.

Cada nó também pode ter um **tipo**. O primeiro nó que vamos criar será a representação do músico "Bob Dylan". Este nó será do **tipo** `Musico`, e possuirá as **propriedades** `nome` e `data_de_nascimento`.

A sintaxe para criar um nó é `CREATE(nome_de_variavel:TipoDoNo {prop1: valor1, prop2: valor2})`. Substituindo com os valores do nosso músico, temos: `CREATE(dylan:Musico {nome : 'Bob Dylan', data_de_nascimento : '1941-05-24'})`.

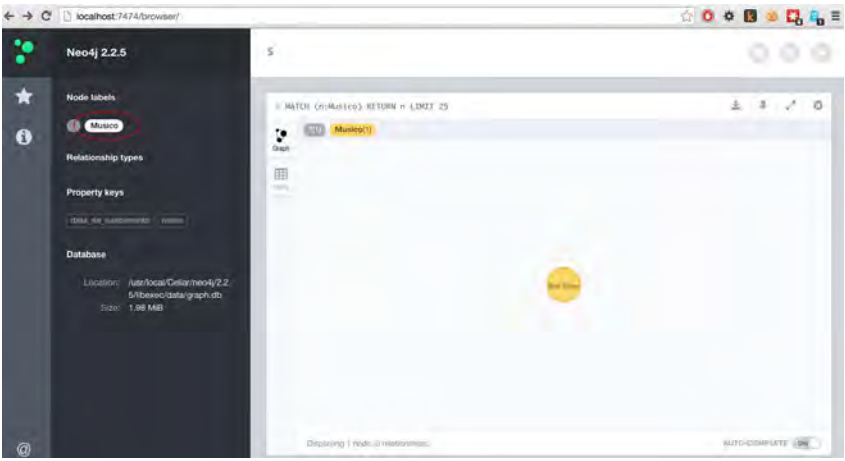
Vamos voltar para a interface web no seu browser e executar o comando para criamos nosso primeiro nó.



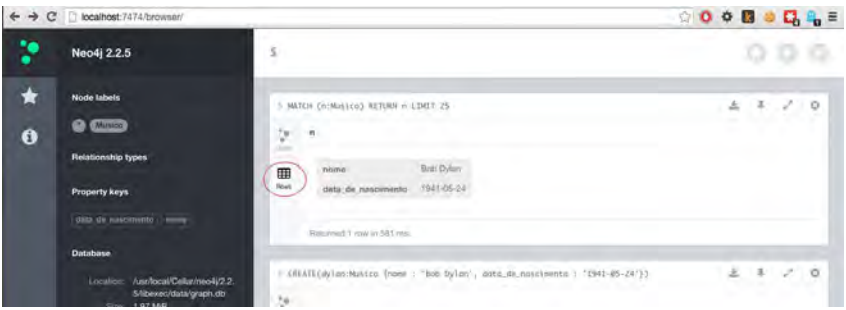

```
$ CREATE(dylan:Musico {nome : 'Bob Dylan',
                       data_de_nascimento : '1941-05-24'})
Added 1 label, created 1 node, set 2 properties,
 statement executed in 671 ms.
```

Agora vamos abrir o menu lateral e visualizar os dados que temos indexados no servidor. Para isso, basta clicar no ícone das 3 bolinhas no canto esquerdo superior.

Nesta área, é possível visualizar todos os tipos de nós (labels), os tipos de relacionamentos, propriedades e algumas outras informações sobre o servidor. Vamos clicar no label `Musico` para visualizarmos todos os nós do tipo `Musico` que existem no banco.



Além desta visualização do grafo, também podemos ver o resultado das nossas queries no formato de tabela. Para isso, basta clicar no botão com ícone de tabela escrito `Rows` .



8.3 CRIANDO RELACIONAMENTOS

Agora que já temos um nó para o músico Bob Dylan, precisamos criar um nó para uma de suas músicas, e então definir o

relacionamento entre eles para começarmos a dar forma para o nosso grafo. Para criar a música, usaremos a mesma sintaxe que utilizamos para criar o músico, mas usando o tipo e as propriedades adequadas ao novo nó.

Já para criar um relacionamento, dependemos de referências para os nós que serão unidos. A maneira mais fácil de conseguir uma referência para um nó é através da variável que definimos quando criamos um nó. Neste próximo passo, vamos criar dois nós e um relacionamento, um nó para o música *"All Along the Watchtower"*, um nó para o músico Jimi Hendrix e o relacionamento entre eles.

Assim como os nós, os relacionamentos também possuem um tipo. Neste exemplo que estamos criando agora, o músico **gravou** a música, então vamos usar este nome. Além do tipo, um relacionamento também pode possuir propriedades, e uma outra característica muita importante é a **direção**. Existem relacionamentos como o `gravou`, nos quais posso afirmar que o músico `hendrix GRAVOU all along`, mas não faz sentido dizer que a música `all along GRAVOU hendrix`.

A sintaxe para criar um relacionamento é `CREATE (no1)-[:TIPO_RELACIONAMENTO]->(no2)`. Reparem que, entre um dos nós e o relacionamento, usamos `-`, enquanto no outro lado do relacionamento usamos `->`, sendo por meio deste detalhe que definimos a direção do relacionamento.

Para criar o relacionamento que desejamos, vamos usar `CREATE (hendrix)-[:GRAVOU]->(all_along)`. A única coisa que precisaremos antes é criar os nós e definir as variáveis para `hendrix` e `all_along`.

```
$ CREATE(all_along:Musica {nome: "All Along the Watchtower"})
  CREATE(hendrix:Musico {nome: "Jimi Hendrix"})
  CREATE (hendrix)-[:GRAVOU]->(all_along)
```

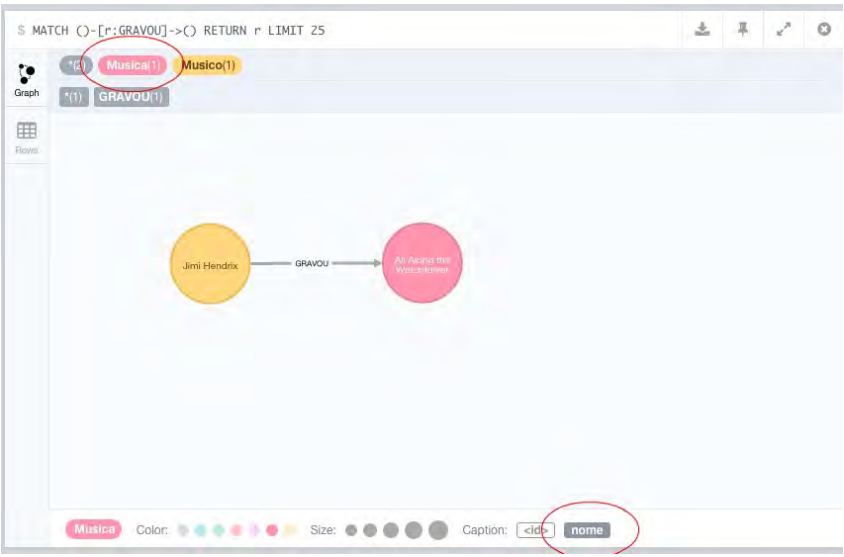
```
Added 2 labels, created 2 nodes, set 2 properties,
created 1 relationship, statement executed in 156 ms.
```

Assim como fizemos após inserir o músico, podemos novamente abrir o menu lateral e, desta vez, além de `Musico`, também encontraremos `Musica` em `Node labels`, e `GRAVOU` em `Relationship types`. Vamos filtrar pelo tipo de relacionamento clicando em `GRAVOU`.

Utilizando esse filtro, além do relacionamento em si, também podemos ver os nós que ele une, nos dando uma visão mais clara de qual músico gravou qual música. Para melhorar ainda mais a visualização, podemos customizar a cor, o tamanho e o atributo que será exibido dentro de cada nó.

Clicando no nome do tipo de nó ou do tipo de relacionamento, no canto esquerdo superior da área de exibição, um menu de customização será exibido na parte inferior desta área. Para facilitar o entendimento dos exemplos, recomendo alterar o nó `Musica` selecionando o atributo `nome` em `Caption` no menu de customização.

Além desta mudança, aproveite a funcionalidade e altere os tamanhos e cores dos nós e relacionamentos da maneira que achar mais fácil para visualizar o grafo.



8.4 FAZENDO BUSCAS NO NEO4J

Nas versões mais atuais do Neo4j, o **cypher** foi introduzido como *query language* oficial do banco de dados. Esta linguagem pode lembrar o tradicional SQL, embora também possua características de uma linguagem imperativa.

A principal palavra chave (`keyword`) para uma query usando o cypher é o `MATCH` . Este comando é usado para definir a estrutura do grafo que estamos buscando. Além do `MATCH` , a outra parte obrigatória em uma query de busca é o `RETURN` , que é usado para definir o que nossa query retornará.

Tentando fazer uma analogia com o SQL, podemos dizer que o `RETURN` é equivalente ao `SELECT` , enquanto o `MATCH` faz o papel do `FROM` .

Vamos começar listando o nome de todos os artistas do banco de dados. Em um banco relacional, usaríamos a seguinte query SQL:

```
SELECT m.nome
```

```
FROM musicos m
```

O equivalente desta query usando o cypher é:

```
MATCH (m:Musico)
RETURN m.nome
```

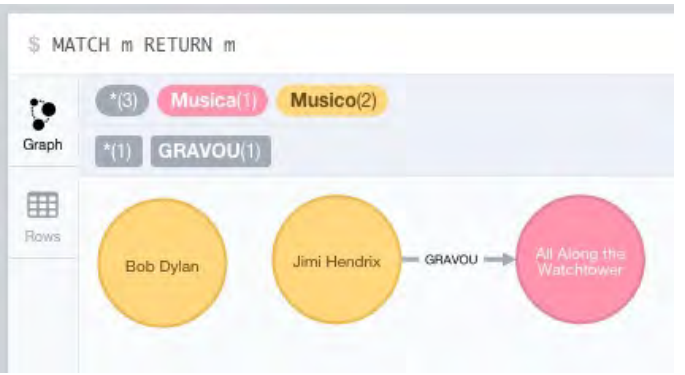
Neste exemplo, `MATCH (m:Musico)` define que estamos buscando todos os nós do tipo `Musico` e atribuindo para a variável `m`, semelhante com `FROM musicos m`. Depois, no `RETURN m.nome`, selecionamos apenas o atributo `nome` dos nós `m` que se encaixam nas restrições do `MATCH`. Ou seja, neste exemplo, recuperamos os nomes de todos os músicos.

Uma das grandes vantagens de utilizar um banco orientado a grafo é que, além de retornar atributos, também podemos retornar nós inteiros, equivalente a retornar um registro todo de uma tabela relacional. Com isso, aproveitamos a visualização do grafo.

```
MATCH (m:Musico)
RETURN m
```

Outra diferença ao buscar usando o cypher é que não somos obrigados a filtrar por um tipo de nó, o que torna possível carregar o grafo todo.

```
MATCH m
RETURN m
```



Além do `MATCH` e do `RETURN`, outra palavra-chave que devemos conhecer é o `WHERE`, que, assim como no SQL, é utilizado para aplicar filtros na nossa query. Por exemplo, para buscar o nó referente ao músico Bob Dylan, podemos usar o `MATCH (m:Musico)` acompanhado de um `WHERE`, para filtrar entre os músicos aquele que possui o atributo `nome` igual a `Bod Dylan`. Para isso, usamos `WHERE m.nome = 'Bob Dylan'`.

```
MATCH (m:Musico)
WHERE m.nome = 'Bob Dylan'
RETURN m
```

Uma outra forma de filtrar nós por um atributo é dentro do `MATCH`, utilizando `{atributo : valor }`. Por exemplo, uma busca equivalente à anterior para buscar nós do tipo `Musico` com o atributo `nome` igual a `Bob Dylan` seria: `MATCH (m:Musico {nome: 'Bob Dylan'})`.

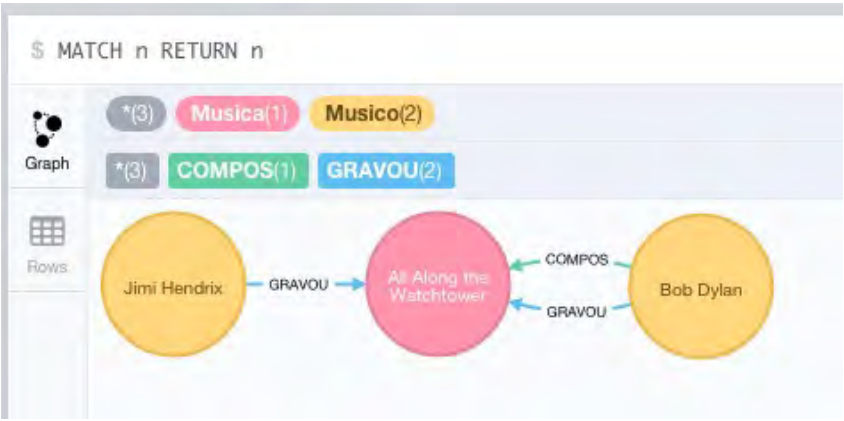
```
MATCH (m:Musico { nome : 'Bob Dylan' })
RETURN m
```

Agora que já sabemos como filtrar nossos nós, podemos criar o relacionamento entre o músico `Bob Dulan` e a música `All Along the Watchtower`. A criação do relacionamento é igual a criação que já fizemos, `CREATE (no1)-[:RELACIONAMENTO]->(no2)`, mas vamos definir o valor das variável usando o que aprendemos de

buscas.

```
MATCH (bob:Musico { nome : 'Bob Dylan' }),
      (all_along:Musica { nome : 'All Along the Watchtower'})
CREATE (bob)-[:GRAVOU]->(all_along)
CREATE (bob)-[:COMPOS]->(all_along)
```

Podemos visualizar o grafo completo novamente usando `MATCH n RETURN n`.



8.5 EXPLORANDO MAIS OPÇÕES COM O MATCH

Apesar de termos utilizado o `MATCH` buscando apenas por um nó até agora, ele deve ser usado para descrever a estrutura do grafo que estamos buscando. Um detalhe importante é que ele possui uma sintaxe diferente para descrever nós e relacionamentos.

Até agora, usamos apenas o nome da variável no caso do `MATCH n`, ou declaramos a variável dentro de parênteses, como em `MATCH (m:Musico)`, para fazer referência a um nó. Quando queremos descrever um relacionamento, devemos utilizar colchetes, como `[relacionamento]`.

Podemos usar o `MATCH` para buscar em nosso grafo pela

estrutura `MUSICO` ligado a uma `MUSICA`. Para descrever esta estrutura, usaremos como base a estrutura `()-[]-()`. Apesar de isso parecer não fazer sentido, com esses argumentos estamos buscando por um nó `()` com um relacionamento `[]` com outro nó `()`.

Para executarmos a query, vamos precisar um `RETURN` que depende de ter, pelo menos, uma variável. Então, podemos modificar um pouco nosso `MATCH` adicionando um nome a um dos nós, e então executar nosso filtro.

```
MATCH (n)-[]-()
RETURN n
```

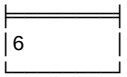
Usando a visualização de gráfico, o resultado vai ser idêntico. Porém, se abrirmos a visualização de tabela e compararmos com o simples `MATCH n RETURN n`, poderemos ver que o resultado foi um pouco diferente.

Vou introduzir o agregador `count` para facilitar a comparação entre as duas queries. Por enquanto, continuamos seguindo a mesma regra na hora de traduzir as queries SQL para cypher. Como usamos o `count` dentro da área do `SELECT`, ele vai para dentro do `RETURN` no Neo4j.

```
MATCH (n)
RETURN COUNT(n)
╒═══════╕
|COUNT(n)|
╞═══════╡
|3      |
└───────┘
```

Entretanto, quando usamos a match `(n)-[]-()`, o resultado do `count` é bem diferente:

```
MATCH (n)-[]-()
RETURN COUNT(n)
╒═══════╕
|COUNT(n)|
```



Aqui começamos a ver o poder de um banco orientado a grafo e como o cypher, apesar de simples, é muito poderoso, às vezes podendo até nos pregar peças.

Na primeira query `MATCH n`, vamos buscar por nós independentemente de seus relacionamentos. Como temos três nós no nosso banco por enquanto, temos como retorno todos eles; por isso o `count` retorna o número 3. Na segunda query `(n)-[]-()`, estamos buscando pelo padrão **NO**-relacionado-**NO**. Se olharmos bem nosso grafo, é possível identificar esse padrão nas seguintes situações:

```
Jimi Hendrix - GRAVOU - All Along the Watchtower
Bob Dylan - GRAVOU - All Along the Watchtower
Bob Dylan - COMPOS - All Along the Watchtower
All Along the Watchtower - GRAVOU - Jimi Hendrix
All Along the Watchtower - GRAVOU - Bob Dylan
All Along the Watchtower - COMPOS - Bob Dylan
```

Por isso o `count` retornou o número 6 usando a estrutura mais complexa. Uma outra funcionalidade fundamental do `MATCH` é filtrar pela direção do relacionamento. Nós podemos criar um relacionamento sem direção. Mas, quando criamos os nossos, usamos a sintaxe `CREATE (bob)-[:GRAVOU]->(all_along)`, o que significa que a direção do relacionamento vai do músico para a música.

Com uma pequena mudança na nossa query, podemos fazer o `count` retornar apenas três elementos novamente. Assim como no `create`, adicionar o `>` ou `<` define a direção que estamos buscando `(n)-[]->()`.

```
MATCH (n)-[]->()
RETURN n

| Jimi Hendrix - GRAVOU - All Along the Watchtower |
```

```
| Bob Dylan - GRAVOU - All Along the Watchtower    |
| Bob Dylan - COMPOS - All Along the Watchtower    |
```

```
MATCH (n)<-[]-()
RETURN n
```

```
| All Along the Watchtower - GRAVOU - Jimi Hendrix |
| All Along the Watchtower - GRAVOU - Bob Dylan    |
| All Along the Watchtower - COMPOS - Bob Dylan    |
```

Como vocês podem notar, inverter a direção da "seta" mudou totalmente o resultado da nossa busca. Embora estamos mostrando o relacionamento completo como resultado das nossas buscas, o resultado que você deve estar vendo no seu computador deve ser apenas o nó do músico na busca com o relacionamento `saindo` do nó `n`, ou só o nó da música no segundo caso com o relacionamento `chegando` no nó `n`.

Para ter um resultado mais parecido com o exibido, aqui precisamos retornar não só o nó `n`, mas também o relacionamento e o nó adjacente. Para isso, é necessário darmos nomes a eles; `(n)-[r]->(n2)`, por exemplo.

```
MATCH (n)-[r]->(n2)
RETURN n, r, n2
```

```
╒═══════════════════════════╤═══╤═══════════════════════╕
│n                          │r  │n2                     │
╞═══════════════════════════╪═══╪═══════════════════════╡
│{nome: Bob Dylan, data_de  │{} │{nome: All Along the Watc│
│_nascimento: 1941-05-24}   │   │htower}                │
├───────────────────────────┼───┼───────────────────────┤
│{nome: Bob Dylan, data_de  │{} │{nome: All Along the Watc│
│_nascimento: 1941-05-24}   │   │htower}                │
├───────────────────────────┼───┼───────────────────────┤
│{nome: Jimi Hendrix}       │{} │{nome: All Along the Watc│
│                           │   │htower}                │
└───────────────────────────┴───┴───────────────────────┘
```

No Neo4j, assim como os nós, os relacionamentos também podem ter atributos, mas nenhum dos relacionamentos que criamos possui informação alguma. Por isso, quando retornamos apenas `r`, temos apenas um `empty` como resultado. Podemos resolver esse

problema usando a função `type`, e retornar o tipo do relacionamento que passamos como argumento para ela.

```
MATCH (n)-[r]->(n2)
RETURN n, type(r), n2
```

```
╒═══════════════════════════╤═══════╤═══════════════════════╕
│n                          │type(r)│n2                     │
╞═══════════════════════════╪═══════╪═══════════════════════╡
│{nome: Bob Dylan, data_de  │COMPOS │{nome: All Along the Watc│
│_nascimento: 1941-05-24}   │       │htower}                │
├───────────────────────────┼───────┼───────────────────────┤
│{nome: Bob Dylan, data_de  │GRAVOU │{nome: All Along the Watc│
│_nascimento: 1941-05-24}   │       │htower}                │
├───────────────────────────┼───────┼───────────────────────┤
│{nome: Jimi Hendrix}       │GRAVOU │{nome: All Along the Watc│
│                           │       │htower}                │
└───────────────────────────┴───────┴───────────────────────┘
```

Mais um pequeno detalhe muito importante na nossa query é que declaramos os nós com nomes diferentes, `n` e `n2`. Se usarmos `(n)-[r]-(n)`, estamos dizendo que o relacionamento deve sair do nó `n` e ir de volta para o mesmo nó `n`.

```
MATCH (n)-[r]->(n)
RETURN n, type(r)

(no rows)
```

Nós podemos misturar tudo que vimos sobre o `MATCH` até agora e adicionar o filtro pelo tipo de nó junto com padrão. Vamos declarar explicitamente o tipo dos nós com `:Musico` ou `:Musica`, e deixar nossos filtros ainda mais específicos.

Neste próximo exemplo, buscaremos os relacionamentos de músicas para músicos:

```
MATCH (n:Musica)-[r]->(n2:Musico)
RETURN type(r)

(no rows)
```

Como todos os nossos relacionamentos são de músicos para músicas, nossa query retorna uma tabela vazia. Mas se invertermos

a direção do relacionamento, podemos encontrar nossas informações.

```
MATCH (n:Musica)<-[r]-(n2:Musico)
RETURN type(r)
```

```
╒═══════╕
│type(r)│
╞═══════╡
│COMPOS │
├───────┤
│GRAVOU │
├───────┤
│GRAVOU │
└───────┘
```

Além do filtro pelo tipo do nó, que já havíamos usado, também podemos filtrar pelo tipo do relacionamento usando a mesma sintaxe. Entretanto, no local destinado aos relacionamentos, por exemplo, para filtrar pelos relacionamentos do tipo `COMPOS` , usaremos `[r:COMPOS]` . Em um exemplo mais prático, vamos comparar os músicos que já gravaram músicas *versus* os músicos que já compuseram músicas.

```
MATCH (n:Musico)-[r:COMPOS]-(n2:Musica)
RETURN n
```

```
╒═══════════════════════════════════╕
│n                                  │
╞═══════════════════════════════════╡
│{nome: Bob Dylan, data_de_nasc│
│imento: 1941-05-24}                │
└───────────────────────────────────┘
```

```
MATCH (n:Musico)-[r:GRAVOU]-(n2:Musica)
RETURN n
```

```
╒═══════════════════════════════════╕
│n                                  │
╞═══════════════════════════════════╡
│{nome: Bob Dylan, data_de_nasc│
│imento: 1941-05-24}                │
├───────────────────────────────────┤
│{nome: Jimi Hendrix}               │
└───────────────────────────────────┘
```

8.6 APAGANDO E EDITANDO NÓS E

RELACIONAMENTOS

Antes de entrar nas buscas mais avançadas e descobrirmos como encontraremos quem gravou músicas escritas por compositores que escreveram músicas que um determinado músico gravou, vamos terminar as operações básicas de escrita vendo como deletar e editar nós e relacionamentos no Neo4j.

Por enquanto, temos dois nós do tipo `Musico`, mas apenas um deles possui um atributo com a data de nascimento. Então, vamos alterar o nó que representa o `"Jimi Hendrix"` e adicionar a data de seu nascimento, que é `27/11/1942`.

Semelhante com o SQL, o cypher possui uma cláusula `SET` que é usada para atribuir um novo valor para um atributo. Assim como no SQL, o `SET` também afeta todos os registros que retornarem do nosso filtro, a diferença é que no cypher usamos os já conhecidos `MATCH` e `WHERE` para filtrar os registros que serão atualizados.

Em vez de usar uma cláusula especial como o `UPDATE` do SQL, no cypher fazemos uma query como outra qualquer, e se adicionarmos o `SET`, os registros serão modificados. Para adicionar a data de nascimento ao músico `Jimi Hendrix`, vamos precisar que uma query que filtre apenas este nó como `MATCH (hendrix:Musico { nome : 'Jimi Hendrix' })`. Então, adicionamos o `SET` com o atributo e valor que desejamos, como: `SET hendrix.data_de_nascimento = '1942-11-27'`.

```
MATCH (hendrix:Musico { nome : 'Jimi Hendrix' })
SET hendrix.data_de_nascimento = '1942-11-27'
RETURN hendrix
```

```
╒═══════════════════════════════════╕
│hendrix                            │
╞═══════════════════════════════════╡
│{nome: Jimi Hendrix,               │
│data_de_nascimento: 1942-11-27}    │
└───────────────────────────────────┘
```

Se quisermos remover um atributo de um nó, basta usar o `SET` atribuindo o valor `null` para o atributo que queremos remover.

```
MATCH (hendrix:Musico { nome : 'Jimi Hendrix' })
SET hendrix.data_de_nascimento = null
RETURN hendrix
```

```
╒═══════════════════════╕
|hendrix                |
╞═══════════════════════╡
|{nome: Jimi Hendrix}|
└───────────────────────┘
```

Mais para a frente, recriaremos os nós dos músicos e das músicas. Então, vamos apagar os nós que já possuímos, começando pelo nó do músico Jimi Hendrix. No cypher, temos a cláusula `DELETE` para apagar nós e relacionamentos. Assim como o `SET`, o `delete` remove todos os dados que foram filtrados pelo `MATCH` e `WHERE`.

Para apagar o nós do Hendrix, vamos utilizar o mesmo `MATCH` do exemplo anterior, adicionando `DELETE hendrix` para apagar o nó:

```
MATCH (hendrix:Musico { nome : 'Jimi Hendrix' })
DELETE hendrix

org.neo4j.kernel.api.exceptions.TransactionFailureException:
 Node record Node[2,used=false,rel=0,prop=-1,
 labels=Inline(0x0:[]),light]
 still has relationships
```

Apesar de a sintaxe estar correta, tivemos um erro ao tentar apagar o nó. A última frase da mensagem de erro deixa bem claro a razão pela qual nosso comando falhou: `Node ... still has relationships`. Ou seja, o nó ainda possui relacionamentos.

Como no Neo4j os relacionamentos obrigatoriamente ligam dois nós, se pudéssemos apagar um nó que possui um relacionamento, este ficaria inconsistente. Então, se quisermos apagar o nó do Hendrix, precisaremos apagar todos os seus

relacionamentos antes.

Vamos adicionar no `MATCH` uma referência para os relacionamentos do nosso nó de qualquer tipo, para qualquer direção que ligue para qualquer outro nó: `MATCH (hendrix:Musico { nome : 'Jimi Hendrix' })-[rel]-()`.

```
MATCH (hendrix:Musico { nome : 'Jimi Hendrix' })-[rel]-()
RETURN hendrix, type(rel)
```

hendrix	type(rel)
{nome: Jimi Hendrix}	GRAVOU

O `DELETE` funciona tanto para nós quanto para relacionamentos. Logo, podemos aproveitar a query anterior e adicionar `DELETE rel` para apagar todos os relacionamentos do Jimi Hendrix.

```
MATCH (hendrix:Musico { nome : 'Jimi Hendrix' })-[rel]-()
DELETE rel

Deleted 1 relationship, statement executed in 171 ms.
```

Agora que nosso nó não possui mais relacionamentos, podemos tentar apagá-lo novamente.

```
MATCH (hendrix:Musico { nome : 'Jimi Hendrix' })
DELETE hendrix

Deleted 1 node, statement executed in 90 ms.
```

Embora tenhamos feito duas operações de `DELETE` separadas, podemos apagar os nós e os relacionamentos ao mesmo tempo. Vamos agora apagar o nó referente ao músico Bob Dylan e todos os seus relacionamentos. Usaremos o `MATCH` igual ao que fizemos para encontrar o nó Jimi Hendrix e seus relacionamentos, apenas trocando o nome do músico: `MATCH (dylan:Musico { nome : 'Bob Dylan' })-[rel]-()`.

No `DELETE`, em vez de usar apenas `DELETE rel` e depois `DELETE dylan`, podemos usar `DELETE rel, dylan`.

```
MATCH (dylan:Musico { nome : 'Bob Dylan' })-[rel]-()
DELETE rel, dylan

Deleted 1 node, deleted 2 relationships,
statement executed in 74 ms.
```

Agora que apagamos os nós que representam os músicos `Bob Dylan` e `Jimi Hendrix` juntamente com seus relacionamentos, o único nó restante em nosso banco é o da música `All Along the Watchtower`. Podemos fazer uma query específica como fizemos para apagar os músicos, ou fazer uma genérica para apagar todos os nós que acabará a apagando também.

Vamos aproveitar que estamos apagando o banco todo e explorar essa possibilidade. Para apagar todos os nós, podemos tentar executar `MATCH (n) DELETE n`, que funcionaria para o nosso estado atual.

Porém, não podemos esquecer que o Neo4j não nos permite apagar nós que possuem relacionamentos. Então, faremos como no último exemplo, e apagaremos junto os relacionamentos. Para isso, podemos tentar o seguinte: `MATCH (n)-[rel]-() DELETE rel, n`.

```
MATCH (n)-[rel]-()
DELETE rel, n

(no changes, no rows)
```

Como vocês podem ver, nenhum registro foi apagado. Novamente, uma situação em que o cypher nos prega uma peça. Estamos buscando um nó que possui um relacionamento, mas nosso nó remanescente não possui relacionamentos, logo, ele não casa com o `MATCH` e não é apagado.

Se tirarmos o filtro dos relacionamentos, não conseguimos

apagar os nós com relacionamento; e se mantivermos o filtro, não conseguimos apagar os nós que não possuem relacionamento. Para essas situações, temos uma outra palavra-chave que vai nos ajudar, a `OPTIONAL`.

A estrutura que estamos buscando é um nó que pode ou não ter relacionamentos. Então, usamos o `MATCH (n)`, que nos retorna todos os nós. Além disso, vamos utilizar o `OPTIONAL MATCH (n)-[rel]-()` para termos uma referência para os relacionamentos caso eles existam. Entretanto, também vamos ter referências para os nós que não possuem relacionamentos, pois o `MATCH` para relacionamentos é opcional.

```
MATCH (n)
OPTIONAL MATCH (n)-[rel]-()
DELETE rel, n

Deleted 1 node, statement executed in 89 ms.
```

Após executar esse comando, limparemos todos os nós e relacionamentos. Os próximos passos agora são popular nosso grafo com alguns nós de músicos e músicas para irmos mais a fundo nas buscas, e explorarmos mais do poder dos grafos.

Capítulo 9

MAIS SOBRE BANCOS ORIENTADOS A GRAFOS

Agora nós já conhecemos a estrutura básica de um banco orientado a grafos, e como o cypher, a *query language* do Neo4j, funciona. Além disso, limpamos todos os nós e relacionamentos criados, e estamos com o banco praticamente como novo.

Para começarmos os exercícios mais complexos como o já citado *"quem gravou músicas escritas por compositores que escreveram músicas que um determinado músico gravou"*, vamos precisar criar novos nós.

9.1 IMPORTANDO DADOS PARA O NEO4J

Desta vez, vamos usar a funcionalidade de importação de dados do Neo4j. Criaremos arquivos `csv` (*Comma Separated Values*), um formato de arquivo bem comum para armazenar e transferir dados entre plataformas, em que os dados são escritos como texto puro com o valores separados por vírgulas ou algum caractere especial.

Para a importação dos dados, vamos criar dois arquivos bem simples: um para as composições, sendo que cada linha possuirá o nome da música e o nome do compositor; e outro para as gravações, que também possuirá apenas o nome da música e do artista que gravou.

Vamos começar com os seguintes dados para o arquivo das composições:

```
Bob Dylan,All Along the Watchtower
Bob Dylan,It Ain't Me, Babe
```

Quando estivermos importando os dados, vamos precisar lidar com nomes de músicas e de músicos repetidos. Independente de como vamos efetivamente carregar os dados, temos de pensar em como vamos inseri-los para evitar a criação de nós duplicados para o mesmo músico.

Nós já conhecemos o `CREATE` para criar um novo nó e o `MATCH` para buscar nós. O que queremos é algo como `MATCH (compositor:Musico {nome: NOME_NO_ARQUIVO})` . Se existir, criamos o relacionamento; caso contrário, criamos primeiro o nó para o músico e depois o relacionamento.

No mundo relacional, existem algumas soluções para esse tipo de problema, dependendo do banco de dados que esteja usando, podemos usar `INSERT IGNORE` , `INSERT ... ON DUPLICATE KEY UPDATE` , `INSERT ... WHERE NOT EXISTS` , ou alguma outra solução específica que permita tomar ações diferentes caso já exista um registro para aquelas informações ou não.

O Neo4j tem a própria solução para esse problema, o `MERGE` . Sua sintaxe é praticamente idêntica à do `MATCH` , a diferença é que se não encontrar um nó que case com o filtro, um novo nó será criado.

```
MERGE (n1:Musico {nome: "Bob Dylan"})
MERGE (n2:Musico {nome: "Bob Dylan"})
RETURN id(n1), id(n2)
```

id(n1)	id(n2)
280	280

```
Added 1 label, created 1 node, set 1 property, statement executed
in 25 ms.
```

Reparem que apesar de dois comandos iguais, apenas um nó foi criado, mas duas variáveis foram declaradas, `n1` e `n2`, ambas apontando para o mesmo nó. O primeiro `MERGE` funcionou como um `CREATE` e adicionou um novo nó ao banco de dados, enquanto o segundo funcionou como um `MATCH` e carregou o nó criado anteriormente.

Usando o `MERGE`, poderemos importar o arquivo `csv` sem nos preocuparmos se é a primeira ocorrência ou não do músico ou música, pois sempre vamos ter uma variável para o músico e para música para poder criar o relacionamento. Um outro diferencial do `MERGE` é que ele também pode ser usado para a criação (ou não) dos relacionamentos.

Ao usar, por exemplo, `MERGE (compositor)-[COMPOS]->(musica)`, caso já exista um relacionamento do tipo `COMPOS` entre o compositor e a música, nada acontecerá. Mas se ainda não existe um relacionamento deste tipo entre os nós, um novo será criado.

Agora que já temos uma estratégia para criar/carregar os nós e os relacionamentos baseado nos valores encontrados no arquivo, então precisamos de uma maneira de ler o conteúdo do arquivo. Usando o cypher, podemos utilizar o `LOAD CSV WITH HEADERS` para carregar arquivos do tipo `csv`.

Para poder importar o arquivo, precisamos passar o endereço dele, que pode ser informado pelo `FROM` juntamente com uma URL para baixar o arquivo, ou `file:/ENDERECO/DO/arquivo.csv`. Usando o `LOAD CSV` junto com `FROM`, o Neo4j já é capaz de abrir e ler o arquivo. O que falta para nos permitir importar os dados é iterar pelo arquivo todo atribuindo os valores de cada linha para alguma variável.

O LOAD já vai iterar o arquivo todo e atribuirá o valor de cada linha na variável que declararmos usando `AS nomeDaVariavel` no final do comando. Ou seja, para ler e imprimir todo o conteúdo de um arquivo `csv` usando o Neo4j, podemos usar o seguinte comando:

```
LOAD CSV WITH HEADERS
FROM "file:/ENDERECO/DO/arquivo.csv"
AS linha
RETURN linha
```

Para fazermos os testes de importação de arquivos, vamos criar os dois arquivos, `gravacoes.csv` e `composicoes.csv` , com o seguinte conteúdo, respectivamente.

O primeiro arquivo será:

```
interprete,musica
"Jimi Hendrix","All Along the Watchtower"
"Johnny Cash","It Ain't Me, Babe"
"Jack White","One More Cup of Coffee"
"George Harrison","If Not For You"
"Joey Ramone","My Back Pages"
"Jon Bon Jovi","Knockin' on Heavens"
"Steve Tyler","Crazy"
"Ricky Martin","Livin' la Vida Loca"
"Jon Bon Jovi","Livin' on a Prayer"
"Jon Bon Jovi","You Give Love a Bad Name"
```

E o segundo será:

```
compositor,musica
"Bob Dylan","All Along the Watchtower"
"Bob Dylan","It Ain't Me, Babe"
"Bob Dylan","One More Cup of Coffee"
"Bob Dylan","If Not For You"
"Bob Dylan","My Back Pages"
"Bob Dylan","Knockin' on Heavens"
"Desmond Child","Crazy"
"Desmond Child","Livin' la Vida Loca"
"Desmond Child","Livin' on a Prayer"
"Desmond Child","You Give Love a Bad Name"
```

Vejamos o resultado do `LOAD CSV` usando o arquivo

composicoes.csv :

```
LOAD CSV WITH HEADERS
FROM "file:/CAMINHO/COMPLETO/composicoes.csv"
AS linha
RETURN linha
```

```
╒═══════════════════════════════════════════════════════════════════╕
│linha                                                              │
╞═══════════════════════════════════════════════════════════════════╡
│{compositor: Bob Dylan, musica: All Along the Watchtower}          │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Bob Dylan, musica: It Ain't Me, Babe}                 │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Bob Dylan, musica: One More Cup of Coffee}            │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Bob Dylan, musica: If Not For You}                    │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Bob Dylan, musica: My Back Pages}                     │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Bob Dylan, musica: Knockin' on Heavens}               │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Desmond Child, musica: Crazy}                         │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Desmond Child, musica: Livin' la Vida Loca}           │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Desmond Child, musica: Livin' on a Prayer}            │
├───────────────────────────────────────────────────────────────────┤
│{compositor: Desmond Child, musica: You Give Love a Bad Name}      │
└───────────────────────────────────────────────────────────────────┘
```

Note que a variável `linha` foi populada com o valor de cada uma das linhas do arquivo, e a iteração dentro do arquivo aconteceu automaticamente. Além disso, a primeira linha é usada como cabeçalho, então em vez de considerá-la como uma linha de dado do arquivo, ela é usada para definir o nome dos atributos. Por exemplo, no código anterior, poderíamos ter usado `linha.compositor` para acessar apenas o nome do música que compôs a música.

Para finalizar a importação dos arquivos, vamos usar o `LOAD CSV` e, para cada linha, vamos usar o `MERGE` buscando ou criando um músico com o nome do compositor.
Depois usaremos novamente o `MERGE` para carregar ou criar um

nó para música e, por último, vamos utilizar mais uma vez o `MERGE`, desta vez para definir o relacionamento do tipo `COMPOS` entre o músico e a música.

```
LOAD CSV WITH HEADERS
FROM "file:/CAMINHO/ABSOLUTO/ATE/composicoes.csv"
AS linhaCsv
MERGE (compositor:Musico {nome: linhaCsv.compositor})
MERGE (musica:Musica {nome: linhaCsv.musica})
MERGE (compositor)-[:COMPOS]->(musica)

Added 11 labels, created 11 nodes, set 11 properties,
created 10 relationships, statement executed in 191 ms.
```

Agora que terminamos de importar as composições, podemos importar as gravações utilizando um código muito semelhante, apenas mudando o tipo do relacionamento de `COMPOS` para `GRAVOU`. Também mudamos o nome do atributo que vamos ler do `csv`, dado que o cabeçalho dos arquivos está diferente. No arquivo de composições, utilizamos o `compositor`, enquanto no arquivo de gravações, nomeamos a coluna `musico`.

```
LOAD CSV WITH HEADERS
FROM "file:/CAMINHO/ABSOLUTO/ATE/gravacoes.csv"
AS linhaCsv
MERGE (interprete:Musico {nome: linhaCsv.interprete})
MERGE (musica:Musica {nome: linhaCsv.musica})
MERGE (interprete)-[:GRAVOU]->(musica)

Added 8 labels, created 8 nodes, set 8 properties,
created 10 relationships, statement executed in 185 ms.
```

9.2 MAIS SOBRE O MATCH

Estamos cada vez mais perto de descobrirmos como encontrar *"quem gravou músicas escritas por compositores que escreveram músicas que um determinado músico gravou"*, mas antes vamos responder uma pergunta um pouco mais simples. Vamos encontrar quem compôs as músicas que cada músico gravou.

Para encontrarmos um compositor, temos de procurar um músico com um relacionamento do tipo `COMPOS` para uma música, usando o cypher `MATCH (compositor:Musico)-[COMPOS]->(musica:Musica)`. Para encontrar quem gravou a música, é só usar a mesma estrutura, mas trocando o tipo do relacionamento para `GRAVOU`. Veja: `MATCH (interprete:Musico)-[GRAVOU]->(musica:Musica)`.

No começo do capítulo anterior, vimos que podemos definir a direção do relacionamento que queremos buscar. Consequentemente, `MATCH (x:X)-[REL]->(y:Y)` é o mesmo que `MATCH (y:Y)<-[REL]-(x:X)`. Dado que sabemos como encontrar tanto quem gravou a música quanto quem a compôs, tudo o que precisamos fazer é juntar as expressões usando a música como o elo que une compositores e intérpretes.

Para isso, basta montar um lado da expressão com o músico a esquerda da música, e e do outro lado da expressão o músico a direita da música. Veja: `MATCH (i:Musico)-[g:GRAVOU]->(m:Musica)<-[e:COMPOS]-(c:Musico)`.

Podemos ler essa expressão como *"Intérprete que gravou uma música escrita por um compositor"*, que retornará todas as músicas que possui um intérprete e um compositor. Vamos verificar o resultado que este filtro retorna.

```
MATCH (i:Musico)-[g:GRAVOU]->(m:Musica)<-[e:COMPOS]-(c:Musico)
RETURN i.nome, m.nome, c.nome
```

i.nome	m.nome	c.nome
Jimi Hendrix	All Along the Watchtower	Bob Dylan
Johnny Cash	It Ain't Me, Babe	Bob Dylan
Jack White	One More Cup of Coffee	Bob Dylan
George Harrison	If Not For You	Bob Dylan

```
|Joey Ramone    |My Back Pages            |Bob Dylan    |
|---------------|-------------------------|-------------|
|Jon Bon Jovi   |Knockin' on Heavens      |Bob Dylan    |
|---------------|-------------------------|-------------|
|Steve Tyler    |Crazy                    |Desmond Child|
|---------------|-------------------------|-------------|
|Ricky Martin   |Livin' la Vida Loca      |Desmond Child|
|---------------|-------------------------|-------------|
|Jon Bon Jovi   |Livin' on a Prayer       |Desmond Child|
|---------------|-------------------------|-------------|
|Jon Bon Jovi   |You Give Love a Bad Name|Desmond Child|
+---------------+-------------------------+-------------+
Returned 10 rows in 59 ms.
```

Até agora, só usamos uma cláusula `MATCH` por query. No final do capítulo anterior, usamos uma `MATCH` junto com uma `OPTIONAL MATCH`, mas podemos usar quantas cláusulas `MATCH` quisermos. Quando usamos mais de um `MATCH`, eles se juntam de forma que o grafo precisa ser compatível com todos os `MATCH` s da query, como se fossem cláusulas `WHERE` agrupadas por `AND` s no SQL.

Por exemplo, a nossa query anterior possui dois relacionamentos interligados pelo mesmo nó. Em vez de usar `MATCH (i:Music)-[g:GRAVOU]->(m:Musica)<-[e:COMPOS]-(c:Musico)`, podemos quebrar em duas partes. A única restrição para que esta query funcione com dois `MATCH` s é usarmos o mesmo nome de variável para o nó que as une. Convertendo o exemplo anterior, podemos usar `MATCH (i:Music)-[g:GRAVOU]->(m:Musica)` junto com `(m:Musica)<-[e:COMPOS]-(c:Musico)`, desde que usemos o mesmo nome `(m:Musica)`.

```
MATCH (i:Musico)-[g:GRAVOU]->(m:Musica)
MATCH (m:Musica)<-[e:COMPOS]-(c:Musico)
RETURN i, m, c

//Mesmo resultado
Returned 10 rows in 59 ms.
```

Lembrando de que a ordem que escrevemos os nós não faz diferença, o que importa é a direção que definimos o

relacionamento. Logo, a query anterior também pode ser escrita desta forma:

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[compos:COMPOS]->(musica:Musica)
RETURN interprete, musica, compositor

//Mesmo resultado
Returned 10 rows in 19 ms.
```

O que importa é que estamos procurando por **um** `músico` que `GRAVOU` **uma** `música` e **outro** `músico` (nomes de variáveis diferentes) que `COMPOS` a **mesma** `música` (mesmo nome de variável).

Ainda não usamos nenhuma função de agregação, como `COUNT`, `SUM`, `AVG` ou todas as outras que você já está acostumado a usar no SQL. O cypher também suporta estas funções de maneira muito semelhante ao SQL. Não vamos entrar a fundo neste tema, mas, para exemplificar, vamos alterar a query anterior para, em vez de listar todas as músicas, exibir quantas músicas cada intérprete gravou de cada compositor.

A única coisa que vamos precisar fazer é alterar dentro do `RETURN`, onde, em vez de retornar o nó da `musica`, vamos retornar o total de músicas usando `COUNT(musica)`.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[compos:COMPOS]->(musica:Musica)
RETURN interprete.nome, COUNT(musica), compositor.nome
```

interprete.nome	COUNT(musica)	compositor.nome
Jimi Hendrix	1	Bob Dylan
Joey Ramone	1	Bob Dylan
Jon Bon Jovi	1	Bob Dylan
George Harrison	1	Bob Dylan
Jon Bon Jovi	2	Desmond Child

```
|Ricky Martin   |1            |Desmond Child  |
|Johnny Cash    |1            |Bob Dylan      |
|Steve Tyler    |1            |Desmond Child  |
|Jack White     |1            |Bob Dylan      |
Returned 9 rows in 135 ms.
```

9.3 QUERIES COMPLEXAS

Até agora, já resolvemos um passo do problema. Nós já conseguimos encontrar os compositores para cada músico. Se usarmos um filtro pelo nome do músico, podemos identificar com quais compositores ele trabalhou. Vamos adicionar novamente a cláusula `WHERE` para filtrar o intérprete, e então comparar os compositores que `Jimi Hendrix` trabalhou com os que `Jon Bon Jovi` trabalhou.

```
// Buscando por Jimi Hendrix
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[compos:COMPOS]->(musica:Musica)
WHERE interprete.nome = "Jimi Hendrix"
RETURN interprete.nome, COUNT(musica), compositor.nome
```

```
|interprete.nome|COUNT(musica)|compositor.nome|
|Jimi Hendrix   |1            |Bob Dylan      |
Returned 1 rows in 95 ms.
```

```
// Buscando por Jon Bon Jovi
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[compos:COMPOS]->(musica:Musica)
WHERE interprete.nome = "Jon Bon Jovi"
RETURN interprete.nome, COUNT(musica), compositor.nome
```

```
|interprete.nome|COUNT(musica)|compositor.nome|
|Jon Bon Jovi   |1            |Bob Dylan      |
|Jon Bon Jovi   |2            |Desmond Child  |
```

```
└─────────────────┴─────────────────┴─────────────────┘
Returned 2 rows in 23 ms.
```

A pergunta que queremos responder é *"quem gravou músicas escritas por **compositores que escreveram músicas que um determinado músico gravou**"*. Reparem que a última query que executamos responde metade da pergunta, a única coisa que falta é mais um salto no grafo para identificar quem gravou as músicas que os compositores em questão escreveram.

Partindo da última query, adicionaremos mais um `MATCH` para encontrar todas as músicas que os compositores escreveram: `MATCH (compositor:Musico)-[compos:COMPOS]->(musica:Musica)` . Temos de tomar muito cuidado com os nomes que vamos usar. Lembrem-se de que se usarmos o mesmo nome de variável, significa que estamos procurando pelo mesmo nó. Portanto, temos de usar dois nomes diferentes nos nós do tipo `Musica` .

Além disso, como queremos buscar as músicas escritas pelo mesmo compositor, então **temos** de usar o mesmo nome para nó do compositor, para garantir que estamos buscando as músicas escritas pelo mesmo músico. Assim, além de um `MATCH` a mais, vamos mudar um pouco o `RETURN` para contar a quantidade de músicas escritas pelo compositor.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[compos:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[compos:COMPOS]->(outraMusica:Musica)
WHERE interprete.nome = "Jimi Hendrix"
RETURN interprete.nome, compositor.nome, COUNT(outraMusica)
```

```
╔═══════════════╦═══════════════╦═══════════════════╗
║interprete.nome║compositor.nome║COUNT(outraMusica)║
╠═══════════════╬═══════════════╬═══════════════════╣
║Jimi Hendrix   ║Bob Dylan      ║1                  ║
└───────────────┴───────────────┴───────────────────┘
Returned 1 rows in 94 ms.
```

Na nossa base de dados, temos 10 músicas, sendo 6 escritas pelo Bob Dylan e 4 escritas pelo Desmond Child. Apesar disso, nossa

última busca retornou apenas uma música. Além da importância dos nomes de variáveis nos nós, também precisamos nos preocupar com o nome que usamos nos relacionamentos.

Apesar de realmente estarmos buscando todos as músicas escritas pelos compositores que escreveram músicas que o Jimi Hendrix gravou, como estamos usando o mesmo nome de variável no relacionamento `compos` , significa que estamos ignorando as músicas que não passam pelo relacionamento original. Vejam a mesma query filtrada para o intérprete Jon Bon Jovi:

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[compos:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[compos:COMPOS]->(outraMusica:Musica)
WHERE interprete.nome = "Jon Bon Jovi"
RETURN interprete.nome, compositor.nome, COUNT(outraMusica)
```

```
╒═══════════════╤═══════════════╤═══════════════════╕
│interprete.nome│compositor.nome│COUNT(outraMusica)│
╞═══════════════╪═══════════════╪═══════════════════╡
│Jon Bon Jovi   │Bob Dylan      │1                  │
├───────────────┼───────────────┼───────────────────┤
│Jon Bon Jovi   │Desmond Child  │2                  │
└───────────────┴───────────────┴───────────────────┘
Returned 2 rows in 74 ms.
```

Novamente o resultado foi igual ao número de músicas escritas pelo compositor que o mesmo intérprete gravou. Para resolver a única mudança que precisamos fazer, devemos renomear a variável de um dos relacionamentos do tipo `COMPOS` .

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
WHERE interprete.nome = "Jimi Hendrix"
RETURN interprete.nome, compositor.nome, COUNT(outraMusica)
```

```
╒═══════════════╤═══════════════╤═══════════════════╕
│interprete.nome│compositor.nome│COUNT(outraMusica)│
╞═══════════════╪═══════════════╪═══════════════════╡
│Jimi Hendrix   │Bob Dylan      │6                  │
└───────────────┴───────────────┴───────────────────┘
Returned 1 rows in 68 ms.
```

Agora conseguimos o resultado que queríamos, e encontramos

todas as músicas escritas pelos compositores que escreveram as músicas que o Jimi Hendrix gravou. Vamos conferir o resultado para o intérprete Jon Bon Jovi:

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
WHERE interprete.nome = "Jon Bon Jovi"
RETURN interprete.nome, compositor.nome, COUNT(outraMusica)
```

interprete.nome	compositor.nome	COUNT(outraMusica)
Jon Bon Jovi	Bob Dylan	6
Jon Bon Jovi	Desmond Child	8

```
Returned 2 rows in 74 ms.
```

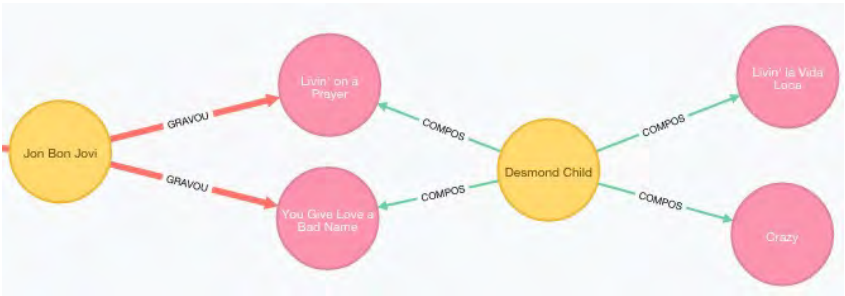
Se repararmos bem, podemos notar que o resultado não é exatamente o que estávamos esperando. A nossa busca encontrou todas as 6 músicas compostas pelo Bob Dylan, mas encontrou 8 músicas escritas pelo Desmond Child, enquanto o resultado esperado deveria ser 4 músicas.

Não é coincidência que o resultado seja exatamente o dobro do valor esperado. Na verdade, não é simplesmente o dobro. Neste exemplo, o valor é 2 vezes maior porque nosso intérprete gravou 2 músicas do compositor. Novamente encontramos um dos casos que a simplicidade do cypher pode complicar as coisas.

Se formos seguir nó por nó, relacionamento por relacionamento, encontraremos dois caminhos que levam do nó `Jon Bon Jovi` até os nós de cada uma das músicas escritas pelo `Desmond Child`. Vamos analisar o grafo do `Bon Jovi` até a música `Crazy`.

Estamos buscando a estrutura *(Musico)-**[GRAVOU]**->(Musica) <-**[COMPOS]**-(Musico)-**[COMPOS]**->(Musica).* Na figura a seguir, podemos ver que, entre o músico `Bon Jovi` e a música `Crazy`,

temos dois caminhos compatíveis com a estrutura que buscamos.



O primeiro caminho é `Bom Jovi-[GRAVOU]->Livin' on a Prayer<-[COMPOS]-Desmond Child-[COMPOS]->Crazy`, e o segundo caminho é `Bom Jovi-[GRAVOU]->You Give Love a Bad Name<-[COMPOS]-Desmond Child-[COMPOS]->Crazy`. Por isso, ao executarmos o `COUNT` de `outraMusica`, o Neo4j conta cada uma das músicas escritas pelo `Desmond Child` 2 vezes.

Assim como no SQL, podemos usar o `DISTINCT`, que remove os itens duplicados. Com isso, resolvemos o problema da nossa busca.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
WHERE interprete.nome = "Jon Bon Jovi"
RETURN interprete.nome,
       compositor.nome,
       COUNT(DISTINCT outraMusica)
```

interprete.nome	compositor.nome	COUNT(DISTINCT outraMusica)
Jon Bon Jovi	Bob Dylan	6
Jon Bon Jovi	Desmond Child	4

```
Returned 2 rows in 17 ms.
```

Finalmente podemos responder a grande questão! Agora que já chegamos a todas as músicas escritas pelos compositores que

escreveram músicas que um determinado músico gravou, só falta mais um passo para chegarmos aos intérpretes que as gravaram. Provavelmente, já não é grande surpresa que a solução envolva adicionar mais um `MATCH` ligando o nó `outraMusica` a um nó do tipo `Musico` por um relacionamento do tipo `GRAVOU`.

Assim como nos exemplos anteriores, é muito importante prestar atenção aos nomes dos nós e dos relacionamentos para garantir que não estamos nos referenciando aos mesmo nós ou relacionamentos anteriores. Vamos adicionar o seguinte fragmento na nossa query `MATCH (outraMusica:Musica)<-[gravou2:GRAVOU]-(outroInter:Musico)`, e no final, no `RETURN`, vamos adicionar o nome nó `outroInter`.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
MATCH (outraMusica:Musica)<-[gravou2:GRAVOU]-(outroInter:Musico)
WHERE interprete.nome = "Jimi Hendrix"
RETURN interprete.nome, compositor.nome, outroInter.nome
```

interprete.nome	compositor.nome	outroInter.nome
Jimi Hendrix	Bob Dylan	Jon Bon Jovi
Jimi Hendrix	Bob Dylan	Joey Ramone
Jimi Hendrix	Bob Dylan	George Harrison
Jimi Hendrix	Bob Dylan	Jack White
Jimi Hendrix	Bob Dylan	Johnny Cash
Jimi Hendrix	Bob Dylan	Jimi Hendrix

```
Returned 6 rows in 29 ms.
```

Por enquanto, a busca parece estar correta, mas com um detalhe que ainda podemos melhorar. O intérprete Jimi Hendrix também aparece na coluna `outroInter` em uma linha `| Jimi Hendrix | Bob Dylan | Jimi Hendrix |`, o que não parece fazer muito

sentido. Para eliminar esta linha, podemos adicionar uma cláusula a mais para garantir que o `outroInterprete` seja de fato **outro** intérprete.

Assim como no SQL, quando queremos usar duas ou mais restrições na nossa busca, precisamos usar o `AND` em vez de um segundo `WHERE`. O filtro que queremos adicionar agora é garantir que o nó `outroInter` seja **diferente** do nó `interprete`.

No cypher, o operador de comparação para diferente é o `<>`. Então, adicionaremos na nossa query anterior `AND interprete <> outroInter`.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
MATCH (outraMusica:Musica)<-[gravou2:GRAVOU]-(outroInter:Musico)
WHERE interprete.nome = "Jimi Hendrix"
AND interprete <> outroInter
RETURN interprete.nome, compositor.nome, outroInter.nome
```

interprete.nome	compositor.nome	outroInter.nome
Jimi Hendrix	Bob Dylan	Jon Bon Jovi
Jimi Hendrix	Bob Dylan	Joey Ramone
Jimi Hendrix	Bob Dylan	George Harrison
Jimi Hendrix	Bob Dylan	Jack White
Jimi Hendrix	Bob Dylan	Johnny Cash

```
Returned 5 rows in 524 ms.
```

Agora que resolvemos este detalhe, vamos verificar os resultados para outros intérpretes.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
MATCH (outraMusica:Musica)<-[gravou2:GRAVOU]-(outroInter:Musico)
WHERE interprete.nome = "Steve Tyler"
AND interprete <> outroInter
```

```
RETURN interprete.nome, compositor.nome, outroInter.nome
```

interprete.nome	compositor.nome	outroInter.nome
Steve Tyler	Desmond Child	Jon Bon Jovi
Steve Tyler	Desmond Child	Jon Bon Jovi
Steve Tyler	Desmond Child	Ricky Martin

```
Returned 3 rows in 13 ms.
```

Novamente temos dados duplicados no nosso resultado. Assim como no passo anterior, tínhamos mais de um caminho do intérprete até as músicas, agora temos mais de um caminho entre os dois intérpretes. Apesar de parecer uma boa ideia, não podemos simplesmente adicionar `DISTINCT` no nó `outroInter`, pois essa não é uma utilização válida.

O `DISTINCT` é na verdade um modificador da função de agregação, e não pode ser usada fora dessa função, como o `COUNT`. Uma das formas de resolver o nosso problema é remover os caminhos duplicados justamente através de uma agregação. Se ignorarmos quais são as músicas entre o compositor e o outro intérprete, e simplesmente contar quantos nós existem nesta posição, vamos remover os caminhos duplicados, e ainda adicionar uma informação relevante para definirmos a afinidade, ou o quão parecidos são os intérpretes, baseando-se no número de músicas do mesmo compositor que eles compartilham.

A solução então será adicionar no `RETURN` o valor `COUNT(outraMusica)`. Além disso, para facilitar a legibilidade, vamos usar o `AS` que nos permite definir qual será o nome da coluna na saída da busca. Então, no final, usaremos `COUNT(outraMusica) AS totMusicas`.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
```

```
MATCH (outraMusica:Musica)<-[gravou2:GRAVOU]-(outroInter:Musico)
WHERE interprete.nome = "Steve Tyler"
AND interprete <> outroInter
RETURN interprete.nome,
       compositor.nome,
       COUNT(outraMusica) AS totMusicas,
       outroInter.nome
```

interprete.nome	compositor.nome	totMusicas	outroInter.nome
Steve Tyler	Desmond Child	2	Jon Bon Jovi
Steve Tyler	Desmond Child	1	Ricky Martin

```
Returned 2 rows in 137 ms.
```

Para finalizar vamos conferir o resultado para o intérprete `Jon Bon Jovi`:

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
MATCH (outraMusica:Musica)<-[gravou2:GRAVOU]-(outroInter:Musico)
WHERE interprete.nome = "Jon Bon Jovi"
AND interprete <> outroInter
RETURN interprete.nome,
       compositor.nome,
       COUNT(outraMusica) AS totMusicas,
       outroInter.nome
```

interprete.nome	compositor.nome	totMusicas	outroInter.nome
Jon Bon Jovi	Bob Dylan	1	Jack White
Jon Bon Jovi	Bob Dylan	1	Johnny Cash
Jon Bon Jovi	Bob Dylan	1	George Harrison
Jon Bon Jovi	Desmond Child	2	Ricky Martin
Jon Bon Jovi	Bob Dylan	1	Jimi Hendrix
Jon Bon Jovi	Desmond Child	2	Steve Tyler
Jon Bon Jovi	Bob Dylan	1	Joey Ramone

```
Returned 7 rows in 17 ms.
```

Estamos muito perto do resultado esperado, mas ainda temos um problema. O Rick Martin e Steve Tyler gravaram apenas uma música do Desmond Child cada, mas como o Jon Bon Jovi gravou duas, novamente temos o mesmo problema dos múltiplos caminhos. Assim como no caso anterior do `COUNT` de músicas, podemos facilmente resolver este problema usando o `DISTINCT`.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
MATCH (outraMusica:Musica)<-[gravou2:GRAVOU]-(outroInter:Musico)
WHERE interprete.nome = "Jon Bon Jovi"
AND interprete <> outroInter
RETURN interprete.nome,
       compositor.nome,
       COUNT(DISTINCT outraMusica) AS totMusicas,
       outroInter.nome
```

interprete.nome	compositor.nome	totMusicas	outroInter.nome
Jon Bon Jovi	Bob Dylan	1	Jack White
Jon Bon Jovi	Bob Dylan	1	Johnny Cash
Jon Bon Jovi	Bob Dylan	1	George Harrison
Jon Bon Jovi	Desmond Child	1	Ricky Martin
Jon Bon Jovi	Bob Dylan	1	Jimi Hendrix
Jon Bon Jovi	Desmond Child	1	Steve Tyler
Jon Bon Jovi	Bob Dylan	1	Joey Ramone

```
Returned 7 rows in 18 ms.
```

Reorganizando o `RETURN` e adicionando o `ORDER BY` para ordenar o resultado, chegamos exatamente ao mesmo resultado do SQL apresentado antes de entrarmos no tópico dos bancos orientados a grafos.

```
MATCH (interprete:Musico)-[gravou:GRAVOU]->(musica:Musica)
MATCH (compositor:Musico)-[com1:COMPOS]->(musica:Musica)
MATCH (compositor:Musico)-[com2:COMPOS]->(outraMusica:Musica)
```

```
MATCH (outraMusica:Musica)<-[gravou2:GRAVOU]-(outroInter:Musico)
WHERE interprete.nome = "Jon Bon Jovi"
AND interprete <> outroInter
RETURN outroInter.nome AS interprete,
       compositor.nome AS compositor,
       COUNT(DISTINCT outraMusica) AS total_musicas
ORDER BY compositor.nome
```

interprete	compositor	total_musicas
Jimi Hendrix	Bob Dylan	1
Joey Ramone	Bob Dylan	1
George Harrison	Bob Dylan	1
Johnny Cash	Bob Dylan	1
Jack White	Bob Dylan	1
Ricky Martin	Desmond Child	1
Steve Tyler	Desmond Child	1

Substituindo o nome do intérprete que estamos filtrando, temos os seguintes resultados:

```
# Filtrando pelo Ricky Martin
```

interprete	compositor	total_musicas
Jon Bon Jovi	Desmond Child	2
Steve Tyler	Desmond Child	1

```
# Filtrando pelo Jimi Hendrix
```

interprete	compositor	total_musicas
Joey Ramone	Bob Dylan	1
Jon Bon Jovi	Bob Dylan	1
George Harrison	Bob Dylan	1
Johnny Cash	Bob Dylan	1

```
┌──────────────┬──────────┬──────────────┐
│Jack White    │Bob Dylan │1             │
└──────────────┴──────────┴──────────────┘
```

Apesar de nossa query final não ser pequena, dependendo de como escrever e agrupamos os `MATCH` , podemos ter queries bem legíveis e, normalmente,
mais fáceis de entender do que os `JOIN` do SQL. Além disso, poder contar com a representação gráfica do grafo nos ajuda muito a visualizar os dados e montar os filtros de forma quase intuitiva.

A ausência de chaves estrangeiras e a possibilidade de qualquer nó se relacionar de qualquer maneira com qualquer nó é, sem dúvidas, uma grande vantagem dos bancos orientados a grafos comparado com os outros tipos de bancos de dados.

CAPÍTULO 10

TUDO JUNTO E UM POUCO MAIS

Se você chegou até este ponto, tenho muito a agradecê-lo. Sem dúvidas, este livro foi um dos projetos pessoais mais importantes da minha vida, e espero que a leitura dele tenha um impacto positivo na sua carreira.

Junto com o livro, existe um projeto com imagens docker prontas para testar os bancos de dados, como também uma suíte de testes que exercitam uma parte do código usado no decorrer do livro. Esta pode ser encontrada no GitHub, em https://github.com/davidpaniz/nosql-project.

A Casa do Código, além de publicar livros como este, também mantém um fórum no qual os assuntos relacionados podem ser discutidos. Vocês são bem-vindos a participar, perguntando e respondendo neste fórum, http://forum.casadocodigo.com.br/.

10.1 O CONSELHO FINAL

Durante o livro, passamos pelos principais tipos de bancos de dados em situações em que eles eram ferramentas adequadas, sempre tentando usar a melhor implementação disponível para os problemas em questão. Após a leitura deste livro, meu principal conselho neste momento é: pense duas vezes antes de escolher o seu próximo banco de dados.

Isso inclui um senso crítico se realmente um banco NoSQL é a melhor ferramenta para seu problema. Lembre-se de que desenvolvimento de software vai muito além de construir. Não podemos, em hipótese alguma, negligenciar a manutenção das coisas que criamos.

Assim como quando um programador que acabou de aprender sobre *design patterns* fica superempolgado para implantar todos eles como se fossem troféus de videogames, é comum enxergamos o problema certo para usar um banco de dados específico em que ele não necessariamente é a melhor solução.

Tenha muito cuidado com a tentação de testar novas tecnologias, especialmente quando estamos falando dos dados de sua aplicação. Como acabei de alertar, além de desenvolver, é preciso manter. E quando o assunto é banco de dados, existem questões de segurança, disponibilidade, performance, políticas e práticas de backups e muitas outras que devem ser levadas em conta.

Apesar de frustrante, em algumas situações escutar um "não" do DBA é, sem dúvidas, a melhor decisão para o produto, dado o conhecimento dos profissionais envolvidos e os recursos disponíveis. Acredito que não podemos estagnar e ter medo de testar novas tecnologias, mas quando um time compra a ideia de usar uma ferramenta nova, é preciso que todos os envolvidos de ponta a ponta do processo estejam comprometidos com a decisão para que ela seja um sucesso de fato.

10.2 OUTRAS FERRAMENTAS RELACIONADAS

Além dos bancos de dados que utilizamos no decorrer do livro, existem outras implementações destes tipos de bancos de dados usados, e algumas outras ferramentas relacionadas a eles. Existem

também outras que nem podem ser consideradas bancos de dados, mas também podem ser ótimas opções para persistência de dados, e muito úteis em determinadas situações.

Elasticsearch

O Elasticsearch é uma dessas ferramentas que não são exatamente um banco de dados. A maior parte dos bancos de dados, incluindo o NoSQL, não é muito boa para trabalhar com queries envolvendo texto.

Quando precisamos fazer uma busca, por exemplo, pelo nome de produtos em uma loja virtual, se formos implementar a busca direto no banco de dados, vamos precisar usar `LIKE` com o uso de `wyldcard` para poder procurar o termo em qualquer parte do nome do produto. Além de lento, este tipo de busca não é muito poderoso para o cliente, pois ele precisa buscar o termo exato, assim como está no nome do produto.

O Elasticsearch é implementado usando o Lucene, uma ferramenta de indexação de texto muito poderosa. Com ele, podemos implementar a busca de texto chamada de `Full-Text search` , que em vez de buscas booleanas — na quais ou o termo é igual ou não é —, ele pode utilizar lógica difusa (*fuzzy logic*) — em que cada palavra recebe uma nota (`score`) baseado no termo buscado e podemos retornar as palavras semelhantes.

Além disso, ele permite a indexação inteligente baseado no radical das palavras, facilitando a busca por palavras que flexionam gênero, número e grau. Isso quer dizer que, se buscarmos "boneca vermelha", podemos encontrar termos como "bonequinho vermelho", desde que configurado para isso.

Além de ser relativamente fácil implementar `full-text search` utilizando o Elasticsearch, ele ainda possui ferramental

para replicação de dados e alta disponibilidade. Ele também nos permite coletar métricas sobre os documentos e, até mesmo, sugerir termos para busca, como o famoso *"Did you mean"*, ou *"Você quis dizer"*, do Google.

openCypher

Como vimos nos capítulos sobre grafos, o **Cypher** é a linguagem de buscas (*query language*) utilizada pelo Neo4j. Apesar de desenvolvida pela *Neo Technology*, essa linguagem chamou a atenção de outras empresas que desenvolvem outros bancos de dados, como *Oracle*, *Databricks* (empresa por trás do **Spark**) e algumas outras.

Essas empresas se uniram aos criadores do **Cypher** e decidiram começar uma implementação pública de código aberto (*open source*) desta linguagem. Então, surgiu o **openCypher**.

Assim como ter um SQL minimamente compativel com todos os RDBMS, foi fundamental para adoção desse tipo de banco de dados. Acredita-se que ter uma versão comum do Cypher disponível em todos os bancos orientados a grafo deva ajudar na adoção desta tecnologia.

Datomic

O Datomic é um banco de dados muito diferente dos outros. Enquanto os bancos de dados, tanto os RDBMS quanto os NoSQL, armazenam apenas o estado atual dos dados, no Datomic nós temos todo o histórico de atualizações.

O Datomic se baseia no princípio de imutabilidade utilizado principalmente por linguagens funcionais, e leva esse conceito para dentro do banco de dados. Quando alteramos um registro, estamos na verdade adicionando um novo fato nele. Por isso, podemos

facilmente carregar apenas um conjunto de dados atual de um registro, ou podemos carregar todas as transações que "modificaram" o registro e verificar cada valor antes e depois de cada transação. Isso o torna uma ótima ferramenta quando existe a necessidade de auditoria na camada de dados.

O Datomic é dividido em três partes: os *peers*, responsáveis pela leitura; o *transactor*, responsável pela escrita; e a camada de persistência, que pode ser apenas uma persistência temporária em memória até um outro banco de dados como o *Riak*, *Cassandra*, *DynamoDB*, ou *Postgre*, entre outros. Além disso, diferente da maior parte dos bancos NoSQL, o Datomic segue o padrão *ACID*, e garante a consistência e integridade dos dados.

Spark

O Spark é mais uma dessas ferramentas que não são um banco de dados. Ele é na verdade um motor de processamento (*processing engine*) para processamento de fluxo (*stream processing*).

Com ele, é possível montar e gerenciar um cluster de máquinas para executar processamentos em cima de grandes volumes de dados. É possível plugar vários tipos de origem de dados, incluindo bancos relacionais, Cassandra e outros.

PostgreSQL Document Store

Apesar de o PostgreSQL ser um famoso banco de dados relacional, nas suas últimas versões ele ganhou suporte para armazenar JSON, tanto no formato JSON puro quanto binário (`jsonb`), semelhante ao BSON usado pelo MongoDB. Armazenando dados no formato `jsonb` , podemos efetuar buscas por atributos, elementos aninhados e em arrays, assim como em uma banco orientado a documentos.

Diferente do MongoDB, o PostgreSQL continua garantindo consistência e durabilidade, podendo ser classificado como **CP**, se baseado no teorema CAP. Além do MongoDB ser duramente criticado pela sua arquitetura e os vários bugs reportados sobre perda de dados e vazamento de memória, na maioria dos testes de performance, o PostgreSQL se mostrou mais rápido e resiliente que ele.

Apesar de o MongoDB ser o banco de dados NoSQL mais popular atualmente, provavelmente o PostgreSQL é uma implementação melhor de um banco de dados orientado a documentos, com a "desvantagem" de priorizar a consistência sobre a disponibilidade.

Redis, Memcached e DynamoDB

O Redis e o Memcached são provavelmente os bancos de dados do tipo chave/valor mais utilizados. O **Redis** usa uma forma de replicação de dados mais tradicional que o Riak, baseado em `master / slave` , e também suporta tipos de dados especiais como conjuntos (`Set`) e listas (`List`).

O **Memcached** segue uma filosofia mais simplista. Parte da lógica deve ficar no cliente, pois o servidor não suporta nenhum tipo de replicação ou dados especiais. Como o próprio nome sugere, sua melhor aplicação é para cache de dados.

O **DynamoDB**, que foi a primeira implementação de banco de dados do tipo chave/valor, não pode ser baixado e instalado. Ele só está disponível como serviço dentro da *Amazon Web Services* (AWS).

Uma grande vantagem de usar o DynamoDB, é que, como ele é contratado como um serviço, não precisamos nos preocupar com disponibilidade e escalabilidade. Só precisamos configurar qual a

quantidade de leitura e de escritas que queremos, e sair usando.

O modelo de cobrança é baseado nestes parâmetros. Então, se sua aplicação é pequena, você pode ter um banco chave/valor altamente disponível, pagando relativamente pouco, sem precisar de várias máquinas e zonas diferentes etc.

Capítulo 11

REFERÊNCIAS BIBLIOGRÁFICAS

BREWER, Eric. *CAP twelve years later:* How the "rules" have changed. Maio 2012. Disponível em: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed.

BROWNE, Julian. *Brewer's CAP theorem:* The kool aid Amazon and Ebay have been drinking. Jan. 2009. Disponível em: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.

GHEMAWAT, Sanjay; HSIEH, Wilson C.; WALLACH, Deborah A.; BURROWS, Mike; et al. *Bigtable:* A distributed storage system for structured data. 2006. Disponível em: http://static.googleusercontent.com/media/research.google.com/pt-BR//archive/bigtable-osdi06.pdf.

JAMPANI, Madan; KAKULAPATI, Gunavardhan; LAKSHMAN, Avinash; PILCHIN, Alex; et al. *Dynamo:* Amazon's highly available key-value store. 2007. Disponível em: http://dl.acm.org/citation.cfm?id=1294281.

HAERDER, Theo; REUTER, Andreas. *Principles of transaction-oriented database recovery.* 1983. Disponível em: http://web.stanford.edu/class/cs340v/papers/recovery.pdf.

HERLIHY, Maurice P.; WING, Jeannette M. *Linearizability:* A

correctness condition for concurrent objects. 1989. Disponível em: http://dl.acm.org/citation.cfm?id=78972.

KOHAVI, Ron; LONGBOTHAM, Roger. *Online experiments:* Lessons learned. 2007. Disponível em: http://robotics.stanford.edu/~ronnyk/2007IEEEComputerOnlineExperiments.pdf.

LAKSHAMN, Avinash; MALIK, Prashant. *Cassandra - A decentralized structured storage system.* 2009. Disponível em: http://dl.acm.org/citation.cfm?id=1773922.

KLEPPMANN, Martin. *Designing Data-Intensive Applications:* The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, 2014.