Banco de Dados NoSQL

Aula 10: Modelo de dados NoSQL baseado em grafos – parte II

Apresentação

Praticaremos, por meio de exemplos, os comandos de manipulação no modelo de dados em grafos, usando o sistema gerenciador de banco de dados Neo4J, com sua linguagem Cypher.

Objetivo

- Escrever comandos Cypher para manipulação de dados: create, update e delete;
- Escrever comandos Cypher para consultas simples e com o uso de funções;
- Escrever comandos Cypher para consultas envolvendo agregações.

Características da Unreal Engine

Ao trabalharmos com UDK, estávamos utilizando a versão 3 da Unreal Engine, mas, a partir da versão 4, passamos a trabalhar com um ambiente bem mais profissional, em que o sistema de **Blueprints** conseguiu trazer ainda mais praticidade do que o **Kismet**, além de oferecer muito mais liberdade para as atividades que envolvem programação.

Os Blueprints são editores visuais de fluxo, mas, diferentemente do Kismet, trazem um sistema expansível, no qual as ferramentas apropriadas são disponibilizadas de acordo com o perfil da tarefa executada, como telas voltadas para controle de animação e física, entre outros, além de interagir com componentes programados em linguagem C++. Eles são o coração da Unreal Engine, e saber lidar com Blueprints trará grande produtividade na concepção de jogos.

Comentário

A linguagem oficial da Unreal Engine é o **C++**, porém, nas versões mais recentes, é possível utilizar o **Python**, de forma alternativa, para efetuar diversas tarefas.

Introdução

Dando continuidade ao estudo dos bancos de dados em grafos, iremos praticar, por meio de exemplos, os comandos para a manipulação de dados usando a linguagem Cypher (SADALAGE; FOWLER, 2013) do Neo4J.

Os principais comandos de manipulação serão apresentados com a ideia de comparar os comandos com a SQL, para facilitar o entendimento dessa linguagem. Após verificar as operações básicas como o CRUD (create, match, set e delete), realizaremos consultas, usaremos funções e consultas com agregações.

Mantidas as devidas diferenças, você vai observar que a ideia do comando é semelhante aos comandos da SQL.

Comandos Cypher para manipulação de dados: create, update e delete

Antes de começarmos a trabalhar nos exemplos das operações de manipulação, comentaremos sobre o estudo de caso que será usado inicialmente. Trata-se de uma aplicação para o controle de projetos usando três labels (rótulos) diferentes:

Gerente

Projeto

Funcionário

Comentário

O que comparativamente ao modelo relacional seriam três tabelas indicando que um gerente pode coordenar vários projetos e um projeto pode ter vários funcionários trabalhando nele; assim como um funcionário pode trabalhar em vários projetos.

Para sua construção, vamos iniciar com a criação dos nós para os dois gerentes dos projetos. O *label* definido é Gerente e foi definida a propriedade "nome" para os nós, no formato chave-valor.

Na interface gráfica do Neo4J (ELMASRI; NAVATHE, 2018), você pode alterar a cor do nó clicando na parte de cima no *label* Gerente e escolhendo a cor desejada na parte debaixo da tela, inclusive selecionar o tamanho dos círculos que representam os nós. Isso facilita a visualização, principalmente quando existem muitos nós na aplicação.

Os comandos para a criação são apresentados a seguir (note que o comando CREATE da Cypher corresponde ao comando INSERT da SQL):

```
CREATE ( :Gerente { nome : 'Léa' } )

CREATE ( :Gerente { nome : 'Luís' } )
```

Para visualizar os nós criados, basta executar o comando simples a seguir, em que n é variável e irá armazenar os dados dos nós recuperados na consulta. Segue o comando que funciona como um SELECT * FROM, porém, apresenta todos os nós da aplicação, mesmo com *labels* diferentes:

```
MATCH ( n ) RETURN n
```

O próximo passo será a criação dos nós relativos aos três projetos. Neles, serão definidas as propriedades: "nome" e "data_inicio". Seguem os comandos para a criação dos nós dos três projetos:

```
CREATE ( : Projeto { nome : 'Projeto 1', data_inicio : '2019-10-06' } )

CREATE ( : Projeto { nome : 'Projeto 2', data_inicio : '2018-11-06' } )

CREATE ( : Projeto { nome : 'Projeto 3', data_inicio : '2017-09-06' } )
```

Para visualizar:

```
MATCH ( n ) RETURN n
```

Neste momento, serão criados os nós dos funcionários com as propriedades indicando o nome, o cargo e o salário desses funcionários. Vejamos:

```
CREATE ( : Funcionario { nome : 'Lia', cargo : 'analista de sistemas júnior', salario : 4000.00 } )

CREATE ( : Funcionario { nome : 'Rui', cargo : 'desenvolvedor web júnior', salario : 5000.00 } )

CREATE ( : Funcionario { nome : 'Nei', cargo : 'administrador de banco de dados sênior', salario : 9500.00 } )


MATCH (n) RETURN n
```

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Agora serão criados os relacionamentos por meio da definição das arestas. Observe que primeiro é necessário usar o comando MATCH para encontrar os nós e depois associá-los. Serão criados os relacionamentos para indicar os gerentes dos projetos:

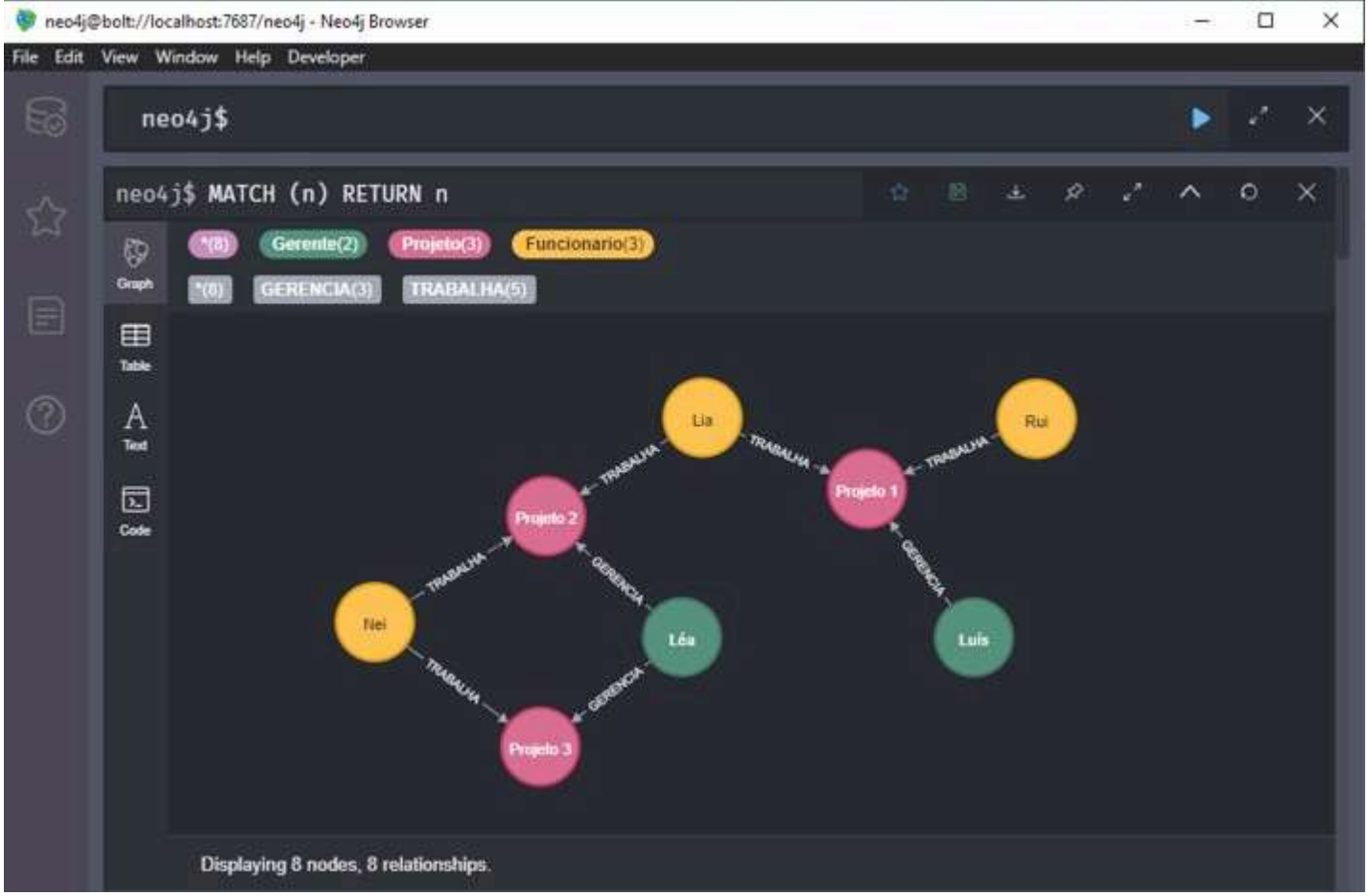
```
MATCH ( p1 : Gerente { nome : 'Luís' } ) , ( p2 : Projeto { nome : 'Projeto 1' } )
    CREATE ( p1 )-[ : GERENCIA ] -> ( p2 ) ;
MATCH ( p1 : Gerente { nome: 'Léa' } ) , ( p2 : Projeto {nome : 'Projeto 2' } )
    CREATE ( p1 )-[ :GERENCIA ] -> ( p2 ) ;
MATCH ( p1 : Gerente { nome : 'Léa' } ) , ( p2 : Projeto { nome : 'Projeto 3' } )
    CREATE ( p1 )-[ : GERENCIA ] -> ( p2 )

MATCH ( n ) RETURN n
```

Agora serão criados os relacionamentos que indicam as alocações dos funcionários nos projetos por meio dos comandos a seguir:

```
MATCH ( p1 : Funcionario { nome : 'Lia' } ) , ( p2 : Projeto { nome : 'Projeto 1' } )
    CREATE ( p1 )-[ : TRABALHA ] -> ( p2 ) ;
MATCH ( p1 : Funcionario { nome : 'Rui' } ) , ( p2 : Projeto { nome : 'Projeto 1' } )
    CREATE ( p1 )-[ : TRABALHA ] -> ( p2 ) ;
MATCH ( p1 : Funcionario { nome : 'Nei' } ) , ( p2 : Projeto { nome : 'Projeto 2' } )
    CREATE ( p1 )-[ : TRABALHA ] -> ( p2 ) ;
MATCH ( p1 : Funcionario { nome : 'Lia'} ) , ( p2 : Projeto { nome : 'Projeto 2' } )
    CREATE ( p1 )-[ : TRABALHA ] -> ( p2 ) ;
MATCH ( p1 : Funcionario { nome : 'Nei'} ) , ( p2 : Projeto { nome : 'Projeto 3' } )
    CREATE ( p1 )-[ : TRABALHA] -> ( p2 )

MATCH ( n ) RETURN n
```

Após a execução dos comandos apresentados, teremos como resultado o grafo exibido na Figura 1 a seguir:

Figura 1 - Resultado dos comandos para a criação do estudo de caso de controle de projetos.



Fonte: Neo4J.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

```
MERGE ( f : Funcionario { nome : 'João', cargo : 'gerente de suporte pleno', salario : 4000.00 } ) RETURN f
```

Se, em algum momento, houver a necessidade de excluir um ou mais nós de seu estudo de caso, pode-se usar os comandos a seguir para esse fim. Vejamos o comando para excluir todos os nós em uma só linha. Cuidado com esse comando!

```
MATCH ( n : Funcionario { nome : Rui } ) DELETE n
```

Ainda em relação às operações básicas do CRUD, apresentaremos um caso de uma atualização do valor de uma propriedade de um nó; pode-se usar o comando SET, o que seria feito com um UPDATE na linguagem SQL. No exemplo a seguir, é apresentado o comando para alterar o valor do salário da funcionária Lia para R$ 4.500,00. Vejamos:

```
MATCH ( p : Funcionario { nome : 'Lia' } )

SET p.salario = 4500.00

RETURN p
```

Ou, ainda, caso seja necessário realizar a atualização de mais de uma propriedade de uma só vez.

Mas cuidado! Com esse comando você não está acrescentando propriedades, e sim substituindo as existentes. Observe que houve o cuidado de manter as propriedades anteriores no conjunto p, mesmo a que não foi alterada; no caso, salário e cargo.

Se isso não fosse feito, essa propriedade seria excluída:

```
MATCH ( p : Funcionario { nome : 'Lia' } )
SET p = { salario : 4500.00, nome : 'Lia Maria' , cargo : 'analista de sistemas júnior' }
RETURN p
```

Para adicionar uma propriedade a um nó, podemos fazer como o exemplo seguinte, que acrescenta o *e-mail* à funcionária de nome Lia. Para a implementação do comando, usa-se o += para indicar a adição da propriedade. Vejamos:

```
MATCH ( p : Funcionario { nome : 'Lia Maria' } )
SET p += { email : 'lia@empresa.com.br' }
```

Caso seja necessário remover uma propriedade de um nó, basta indicar a propriedade pelo nome e "setar" o valor null. Segue o comando:

```
MATCH ( p : Funcionario { nome : 'Lia Maria' } )
SET p.email = null
```

Comandos Cypher para consultas simples e com o uso de funções

Continuaremos o estudo da linguagem escrevendo comandos direcionados para a recuperação ou seleção dos dados armazenados nos nós. Para iniciar, iremos escrever um comando para listar todos os nomes e salários dos funcionários. Algo que seria um SELECT nome, salario FROM Funcionario:

```
MATCH ( p : Funcionario )

RETURN p.nome, p.salario
```

O comando seguinte permite recuperar os nomes dos projetos gerenciados por Ana. Nele foi usado o símbolo -- que representa relacionado com. Consultas explorando os relacionamentos são muito comuns no Neo4J. Vejamos:

```
MATCH ( Gerente { nome : 'Léa' } )--( Projeto )

RETURN Projeto.nome
```

O comando seguinte serve para criar um nó para um estagiário de nome Pedro e, simultaneamente, criar o relacionamento (ESTAGIA) aproveitando o mesmo comando para associá-lo ao Projeto 1. Ou seja, usa o comando MATCH e o CREATE em uma só operação:

```
MATCH ( p : Projeto { nome : 'Projeto 1' } )

CREATE ( r : Estagiario { nome : 'Pedro' } ) ,

( r )-[ : ESTAGIA ] -> ( p )
```

Agora vamos estudar o uso dos comandos de consulta com a condição WHERE. Vejamos um exemplo aplicando uma condição para recuperar os dados dos funcionários que ganham salário igual ou acima de R$ 5.000,00:

```
MATCH ( p : Funcionario )

WHERE p.salario >= 5000

RETURN p
```

Se a necessidade fosse recuperar os funcionários que ganham entre R$ 4.000,00 e R$ 5.000,00, inclusive, ficaria da seguinte forma:

```
MATCH ( p : Funcionario )

WHERE 4000 <= p.salario <= 5000

RETURN p
```

Se fosse necessário recuperar os dados dos funcionários que pertencem ao cargo de administrador de banco de dados sênior, com salário superior a R$ 5.000,00, poderíamos escrever o comando da seguinte maneira, usando o operador AND:

```
MATCH ( p : Funcionario )

WHERE p.cargo = 'administrador de banco de dados sênior'

AND p.salario > 5000

RETURN p
```

Outro exemplo de consulta com condição composta é apresentado a seguir para recuperar os dados dos funcionários que exercem os cargos de analista de sistemas júnior ou de administrador de banco de dados sênior, fazendo uso do operador OR:

```
MATCH ( p : Funcionario )

WHERE p.cargo = 'analista de sistemas júnior'

OR p.cargo = 'administrador de banco de dados sênior'

RETURN p
```

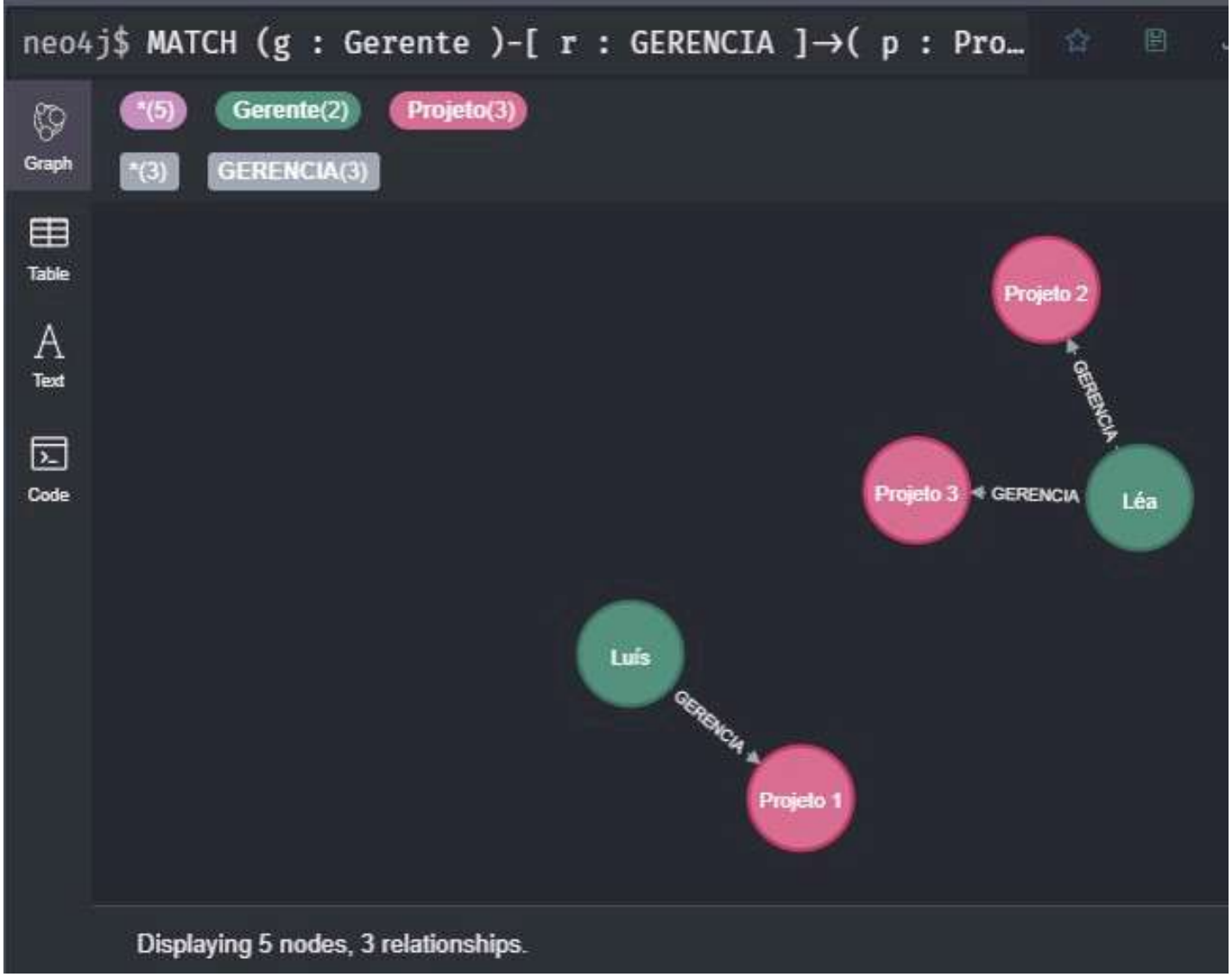
A consulta anterior pode ser escrita com o uso do IN como em um conjunto. Observe que o IN usa colchetes ao invés de parênteses como em SQL, mas seu funcionamento é idêntico. Segue o comando:

```
MATCH ( p : Funcionario )

WHERE p.cargo IN [ 'analista de sistemas júnior', 'administrador de
banco de dados sênior' ]

RETURN p
```

Fazendo uma alusão às consultas envolvendo junções com várias tabelas no modelo relacional, a Cypher permite recuperar dados dos nós envolvidos em relacionamentos. No exemplo a seguir, iremos recuperar os dados dos gerentes e dos projetos gerenciados por eles (Figura 2), mas você pode também ver o resultado em formato tabular ou, ainda, em formato textual, como apresentado na Figura 3:

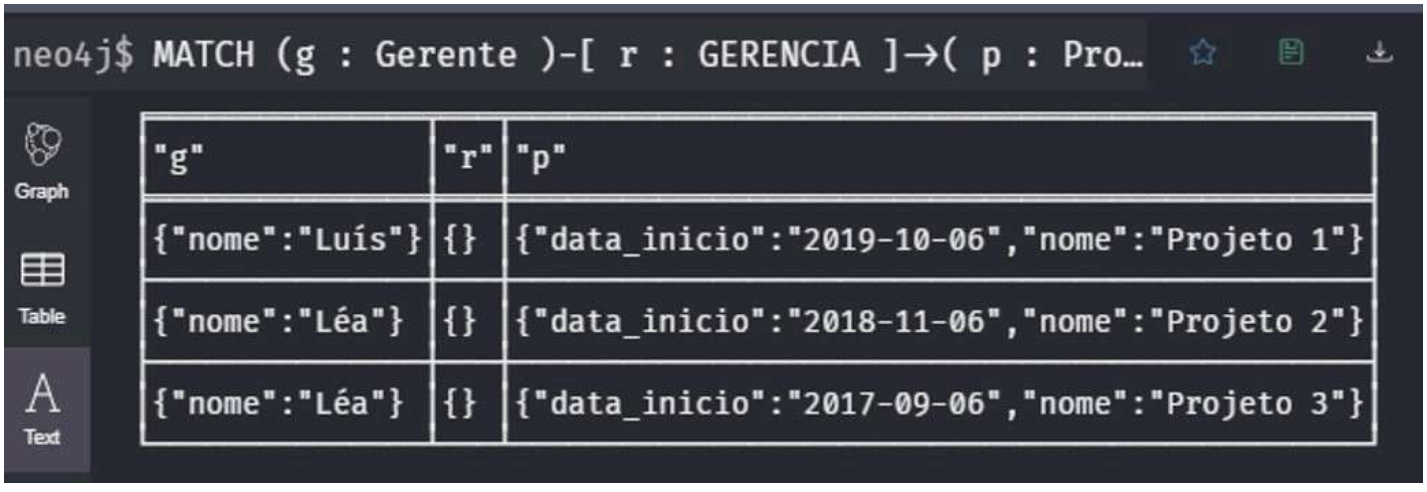
```
MATCH ( g : Gerente )-[ r : GERENCIA ] -> ( p : Projeto )

RETURN g, r, p
```

Figura 2 - Resultado de uma consulta com os dados dos gerentes e dos projetos gerenciados por eles.



Fonte: Neo4J.

Figura 3 - Apresentação do resultado da consulta exibida na Figura 2 em formato textual.



"g"	"r"	"p"
{"nome":"Luís"}	{}	{"data_inicio":"2019-10-06","nome":"Projeto 1"}
{"nome":"Léa"}	{}	{"data_inicio":"2018-11-06","nome":"Projeto 2"}
{"nome":"Léa"}	{}	{"data_inicio":"2017-09-06","nome":"Projeto 3"}

Fonte: Neo4J.

Outra forma de consulta similar às usadas em SQL é a consulta do tipo UNION. No exemplo a seguir, iremos recuperar os nomes de todos os funcionários e os nomes de todos os projetos. Vejamos:

```
MATCH ( f : Funcionario )

RETURN f.nome AS nome
```

```
UNION ALL MATCH ( p : Projeto )

RETURN p.nome AS nome
```

Comandos Cypher para consultas envolvendo agregações

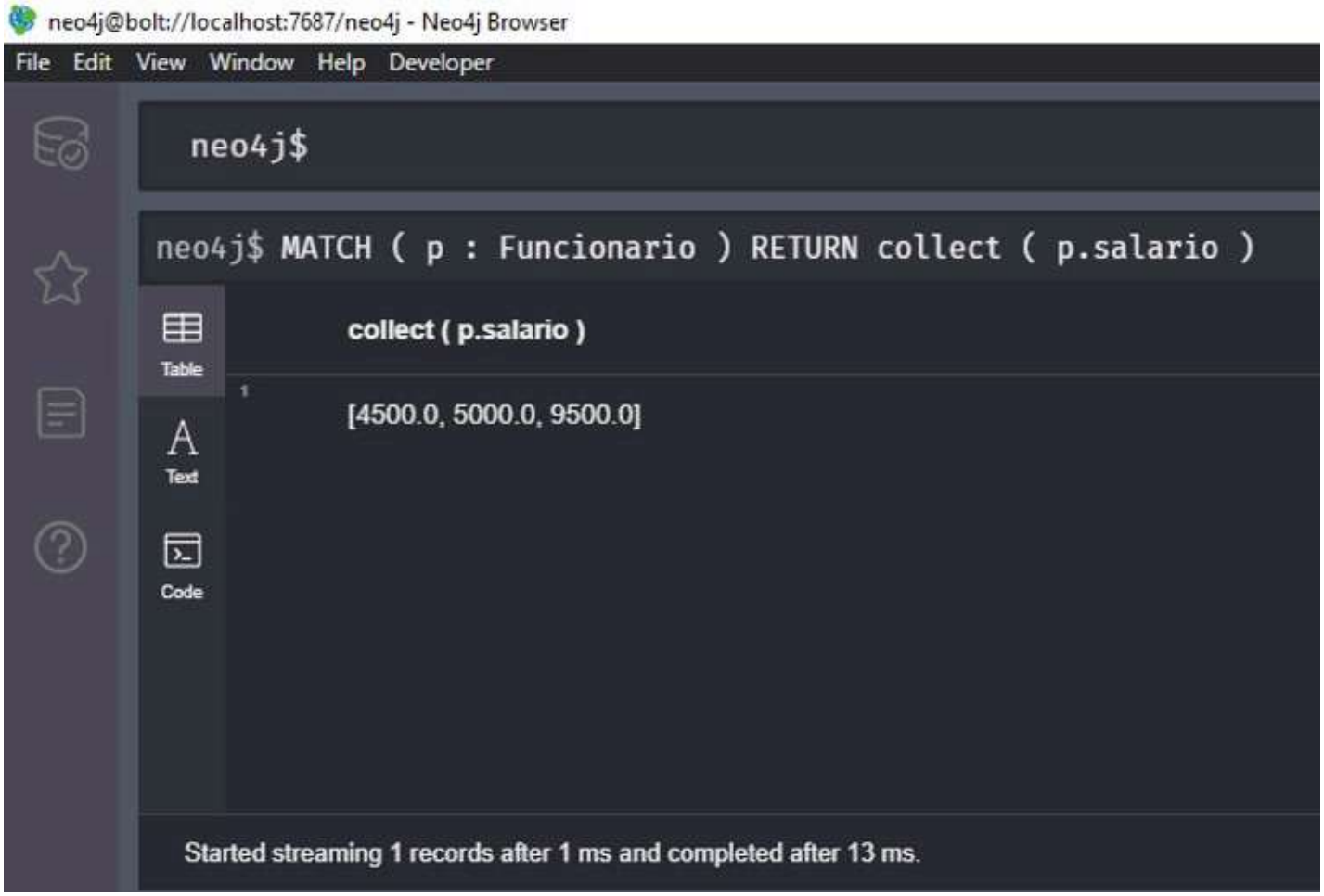
A linguagem Cypher também possui comandos e funções que podem ser usados para recuperar valores isolados ou na forma de agregados quando aplicados aos nós. Vamos iniciar apresentando a média de salários dos funcionários com o uso da função avg(), em que a função tem o mesmo nome usado na SQL:

```
MATCH ( p : Funcionario )

RETURN avg ( p.salario ) as media
```

Digamos que a necessidade agora seja listar propriedades da forma de uma coleção. Para isso, a Cypher possui a função collect(). Vejamos um exemplo para listar a coleção de salários dos funcionários. O resultado da consulta é apresentado na Figura 4:

```
MATCH ( p : Funcionario )

RETURN collect ( p.salario ) as salarios
```

Figura 4 - Resultado de uma consulta com o uso da função collect().



Fonte: Neo4J.

A linguagem Cypher permite realizar contagens de nós por meio de uma função count(), que funciona de forma igual à SQL. Veja os três exemplos a seguir, que apresentam os totais de funcionários, de gerentes e de estagiários presentes no banco de dados. Seguem os comandos:

```
MATCH ( p : Funcionario )
RETURN count(*) as total

MATCH ( p : Gerente )
RETURN count(*) as total

MATCH ( p : Estagiario )
RETURN count(*) as total
```

Podemos também usar as funções para verificar os maiores e os menores valores presentes em uma propriedade com o uso das funções max(), que indica o valor máximo, e a função min(), que indica o menor valor presente em uma determinada propriedade nos nós envolvidos na consulta. O exemplo de comando a seguir apresenta o maior e o menor salário de todos os funcionários:

```
MATCH ( p : Funcionario )
```

```
RETURN max ( p.salario ) as maior_salario, min ( p.salario ) as menor_salario
```

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Atividade

1. Assinale a opção correta quanto ao código necessário para criar um relacionamento indicando que Luís pratica futebol. Você pode usar o código abaixo para criar os nós e testar sua resposta antes de responder:

```
CREATE ( ana : Pessoa { nome : 'Ana' , email : 'ana@meumail.com' } )
CREATE ( lia :Pessoa { nome : 'Lia' , email : 'lia@meumail.com' } )
CREATE ( luis : Pessoa { nome : 'Luís' , email : 'luis@meumail.com' } )
CREATE ( rui : Pessoa { nome : 'Rui' , email : 'rui@meumail.com' } )
CREATE ( nei : Pessoa { nome : 'Nei' , email : 'nei@meumail.com' } )
CREATE ( ana )-[:AMIGO] -> ( lia )-[:AMIGO] -> ( luis )
CREATE ( ana )-[:AMIGO] -> ( rui )-[:AMIGO] -> ( nei )
CREATE ( futebol : Esporte { nome : 'Futebol' } )
MATCH ( n ) RETURN n
```

a) MATCH (p1 : Pessoa { nome : 'Luís' }) , (p2 : Esporte { nome : 'Futebol' })
CREATE (p1)-[: PRATICA] -> (p2)

b) MATCH (p1 : Pessoa { nome : 'Luís' }) , (p2 : Esporte { nome : 'Futebol' })
CREATE (p1)-(: PRATICA) -> (p2)

c) MATCH (p1 : { nome :'Luís' }) , (p2 : Esporte { nome : 'Futebol' })
CREATE (p1)-[: PRATICA] -> (p2)

d) MATCH (: Pessoa { nome : 'Luís' }) , (: Esporte { nome : 'Futebol' })
CREATE (p1)-[: PRATICA] -> (p2)

e) MATCH (p1 : Pessoa { nome : 'Luís' }) , (p2 : Esporte { nome : 'Futebol' })
CREATE (p1)->[: PRATICA] -> (p2)

---

2. Assinale a afirmativa que contém o comando correto para listar todos os dados das Pessoas cujo nome está contido em Ana ou Lia:

a) MATCH [p : Pessoa] WHERE p.nome IN ['Ana', 'Lia'] RETURN p

b) MATCH (p : Pessoa) WHERE p.nome IN ['Ana', 'Lia'] RETURN p

c) MATCH (p : Pessoa) WHERE p.nome IN ('Ana', 'Lia') RETURN p

d) SELECT (p : Pessoa) WHERE p.nome IN ['Ana', 'Lia'] RETURN p

e) MATCH (p : Pessoa) WHERE p.nome = 'Ana' and p.nome = 'Lia' RETURN p

---

3. Considere o código apresentado abaixo para uma rede de usuários, em que há a indicação do estado civil de cada um deles. Assinale a afirmativa que contém o comando correto para exibir o total de usuários solteiros:

```
CREATE ( : Usuario { nome : 'Ana' , pais : 'Canadá' , estado_civil : 'solteiro' } )
CREATE ( : Usuario { nome : 'Lia' , pais : 'Brasil' , estado_civil : 'casado' } )
CREATE ( : Usuario { nome : 'Rui' , pais : 'Brasil' , estado_civil : 'solteiro' } )
CREATE ( : Usuario { nome : 'Nei' , pais : 'Brasil' , estado_civil : 'casado' } )
CREATE ( : Usuario { nome : 'Joe' , pais : 'França' , estado_civil : 'solteiro'} )
CREATE ( : Usuario { nome : 'Carol' , pais : 'Estados Unidos da América' , estado_civil : 'namorando' } )
CREATE ( : Usuario { nome : 'Luís' , pais : 'México' , estado_civil : 'solteiro'} )
CREATE ( : Usuario { nome : 'Carol' , pais : 'Brasil' , estado_civil : 'casado' } )
```

a) MATCH (p : Usuario) WHERE p.estado_civil IN 'solteiro' RETURN GROUP BY count(*)
b) SELECT (p : Usuario) RETURN p.estado_civil = 'solteiro' WHERE count(*) as total
c) MATCH (p : Usuario) WHERE p.estado_civil = 'solteiro' RETURN count(*) as total
d) SELECT count(*) as total (p : Usuario) WHERE p.estado_civil = 'solteiro' RETURN
e) FIND (p : Usuario) WHERE p.estado_civil = 'solteiro' RETURN count(*) as total

---

Notas
Referências

ELMASRI, R.; NAVATHE, S.B. **Sistemas de banco de dados**. 6.ed. São Paulo: Pearson Education do Brasil, 2018.

SADALAGE, P.J.; FOWLER, M. **NoSQL essencial** – um guia conciso para o mundo emergente da persistência poliglota. 1.ed. São Paulo: Novatec, 2013.

Próxima aula

-

Explore mais

- Leia o livro ELMASRI, R.; NAVATHE, S.B. **Sistemas de banco de dados**. 6.ed. São Paulo: Pearson Education do Brasil, 2018.
- Leia o Capítulo 11 do livro SADALAGE, P.J.; FOWLER, M. **NoSQL essencial** – um guia conciso para o mundo emergente da persistência poliglota. 1.ed. São Paulo: Novatec, 2013.