
Neo4J com Java e Python

Fernando Anselmo

<http://fernandoanselmo.orgfree.com/wordpress/>

Versão 1.0 em 24 de outubro de 2021

Resumo

Neo4j é um banco do tipo NoSQL que é considerado como um sistema gerenciador para banco de dados Grafo. Oferece aos desenvolvedores e cientistas de dados as ferramentas mais confiáveis e avançadas para criar rapidamente os aplicativos inteligentes e fluxos para aprendizado de máquina. Disponível como um serviço de nuvem totalmente gerenciado ou auto-hospedado. Neste tutorial veremos o que vem a ser o banco NoSQL Neo4j [1] e como proceder sua utilização utilizando como pano de fundo a linguagem de programação Java [2] e Python [3].

1 Parte inicial

Neo4j foi criado em 2000, quando *Emil Eifrem*, *Johan Svensson* e *Peter Naubauer* começaram a notar uma quantidade significativa de sobrecarga tanto no desempenho quanto ao trabalho exigido em uma de suas aplicações. O primeiro e mais significativo aspecto da sobrecarga pode ser atribuído à incompatibilidade do modelo na forma como o SGBD Relacional estava trabalhando. Observaram que as conexões entre os dados impunham um longo tempo de processamento para as consultas. Além disso, o desempenho dessas ficou pior à medida que as conexões entre os dados se tornaram mais complexas. Finalmente, o tempo e esforço necessários para gerenciar esses relacionamentos colocaram ainda mais sobrecarga no ciclo de vida no desenvolvimento do aplicativo.



Figura 1: Logo do Neo4j

Depois de buscarem alternativas e algumas pesquisas, começaram a construir um projeto denominado **Neo**. O objetivo era apresentar um banco de dados que oferecesse uma maneira melhor de modelar, armazenar e recuperar dados, ao mesmo tempo em que mantinha todos os conceitos básicos, tais como ACID ou transações, que tornavam os SGBD Relacionais seguros e conhecidos.

Uma transação num banco relacional é essencialmente um grupo de consultas que devem ser bem-sucedidas para serem aplicadas. Se algo em uma transação falhar, toda ela deve também falhar. As transações possuem dois propósitos básicos, ambos envolvendo consistência, apenas de maneiras diferentes: O primeiro é garantir que todas os comandos dentro dessa sejam realmente executados. O segundo diz respeito a duas ações que acontecem ao mesmo tempo: se um banco de dados está sendo consultado simultaneamente por várias fontes, então existe a possibilidade da integridade dos dados ser comprometida.

Os princípios usados no ACID (Atomicidade, Consistência, Isolação e Durabilidade) são relativos ao Teorema CAP, também conhecido como *Teorema de Brewer*. *Eric Brewer* afirmou que é impossível para um sistema de computador distribuído (ou banco de dados) garantir simultaneamente as três condições a seguir:

- **Disponibilidade** cada solicitação recebe uma resposta sobre se foi bem-sucedida ou falha.
- **Consistência** os dados estão disponíveis para todos os nós ao mesmo tempo.
- **Tolerância de partição** o sistema ainda pode operar apesar de perder contato com outros nós devido a problemas de rede.

Assim pegaram esses conceitos e adicionaram os princípios da **teoria dos grafos** aplicando-os ao projeto **Neo**. Graças a este trabalho árduo, os relacionamentos entre dados é a característica levada muito a sério. Em um sentido matemático, a teoria dos grafos é o estudo das estruturas usadas para modelar o relacionamento entre os objetos.

Neo4j armazena dados como vértices e arestas ou, na terminologia dos grafos, nós e relacionamentos. Por exemplo donos são representados como nós e carros serão representadas como relacionamentos entre nós de usuários. Assim rapidamente podemos associar todas as pessoas que compraram determinado veículo (com apenas um único relacionamento).

1.1 O que é um Grafo?

O primeiro artigo conhecido sobre a teoria dos grafos foi escrito em 1736, chamado de "*As 7 Pontes de Königsberg*" por Leonhard Euler, um matemático e físico, considerado preeminente do século XVIII. A cidade de *Königsberg*, na Prússia (atual Rússia), foi construída no topo do rio *Pregel* e incluía duas grandes ilhas que eram conectadas entre si e ao continente por sete pontes. O problema era conhecer a possibilidade em se cruzar cada uma das sete pontes de Königsberg apenas uma vez e visitar todas as partes da cidade.

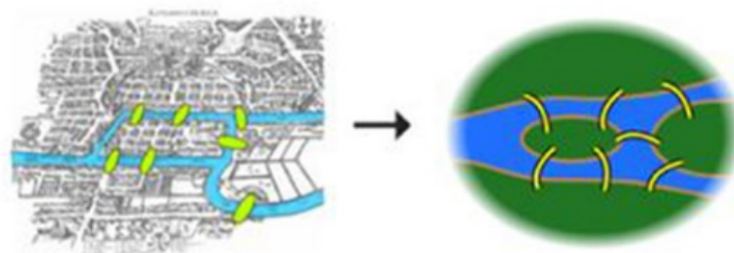


Figura 2: *As 7 Pontes de Königsberg*

Após abstrair o problema em um gráfico, Euler percebeu um padrão, baseado no número de vértices e arestas. No gráfico de Königsberg, existem 4 vértices e 7 arestas. No sentido literal, Euler percebeu que se caminhasse para uma das ilhas e sair para outra, usaria uma ponte de entrada e uma ponte de saída. Essencialmente, para atravessar um caminho sem cruzar uma aresta mais de uma vez, seria então necessário um número par de arestas.

Euler teorizou que percorrer um gráfico inteiramente, usando cada aresta apenas uma vez, depende dos graus de um nó. No contexto de um nó ou vértice, graus se refere à quantidade de arestas que tocam o nó. Se percorrermos um gráfico completamente (usando apenas uma aresta por vez), podemos ter de 0 ou 2 nós com graus ímpares (cruzar um gráfico dessa forma é conhecido como *Caminho de Euler*).

Nesse contexto, um grafo é composto de nós (ou vértices) e potencialmente arestas que os conectam. Para visualizarmos isso, podemos usar uma seta para indicar que um nó está conectado a outro nó. Por exemplo, se tivéssemos dois nós, A e B, aos quais A está conectado a B com uma aresta, isso poderia ser expresso como $A \triangleright B$. A direção é mostrada aqui em que A está conectado a B, mas B não está conectado a A. Se as arestas que compõem um gráfico não têm uma direção associada (por exemplo, $A \triangleright B$ é o mesmo que $B \triangleright A$), então o gráfico é classificado como não direcionado. Se, no entanto, a orientação da aresta tiver algum significado, o gráfico é direcionado.



Figura 3: Cena da Série *Elementary* (CBS)

Existem outras aplicações para a teoria dos grafos fora do mundo da matemática. Uma vez que a teoria dos gráficos, em seu nível mais baixo, descreve como os dados se relacionam uns com os outros, pode ser aplicada a uma série de diferentes setores e cenários onde relacionar os dados é importante. Pode ser usada para mapear estruturas químicas, criar diagramas de estradas e até mesmo analisar dados de redes sociais. As aplicações são bastante amplas.

1.2 Criar o contêiner Docker

A forma mais simples para obtermos o Redis é através de um contêiner no Docker, deste modo podemos ter várias versões do banco instalada e controlar mais facilmente qual banco está ativo ou não. E ainda colhemos o benefício adicional de não termos absolutamente nada deixando sujeira em nosso sistema operacional ou áreas de memória.

Baixar a imagem oficial:

```
$ docker pull redis
```

Antes de criarmos o contêiner precisamos criar uma pasta em nosso disco para hospedar o banco, por exemplo minha pasta chama-se: `"/home/fernando/neo4j"`. Deste modo a criação do contêiner será:

```
$ docker run -d -p 7687:7687 -p 7474:7474 -v $HOME/neo4j/data:/data
-v $HOME/neo4j/logs:/logs -v $HOME/neo4j/import:/import
-v $HOME/neo4j/plugins:/plugins --env NEO4J_AUTH=neo4j/test --name meu-neo4j neo4j
```

Podemos entrar no *Cypher-shell* dentro do nosso contêiner:

```
$ docker exec -it meu-neo4j bash
```

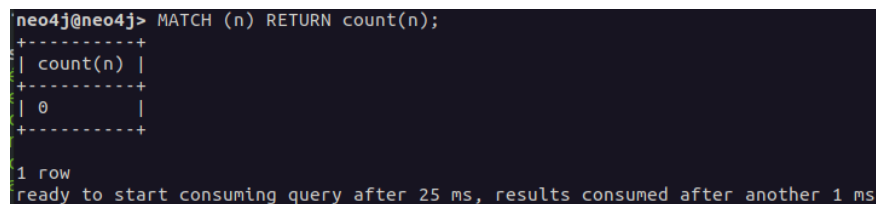
Acessar o Cypher-shell:

```
$ cypher-shell -u neo4j -p test
```

Quando criamos o contêiner usamos a opção: `--env NEO4J_AUTH=neo4j/test`, neste ponto indicamos a senha **test** para o *Cypher-shell* e devemos usá-la quando entramos neste. Executamos uma consulta para verificarmos que está tudo OK:

```
neo4j> MATCH (n) RETURN count(n);
```

E obtemos:



```
neo4j@neo4j> MATCH (n) RETURN count(n);
+-----+
| count(n) |
+-----+
| 0        |
+-----+
1 row
ready to start consuming query after 25 ms, results consumed after another 1 ms
```

Figura 4: Execução do comando no *Cypher-shell*

Podemos sair do *Cypher-shell* com o comando:

```
neo4j> :exit
```

E com o comando:

```
$ exit
```

Para sairmos do contêiner. Na seção **Cypher** investigaremos mais sobre essa linguagem. Podemos parar o contêiner com:

```
$ docker stop meu-neo4j
```

Ou iniciá-lo novamente:

```
$ docker start meu-neo4j
```

1.3 Gerenciar o Neo4J

Uma vez que subimos nosso contêiner, após alguns segundos, está a nossa disposição uma das ferramentas mais úteis incluídas em um banco de dados o "Neo4j Browser", no endereço: <http://localhost:7474>.

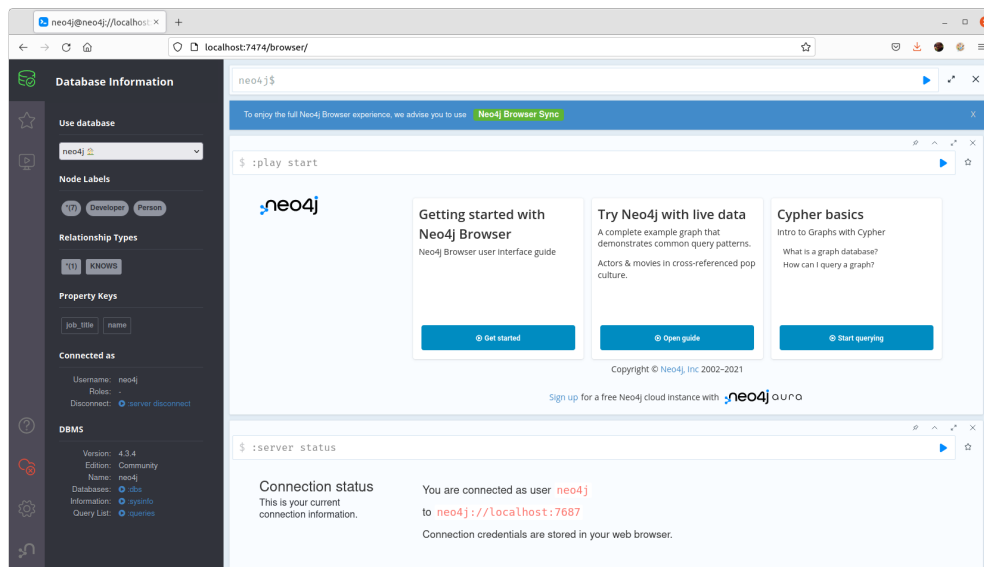


Figura 5: *Neo4j Browser*

É possível com este aplicativo realizar melhorias significativas para os recursos, velocidade e testes no banco através dessa ferramenta de visualização com base na Web. Na barra superior da janela principal, devemos dar o máximo de atenção na indicação **neo4j\$** pois neste ponto podemos inserir comandos em linguagem **Cypher**.

2 Cypher

A linguagem utilizada para trabalhar com o Neo4j chama-se Cypher, a primeira representação que devemos conhecer é:

- **Parênteses** devem ser usados para os nós.
- **Chaves** devem ser usadas para as propriedades.
- **Colchetes** devem ser usados para os relacionamentos.

Uma dica simples podemos tentar comparar com os comandos SQL, porém essa linguagem nada tem a ver, por exemplo, um comando básico usado frequentemente é:

```
1 MATCH (a)-[:DONA]->(b) RETURN a, b;
```

Realiza uma consulta que mostra como obter todos os nós que estão relacionados entre si por um determinado relacionamento chamado **DONA**. Mas vamos começar com muita calma e ir avançado, para executar qualquer comando, na barra de comandos do *Neo4j Browser* digitamos:

```
1 MATCH (n) RETURN n;
```

E pressionamos o botão para executar, o resultado deve ser algo assim:

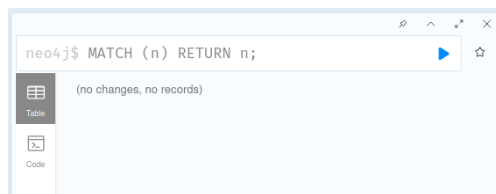


Figura 6: Execução do comando no Neo4j Browser

Essa consulta nos mostra que não existe nenhum banco ou registro cadastrado. Em seguida, veremos uma estrutura completa do banco de dados, para facilitar o entendimento tentaremos associar isso aos comandos SQL.

2.1 Comando CREATE / CREATE UNIQUE

Este comando é utilizado para criarmos qualquer elemento no banco de dados: nós, relacionamentos, propriedades ou combinações entre eles. Podemos associá-lo com **INSERT**, porém age de forma bem diferente do SQL, primeiro um Nó (que seria o correspondente a uma entidade) pode ter vários nomes, por exemplo:

```
1 CREATE (:Pessoa:Proprietario);
```

Podemos criar algumas propriedades para este nó:

```
1 CREATE (:Pessoa { nome : 'Fernando', modelo_carteira : "AB" });
2 CREATE (:Pessoa { nome : 'Anselmo', modelo_carteira : "B" });
3 CREATE (:Automovel { nome : 'Ecosport' });
```

Os relacionamentos são criados assim, para a pessoa com o nome "Fernando":

```
1 MATCH (p:Pessoa { nome: "Fernando" }) OPTIONAL MATCH (a:Automovel { nome: "Ecosport" })
   CREATE (p)-[:DONA]->(a)
```

E para a pessoa com o nome "Anselmo":

```
1 MATCH (p:Pessoa { nome: "Anselmo" }) OPTIONAL MATCH (a:Automovel { nome: "Ecosport" })
   CREATE (p)-[:DONA]->(a)
```

E com o comando:

```
1 MATCH (n) RETURN n;
```

Podemos visualizar o seguinte Grafo:

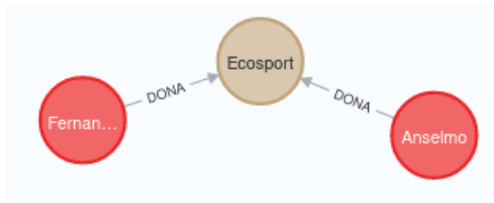


Figura 7: Grafo entre Pessoa e Automóvel

Também podemos criar simultaneamente todas as pessoas e seus relacionamentos com um único comando que seria:

```
1 CREATE (:Pessoa {nome: "Fernando", modelo_carteira : "AB"})-[:DONA]->(:Automovel {nome:
    "Ecosport"})<-[:DONA]-(:Pessoa {nome: "Anselmo", modelo_carteira : "B"});
```

2.2 Comando MATCH - Subcomando SET

Normalmente o comando **MATCH** é associado com o **SELECT**, porém faz muito mais já o vimos associado ao **CREATE**, com sua associação a SET podemos adicionar ou criar propriedades nos nós:

```
1 MATCH (n) WHERE n.nome = "Fernando" SET n.profissao = "Desenvolvedor"
```

Se reparar na sintaxe podemos fazer uma associação ao comando **UPDATE**.

2.3 Comando MATCH - Para consultas

Nota-se que este é o principal comando em Cypher, o comando MATCH é extremamente versátil, para retornarmos todas as pessoas com um nome específico:

```
1 MATCH (n:Pessoa {nome: "Fernando"}) RETURN n;
```

Retornar um conjunto específico de propriedades (não todas):

```
1 MATCH (p:Pessoa {nome: "Fernando"})
2 RETURN p.modelo_carteira, p.profissao
```

Retornar todas as pessoas que não possuem o valor de alguma propriedade específica:

```
1 MATCH (p:Pessoa)
2 WHERE NOT p.nome = 'Anselmo'
3 RETURN j
```

A subcláusula WHERE funciona quase da mesma forma que o SQL, por exemplo, vamos supor que as pessoas tivessem uma propriedade "experiencia" (em anos) e desejamos conhecer todas em um determinado intervalo:

```
1 MATCH (p:Pessoa)
2 WHERE 3 <= p.experiencia <= 7
```

```
3 RETURN p
```

Ou para avaliar a existência de um determinado valor em um relacionamo:

```
1 MATCH (p:Pessoa)-[rel:DONA]->(a:Automovel)
2 WHERE p.modelo_carteira IS NOT NULL
3 RETURN p, rel, a;
```

Para combinarmos os nós que estão relacionados, independentemente da direção (observe a falta de uma seta) e retornar de ambos os lados:

```
1 MATCH (a)--(b) RETURN a, b;
```

Podemos atribuir os relacionamentos a uma variável "r", e isso significa que podem ser retornados da consulta, portanto, se precisamos do relacionamento (ou qualquer uma de suas propriedades):

```
1 MATCH (a)-[r]-(b) RETURN a, r, b;
```

Quando não desejamos caracterizar tudo, ou seja somente um certo tipo de relacionamento, adicionamos o padrão ":TIPO". Nesse caso, o tipo é DONA e, como o relacionamento não é necessário posteriormente, o alias é descartado:

```
1 MATCH (a)-[:DONA]->(b) RETURN a, b;
```

Em vez de apenas retornar todos os nós envolvidos em um caminho, podemos obter o próprio caminho. Com base no exemplo anterior, isso será transformado em um caminho nomeado:

```
1 MATCH p=(a)-[:DONA]->(b) RETURN p;
```

É possível usar várias cláusulas MATCH em uma consulta, portanto, para retornar dois nós específicos, podemos usar várias causas de correspondência e, em seguida, retornar o resultado:

```
1 MATCH (a:Pessoa {nome: 'Fernando'})
2 MATCH (b:Pessoa {nome: 'Anselmo'})
3 RETURN a, b;
```

Que podemos simplificar para:

```
1 MATCH (a:Pessoa {nome: 'Fernando'}),(b:Pessoa {nome: 'Anselmo'}) RETURN a, b;
```

Esse comando é extremamente importante pois mostra bem o princípio que discutimos sobre transações, retorna os nós solicitados exatamente como esperamos. Se o segundo MATCH falhar, então o primeiro também falha, e a consulta retorna 0 resultados, pois está procurando por um **AND** também b, portanto, caso não conseguir encontrar nada em "b" então a consulta não é válida.

Podemos contornar isso, com a utilização de uma correspondência opcional. Retorna somente se a correspondência estiver presente. Do contrário, retornará **null**:

```
1 MATCH (a:Pessoa {nome: 'Fernando'})
2 OPTIONAL MATCH (b:Pessoa {nome: 'Anselmo'})
```



```
3 RETURN a, b;
```

Este sinalizador **OPTIONAL MATCH** é utilizado para retornar relacionamentos potenciais para um nó. Se apenas o nó pode ter um relacionamento, pode ser usado para remediar isso, assim:

```
1 MATCH (a:Pessoa {nome: 'Fernando'})
2 OPTIONAL MATCH (a)-->(x)
3 RETURN a, x;
```

Assim para todos os nós rotulados como "Pessoa" com o nome de "Fernando", ambos os nós que possuem ou não relacionamentos serão retornados.

2.4 Comando MATCH - Subcomando DELETE

Vamos começar limpando nós ou relacionamentos que não possuem associações:

```
1 MATCH (n) WHERE NOT (n)--() DELETE n
```

Observamos então que basta fazer uma consulta e usar o DELETE para eliminar o que desejamos. Por exemplo, vamos excluir um relacionamento determinado:

```
1 MATCH (p:Pessoa {nome: "Fernando"})-[rel:DONA]->(a:Automovel {nome: "Ecosport"})
2 DELETE rel
```

Eliminar um nó determinado (deve estar sem relacionamentos):

```
1 MATCH (p:Pessoa {nome: "Fernando"}) DELETE p
```

Eliminar todos com o comando (seria um correspondente a **DROP DATABASE**):

```
1 \label{MATCH (n) DETACH DELETE n}
```

3 Conclusão

Neo4j é significativamente mais rápido na consulta de dados relacionados do que usar um banco de dados relacional tradicional. Além disso, uma única instância do Neo4j pode lidar com conjuntos de dados contendo três ordens de magnitude sem penalidades de desempenho. A independência do desempenho transversal no tamanho do gráfico é um dos principais aspectos que tornam o **Neo4j** torna-se um candidato ideal para resolver problemas de grafos, mesmo quando os conjuntos de dados são muito grandes.

Para auxiliar **Neo4j** a ser o mais rápido possível, dois sistemas de cache diferentes são usados: um cache de buffer e um de objetos. O primeiro tem como objetivo acelerar as consultas, ao armazenar uma cópia das informações recuperadas do gráfico, enquanto que o de objetos armazena versões otimizadas de nós, propriedades e relacionamentos para acelerar a travessia do grafo.

As principais linguagens de programação possuem suporte e aqui vimos apenas Java e Python,

porém existem muitas outras como PHP, C, C++, C#, JavaScript, Node.js, Ruby, R e Go. Esta apostila faz parte da série dos quatro tipos para Bancos de Dados no padrão NoSQL que estou tentando desmistificar e torná-los mais acessíveis tanto para as comunidades de Java e Python voltada especificamente para desenvolvedores ou cientistas de dados.

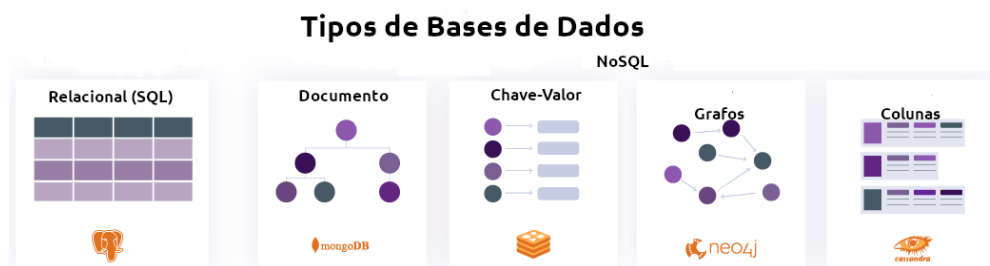


Figura 8: *Tipos de Bancos de Dados*

Sou um entusiasta do mundo **Open Source** e novas tecnologias. Qual a diferença entre Livre e Open Source? Livre significa que esta apostila é gratuita e pode ser compartilhada a vontade. Open Source além de livre todos os arquivos que permitem a geração desta (chamados de arquivos fontes) devem ser disponibilizados para que qualquer pessoa possa modificar ao seu prazer, gerar novas, complementar ou fazer o que quiser. Os fontes da apostila (que foi produzida com o LaTeX) está disponibilizado no GitHub [8]. Veja ainda outros artigos que publico sobre tecnologia através do meu Blog Oficial [6].

Referências

- [1] Página do Neo4j
<https://neo4j.com/>
- [2] Página do Oracle Java
<http://www.oracle.com/technetwork/java>
- [3] Página do Python
<https://www.python.org>
- [4] Editor Spring Tool Suite para códigos Java
<https://spring.io/tools>
- [5] Página do Jupyter
<https://jupyter.org/>
- [6] Fernando Anselmo - Blog Oficial de Tecnologia
<http://www.fernandoanselmo.blogspot.com.br/>
- [7] Encontre essa e outras publicações em
<https://cetrex.academia.edu/FernandoAnselmo>
- [8] Repositório para os fontes da apostila
<https://github.com/fernandoans/publicacoes>