

---

# Neo4j com Java e Python

**Fernando Anselmo**

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

---

Versão 1.0 em 30 de novembro de 2021

## Resumo

**N**eo4j 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:

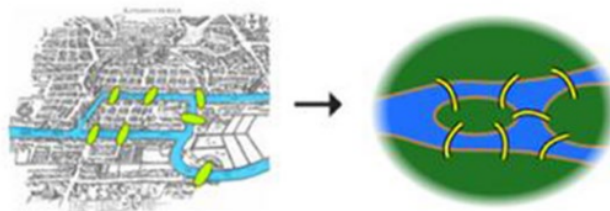
- **Disponibilidade** cada solicitação recebe uma resposta sobre se foi bem-sucedida ou falha.
- **Consistência** os dados estão disponíveis para todos os nós ao mesmo tempo.
- **Tolerância de partição** o sistema ainda pode operar apesar de perder contato com outros nós devido a problemas de rede.

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.

No Neo4j todos esses conceitos estão amplamente aplicados:

- **Nó** representamos entidades com Nó. Identificamos ao criar nosso modelo de dados. Junto com os relacionamentos, nó é uma das unidades fundamentais que formam um gráfico.
- **Propriedades** representam os atributos do nó e relacionamentos. São pares nome-valor usados para adicionar qualidades a nós ou relacionamentos.
- **Rótulo (label) do nó** Podemos atribuir funções ou tipos a um nó usando um ou mais rótulos. São utilizados para agrupar nós onde todos com o mesmo rótulo pertencem ao mesmo grupo.
- **Relação** Usamos um relacionamento para conectar dois nós. Deve ter exatamente um tipo de relacionamento.

## 1.2 Criar o contêiner Docker

A forma mais simples para obtermos o Neo4J 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 neo4j
```

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
```

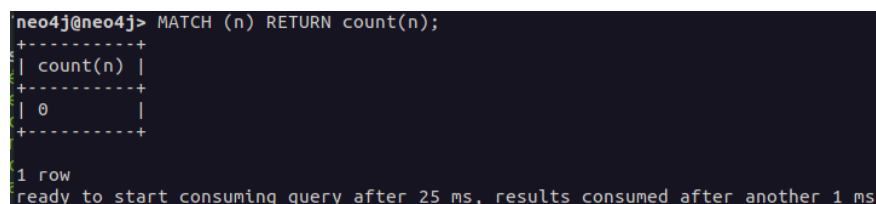
Acessar o Cypher-shell:

```
$ 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`.

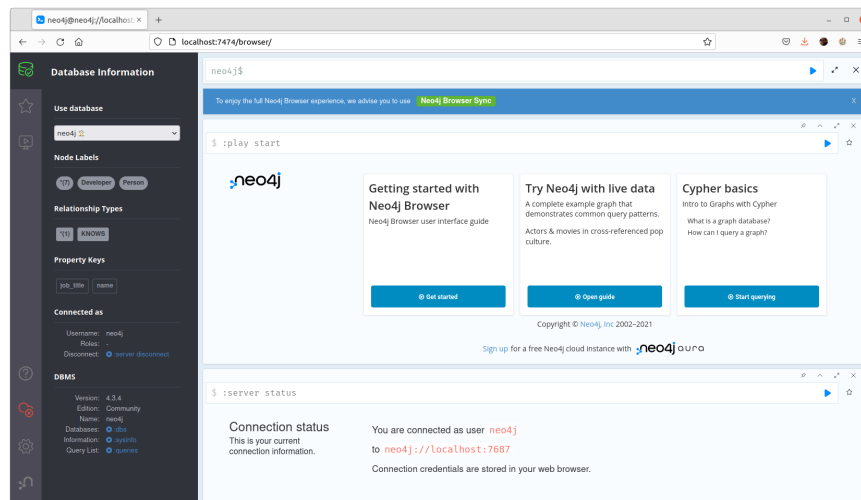


Figura 5: Neo4j Browser

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.

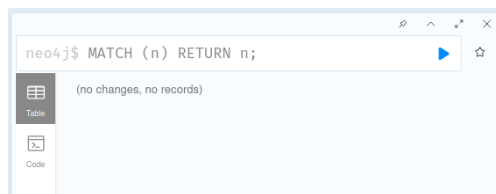
Uma dica simples podemos tentar comparar com os comandos SQL, porém essa linguagem nada tem a ver, por exemplo, um comando básico usado frequentemente é:

```
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)
```

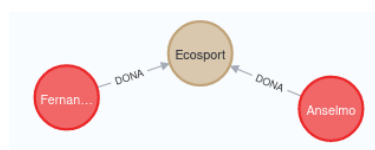
E para a pessoa com o nome "Anselmo":

```
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 MATCH (n) DETACH DELETE n
```

## 3 Linguagem Java

Java é considerada a linguagem de programação orientada a objetos mais utilizada no Mundo, base para a construção de ferramentas como Hadoop, Pentaho, Weka e muitas outras utilizadas comercialmente. Foi desenvolvida na década de 90 por uma equipe de programadores chefiada por *James Gosling* para o projeto Green, na Sun Microsystems - tornou-se nessa época como a linguagem que os programadores mais baixaram e o sucesso foi instantâneo. Em 2008 o Java foi adquirido pela Oracle Corporation.

### 3.1 Driver JDBC de Conexão

Para proceder a conexão com Java, é necessário baixar um driver JDBC (Java Database Connection). Existem vários drivers construídos. Para utilizar o driver é necessário criar um projeto (vamos usar o **Spring Tool Suite 4**, utilize se quiser qualquer outro editor de sua preferência).

No STS4 acessar a seguinte opção no menu: File ▸ New ▸ Java Project. Informar o nome do projeto (TesteNeo4j), não esquecer de modificar a opção "Use an environment JRE" para a versão correta da Java Runtime desejada e pressionar o botão Finish.

Agora devemos convertê-lo para um projeto Apache Maven. Clicar com o botão direito do mouse no projeto e acessar a opção: Configure ▸ Convert to Maven Project. Na janela apenas pressione o botão *Finish*. Se tudo está correto observamos que o projeto ganhou uma letra **M** o que indica agora é um projeto padrão Maven. Então foi criado um arquivo chamado **pom.xml**.

Acessar este arquivo e antes da tag BUILD, inserir a tag DEPENDENCIES:

```
1 <dependencies>
2   <dependency>
3     <groupId>org.neo4j.driver</groupId>
4     <artifactId>neo4j-java-driver</artifactId>
5     <version>4.3.4</version>
6   </dependency>
7 </dependencies>
```

Agora a situação do projeto é esta:

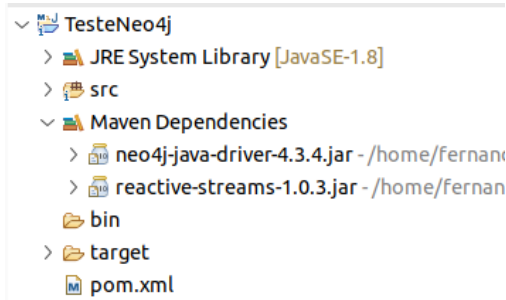


Figura 8: Dependências do Maven

Observamos que na pasta **Maven Dependencias** foi baixado a versão 2.8.1 do driver Jedis - Isso não é erro de digitação realmente o driver se escreve com "J".

## 3.2 Exemplo Prático

Estamos prontos para testarmos a conexão entre Redis e Java. Criamos um pequeno exemplo que nos auxiliará como teste, uma classe chamada **Hello** no pacote **decus.com** e inserimos nesta a seguinte codificação:

```
1 package decus.com;
2
3 import static org.neo4j.driver.Values.parameters;
4
5 import org.neo4j.driver.AuthTokens;
6 import org.neo4j.driver.Driver;
7 import org.neo4j.driver.GraphDatabase;
8 import org.neo4j.driver.Result;
9 import org.neo4j.driver.Session;
10 import org.neo4j.driver.Transaction;
11 import org.neo4j.driver.TransactionWork;
12
13 public class Exemplo implements AutoCloseable {
14     private final Driver driver;
15
16     public Exemplo(String uri, String user, String password) {
17         driver = GraphDatabase.driver(uri, AuthTokens.basic(user, password));
18     }
19
20     @Override
21     public void close() throws Exception {
```

```

22     driver.close();
23 }
24
25 public void exemplo() {
26     try (Session session = driver.session()) {
27         String saudacao = session.writeTransaction(new TransactionWork<String>() {
28             @Override
29             public String execute(Transaction tx) {
30                 Result result = tx.run(
31                     "CREATE (s:Saudacao) SET s.mensagem = $mensagem " +
32                     "RETURN s.mensagem + ', from node ' + id(s)",
33                     parameters("mensagem", "Hello World!"));
34                 return result.single().get(0).asString();
35             }
36         });
37         System.out.println(saudacao);
38     }
39 }
40
41 public static void main(String... args) throws Exception {
42     try (Exemplo hello = new Exemplo("bolt://localhost:7687", "neo4j", "test")) {
43         hello.exemplo();
44     }
45 }
46 }

```

Penso que usar o Neo4j para escrever "Hello World!" seja demais, porém pensemos nisso como exemplo para entendermos como tudo funciona, no construtor da classe passamos os parâmetros para nos conectarmos ao driver, por implementação da *AutoCloseable* sobrepomos o método *close()* que será responsável por encerrar a conexão. Começando nossa execução pelo método *main()* criamos um objeto da classe no qual definimos a URI (caminho de conexão), usuário e senha (que criamos para nosso contêiner) e chamamos o método *exemplo()*. Esse inicia uma sessão que implementa um do tipo *TransactionWork*, exige uma implementação do método *execute(Transaction)* e nesse passamos o comando Cypher que desejamos enviar para o Neo4j.

Foi como usar um canhão para matar um mosquito, porém estrutura básica é esta e a partir dela faremos as modificações para compreendermos como funciona a comunicação. Vamos mudar o método *exemplo()* para a seguinte codificação:

```

1  public void exemplo() {
2      try (Session session = driver.session()) {
3          session.run("CREATE (:Pessoa {nome: $nome, modelo_carteira : $modelo})",
4              parameters("nome", "Fernando", "modelo", "AB")
5          );
6          session.run("CREATE (:Pessoa {nome: $nome, modelo_carteira : $modelo})",
7              parameters("nome", "Anselmo", "modelo", "A")
8          );
9          session.run("CREATE (:Automovel {nome: $nome})",
10             parameters("nome", "Ecosport")
11          );
12     }
13 }

```

Assim teremos nossos dados criados sem muitas modificações conforme vimos na seção **Cypher**. Para criarmos os relacionamentos vamos fazer um pouco diferente (novamente substituindo o método

*exemplo()*:

```
1 public void exemplo() {
2     final String nomeAutomovel = "Ecosport";
3     final String [] nomes = {"Fernando", "Anselmo"};
4     for (String nomePessoa: nomes) {
5         try (Session session = driver.session()) {
6             session.run("MATCH (p:Pessoa { nome: $nomePessoa }) "
7                 + "OPTIONAL MATCH (a:Automovel { nome: $nomeAutomovel }) "
8                 + "CREATE (p)-[:DONA]->(a)",
9                 parameters("nomePessoa", nomePessoa, "nomeAutomovel", nomeAutomovel)
10            );
11        }
12    }
13 }
```

E podemos trazer os dados com (novamente substitua o método *exemplo()*):

```
1 private Result showBase(final Transaction tx) {
2     Result result = tx.run("MATCH (a)-[:DONA]->(b) RETURN a.nome, b.nome");
3     while (result.hasNext()) {
4         Record r = result.next();
5         System.out
6             .println(String.format("%s proprietário de %s", r.get("a.nome").asString(),
7                 r.get("b.nome").toString()));
8     }
9     return result;
10 }
11 private void exemplo() {
12     try (Session session = driver.session()) {
13         session.writeTransaction(tx -> showBase(tx));
14     }
15 }
```

No método *exemplo()* criamos a seção que através de uma transação executará o método *showBase()*, nesse temos a execução de um comando **MATCH** que retorna os relacionamentos encontrados, basicamente este objeto da classe **Result** pode ser comparado com um MAPA, basta percorrê-lo e obter os resultados com um **get**.

Resumidamente, tudo que vimos na seção **Cypher** podemos aplicar com a linguagem Java, porém antes de encerrarmos esta parte vamos criar uma nova classe e fazer um programa que com toda certeza seria um modelo completo ideal para iniciarmos nossas explorações.

### 3.3 Locadora Java

Neste exemplo um pouco mais completo temos um interessante ponto de partida para a construção do que pode ser um sistema sobre Aluguel de Veículos que pode abranger relacionamentos entre vendedores, clientes e empresas.

```
1 package decus.com;
2
3 import static org.neo4j.driver.SessionConfig.builder;
4 import static org.neo4j.driver.Values.parameters;
```

```

5 import java.util.ArrayList;
6 import java.util.List;
7 import org.neo4j.driver.AccessMode;
8 import org.neo4j.driver.AuthTokens;
9 import org.neo4j.driver.Bookmark;
10 import org.neo4j.driver.Driver;
11 import org.neo4j.driver.GraphDatabase;
12 import org.neo4j.driver.Record;
13 import org.neo4j.driver.Result;
14 import org.neo4j.driver.Session;
15 import org.neo4j.driver.Transaction;
16
17 public class Locadora implements AutoCloseable {
18
19     private final Driver driver;
20
21     public Locadora(String uri, String user, String password) {
22         driver = GraphDatabase.driver(uri, AuthTokens.basic(user, password));
23     }
24
25     @Override
26     public void close() throws Exception {
27         driver.close();
28     }
29
30     private Result addEmpresa(final Transaction tx, final String nome) {
31         return tx.run("CREATE (:Empresa {nome: $nome})", parameters("nome", nome));
32     }
33
34     private Result addPessoa(final Transaction tx, final String nome) {
35         return tx.run("CREATE (:Pessoa {nome: $nome})", parameters("nome", nome));
36     }
37
38     private Result empregado(final Transaction tx, final String pessoa, final String
        empresa) {
39         return tx.run("MATCH (p:Pessoa {nome: $pessoa_nome}) " +
40             "MATCH (e:Empresa {nome: $empresa_nome}) " +
41             "CREATE (p)-[:TRABALHA_PARA]->(e)",
42             parameters("pessoa_nome", pessoa, "empresa_nome", empresa));
43     }
44
45     private Result fazerAmigos(final Transaction tx, final String pessoa1, final String
        pessoa2) {
46         return tx.run(
47             "MATCH (a:Pessoa {nome: $pessoa_1}) " +
48             "MATCH (b:Pessoa {nome: $pessoa_2}) " +
49             "MERGE (a)-[:CONHECE]->(b)",
50             parameters("pessoa_1", pessoa1, "pessoa_2", pessoa2));
51     }
52
53     private Result mostrarAmizade(final Transaction tx) {
54         Result result = tx.run("MATCH (a)-[:CONHECE]->(b) RETURN a.nome, b.nome");
55         while (result.hasNext()) {
56             Record r = result.next();
57             System.out.println(String.format("%s conhece %s",
58                 r.get("a.nome").asString(),
59                 r.get("b.nome").toString()));
60         }

```

```

61     return result;
62 }
63
64 public void construir() {
65     List<Bookmark> savedBookmarks = new ArrayList<>();
66     try (Session s1 = driver.session(
67         builder()
68             .withDefaultAccessMode(AccessMode.WRITE).build())) {
69         s1.writeTransaction(tx -> addEmpresa(tx, "Bros Carros"));
70         s1.writeTransaction(tx -> addPessoa(tx, "Mario"));
71         s1.writeTransaction(tx -> empregado(tx, "Mario", "Bros Carros"));
72         savedBookmarks.add(s1.lastBookmark());
73     }
74     try (Session s2 = driver.session(
75         builder()
76             .withDefaultAccessMode(AccessMode.WRITE).build())) {
77         s2.writeTransaction(tx -> addEmpresa(tx, "Usina Carros"));
78         s2.writeTransaction(tx -> addPessoa(tx, "Luigi"));
79         s2.writeTransaction(tx -> empregado(tx, "Luigi", "Usina Carros"));
80         savedBookmarks.add(s2.lastBookmark());
81     }
82     try (Session s3 = driver.session(
83         builder()
84             .withDefaultAccessMode(AccessMode.WRITE)
85             .withBookmarks(savedBookmarks).build())) {
86         s3.writeTransaction(tx -> fazerAmigos(tx, "Mario", "Luigi"));
87         s3.readTransaction(this::mostrarAmizade);
88     }
89 }
90
91 public static void main(String... args) throws Exception {
92     try (Locadora loc = new Locadora("bolt://localhost:7687", "neo4j", "test")) {
93         loc.construir();
94     }
95 }
96 }

```

E ao término da execução dessa classe o seguinte relacionamento será montado:



**Figura 9:** *Relacionamentos da Locadora*

Observaremos cada seção no método **construir()** é formado por um conjunto lógico de chamadas, com uma execução em formato de *Pipeline* para cada transação a ser chamada dentro de uma seção. Creio que essa ser a forma mais elegante de se construir em Java.

## 4 Python

Python é uma linguagem de programação de alto nível, interpretada a partir de um script, Orientada a Objetos e de tipagem dinâmica. Foi lançada por Guido van Rossum em 1991. Não pretendo nesta apostila COMPARAR essa linguagem com Java (espero que nunca o faça), fica claro que os comandos são bem mais fáceis porém essas linguagens possuem diferentes propósitos.

Todos os comandos descritos abaixo foi utilizado no JupyterLab [5], então basta abrir um Notebook e digitá-los em cada célula conforme se apresentam.

### 4.1 Pacote Necessário

Baixar o pacote necessário:

```
!pip install neo4j
```

Importar os pacotes necessários:

```
from neo4j import GraphDatabase
```

Para realizarmos a manipulação dos dados a melhor forma é criarmos uma classe que terá a seguinte estrutura:

```
1 class Teste:
2     def __init__(self, uri, usuario, senha):
3         try:
4             self.driver = GraphDatabase.driver(uri, auth=(usuario, senha))
5         except Exception as e:
6             print("Falhou para acessar o Driver:", e)
7
8     def close(self):
9         self.driver.close()
```

Nossa classe Teste possui o construtor que recebe URI (endereço), usuário e a senha de acesso ao banco e realiza essa conexão através da classe *GraphDatabase*. Ao encerrar o objeto a conexão é encerrada. Implementamos a chamada principal:

```
1 if __name__ == "__main__":
2     app = Teste("bolt://localhost:7687", "neo4j", "test")
3     app.close()
```

Agora podemos testar a vontade sabendo que existem 2 métodos principais: *write\_transaction* que nos permite realizar as operações de gravação na base de dados e *read\_transaction* para realizarmos as transações de consulta. Vamos começar inserindo as pessoas na base, para tanto na classe abaixo do método *close()*, inserimos a seguinte codificação:

```
1     def criarPessoa(self, nome_pessoa, modelo):
2         with self.driver.session() as session:
3             session.write_transaction(
4                 self._criarPessoa, nome_pessoa, modelo)
5
6     @staticmethod
7     def _criarPessoa(tx, nome_pessoa, modelo):
8         tx.run("CREATE (:Pessoa {nome: $n1, modelo_carteira: $m1})", n1=nome_pessoa,
```

```
m1=modelo)
```

O primeiro método abre uma seção com o *Driver* em seguida uma transação de escrita e chama o segundo método para proceder a inclusão do registro em linguagem *Cypher*. Para executar na chamada principal entre a criação do objeto *app* e seu fechamento, inserimos:

```
1 app.criarPessoa('Fernando', 'AB')
2 app.criarPessoa('Anselmo', 'B')
```

Executamos e comentamos essas chamadas pois se rodarmos novamente criará mais objetos. Para testarmos de forma simples a leitura criamos mais 2 métodos na classe:

```
1 def obterPessoas(self):
2     with self.driver.session() as session:
3         result = session.read_transaction(self._obterPessoas)
4         return result
5
6 @staticmethod
7 def _obterPessoas(tx):
8     result = tx.run("MATCH (p:Pessoa) RETURN p.nome AS nome")
9     return [record["nome"] for record in result]
```

E inserimos a chamada deste no ponto que comentamos as chamadas anteriores:

```
1 print(app.obterPessoas())
```

E como resposta teremos a lista de pessoas registradas:  
['Anselmo', 'Fernando']

## 4.2 Sem utilizar uma Estrutura de Classe

Não necessitamos utilizar uma estrutura de classe para criarmos nossas transações basta que elas sejam executadas através de uma seção única, vamos criar um exemplo mais completo para demonstrar isso:

```
1 from neo4j import GraphDatabase
2
3 graphDB_Driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "test"))
4
5 # Consultas
6 cqlUniversidades = "MATCH (u:universidade) RETURN u.nome"
7 cqlDistancia = "MATCH (p:professor {nome:'Fernando Anselmo'})-[r]->(u:universidade)
8                 RETURN u.nome,r.km"
9
10 # Criação dos registros
11 cqlCriar = """CREATE (p:professor { nome: "Fernando Anselmo"}),
12 (s:universidade {nome: "IESB Campus Sul"}),
13 (n:universidade {nome: "IESB Campus Norte"}),
14 (c:universidade {nome: "IESB Campus Ceilândia"}),
15 (p)-[:PERCORRE {km: 8}]->(s),
16 (p)-[:PERCORRE {km: 1}]->(n),
17 (p)-[:PERCORRE {km: 18}]->(c)"""
```



```

17
18 with graphDB_Driver.session() as secao:
19     secao.run(cqlCriar)
20     nodes = secao.run(cqlUniversidades)
21
22     print("Lista dos Campus do IESB:")
23     for node in nodes:
24         print(node["u.nome"])
25
26     nodes = secao.run(cqlDistancia)
27     print("\nDistância Percorrida:")
28     for node in nodes:
29         print("Para ir ao", node["u.nome"], "percorre", node["r.km"], "Km")

```

Criamos a conexão com o banco, e definimos 3 variáveis (apenas para um melhor entendimento): **cqlUniversidades** que mostra os campus criados, **cqlDistancia** que mostra a distância percorrida por um determinado professor e **cqlCriar** que cria todos os registros. Abrimos uma seção com **with** e podemos rodar qualquer comando para obter os resultados desejados. E como saída teremos:

Lista dos Campus do IESB:

IESB Campus Norte

IESB Campus Sul

IESB Campus Ceilândia

Distância Percorrida:

Para ir ao IESB Campus Norte percorre 1 Km

Para ir ao IESB Campus Sul percorre 8 Km

Para ir ao IESB Campus Ceilândia percorre 18 Km

Não pretendo de modo nenhum, nessa apostila, repetir mais do mesmo, sendo assim já temos o básico para realizar todos os exemplos que foram mostrados anteriormente. Se deseja ir além existem excelentes tutoriais na Internet e entre eles recomendo:

- **Create a graph database in Neo4j using Python** de CJ Sullivan  
<https://towardsdatascience.com/create-a-graph-database-in-neo4j-using-python-4172d40f89c4>
- **Build a Restaurant Recommendation Engine Using Neo4j** de Ng Wai Foong  
<https://betterprogramming.pub/build-a-restaurant-recommendation-engine-using-neo4j-9d13ebdd4736>
- **Neo4j Movies Application** do Github de Neoj-examples  
<https://github.com/neo4j-examples/movies-python-bolt>
- **Building a Career Recommendation Engine With Neo4j** de Otavio Santana  
<https://dzone.com/articles/when-neo4j-faces-java-ee-ops-ee4j>

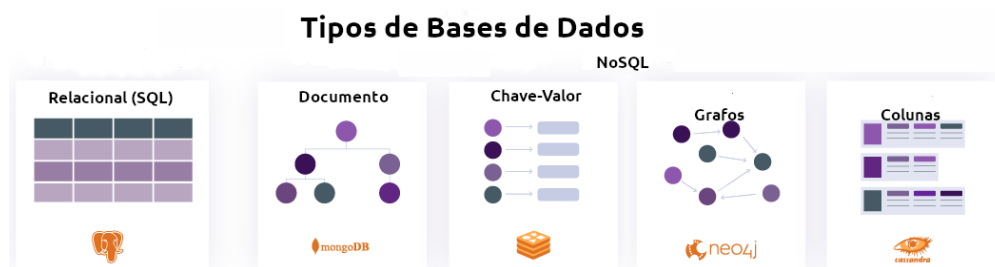
## 5 Conclusão

**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 10:** *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>