

ARQUITETURA NOSQL

GRAPH - BASED *DATABASES* (NEO4J)

MARCELO MANZANO E REGINA CANTELE



5

LISTA DE FIGURAS

Figura 5.1 – Leonhard Euler.....	6
Figura 5.2 – As 7 pontes de Königsberg	7
Figura 5.3 – Grafo	7
Figura 5.4 – Grafos e seus usos	9
Figura 5.5 – Graph Technology Landscape 2020	11
Figura 5.6 – Db-Engines NoSQL Graph	11
Figura 5.7 – Amazon Neptune – Ciências Biológicas.....	13
Figura 5.8 – História e funcionalidades	16
Figura 5.9 – Integrando dados de diversas bases de dados	17
Figura 5.10 -- Fabric com várias bases vistas como uma única.....	18
Figura 5.11 – Fabric e Sharding	19
Figura 5.12 – Fabric e Sharding com gerenciadores de bases distintas	20
Figura 5.13 – Neo4j e cluster	20
Figura 5.14 – Grafos e seus componentes.....	22
Figura 5.15 – Relacionamentos no ambiente grafo.....	23
Figura 5.16 – Relacionamento M:M	24
Figura 5.17 – Multirrelacionamentos no ambiente Grafo.....	24
Figura 5.18 – Nós e relacionamentos (1)	25
Figura 5.19 – Nós e relacionamentos (2)	26
Figura 5.20 – Grafos e seus componentes.....	27
Figura 5.21 – Acesso Sandbox Neo4j	28
Figura 5.22 – Neo4J Sandbox - Selecionando um projeto	29
Figura 5.23 – Base de dados criada.....	29
Figura 5.24 – Criando um novo projeto	30
Figura 5.25 – Sequência para criação e inicialização da base de dados	31
Figura 5.26 – Interface Gráfica – Editor de comandos	32
Figura 5.27 – Interface Gráfica – Stream/tela de resultados	32
Figura 5.28 – Interface Gráfica – Resultados - CODE.....	33
Figura 5.29 – Interface Gráfica – Resultados – TEXT e TABLE.....	33
Figura 5.30 – Sidebar	34
Figura 5.31 – Resultado CREATE.....	36
Figura 5.32 – Exemplo Cypher.....	36
Figura 5.33 – Exemplo Arquivo CSV sem cabeçalho.....	39
Figura 5.34 – Exemplo arquivo CSV com cabeçalho	39
Figura 5.35 – Interface gráfica web	40
Figura 5.36 – Modelo de Grafo “Movies”	41
Figura 5.37 – Modelo de Grafo.....	44
Figura 5.38 – ShortestPath.....	49
Figura 5.39 – Ativando APOC - Banco de Dados - Manage.....	56
Figura 5.40 – Ativando APOC - Plugins APOC	56
Figura 5.41 – Graph Data Science Library	57
Figura 5.42 – Workflow.....	58
Figura 5.43 – Neo4j GraphQL (1).....	60
Figura 5.44 – Neo4j GraphQL (2).....	61
Figura 5.45 – Neo4J Streams.....	62
Figura 5.46 – Subconjunto do arquivo JSON de aeroportos	63
Figura 5.47 – Resultado da ingestão de aeroportos.....	64

Figura 5.48 – Graph Apps	66
Figura 5.49 – Graph Data Science Playground	66
Figura 5.50 – Carga Game of Thrones.....	67
Figura 5.51 – Resultado gráfico da interação dos personagens	67
Figura 5.52 – Resultado gráfico do algoritmo Louvain	68
Figura 5.53 – Connected Components.....	69

EXEMPLO

LISTA DE CÓDIGOS-FONTE

Código-fonte 5.1 – Comando execução tutorial Movies	40
Código-fonte 5.2 – Exibir modelo de dados.....	40
Código-fonte 5.3 – Exemplo do uso do comando <code>Create node Movie</code>	41
Código-fonte 5.4 – Exemplo do uso do comando <code>Create node Person</code>	41
Código-fonte 5.5 – Exemplo do uso do comando <code>Create relationship</code>	41
Código-fonte 5.6 – Exemplo do uso do comando <code>Create relationship</code>	42
Código-fonte 5.7 – Exemplo do uso do comando <code>Create relationship</code>	42
Código-fonte 5.8 – Exemplo do uso do comando <code>Create relationship</code>	42
Código-fonte 5.9 – Exemplo do uso do comando <code>Match 1</code>	47
Código-fonte 5.10 – Exemplo do uso do comando <code>Match 2</code>	47
Código-fonte 5.11 – Exemplo do uso do comando <code>Match 3</code>	48
Código-fonte 5.12 – Exemplo do uso do comando <code>Match 4</code>	48
Código-fonte 5.13 – Exemplo do uso do comando <code>Match 5</code>	48
Código-fonte 5.14 – Exemplo do uso do comando <code>Match e shortestPath</code>	48
Código-fonte 5.15 – Exemplo do uso do comando <code>Match e Where</code>	49
Código-fonte 5.16 – Exemplo do uso do comando <code>Delete</code>	53
Código-fonte 5.17 – Exemplo do uso da cláusula <code>count (*)</code>	53
Código-fonte 5.18 – Exemplo do uso das cláusulas para agregação.....	53
Código-fonte 5.19 – Exemplo do uso do comando <code>Match e count (*) 1</code>	53
Código-fonte 5.20 – Exemplo do uso do comando <code>Match e count (*) 2</code>	53
Código-fonte 5.21 – Exemplo do uso do comando <code>Match 6</code>	54
Código-fonte 5.22 – Exemplo do uso da cláusula <code>allshortestpaths</code>	54
Código-fonte 5.23 – Exemplo do uso do comando <code>Set 1</code>	54
Código-fonte 5.24 – Exemplo do uso do comando <code>Set 2</code>	54
Código-fonte 5.25 – Exemplo do uso do comando <code>Set 3</code>	54
Código-fonte 5.26 – Exemplo do uso do comando <code>Delete</code>	55
Código-fonte 5.27 – Acessando módulo APOC	56
Código-fonte 5.28 – Consulta exemplo GraphQL.....	60
Código-fonte 5.29 – Resultado da consulta.....	61
Código-fonte 5.30 – Exemplo do uso do comando <code>Create Constraint e Index 1</code>	64
Código-fonte 5.31 – Exemplo do uso do comando <code>Create Constraint e Index 2</code>	64
Código-fonte 5.32 – Exemplo do uso do comando <code>Match</code>	65
Código-fonte 5.33 – Exemplo do uso do comando para calcular distância	65
Código-fonte 5.34 – Exemplo do uso do comando menor caminho entre aeroportos	65
Código-fonte 5.35 – Exemplo do uso do comando para calcular distância	65

SUMÁRIO

5 GRAPH-BASED DATABASES (NEO4J)	6
5.1 Introdução	6
5.2 Tecnologias	10
5.2.1 Amazon Neptune	12
5.2.2 Stardog	13
5.2.3 Giraph	14
5.2.4 Neo4j	15
5.2.4.1 Arquitetura e suas camadas	17
5.2.4.2 Características principais	21
5.2.4.3 Modelo de dados	23
5.3 Usando Neo4j	26
5.3.1 Neo4j sandbox	28
5.3.2 Neo4j Desktop	29
5.3.3 Interface gráfica	31
5.3.4 Criando Nós	34
5.3.5 Um pouco do Cypher	36
5.3.6 Tutorial Movies	39
5.3.7 Consultando os dados	43
5.3.8 Filtrando as consultas	46
5.3.9 Alterando os dados	49
5.3.10 Removendo os dados	51
5.3.11 Fixação dos comandos e consultas agregadas	53
5.3.12 Módulos e suas funcionalidades	55
5.3.12.1 APOC	55
5.3.12.2 Graph Data Science Library	57
5.3.12.3 GraphQL	59
5.3.12.4 Neo4j Streams	62
5.3.12.5 Exemplo Aeroportos	63
5.3.12.6 Exemplo Graph Data Science Library	65
REFERÊNCIAS	70

5 GRAPH-BASED DATABASES (NEO4J)

5.1 Introdução

A ideia de Grafo surgiu independente das diversas áreas de conhecimento, no entanto é considerada como uma área da matemática aplicada. A mais antiga menção sobre o assunto ocorreu no trabalho de Euler (pronuncia-se Óiler).



Figura 5.1 – Leonhard Euler
Fonte: Adaptado por FIAP (2020)

Assim, essa teoria surgiu pela primeira vez em 1736 e foi criada por Leonhard Euler para resolver um problema chamado de as 7 pontes de Königsberg (atual Kaliningrado). Seis delas interligavam duas ilhas às margens do Rio Pregel e uma que fazia a ligação entre as duas ilhas. O problema consistia na seguinte questão: como seria possível fazer um passeio a pé pela cidade de forma a se passar uma única vez por cada uma das sete pontes e retornar ao ponto de partida?

a



Figura 5.2 – As 7 pontes de Königsberg
Fonte: Adaptado por FIAP (2020)

Euler focou apenas no problema removendo todos os respectivos detalhes geométricos (distância, tamanho das pontes, formas etc.) e apresentou uma solução que, além de provar que não era possível passar apenas uma vez por ponte, mostrou como a forma de solução poderia ser aplicada a outros problemas semelhantes. Assim surgiu a teoria dos grafos.

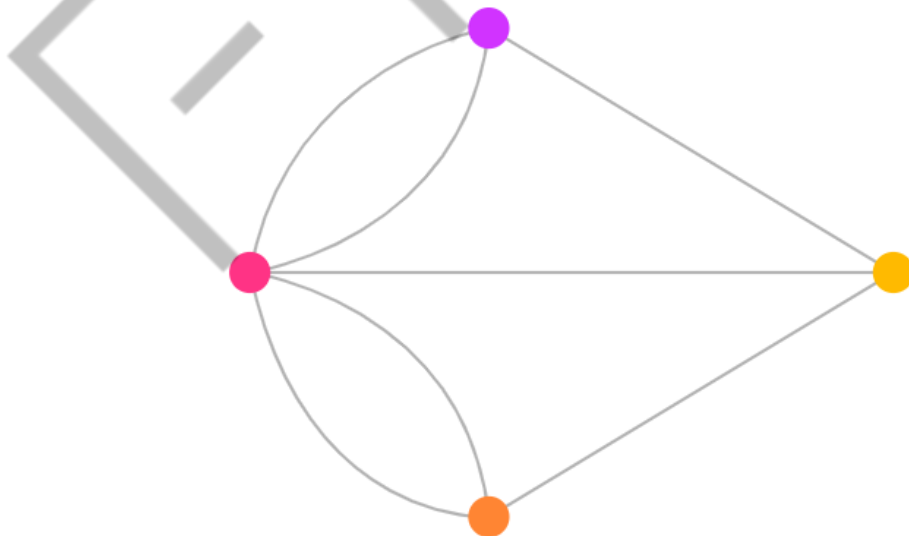


Figura 5.3 – Grafo
Fonte: Elaborado pelo autor (2020)

Mas o que é um grafo? Muito simplesmente, um banco de dados grafo é um banco de dados projetado para tratar os relacionamentos entre dados como o principal aspecto no modelo de dados, ou seja, são vértices unidos por arestas (linhas).

Essa composição do grafo foi adaptada e trazida para a ótica do banco de dados, o que mudou um pouco os conceitos para nós e relacionamentos. Em uma base de grafos, o nó armazena as informações de uma entidade e os relacionamentos, as informações de como as entidades se relacionam.

Os bancos de dados grafos são adequados para armazenar não apenas informações sobre objetos, mas também todos os relacionamentos existentes entre eles. Eles contam com um modelo de grafo sem esquema para modelar e representar facilmente os dados conectados.

Modelos grafos usam vértices e arestas para representar conexões entre dados. Um grafo pode se referir a uma rede profissional, por exemplo. Nesse caso, os vértices representam profissionais, enquanto as arestas direcionadas representam vínculos e relacionamentos entre esses profissionais. Cada vértice também é inicializado com um valor.

Vale a pena mencionar que mesmo que os bancos de dados grafos salvem relacionamentos, eles não são muito distintos dos bancos de dados relacionais. Eles são úteis para armazenar, acessar e analisar a força e a natureza das relações entre dois ou mais itens. Por exemplo, quão próxima é a relação entre duas pessoas? A que distância está um motorista de táxi de outro ou de um local turístico. Responder a essas perguntas torna possível formular recomendações valiosas em muitas indústrias.

Os grafos representam entidades como nós e as maneiras pelas quais essas entidades se relacionam com o mundo como relacionamentos. Um grafo contém nós e relacionamentos como mencionado acima, o grafo mais simples possível é um único nó, um registro que tem valores nomeados chamados de Propriedades. Um nó pode começar com uma única propriedade e adicionar milhares. Formalmente, um grafo é apenas uma coleção de vértices e arestas ou um conjunto de nós e as relações que os conectam.

Bancos de dados grafos, para muitos casos de uso, oferecem um excelente desempenho com latência menor em comparação com o processamento de outras

categorias de NoSQL. Um modelo de dados flexível com uma maneira fácil de expressar relacionamentos; e um enriquecimento constante do grafo como novos dados dando para os aplicativos a capacidade de evoluir de maneira controlada alinhada com práticas de desenvolvimento de software ágeis e orientadas a testes.

Os modelos de banco de dados grafo são aplicados em áreas onde as informações sobre a interconectividade ou topologia de dados são mais importantes ou tão importantes quanto os próprios dados.

Encontram-se exemplos de uso em redes sociais, redes de informação, redes tecnológicas (por exemplo, Internet, companhias aéreas, rotas e redes telefônicas), redes biológicas (por exemplo, área de genômica) e web semântica. Bancos de dados grafos são uma ferramenta poderosa para consultas baseadas na teoria dos grafos como computar o caminho mais curto entre dois nós, detecção de comunidades, pagerank, influência e autoridade na rede.

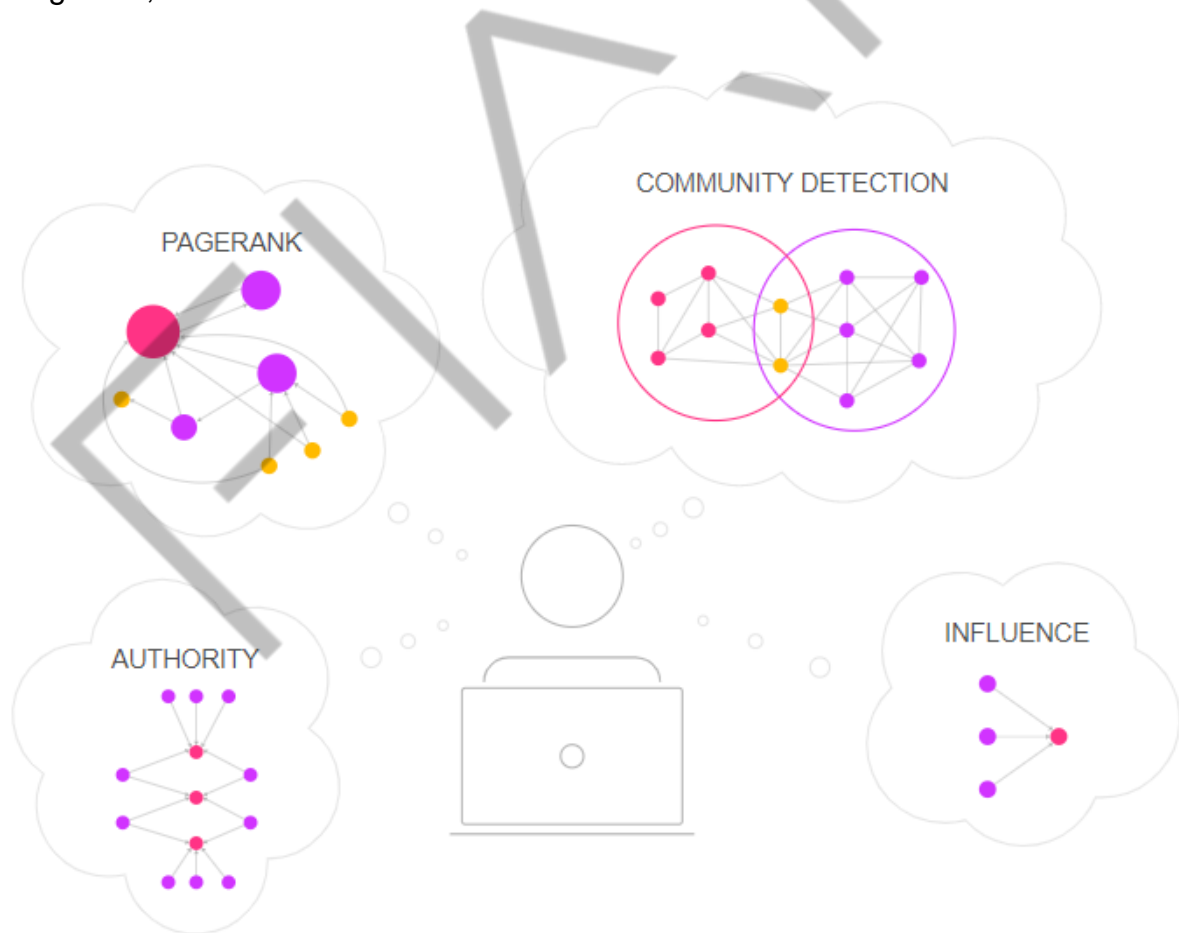


Figura 5.4 – Grafos e seus usos
Fonte: Adaptado por FIAP (2020)

A teoria dos grafos é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. Centralidade é uma medida de importância de um vértice em um grafo, e está relacionada com o número de vezes que um nó age como ponte ao longo do caminho mais curto entre dois outros nós.

Os grafos são extremamente úteis na compreensão de uma ampla diversidade de conjuntos de dados em campos como ciência, governo e negócio. Essa estrutura expressiva de uso geral permite a modelagem de todos os tipos de cenários, da construção de um foguete espacial, a um sistema de estradas, e da cadeia de abastecimento ou proveniência de alimentos, a histórico médico para as populações. Depois de entender os grafos, começamos a vê-los em todos os tipos de soluções.

O Gartner identifica cinco grafos no mundo dos negócios – social, intenção, consumo, interesse e mobilidade – e diz que a capacidade de aproveitar esses grafos fornece uma "vantagem competitiva sustentável". Por exemplo, os dados do Twitter são facilmente representados como um grafo.

5.2 Tecnologias

Szendi-Varga (2020) mapeou as tecnologias relacionadas ao modelo de grafos no Graph Technology Landscape. Nesse cenário as tecnologias estão distribuídas em:

- Infraestrutura com os bancos de dados de grafos apresentados como Graph DBMS, Multi model e padrão RDF; engines para processamento de dados; plataforma IAAS e PAAS; e Integração.
- Aplicações com visual analytics, soluções para grafos de conhecimento; detecção de fraude; cibersegurança e aprendizado de máquina entre outras.
- Ferramentas de desenvolvimento com bibliotecas e frameworks, integração com ferramentas de Business Intelligence; ferramentas para modelagem e linguagens para consultas como graphql, SPARQL e Gremlin.
- Fontes de informação com livros e conferências significativos no mundo de grafos.

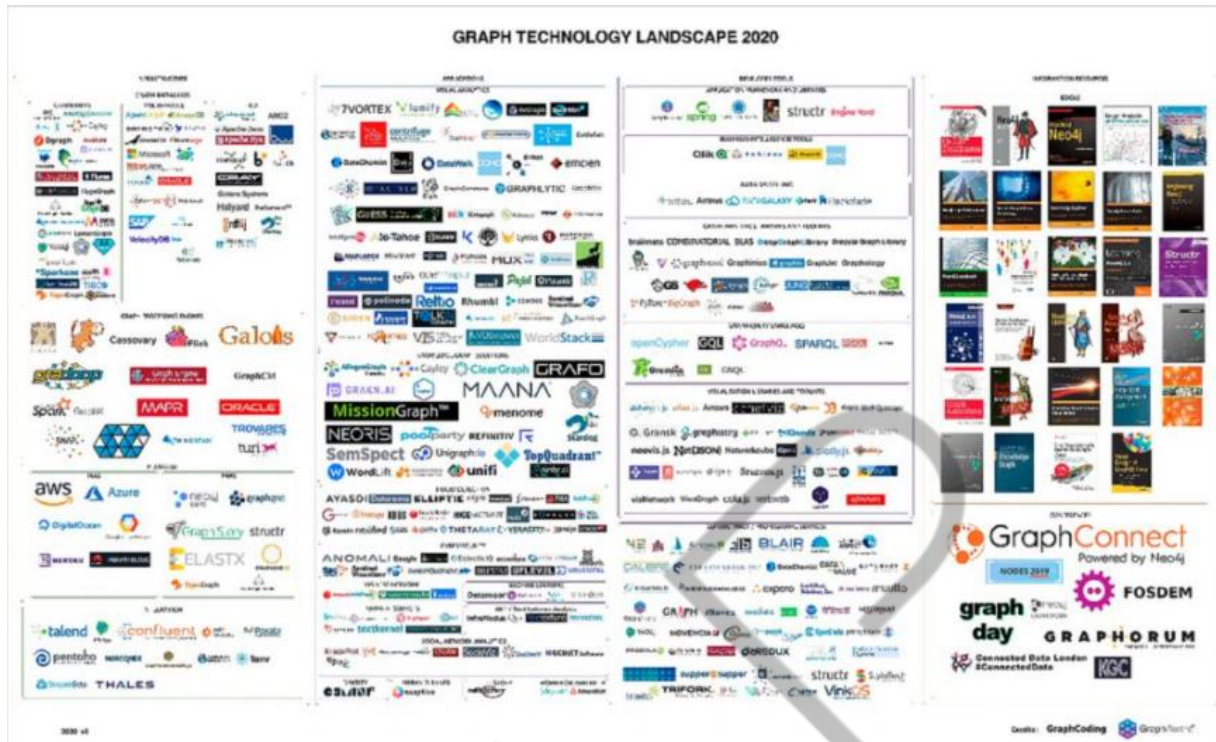


Figura 5.5 – Graph Technology Landscape 2020
Fonte: Szendi-Varga (2020)

Assim os bancos de dados de grafos oferecem muito mais do que apenas um armazenamento de dados. Eles vêm embalados com algoritmos para análise de grafos, recursos de visualização, recursos de aprendizado de máquina e ambientes de desenvolvimento. O site Db-Engines relaciona 32 bancos de dados nessa categoria em 2020.

























32 systems in ranking, August 2020								
Rank			DBMS	Database Model	Score			
Aug 2020	Jul 2020	Aug 2019			Aug 2020	Jul 2020	Aug 2019	
1.	1.	1.	Neo4j 	Graph	50.18	+1.26	+1.79	
2.	2.	2.	Microsoft Azure Cosmos DB 	Multi-model 	30.73	+0.32	+0.79	
3.	3.	 4.	ArangoDB 	Multi-model 	5.73	-0.11	+0.61	
4.	4.	 3.	OrientDB	Multi-model 	5.02	+0.14	-1.27	
5.	5.	5.	Virtuoso 	Multi-model 	2.65	+0.21	-0.41	
6.	6.	 7.	Amazon Neptune	Multi-model 	2.15	-0.06	+0.51	
7.	7.	 6.	JanusGraph	Graph	2.02	+0.00	+0.08	
8.	 9.	 18.	FaunaDB	Multi-model 	1.70	+0.22	+1.30	
9.	 8.	 8.	Dgraph 	Graph	1.47	-0.08	+0.16	
10.	10.	10.	GraphDB 	Multi-model 	1.39	+0.07	+0.25	
11.	11.	 12.	Stardog 	Multi-model 	1.36	+0.10	+0.47	

Figura 5.6 – Db-Engines NoSQL Graph
Fonte: DB-Engines (2020)

Segue uma breve descrição de quatro principais: Neo4j, Amazon Neptune, Stardog e Giraph.

5.2.1 Amazon Neptune

Embora os sistemas de banco de dados relacional tenham dominado o mercado de banco de dados por muitas décadas, nos últimos anos temos testemunhado uma diversificação contínua dos bancos de dados NoSQL.

Abordando a ampla gama de casos de uso de processamento de dados de hoje, o ecossistema apresentado pela Amazon Web Service (AWS) oferece uma variedade de serviços adaptados e otimizados para as necessidades específicas de uso.

Além dos sistemas clássicos de gerenciamento de dados relacionais oferecidos pela Amazon RDS, serviços AWS incluem Amazon Redshift para análises, Amazon DynamoDB para chave-valor, Amazon DocumentDB para gerenciamento de documentos JSON, Amazon ElastiCache para processamento otimizado na memória, Amazon Timestream como uma solução para dados de séries temporais, Amazon Elasticsearch Service fornecendo funcionalidade de pesquisa de texto, serviços de análise de dados em grande escala como Amazon Elastic MapReduce (EMR) e serviços como Amazon SageMaker que suportam fluxos de trabalho de aprendizado de máquina e algoritmos sobre os dados.

O Amazon Neptune é um banco de dados para grafos seguindo os padrões do W3C (World Wide Web Consortium) como RDF (Resource Description Framework) e SPARQL, com acesso realizado com Gremlin e a plataforma de desenvolvimento Apache Tinkerpop.

Um exemplo do uso do Amazon Neptune é para criar aplicativos que armazenam e navegam em informações de ciências biológicas. Por exemplo, você pode usar o Neptune para armazenar modelos de doenças e interações genéticas, bem como pesquisar padrões grafos em vias de proteína para encontrar outros genes que podem estar associados a uma doença. Você pode modelar compostos químicos como um grafo e consultar padrões em estruturas moleculares.

A startup Blackfynn pesquisa maneiras de mudar a forma como a epilepsia, a doença de Alzheimer, a doença de Parkinson, a ELA e outros distúrbios neurológicos são tratados. Chris Baglieri (AWS, 2018) destacou como conectar os pontos entre dados clínicos de genomas, patologia, neuroquímica, dispositivos e pacientes, para de forma eficiente e em escala, ajudar a startup a descobertas inovadoras.

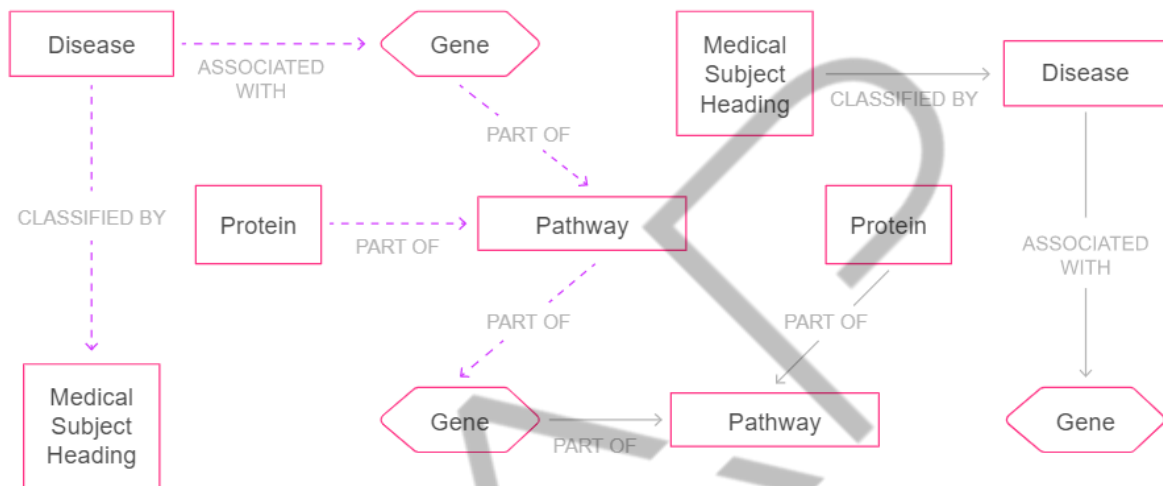


Figura 5.7 – Amazon Neptune – Ciências Biológicas
Fonte: Heller (2019)

Veja este laboratório para criar um mecanismo de recomendações para jogos com o Amazon Neptune: <https://aws.amazon.com/pt/getting-started/hands-on/recommendation-engine-for-games-amazon-neptune/>.

5.2.2 Stardog

Banco de grafos utilizado pela NASA! A plataforma Enterprise Knowledge Graph da Stardog é usada por líderes da indústria, incluindo Morgan Stanley, NASA, Schneider Electric e Bayer. Implementa grafos in memory e segue os padrões semânticos do W3C (World Wide Web Consortium) na sua implementação – RDF (Resource Description Framework), SPARQL e OWL (Web Ontology Language) - permitindo denominar seus grafos como grafos do conhecimento.

Stardog é um sistema comercial para grafo do conhecimento desenvolvido pela Stardog Union, Java puro que aceita ACID e restrições de integridade. Realiza inferências principalmente por reescrita de consulta, resposta a consultas

geoespaciais, e interação programática por meio de várias linguagens e interfaces de rede.

Um grafo do conhecimento rápido, escalonável e com base em padrões e seguro, Stardog é mais comumente usado para antilavagem de dinheiro, otimização da cadeia de suprimentos, descoberta de drogas, monitoramento de fraude, pesquisa baseada em biomarcadores, entre outros.

Explore seus recursos em: <<https://www.stardog.com/sandbox/>>.

Por curiosidade veja o início do Google Knowledge Graph:
<<https://www.youtube.com/watch?v=mmQl6VGvX-c>>.

5.2.3 Giraph

Para o processamento de grafos grandes, milhões de arestas e vértices, Pregel e Giraph são usados pois são baseados no modelo de computação paralela síncrono em massa - Bulk Synchronous Parallel (BSP) – no qual os cálculos são divididos em passos.

Em 2010, o Google anunciou Pregel, um framework para processamento em grande escala de grafos distribuídos. Seu objetivo era fornecer um certo nível de abstração para os programadores não terem que lidar com gerenciamento de memória distribuída ou sincronização de dados.

Apache Giraph é uma alternativa de código aberto de Pregel, uma das estruturas de processamento de grafos mais famosas, amplamente utilizadas e implementa um sincronismo síncrono e protocolos de comunicação de passagem de mensagens. Nenhum algoritmo de particionamento é implementado nessa estrutura. Isso significa que o particionamento do grafo é baseado no particionamento do arquivo de grafo no sistema de arquivos distribuídos Hadoop (HDFS). É importante notar que as soluções Giraph funcionam como tarefas MapReduce e usam Zookeeper na coordenação.

5.2.4 Neo4j

Neo4j é um dos bancos de dados de grafos NoSQL mais populares. É um projeto de código aberto disponível em uma edição da GPL, edições Enterprise disponíveis sob a GPL avançada (AGPL) v3, bem como uma licença comercial. Seu uso é tão difundido e importante que será detalhado em capítulos subsequentes desta disciplina.

Diferentemente de outros tipos de bancos de dados NoSQL, esse está diretamente relacionado a um modelo de dados estabelecido, o modelo de grafos, tem o pressuposto de representar os dados e/ou o esquema dos dados como grafos dirigidos ou como estruturas que generalizem a noção de grafos.

O modelo de grafos é mais interessante que outros, quando informações sobre a interconectividade ou a topologia dos dados são mais importantes ou tão importante quanto os dados propriamente ditos.

O modelo orientado a grafos possui três componentes básicos: os **nós** (são os vértices do grafo), os **relacionamentos** (são as arestas) e as **propriedades** (ou atributos) dos nós e relacionamentos, veremos todos eles em detalhes na sequência.

Nesse caso, o banco de dados pode ser visto como um multigrafo rotulado e direcionado, em que cada par de nós pode ser conectado por mais de uma aresta.

Neo4j é um dos bancos de dados de grafos NoSQL mais famosos. Tem código aberto, fornece um back-end transacional em conformidade com as propriedades ACID e seu desenvolvimento começou em 2003, ficando disponível ao público em 2007, tendo sua primeira versão open em 2010.

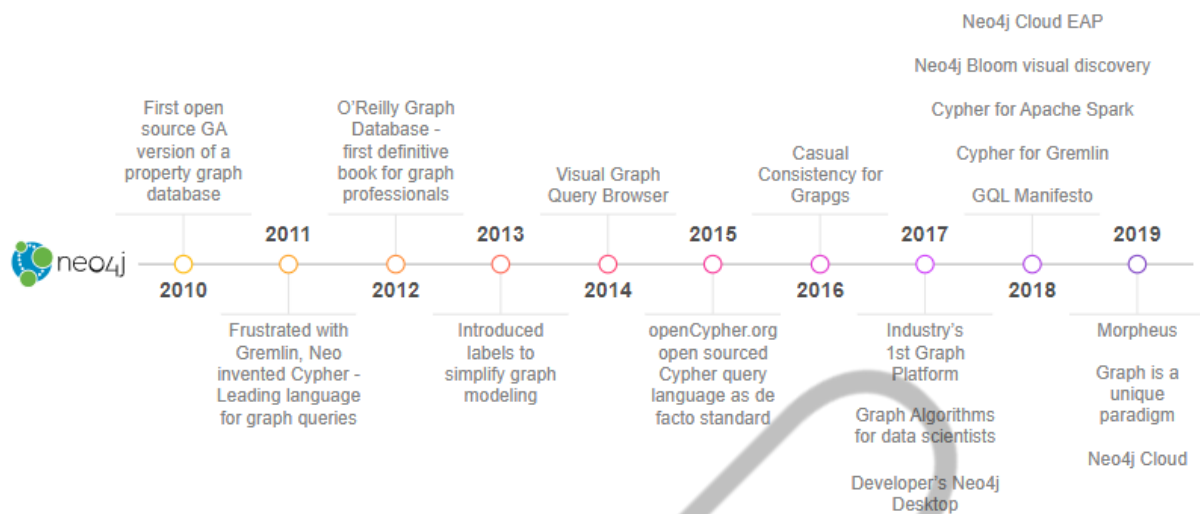


Figura 5.8 – História e funcionalidades
Fonte: NEO4J (2020)

O código fonte do Neo4j, escrito em Java e Scala, está disponível gratuitamente no GitHub ou como um download de aplicativo de desktop a partir do site oficial. Seguem algumas características destacadas por desenvolvedores, arquitetos e engenheiros de dados:

- **Cypher:** uma linguagem de consulta declarativa semelhante ao SQL, mas otimizada para grafos. Agora utilizada por outros bancos de dados, como o SAP HANA Graph e Redis, por meio do projeto openCypher.
- Tempo para retornar os percursos constantes em grandes grafos, com consultas tanto em profundidade quanto em largura, devido à representação eficiente de nós e relacionamentos. Permite expansão para bilhões de nós com requisitos de hardware considerados moderados.
- Esquema flexível para propriedades que pode se adaptar ao longo do tempo.
- Drivers para linguagens de programação populares, incluindo Java, JavaScript, .NET, Python e muito mais.

Neo4j é mais rápido nas suas consultas do que as bases de dados relacionais, tornando-o ideal para gerir dados complexos de vários domínios.

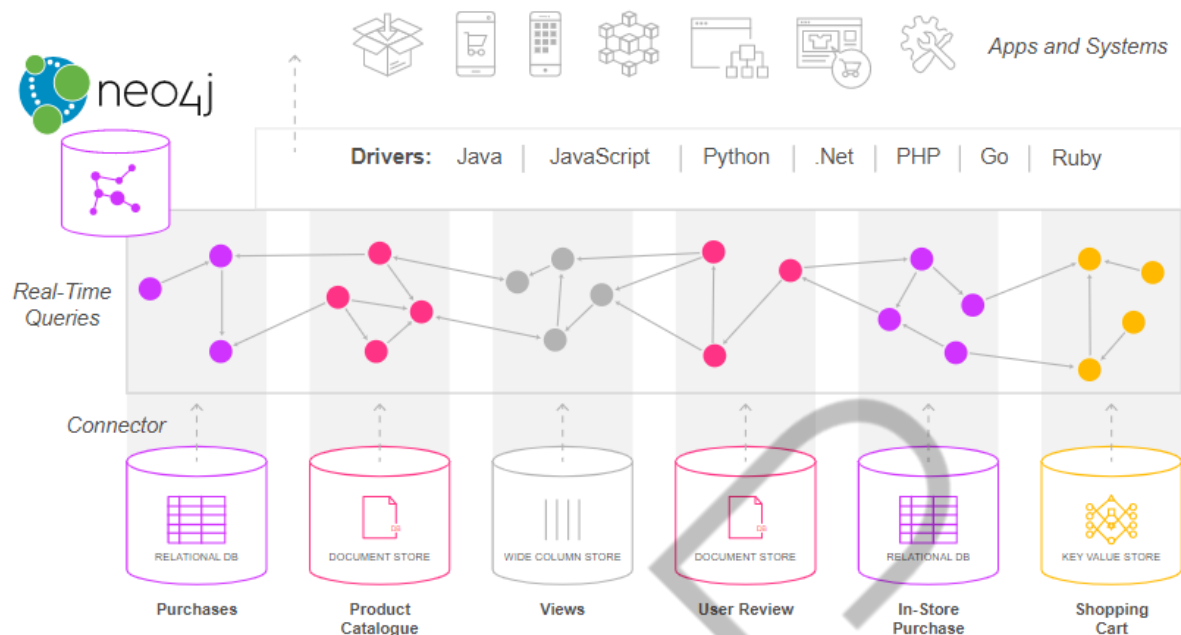


Figura 5.9 – Integrando dados de diversas bases de dados
Fonte: NEO4J (2020)

5.2.4.1 Arquitetura e suas camadas

Neo4j é um banco de dados com uma estrutura física de arquivos organizados em um diretório ou pasta, que tem o mesmo nome do banco de dados. Em termos lógicos, um banco de dados é um contêiner para um ou mais grafos. Na versão Neo4j 4.0, existe apenas um grafo em cada banco de dados e muitos comandos, ao se referirem a um grafo específico, fazem isso usando o nome do banco de dados.

Um banco de dados define um domínio de transação e um contexto de execução. Isso significa que uma transação não pode se estender por vários bancos de dados. Da mesma forma, um procedimento é chamado dentro de um banco de dados, embora sua lógica possa acessar dados armazenados em outros bancos de dados. Uma instalação padrão do Neo4j 4.1 contém dois bancos de dados:

- **system** - o banco de dados do sistema, contendo metadados no DBMS e na configuração de segurança.
- **neo4j** - o banco de dados padrão, um único banco de dados para dados do usuário. Ele tem um nome padrão de neo4j. Um nome diferente pode ser configurado antes de iniciar o Neo4j pela primeira vez.

No Neo4j 4.0 com vários bancos de dados, temos vários bancos de dados ativos por cluster com forte isolamento. Portanto, os bancos de dados são fisicamente separados, apesar de serem executados no mesmo cluster de servidores.

O Neo4j Fabric é um módulo para permitir consultar vários grafos dentro da mesma transação com Cypher, ou seja, realiza a federação de dados – uma visão unificada dos dados locais e distribuídos, acessíveis por meio de uma única conexão de cliente e sessão do usuário.

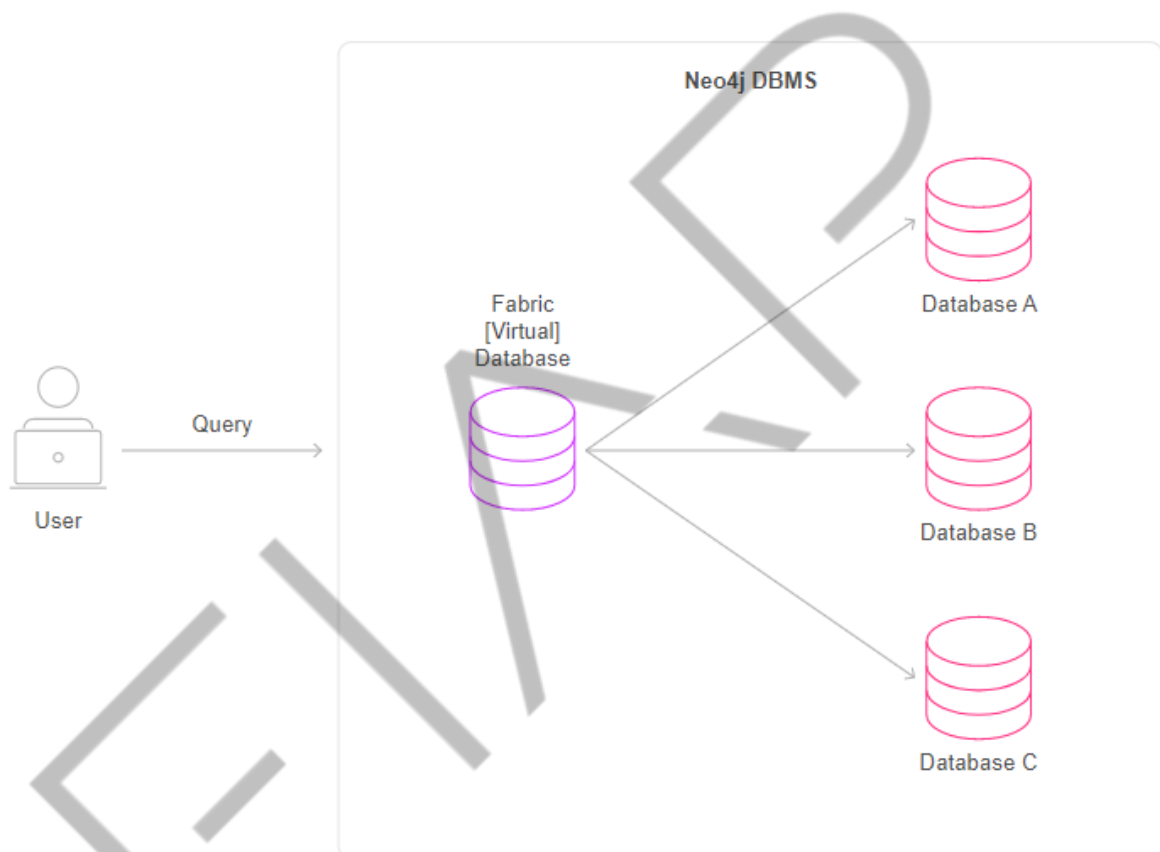


Figura 5.10 -- Fabric com várias bases vistas como uma única
Fonte: NEO4J (2020)

A federação de dados permite recuperar dados presentes em distintos bancos de dados de grafos. Imagine ter grafos construídos em toda a organização, de TI a finanças, operações, vendas, RH, marketing, fabricação, entre outros. A linguagem Cypher permite que os desenvolvedores consultem esses grafos presentes nos silos - mesmo com esquemas diferentes - como se fosse um único grafo.

Fabric também fornece a infraestrutura e as ferramentas para sharding. Sharding é realizado com o armazenamento físico do grafo dividido ou fragmentado

em vários servidores ou clusters. Os usuários dividem um grafo maior em grafos individuais menores e os armazenam em bancos de dados separados.

Para grafos altamente conectados, isso significa algum nível de redundância de dados para manter os relacionamentos entre os nós. Algumas das razões para fragmentar um grafo encontram-se na necessidade de minimizar a latência armazenando shards próximos aos usuários e de dividir grafos muito grandes (dezenas de bilhões de nós).

Permite assim maior escalabilidade para operações de leitura/gravação, volume de dados e simultaneidade; e alta disponibilidade com nenhum ponto único de falha para grandes volumes de dados. O sharding suporta casos de uso como conformidade com regulamentos de privacidade de dados complexos para países cuja lei determina que os dados fiquem em seu território.

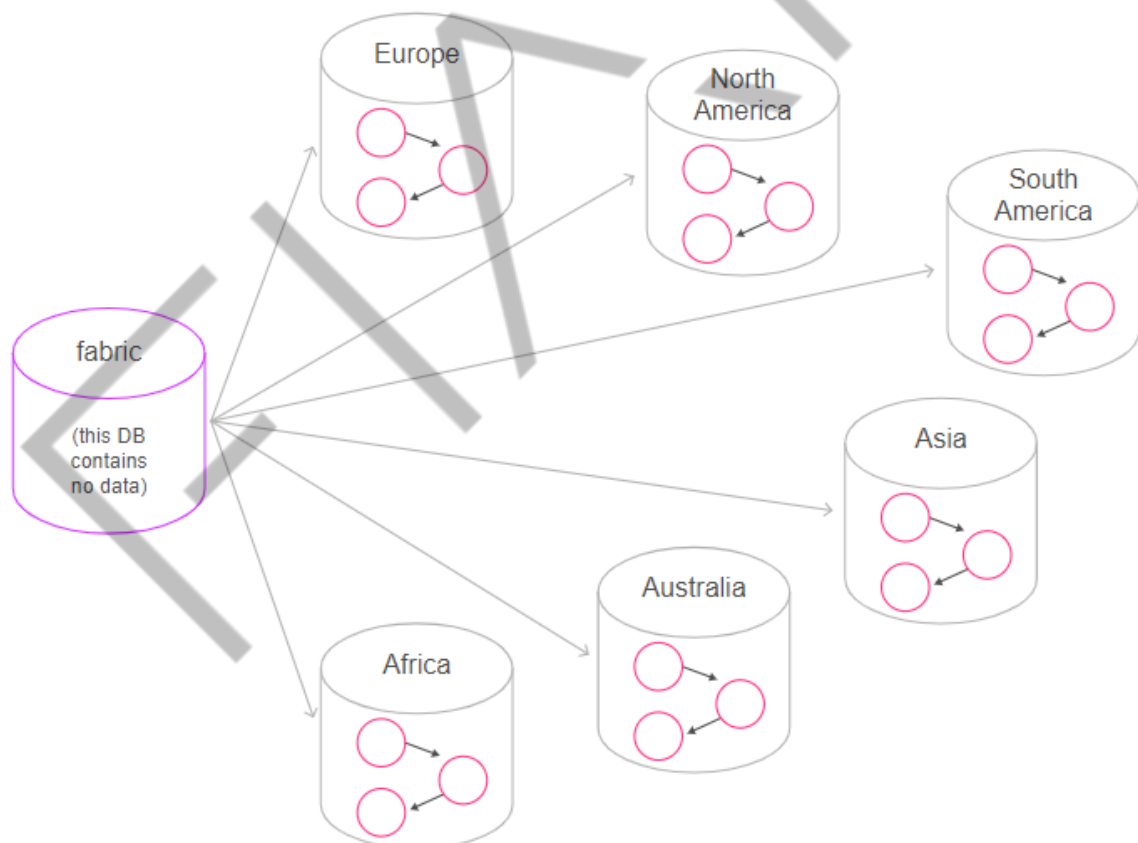


Figura 5.11 – Fabric e Sharding
Fonte: NEO4J (2020)

Fabric administra o sharding como um único banco de dados ou uma combinação de banco de dados.

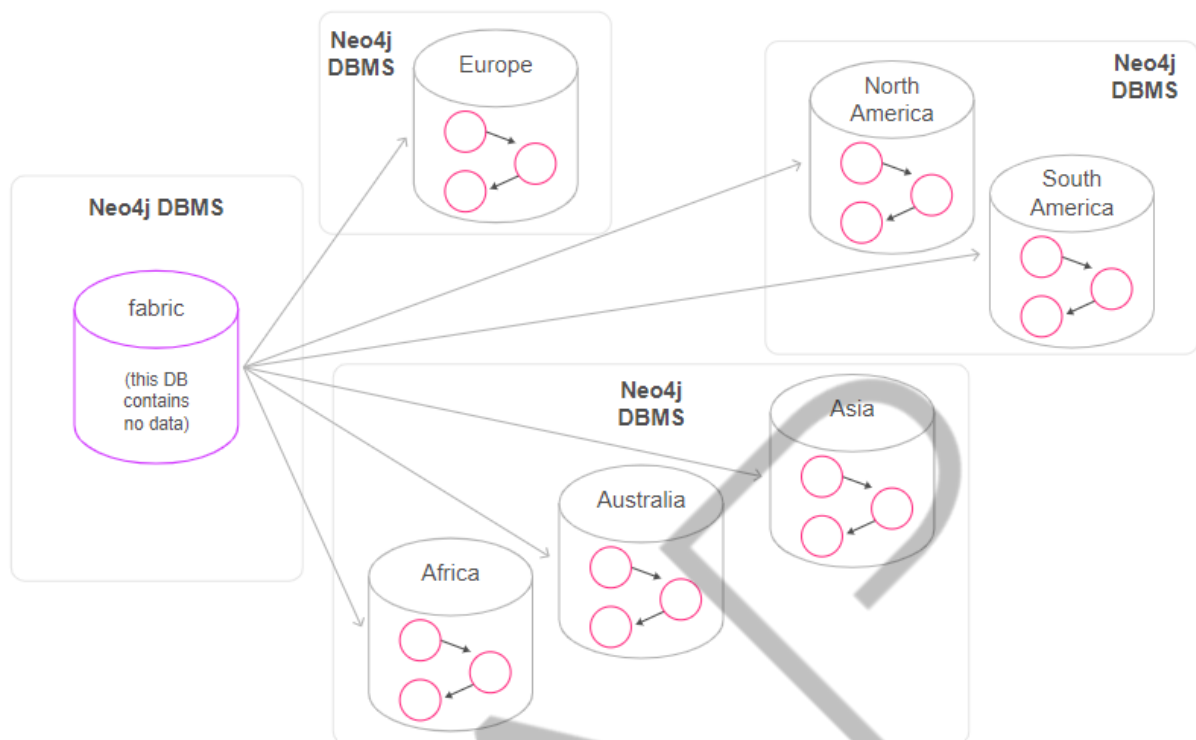


Figura 5.12 – Fabric e Sharding com gerenciadores de bases distintas
Fonte: NEO4J (2020)

Clusters disjuntos podem ser dimensionados de acordo com a carga de trabalho esperada e os bancos de dados podem ser colocados no mesmo cluster ou podem ser hospedados em seu próprio cluster para fornecer maior taxa de transferência.

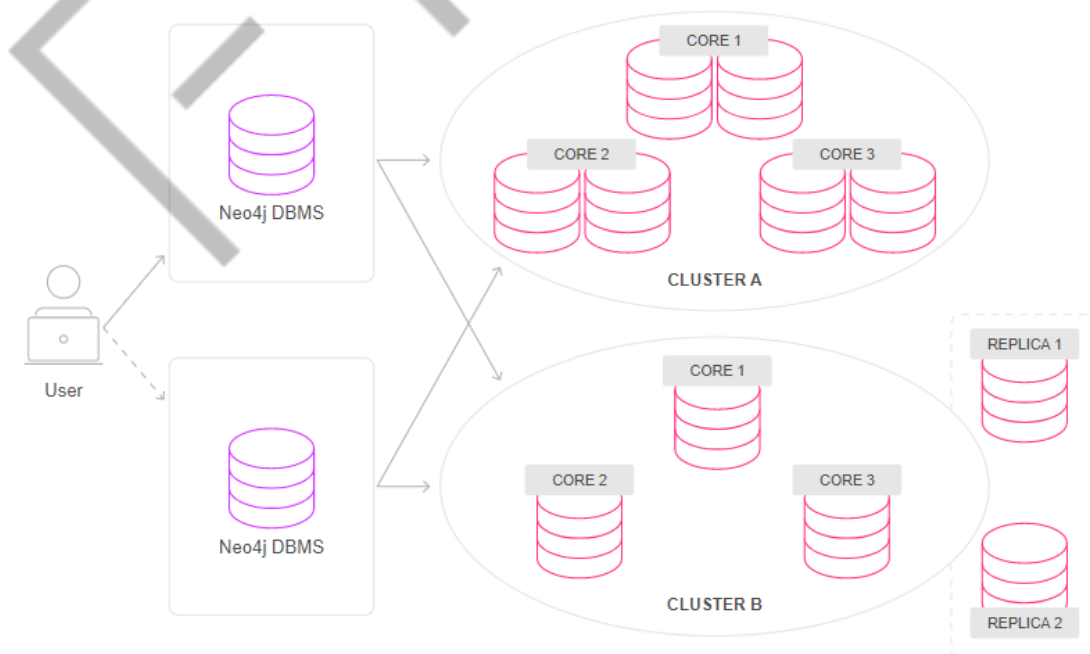


Figura 5.13 – Neo4j e cluster
Fonte: NEO4J (2020)

5.2.4.2 Características principais

Dentre suas principais características, vale a pena ressaltar:

Linguagem de consulta específica do gráfico

Cypher é uma linguagem de consulta legível por humanos projetada especificamente para manipular dados gráficos conectados. Está se tornando uma linguagem de consulta de gráficos neutra de fornecedor por meio do projeto openCypher.

Visualização e ferramentas específicas para gráficos

O Navegador Neo4j permite que você visualize seus dados conectados, simplifique seus comandos Cypher e ofereça ferramentas de desenvolvimento de consulta além da linha de comando.

Mecanismo de armazenamento específico do gráfico

O Neo4j é projetado de cima para baixo para armazenar dados conectados sem uma dependência excessiva em índices (adjacência livre de índice) e sem comprometer a integridade dos dados por meio de transações compatíveis com ACID.

Mecanismo de processamento específico do gráfico

Para desempenho de nível corporativo, um banco de dados de gráfico nativo deve oferecer consultas de gráficos compiladas, planejamento de consulta de gráficos, APIs específicas de gráficos, drivers de aplicativos nativos, otimizadores baseados em gráficos específicos de gráficos e cache de alto desempenho.

Recursos de escalabilidade específicos do gráfico

O Neo4j inclui gerenciamento de memória fora do heap, Causal Clustering, que otimiza tanto o acesso somente leitura quanto o acesso de leitura/gravação, alta disponibilidade (HA), recuperação de desastre e suporte a vários data centers.

Segurança do usuário corporativo específica do gráfico

A tecnologia de gráfico nativa deve oferecer uma arquitetura de banco de dados fundamentalmente robusta que inclua segurança baseada em usuário e função, integração LDAP e Active Directory, autenticação Kerberos, opções de implantação na nuvem e no local e licenciamento comercial e de código aberto.

Suporte Full a ACID

A integridade dos dados por meio de transações compatíveis com ACID. Em termos gerais, ele atua muito bem para:

- Sistemas de recomendações.
- Usado para Business Intelligence.
- Sistema Geoespaciais.
- Índices RDBMS.
- O mundo é baseado em relacionamentos.
- Os dados e suas estruturas são dinâmicos.

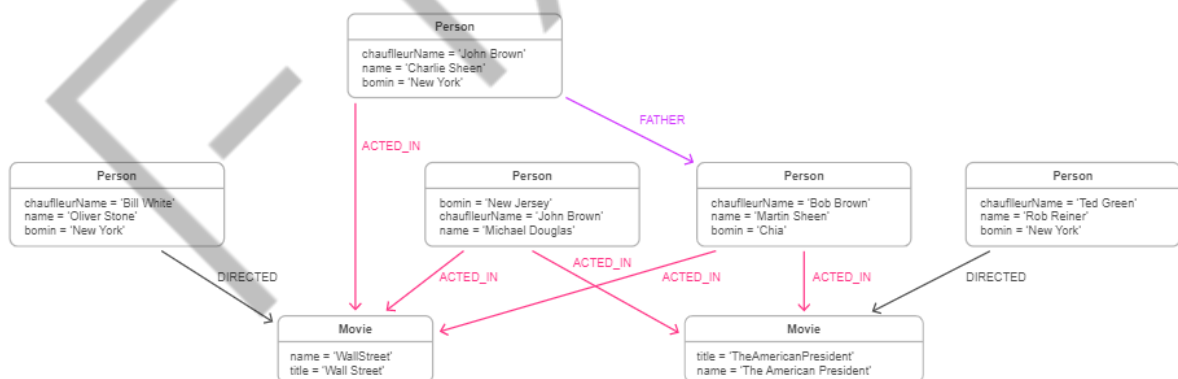


Figura 5.14 – Grafos e seus componentes

Fonte: NEO4J (2020)

5.2.4.3 Modelo de dados

Se você já trabalhou com um modelo de objeto ou um diagrama de relacionamento de entidade, o modelo de gráfico de propriedade rotulado parecerá familiar.

Nós são as entidades no gráfico. Eles podem conter qualquer número de atributos (pares de valor-chave) chamados propriedades. Os nós podem ser marcados com marcadores representando suas diferentes funções no seu domínio. Além de contextualizar propriedades de nó e relacionamento, os rótulos também podem servir para anexar metadados - informações de índice ou restrição - a determinados nós.

Relacionamentos fornecem conexões dirigidas, nomeadas semanticamente relevantes entre duas entidades de nó (por exemplo, Employee WORKS_FOR Company).

Um relacionamento sempre tem uma direção, um tipo, um nó inicial e um nó final. Como os nós, os relacionamentos também podem ter propriedades. Na maioria dos casos, os relacionamentos têm propriedades quantitativas, como pesos, custos, distâncias, classificações, intervalos de tempo ou intensidades.

Como os relacionamentos são armazenados eficientemente, dois nós podem compartilhar qualquer número ou tipo de relacionamento sem sacrificar o desempenho. Observe que, embora sejam direcionados, os relacionamentos sempre podem ser navegados eficientemente em qualquer direção.

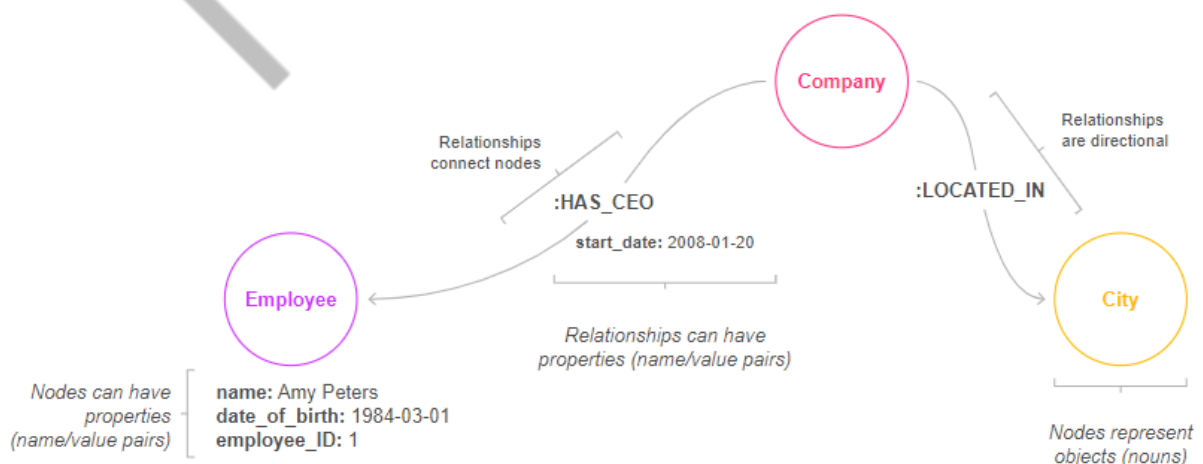


Figura 5.15 – Relacionamentos no ambiente grafo
Fonte: Neo4J (2020)

Como estudado anteriormente, o modelo relacional puro tem algumas limitações, como a resolução dos relacionamentos M:M (muitos para muitos), correto? Temos que resolver esse tipo de necessidade de negócio no modelo para poder implementá-lo fisicamente.

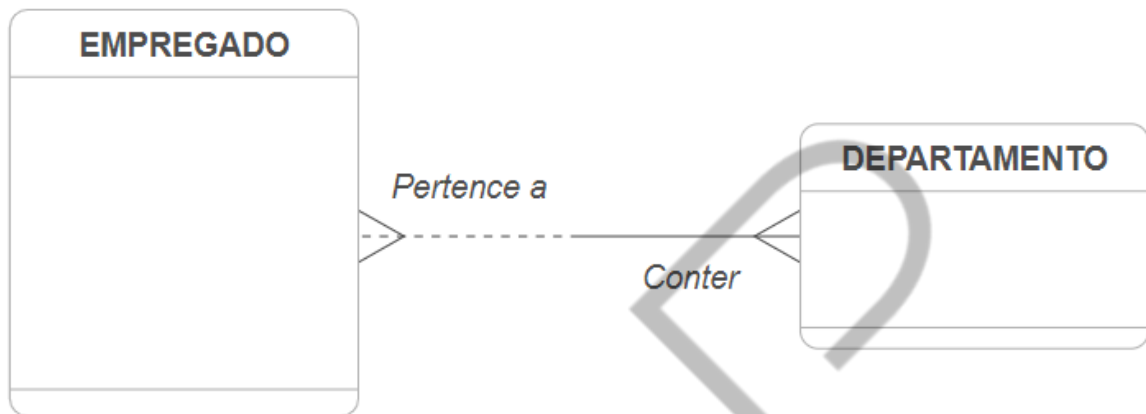


Figura 5.16 – Relacionamento M:M
Fonte: Elaborado pelo autor (2020)

No caso do banco de dados grafo, essa limitação não se impõe, pois ele consegue fazer a implementação física dos modelos M:M.

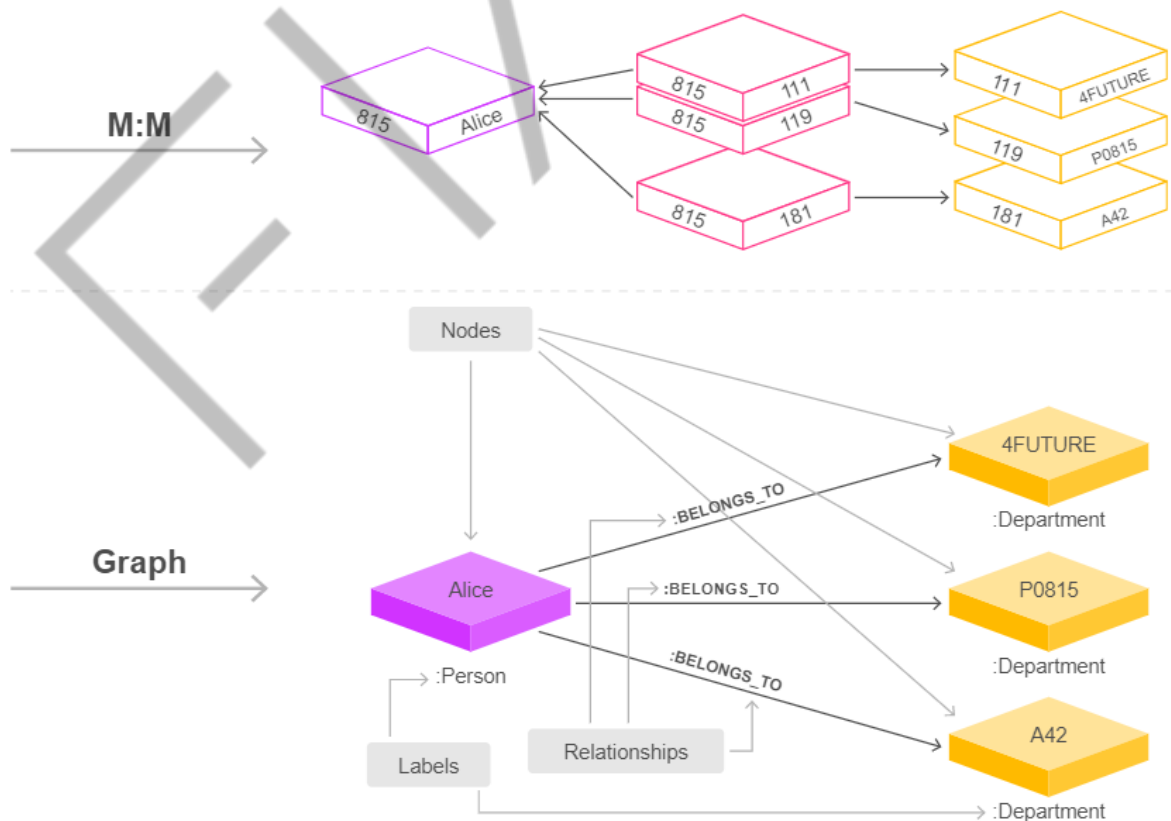


Figura 5.17 – Multirrelacionamentos no ambiente Grafo
Fonte: Elaborado pelo autor (2020)

Vale ressaltar também que, como mencionado, os nós e os relacionamentos são objetos, e como tal, possuem propriedades, veja:

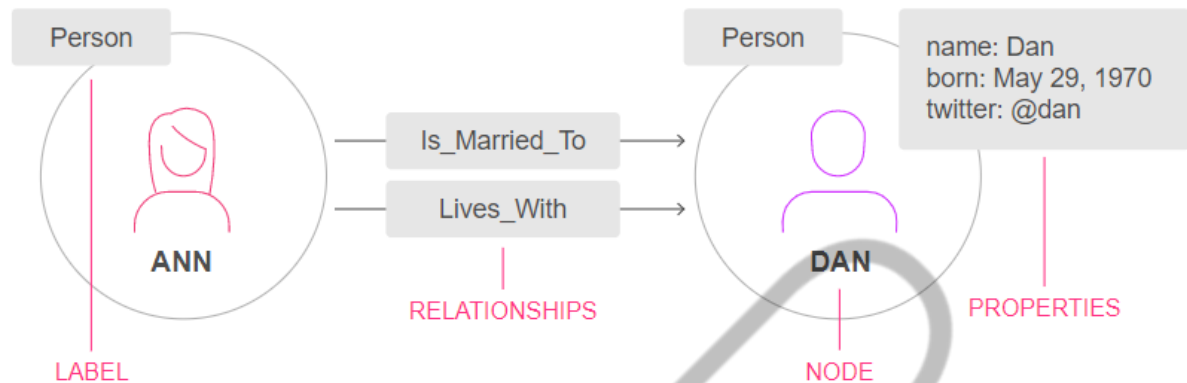


Figura 5.18 – Nós e relacionamentos (1)
Fonte: Neo4J (2020)

Na figura acima, temos os nós e os relacionamentos, sendo que os nós:

- São os principais elementos de dados.
- Os nós estão conectados a outros nós por meio de relacionamentos.
- Os nós podem ter uma ou mais propriedades (isto é, atributos armazenados como pares chave/valor).
- Os nós têm um ou mais rótulos que descrevem seu papel no gráfico.

E os relacionamentos:

- Relacionamentos conectam dois nós.
- Relacionamentos são direcionais.
- Os nós podem ter relacionamentos múltiplos, até recursivos.
- Relacionamentos podem ter uma ou mais propriedades (isto é, atributos armazenados como pares chave / valor).

Ambos possuem propriedades, temos também os **LABELs**:

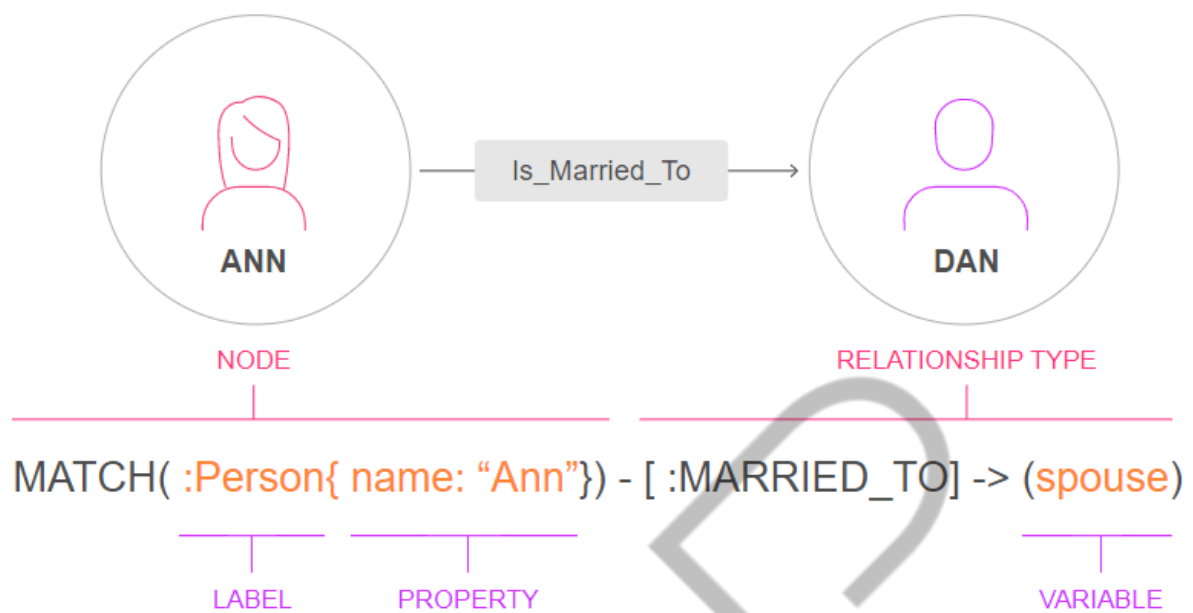


Figura 5.19 – Nós e relacionamentos (2)
Fonte: Neo4J (2020)

Onde os labels:

- Os rótulos são usados para agrupar nós em conjuntos.
- Um nó pode ter vários rótulos.
- Os rótulos são indexados para acelerar a localização de nós no gráfico.
- Índices de rótulos nativos são otimizados para velocidade.

E as propriedades:

- Propriedades são valores nomeados em que o nome (ou chave) é uma string.
- Propriedades podem ser indexadas e restringidas.
- Índices compostos podem ser criados a partir de múltiplas propriedades.

5.3 Usando Neo4j

Um grafo possui nós e arestas denominados, no Neo4J, nodes e relationships, respectivamente. Tanto os nós como as arestas possuem propriedades (properties).

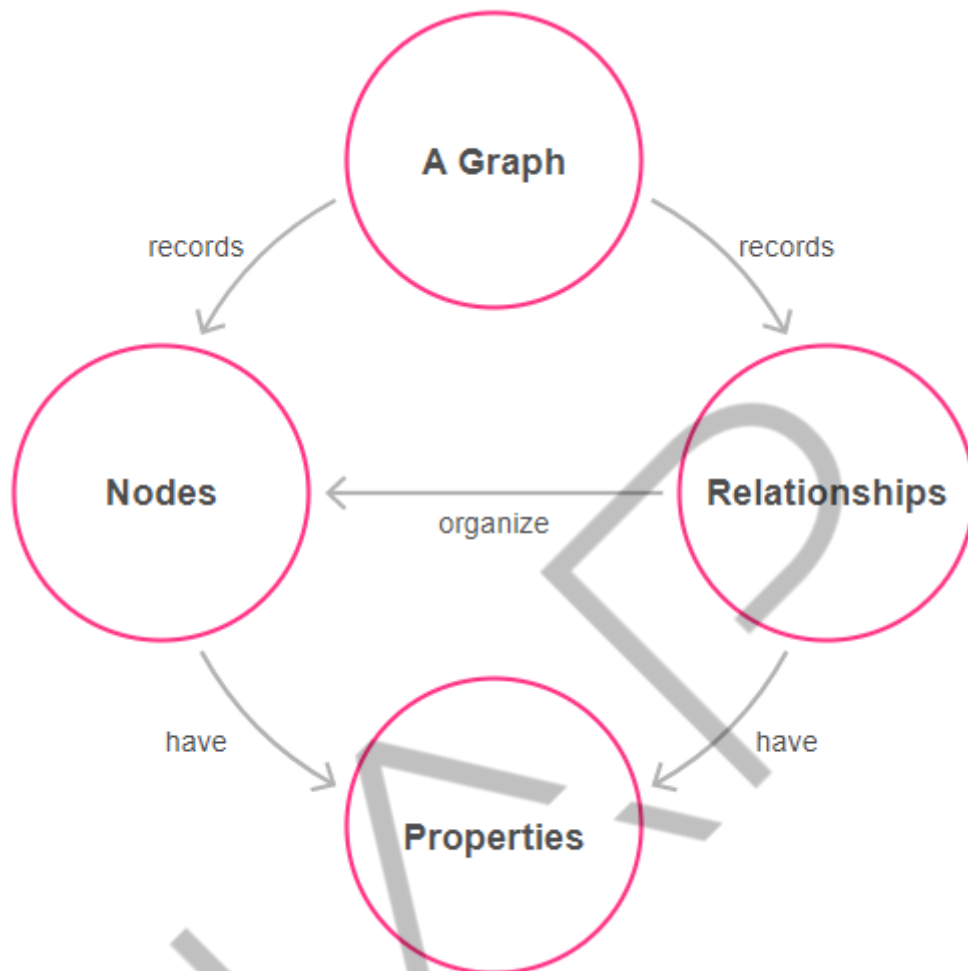


Figura 5.20 – Grafos e seus componentes
Fonte: NEO4J (2020)

Cypher é uma linguagem de consulta declarativa do Neo4j que permite consultas e atualizações expressivas e eficientes dos dados no grafo. Consultas de banco de dados tidas como muito complicadas em modelos como o relacional podem ser facilmente expressas por meio da sintaxe Cypher.

Seus principais comandos são CREATE e MATCH para operações de inserção, alteração, exclusão e consulta dos dados armazenados no banco de dados de grafo. Possui sintaxe elaborada para tratar com padrões de acesso, operadores, transações, funções e cláusulas para tratar os resultados obtidos.

De forma a ser possível reproduzir todos os exemplos presentes neste capítulo, podemos optar pela versão na nuvem (<https://sandbox.neo4j.com/>), ou realizar uma instalação local (<https://neo4j.com/download/>).

5.3.1 Neo4j sandbox

Para acessarmos o sandbox na nuvem, precisamos ter um usuário e senha na plataforma ou utilizarmos algum outro usuário nosso já existentes no Google, **LinkedIn** ou Twitter.



Figura 5.21 – Acesso Sandbox Neo4j
Fonte: NEO4J (2020)

Selecionar Blank Sandbox no Sandbox Neo4j. Observe como tem vários exemplos já construídos para exploração futura e mais detalhes podem ser vistos em (<https://neo4j.com/graphgists>).

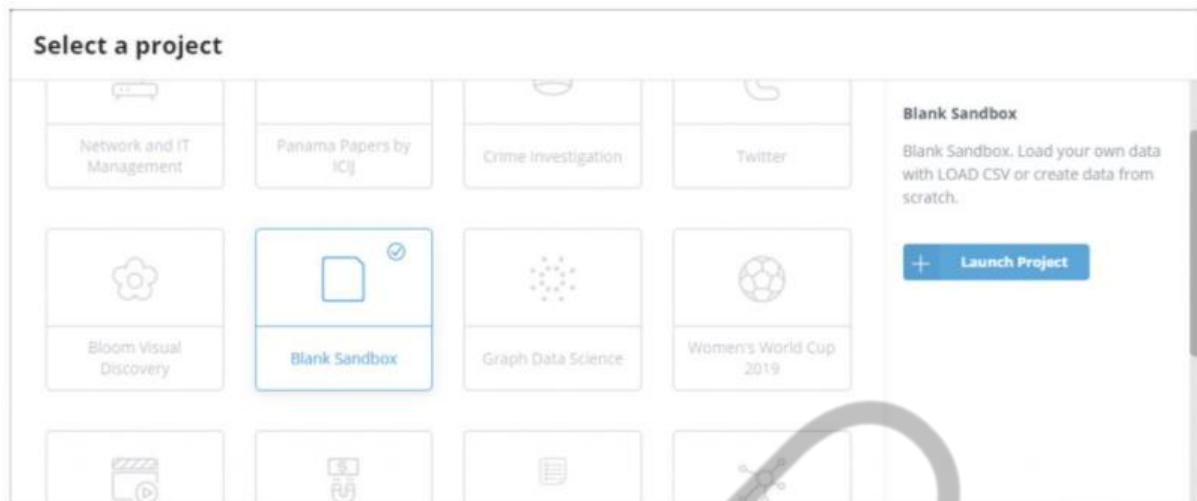


Figura 5.22 – Neo4J Sandbox - Selecionando um projeto
Fonte: NEO4J (2020)

Nosso projeto Blank Sandbox foi criado e temos três dias para realizarmos comandos nesse ambiente. Para isso, abrimos uma interface gráfica no browser (Open with Browser).

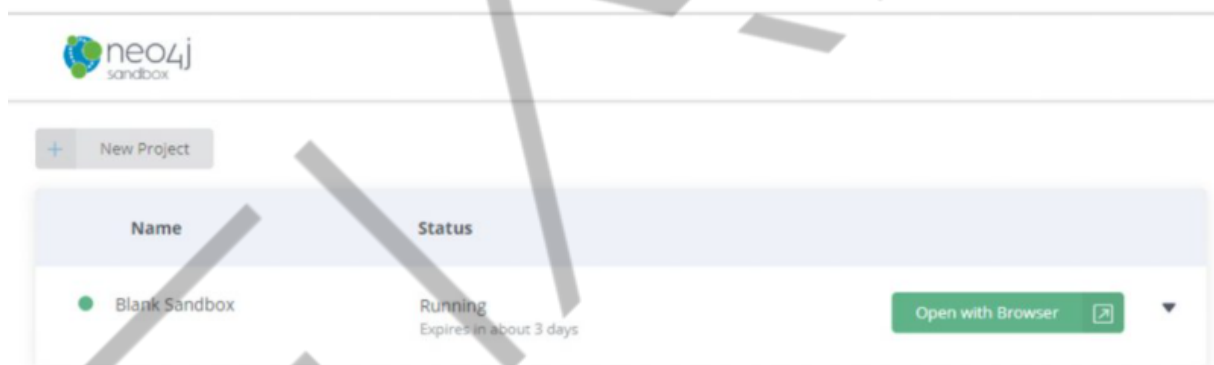


Figura 5.23 – Base de dados criada
Fonte: NEO4J (2020)

5.3.2 Neo4j Desktop

Para instalação local forneça seus dados, o instalador será baixado (neo4j-desktop-offline-1.3.8-setup), copie sua "Activation Key" para sua versão desktop e realize a instalação.

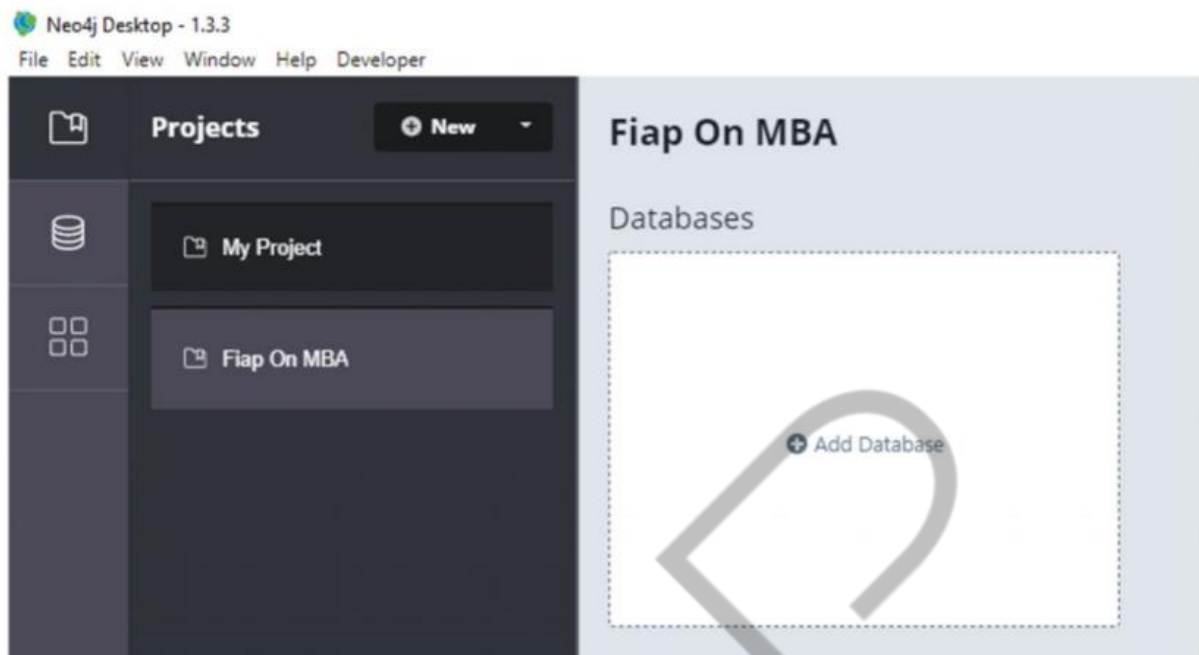


Figura 5.24 – Criando um novo projeto
Fonte: NEO4J (2020)

Na sequência, criamos um projeto aqui denominado “Fiap On MBA” para nossa nova base de dados.

Ao adicionarmos a base de dados (Add Database), informamos o nome da base (FiapOn) e uma senha antes de clicarmos em Create (passo 1). Ao finalizar a criação dos diretórios necessários para a base de dados, aí sim podemos inicializar (Start) a base de dados (passo 2) – deixá-la disponível para os acessos. Ao término desse passo, a base estará criada.

Observe como não temos nenhum nó (node) e nenhuma aresta (relationships) e a base está disponível (Active) (passo 3). Agora só resta acessar a interface web (Open Folder) para realizarmos os comandos para criação e consulta dos nossos grafos.

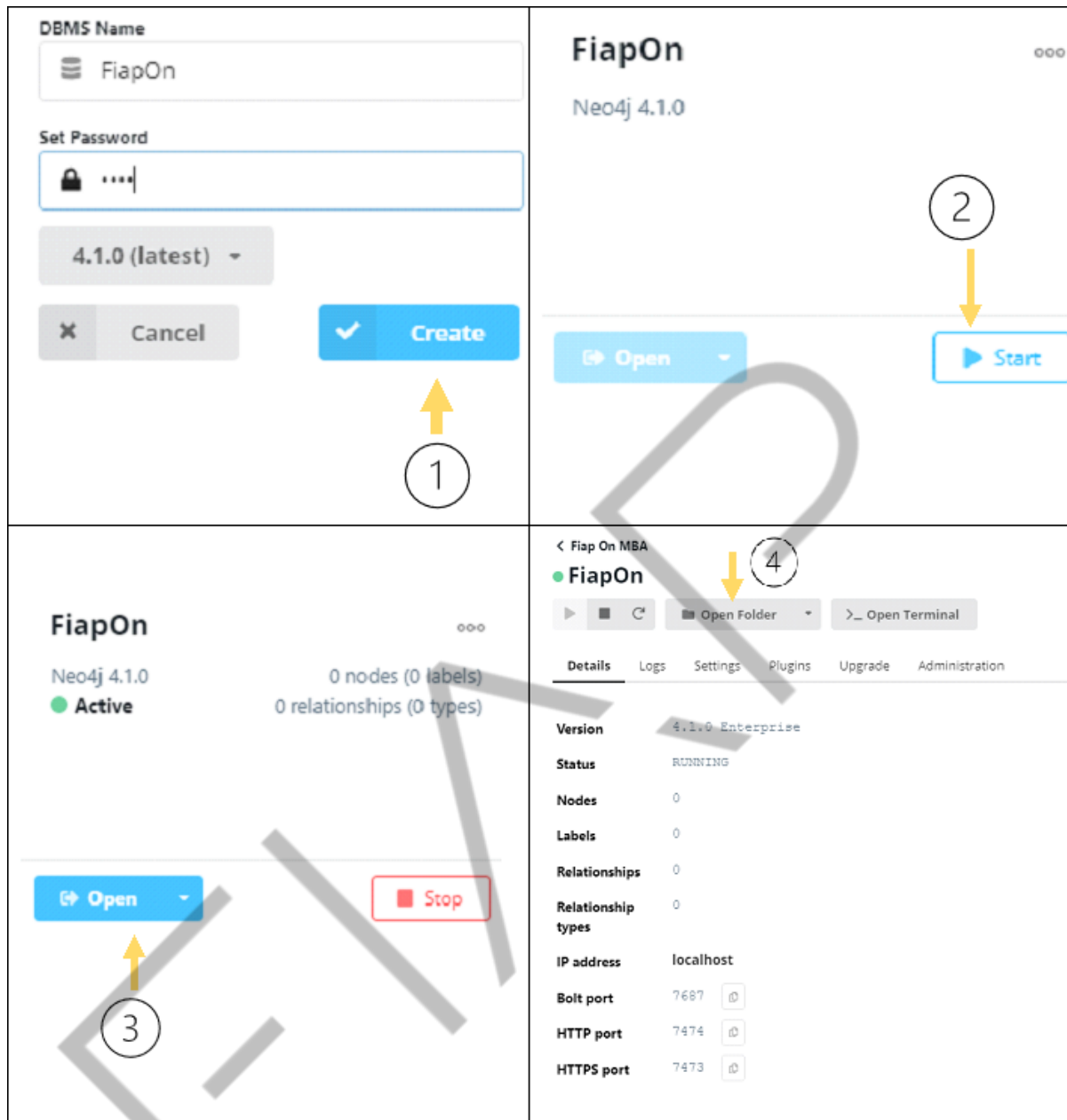


Figura 5.25 – Sequência para criação e inicialização da base de dados
Fonte: NEO4J (2020)

5.3.3 Interface gráfica

O editor é a interface principal para entrar e executar comandos. Insira as consultas do Cypher para trabalhar com dados do gráfico. Use comandos do lado do cliente como: ajuda para outras operações.

- Edição de linha única para breves consultas ou comandos.
- Mudar para edição multi-linha com <shift-enter>.

- Execute uma consulta com <ctrl-enter>.
- O histórico é mantido para recuperar facilmente os comandos anteriores.

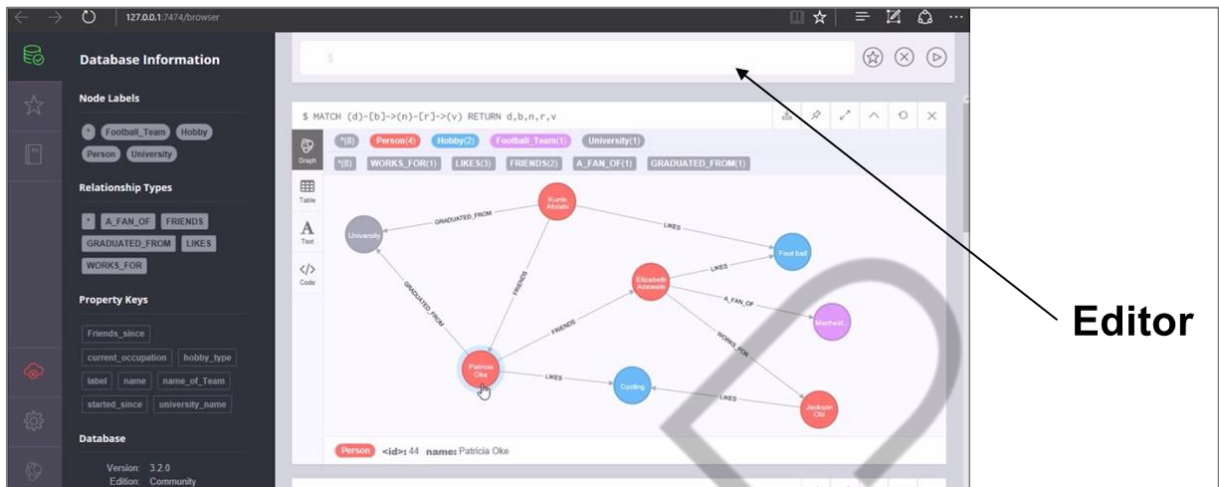


Figura 5.26 – Interface Gráfica – Editor de comandos
Fonte: Elaborado pelo autor (2020)

Um quadro de resultado é criado para cada execução de comando, adicionado à parte superior do fluxo para criar uma coleção rolável em ordem cronológica inversa.

- Quadros especiais como visualização de dados.
- Expandir um quadro para tela cheia.
- Remover um quadro específico do fluxo.
- Limpe o fluxo com o comando **:clear**.

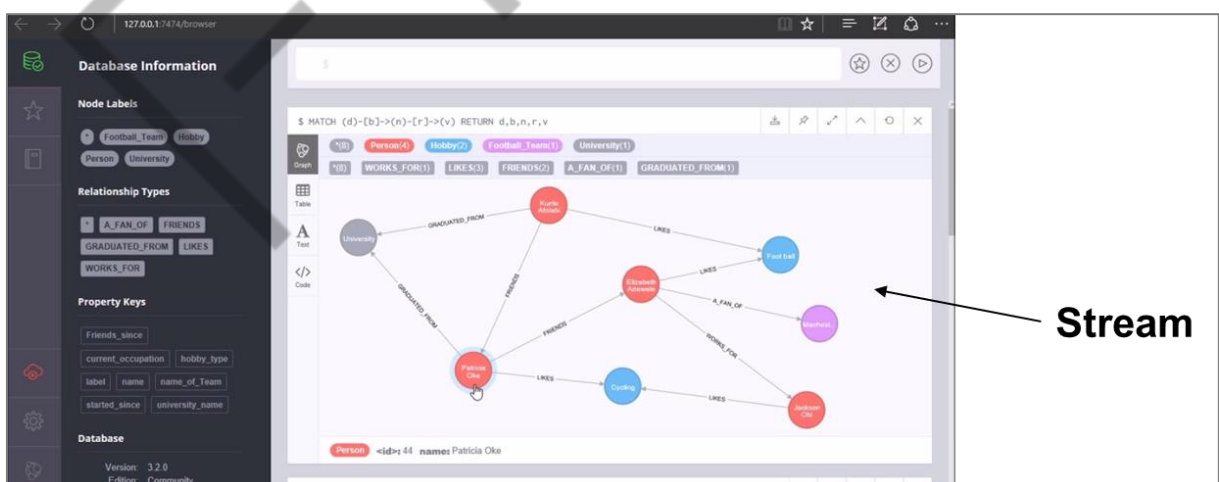


Figura 5.27 – Interface Gráfica – Stream/tela de resultados
Fonte: Elaborado pelo autor (2020)

A guia de código exibe tudo enviado e recebido do servidor Neo4j, incluindo:

- Solicitar URI, método HTTP e cabeçalhos.
- Código e cabeçalhos de resposta HTTP de resposta.
- Pedido bruto e conteúdo da resposta no formato JSON.

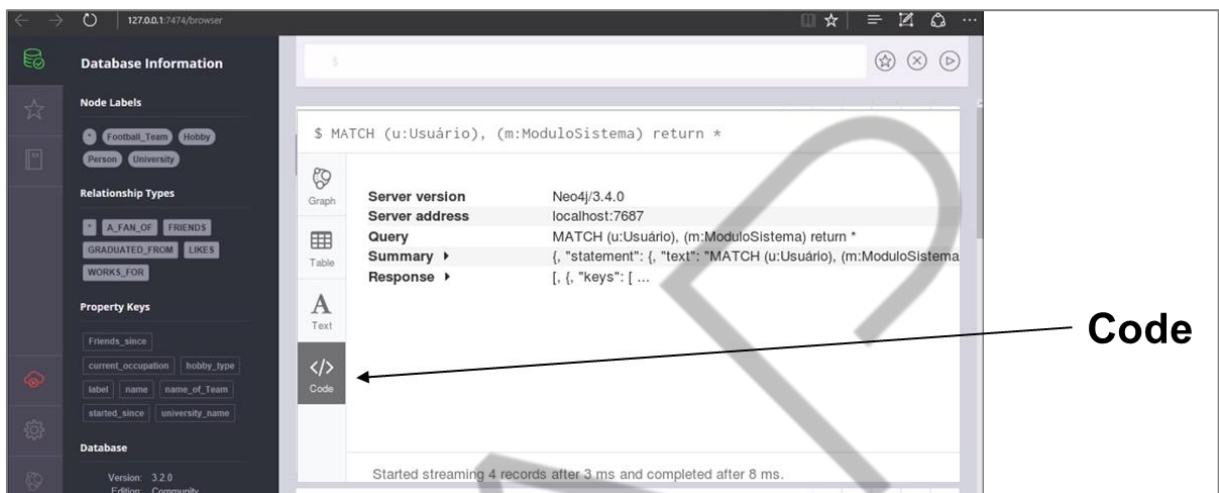


Figura 5.28 – Interface Gráfica – Resultados - CODE

Fonte: Elaborado pelo autor (2020)

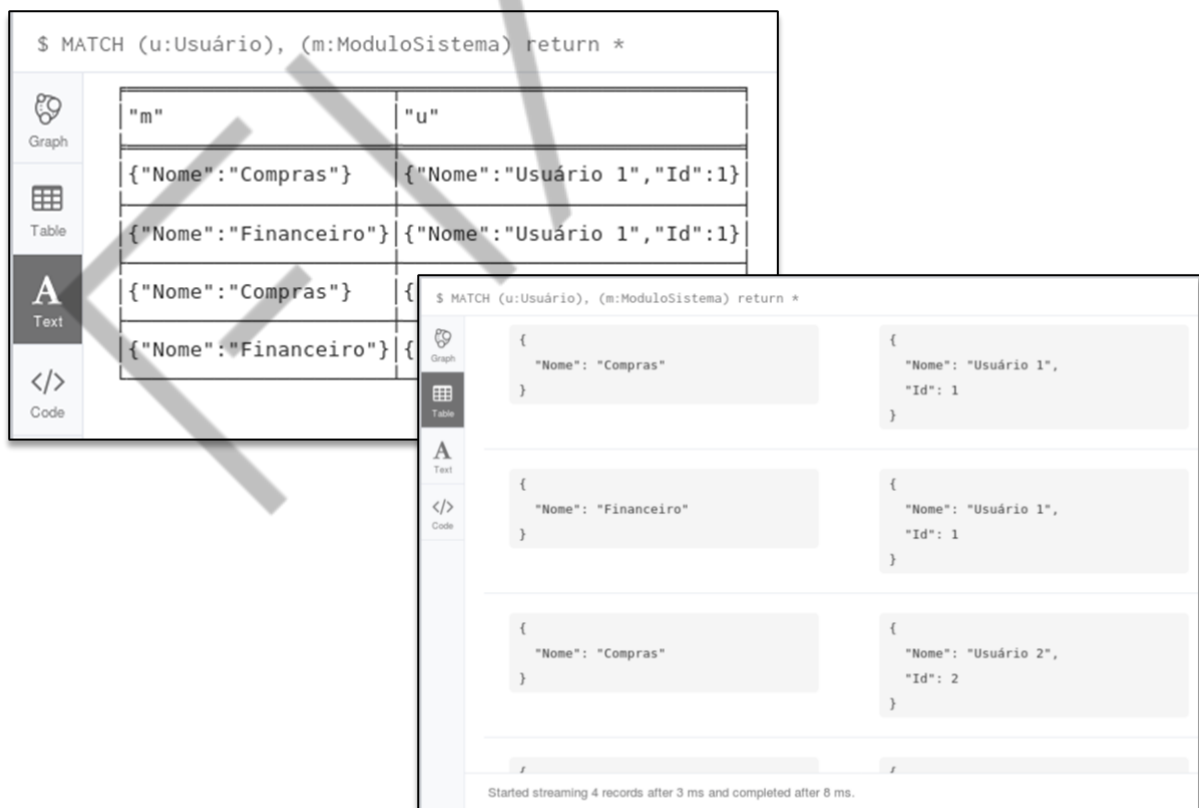


Figura 5.29 – Interface Gráfica – Resultados – TEXT e TABLE

Fonte: Elaborado pelo autor (2020)

A barra lateral se expande para revelar diferentes painéis funcionais para consultas e informações comuns.

- Metadados do banco de dados e informações básicas.
- Scripts salvos organizados em pastas.
- Links de informações para documentos e referência.
- Créditos e informações de licenciamento.

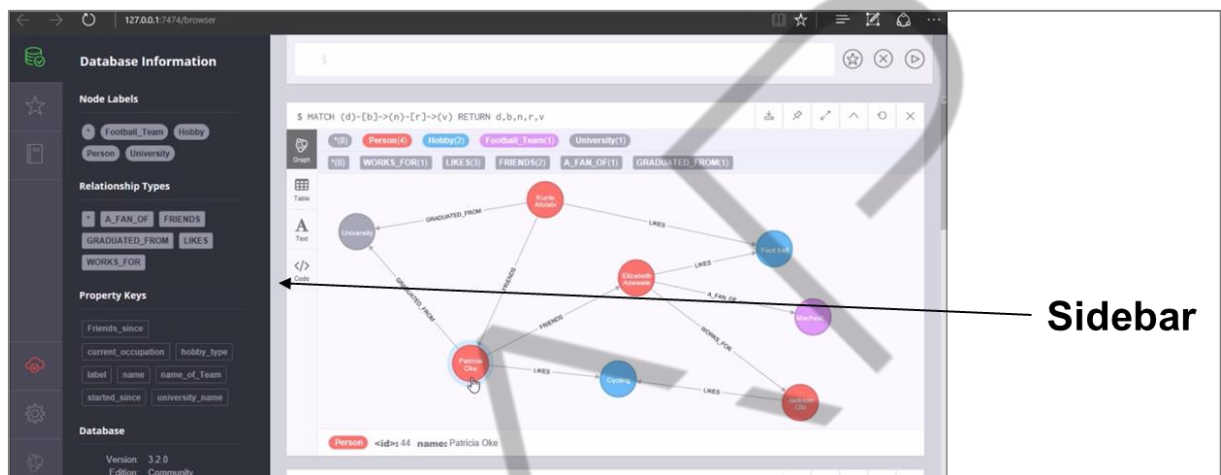


Figura 5.30 – Sidebar
Fonte: Elaborado pelo autor (2020)

5.3.4 Criando Nós

Vamos iniciar nossa jornada **CRUD** (Create, Read, Update e Delete)? Vamos começar com o C de **CREATE**. Usando a linguagem Cypher, iniciaremos com a definição dos objetos dentro do banco de dados. Para criar nós, usaremos o seguinte comando:

CREATE (ee: Pessoa {nome: "Emil", de: "Sweden", klout: 99})

Use:

- **()** parênteses - para indicar um nó
- **ee: Pessoa** - uma variável 'ee' como um alias de banco de dados
- **{}** colchetes para adicionar propriedades ao nó

Atente sempre para a sintaxe, ou seja, o uso de parênteses, chaves e dois pontos.

Para mais detalhes, acesse a documentação online do Neo4J: <https://neo4j.com/docs/developer-manual/3.4/cypher/clauses/create/#create-nodes>.

Veja outros exemplos de criação de nós:

Criando somente um nó:

```
CREATE (n)
```

Criando vários nós:

```
CREATE (n),(m)
```

Criando nós com labels (alias):

```
CREATE (n:Person)
```

```
CREATE (n:Person:American)
```

Repare que, acima, temos a criação do nó N cujo label (alias) é Person, e no comando subsequente, temos a criação do nó N, que possui dois labels: Person e American. Podemos adicionar quantos labels quisermos a um nó. Isso nos traz a flexibilidade de trabalhar com subconjuntos de nós, ou seja, podemos ter nós do tipo Person, que também fazem parte do tipo American, European, Asian etc.

Ideia de subconjuntos, sabe? Criando nó com label e propriedades:

```
CREATE (n:Person { name: 'Andres', title: 'Developer' })
```

Repare que para definir as propriedades de um objeto, seja ele um nó ou um relacionamento (veremos na sequência), os valores devem ser informados dentro das chaves {}, separados por vírgula e no formato chave-valor.

Na criação do nó, pode-se usar um comando do tipo R – Read, para retornar o nó criado, veja:

```
CREATE (a { name: 'Andres' }) RETURN a.name
```

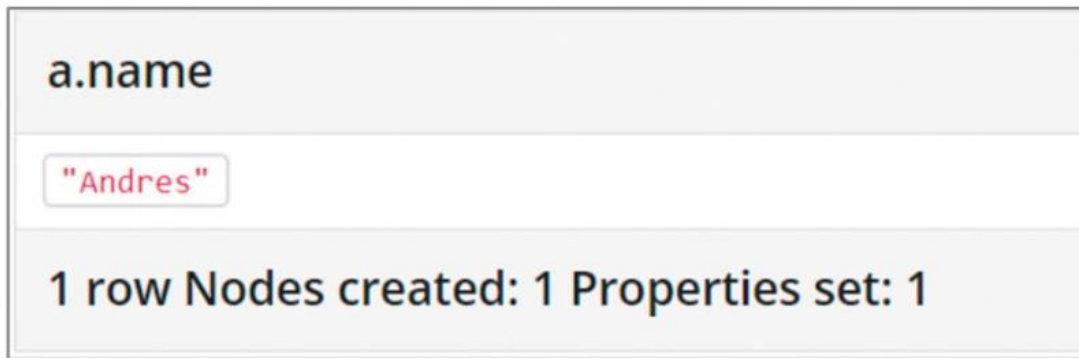


Figura 5.31 – Resultado CREATE
Fonte: Elaborado pelo autor (2020)

Repare que no comando **Return**, estamos usando o label (alias) para especificar que queremos como retorno a propriedade name do nó.

5.3.5 Um pouco do Cypher

A linguagem Cypher usa ASCII-Art para representar padrões. Envolvermos os nós com parênteses que parecem círculos, por exemplo (nó).

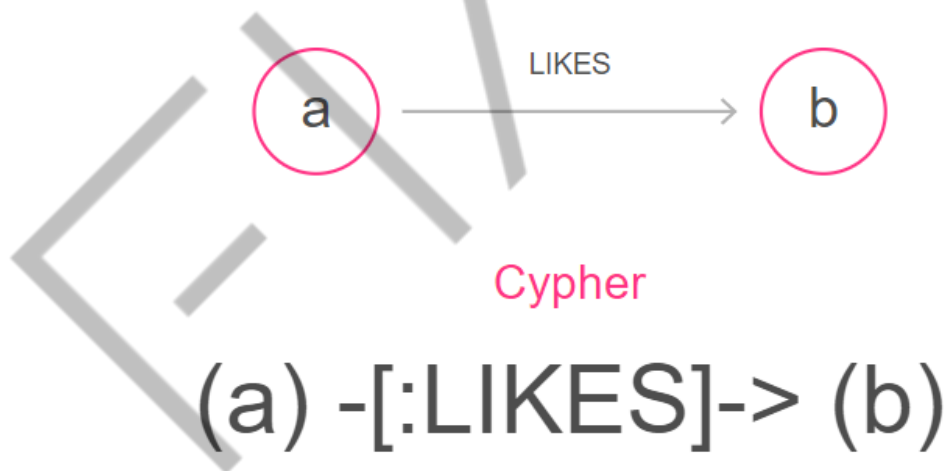


Figura 5.32 – Exemplo Cypher
Fonte: Neo4j (2020)

Se depois quisermos nos referir ao nó, vamos dar uma variável como (p) para pessoa ou (t) para coisa. Em consultas do mundo real, provavelmente usaremos nomes de variáveis mais longos e expressivos, como (pessoa) ou (coisa). Se o nó não for relevante para sua pergunta, você também pode usar parênteses vazios ().

Normalmente, os rótulos relevantes do nó são fornecidos para os relacionamentos. Para utilizar plenamente o poder do banco de dados gráfico,

queremos expressar padrões mais complexos entre nossos nós. Relacionamentos são basicamente uma seta -> entre dois nós. Informações adicionais, como propriedades e labels de relacionamentos, podem ser colocadas entre colchetes dentro da seta.

Isso pode ser:

- tipos de relacionamento como - [: CONHECE |: GOSTA] ->
- um nome de variável - [rel: KNOWS] -> antes do cólon
- propriedades adicionais - [{since: 2010}] ->
- informações estruturais para caminhos de comprimento variável - [: CONHECE * .. 4] ->

Para acessar informações sobre um relacionamento, podemos atribuir uma variável para referência futura. Ele é colocado na frente do cólon - [rel: KNOWS] -> ou fica sozinho - [rel] ->. distinguir entre entidades e otimizar a execução, como (p: Person).

Para ilustrar a criação dos relacionamentos via Cypher, vamos usar outro comando R – **Read**, o comando MATCH, que atua quase como um comando SELECT-WHERE, veja:

```
MATCH (cli:Person) RETURN cli.property
```

Estou solicitando que o nó **PERSON** seja selecionado, e pedindo para que uma propriedade específica seja retornada. Veja o exemplo:

```
MATCH (node1:Label1) - -> (node2:Label2)
```

```
WHERE node1.propertyA = {value}
```

```
RETURN node2.propertyA, node2.property
```

Repare que estamos solicitando dois nós e já criando o relacionamento, representado por (- ->). Isso mesmo, traço, traço, símbolo de maior. Isso indica a direção do relacionamento. Entre os traços (- ->) poderíamos colocar as propriedades do relacionamento e seu label. No caso acima, esses dados foram omitidos.

Observe que os rótulos de nós, os tipos de relacionamentos e os nomes de propriedades fazem distinção entre maiúsculas e minúsculas no Cypher. Todas as

outras cláusulas, palavras-chave e funções não são, mas devem ser encaixadas consistentemente de acordo com o estilo usado aqui.

Para mais informações sobre criação de nós, use o manual online Neo4J: <https://neo4j.com/docs/developer-manual/3.4/cypher/clauses/create/#create-nodes>.

Para criar um relacionamento entre dois nós, primeiro obtemos os dois nós. Quando os nós são carregados, simplesmente criamos um relacionamento entre eles. Veja este outro exemplo:

```
MATCH (a:Person),(b:Person) WHERE a.name = 'A' AND b.name = 'B'  
CREATE (a) -[r:RELTYPE]-> (b)  
RETURN type(r)
```

Repare que no comando acima, informamos, no relacionamento um label, r.

Veja agora alguns padrões de relacionamentos, muito usados:

Registrando e mantendo o “amigo do amigo”:

```
(user)-[:KNOWS]-(friend)-[:KNOWS]-(foaf)
```

Indo pelo menor caminho:

```
path = shortestPath( (user)-[:KNOWS*..5]-(other) )
```

Aqui, veja que dentro do relacionamento, há um label, especificando que nessa consulta, somente deverão ser trazidos os nós cujos relacionamentos têm label = a KNOWS e com no máximo até 5 níveis de relacionamento.

Filtragem colaborativa:

```
(user)-[:PURCHASED]->(product)<-[:PURCHASED]-()-[:PURCHASED]->(otherProduct)
```

Muito usada em recomendações de produtos.

Navegação em árvore:

```
(root)<-[:PARENT*]-(leaf:Category)-[:ITEM]->(data:Product)
```

No momento de criarmos relacionamentos para vários nós, podemos fazer uso do comando FOREACH e, assim, criar múltiplos relacionamentos em um comando só. veja:

```
MATCH (you:Person {name:"You"})
```

```
FOREACH (name in ["Johan","Rajesh","Anna","Julia","Andrew"]
```

```
CREATE (you)-[:FRIEND]->(:Person {name:name}))
```

Outra forma de criar os nós e relacionamentos é por meio do uso dos comandos de LOAD CSV, veja o exemplo para importar dados de um arquivo CSV para o Neo4j:

```
"1","ABBA","1992"
"2","Roxette","1986"
"3","Europe","1979"
"4","The Cardigans","1992"
```

Figura 5.33 – Exemplo Arquivo CSV sem cabeçalho
Fonte: Elaborado pelo autor (2020)

```
LOAD CSV FROM '{csv-dir} / artists.csv' AS line CREATE (:Artist {name: line[1],  
year: toInteger (line[2] ) } )
```

Ou, usando o nome de colunas para nomear os nós:

```
"Id","Name","Year"
"1","ABBA","1992"
"2","Roxette","1986"
"3","Europe","1979"
"4","The Cardigans","1992"
```

Figura 5.34 – Exemplo arquivo CSV com cabeçalho
Fonte: Elaborado pelo autor (2020)

```
LOAD CSV WITH HEADERS FROM '{csv-dir} / artists-with  
headers.csv' AS line CREATE (:Artist {name: line.Name, year: toInteger (line.Year) }  
)
```

5.3.6 Tutorial Movies

Aproveitando os tutoriais existentes no Neo4J, vamos praticar um pouco? A interface gráfica aberta, tanto na nuvem como na estação de trabalho, apresenta as mesmas funcionalidades necessárias para nossos exercícios.

Ao lado esquerdo podemos expandir as opções clicando sobre o ícone da base de dados. Para escondê-las novamente, basta clicar novamente sobre o ícone.

Temos desde informações sobre a base de dados, exemplos de scripts, tutorial até parâmetros de configurações da base.

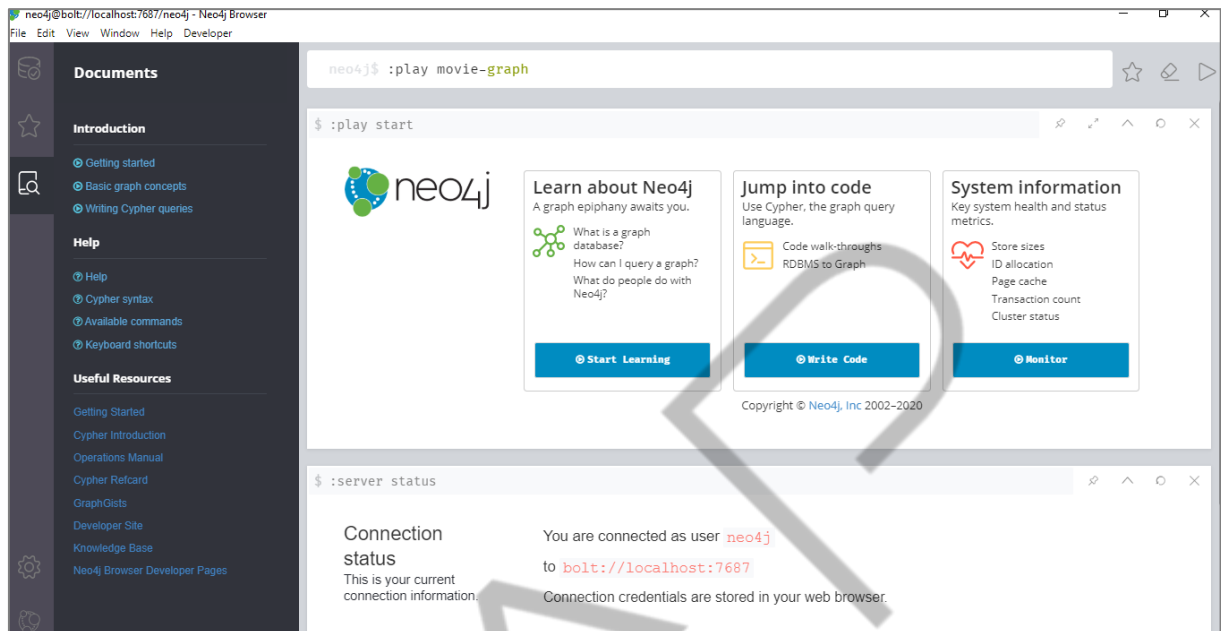


Figura 5.35 – Interface gráfica web
Fonte: NEO4J (2020)

Para entendermos melhor os conceitos de nós e arestas e os principais comandos básicos realizados sobre um grafo, realizaremos o tutorial Movies. Para isso, basta digitar na linha de comando `:play movie-graph` e executar (último ícone no lado direito da linha de comando).

```
:play movie-graph
```

Código-fonte 5.1 – Comando execução tutorial Movies
Fonte: NEO4J (2020)

Este tutorial tem oito etapas, e para cada uma, apresenta uma breve descrição do objetivo e os comandos a serem realizados. A qualquer momento, você pode avançar ou retornar no tutorial com as setas do lado direito ou esquerdo do tutorial.

A etapa dois tem o objetivo de criar o grafo de filmes, e para isso já traz o script com os comandos Cypher. Ao clicar na janela dos comandos, eles são transferidos para a linha de comando (508 linhas); e, ao executarmos, o grafo será criado resultando 171 nodes e 253 relationships. Os nós Person e Movie e os relacionamentos Acted_in, Directed, Follows, Produced, Reviewed e Wrote são criados. O modelo do grafo pode ser obtido com o comando:

```
CALL db.schema.visualization()
```

Código-fonte 5.2 – Exibir modelo de dados

Fonte: NEO4J (2020)

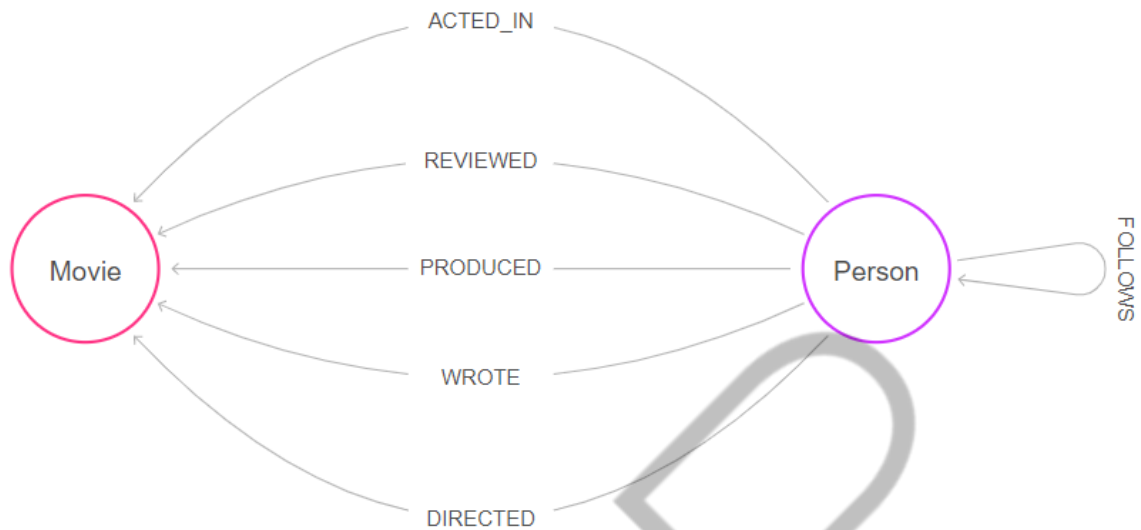


Figura 5.36 – Modelo de Grafo “Movies”

Fonte: NEO4J (2020)

Observemos as sintaxes para criação dos nós – comando `CREATE` – para *Movie* e *Person* – labels associados aos nós. Para o nó *Movie*, temos duas propriedades *title*, *released* e *tagline*. O label é *Movie* e a variável para receber o retorno do comando é *TheMatrix*.

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999,
tagline:'Welcome to the Real World'})
```

Código-fonte 5.3 – Exemplo do uso do comando `Create` node *Movie*

Fonte: NEO4J (2020)

Para o nó *Person* temos as propriedades *name* e *born*. Nesse caso a variável *Keanu* recebe o resultado do comando.

```
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
```

Código-fonte 5.4 – Exemplo do uso do comando `Create` node *Person*

Fonte: NEO4J (2020)

Já para criar o relacionamento *ACTED_IN* entre o nó *Movie* = “Matriz” e o nó *Person* = “Keanu Reeves”, quando realizado no mesmo comando de execução, usar as variáveis associadas na criação dos nós. No exemplo o relacionamento *ACTED_IN* com a propriedade *roles*.

```
CREATE
(Keanu)-[:ACTED_IN {roles:['Neo']}]>(TheMatrix)
```

Código-fonte 5.5 – Exemplo do uso do comando `Create` relationship

Fonte: NEO4J (2020)

Alternativamente, os relacionamentos podem ter outra sintaxe associada.

Se o relacionamento for criado após a criação dos nós em comandos independentes, é necessário recuperar os nós envolvidos (MATCH) e, após, criar o relacionamento (CREATE). Tudo num mesmo bloco de comando.

```
MATCH (Keanu:Person), (TheMatrix:Movie)
WHERE Keanu.name='Keanu Reeves' AND TheMatrix.title='The
Matrix'
CREATE
(Keanu)-[:ACTED_IN {roles:['Neo']}]>(TheMatrix)
Return Keanu, TheMatrix
```

Código-fonte 5.6 – Exemplo do uso do comando Create relationship
Fonte: NEO4J (2020)

Se você deseja garantir que apenas **um** relacionamento seja criado, não importa quantas vezes você execute essa instrução, use o MERGE.

```
MATCH (Keanu:Person), (TheMatrix:Movie)
WHERE Keanu.name='Keanu Reeves' AND TheMatrix.title='The
Matrix'
MERGE
(Keanu)-[:ACTED_IN {roles:['Neo']}]>(TheMatrix)
Return Keanu, TheMatrix
```

Código-fonte 5.7 – Exemplo do uso do comando Create relationship
Fonte: NEO4J (2020)

Ou ainda desta forma:

```
MATCH (Keanu:Person {name: 'Keanu Reeves'}), (TheMatrix:Movie
{title:'The Matrix'})
CREATE (Keanu)-[:ACTED_IN {roles:['Neo']}]>(TheMatrix)
Return Keanu, TheMatrix
```

Código-fonte 5.8 – Exemplo do uso do comando Create relationship
Fonte: NEO4J (2020)

A etapa três recupera dados presentes nos nós. O resultado pode ser apresentado na forma de grafo, tabela, texto ou código (barra ao lado do resultado).

Tente, agora, fazer as seguintes tarefas dentro da base de atores:

- Crie um filme chamado Tropa de Elite, com os atributos (released 2007, tagline Pede pra sair).
- Crie dois atores que atuaram no filme acima: Wagner Moura, nascido em 1976; e Maria Ribeiro, nascida em 1975.
- Relacione os dois atores ao filme criado, informando nos relacionamentos o personagem que cada ator interpretou.

5.3.7 Consultando os dados

Na sequência do processo de criação de nós, relacionamentos, propriedades e labels, vamos ver o R- **Read** do **CRUD**.

Vamos detalhar mais a cláusula **MATCH**, pois ela permite que você especifique os padrões que o Neo4j procurará no banco de dados. Essa é a principal maneira de obter dados no conjunto atual de ligações.

MATCH é frequentemente usado em conjunto da cláusula **WHERE** que adiciona restrições ou predicados aos padrões **MATCH**, tornando-os mais específicos. Os predicados fazem parte da descrição do padrão e não devem ser considerados como um filtro aplicado somente após a correspondência ser feita. Isso significa que **WHERE** sempre deve ser colocado com a cláusula **MATCH** a que pertence.

MATCH pode ocorrer no início da consulta ou mais tarde, possivelmente após um **WITH**. Se for a primeira cláusula, nada terá sido vinculado ainda, e Neo4j projetará uma pesquisa para encontrar os resultados correspondentes à cláusula e quaisquer predicados associados especificados em qualquer parte do **WHERE**.

Isso poderia envolver uma varredura do banco de dados, uma pesquisa de nós com um determinado rótulo ou uma pesquisa de um índice para localizar pontos de início para a correspondência de padrões.

Os nós e relacionamentos encontrados por essa pesquisa estão disponíveis como elementos de padrão vinculados e podem ser usados para correspondência de padrões de subgráficos. Eles também podem ser usados em qualquer outra cláusula **MATCH**, na qual o Neo4j usará os elementos conhecidos, e a partir daí encontrará outros elementos desconhecidos.

O Cypher é declarativo e, portanto, geralmente, a consulta em si não especifica o algoritmo a ser usado para executar a pesquisa. Neo4j trabalhará automaticamente a melhor abordagem para encontrar os nós iniciais e os padrões de correspondência.

Predicados na **WHERE** podem ser avaliados antes da correspondência de padrões, durante a correspondência de padrões ou após encontrar correspondências. No entanto, há casos em que você pode influenciar as decisões tomadas pelo compilador de consulta.

Para alguns exemplos do uso do MATCH, vamos usar este grafo:

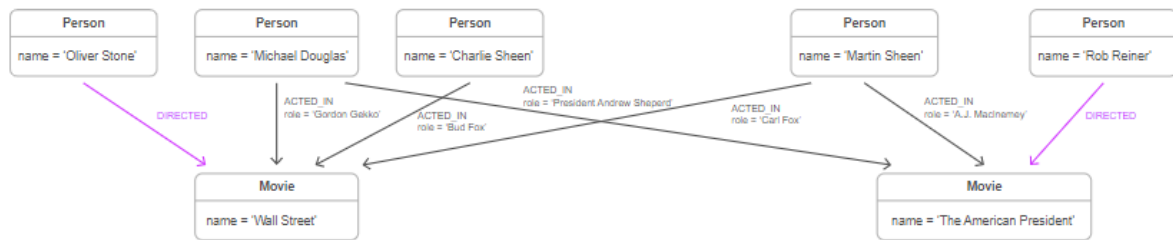


Figura 5.37 – Modelo de Grafo
Fonte: NEO4J (2020)

Retornando os dados dos nós indistintamente:

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

Obter todos os nós com um rótulo neles é feito com um padrão de nó único:

```
MATCH (movie:Movie) RETURN movie.title
```

Trazendo nós relacionados, o símbolo **--** significa relacionado a, sem considerar o tipo ou direção do relacionamento (vamos consultar todos os filmes cujo diretor ou ator é Oliver Stone):

```
MATCH (director { name: 'Oliver Stone' })--(movie) RETURN movie.title
```

Busca com labels: Para restringir seu padrão com rótulos nos nós, adicione-o aos nós do padrão, usando a sintaxe do rótulo (agora vamos consultar somente os filmes que foram dirigidos por Oliver Stone, repare que o relacionamento agora tem uma direção):

```
MATCH (:Person { name: 'Oliver Stone' })-->(movie) RETURN movie.title
```

Se uma variável for necessária, seja para filtrar propriedades do relacionamento ou para retornar o relacionamento, é assim que você introduz a variável (retornará o tipo de relacionamento de saída do Oliver Stone, ou seja: DIRECTED):

```
MATCH (:Person { name: 'Oliver Stone' })-[r]->(movie) RETURN type(r)
```

Quando você conhece o tipo de relacionamento com o qual deseja corresponder, é possível especificá-lo usando dois pontos com o tipo de relacionamento:

```
MATCH (wallstreet:Movie { title: 'Wall Street' })<-[:ACTED_IN]-<(actor)
RETURN actor.name
```

Para combinar em um dos vários tipos, você pode especificar isso encadeando-os com o símbolo pipe |, praticamente uma cláusula **OR**:

```
MATCH (wallstreet {title: 'Wall Street' })<-[:ACTED_IN|DIRECTED]-<(person)
RETURN person.name
```

Consultas de menor caminho. Sabe aquela máxima que tenta buscar a quantas pessoas de distância você está do presidente da república? É mais ou menos isso:

```
MATCH (martin:Person { name: 'Martin Sheen' }),(oliver:Person { name: 'Oliver Stone' })
p = shortestPath((martin)-[*..15]-(oliver))
RETURN p
```

Isso significa: encontrar um único caminho mais curto entre dois nós, desde que o caminho tenha no máximo 15 relacionamentos. Dentro dos parênteses, você define um único link de um caminho - o nó inicial, o relacionamento de conexão e o nó final. As características que descrevem o relacionamento como tipo de relacionamento, salto máximo e direção são usadas ao encontrar o caminho mais curto. Se houver uma cláusula **WHERE** após a correspondência de um **shortestPath**, os predicados relevantes serão incluídos no **shortestPath**.

```
(1)-[:ACTED_IN,1]->(5)<-[:DIRECTED,3]-(3)
match(kea:Person {name: 'Keanu Reeves'}), (tom: Person { name: 'Tom Cruise'})
return shortestPath((kea)-[*]-(tom))
match(p1:Person), (p2: Person),
p = shortestPath((p1:Person)-[*]-(p2:Person))
where p1.name <> p2.name
return max(length(p))
```

5.3.8 Filtrando as consultas

Você pode usar os operadores booleanos **AND**, **OR**, **XOR** e **NOT**. Por exemplo:
`MATCH (n) WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = 'Tobias') OR NOT (n.name = 'Tobias' OR n.name = 'Peter') RETURN n.name, n.age.`

Veja alguns outros exemplos:

Filtrando em labels:

```
MATCH (n)
WHERE n:Swedish
RETURN n.name, n.age
```

Filtrando em propriedades dos nós:

```
MATCH (n)
WHERE n.age < 30
RETURN n.name, n.age
```

Filtrando em propriedades dos relacionamentos:

```
MATCH (n)-[k:KNOWS]->(f)
WHERE k.since < 2000
RETURN f.name, f.age, f.email
```

Repare que usamos sempre os labels (alias) para referenciar as propriedades dentro do comando. Filtrando em relações existentes, quando uma propriedade existir:

```
MATCH (n)
WHERE exists(n.belt)
RETURN n.name, n.belt
```

Filtrando com base em strings:

```
MATCH (n)
WHERE n.name STARTS WITH 'Pet' OR ENDS WITH 'ter'
```

```
RETURN n.name, n.age
```

Filtrando usando negações:

```
MATCH (n)
WHERE NOT n.name CONTAINS 'ete'
RETURN n.name, n.age
```

Filtrando usando “patterns”:

```
MATCH (tobias { name: 'Tobias' }),(others)
WHERE others.name IN ['Andres', 'Peter'] AND (tobias)<--(others)
RETURN others.name, others.age
```

Filtrando usando NULLS e Order By:

```
MATCH (n)
WHERE n.belt = 'white' OR n.belt IS NULL RETURN n.name, n.age, n.belt
ORDER BY n.name
```

Se parar para analisar, uma vez de posse da sintaxe dos comandos Cypher, tudo vai ficando bem intuitivo e fácil, não é?

Para mais informações sobre derivações dos comandos apresentados, veja documentação on-line:

<<https://neo4j.com/docs/developer-manual/3.4/cypher/clauses/where/>>

Vamos seguir praticando um pouco mais e sempre observando as sintaxes para recuperação de dados nos nós. Pode-se buscar um nó com uma condição de igualdade ou com limites com a cláusula where.

```
MATCH (tom {name: "Tom Hanks"}) RETURN tom
```

Código-fonte 5.9 – Exemplo do uso do comando Match 1
Fonte: NEO4J (2020)

```
MATCH (nineties:Movie) WHERE nineties.released >= 1990 AND
nineties.released < 2000 RETURN nineties.title
```

Código-fonte 5.10 – Exemplo do uso do comando Match 2
Fonte: NEO4J (2020)

A etapa quatro explora a recuperação de dados percorrendo os nós e relacionamentos entre eles. O nome do diretor do filme “Cloud Atlas” será obtido do nó Person por meio do relacionamento DIRECTED.

```
MATCH (cloudAtlas {title: "Cloud Atlas"})<-[:DIRECTED]-  
      (directors)  
RETURN directors.name
```

Código-fonte 5.11 – Exemplo do uso do comando Match 3

Fonte: NEO4J (2020)

Todos os atores que atuaram junto com o ator “Tom Hanks”. A partir do nó Person igual a Tom Hanks percorrendo o relacionamento ACTED_IN e chega no nó m (Movie) e verifica todos os nós Person que chegam aos filmes recuperados.

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-  
      [:ACTED_IN]-(coActors) RETURN coActors.name
```

Código-fonte 5.12 – Exemplo do uso do comando Match 4

Fonte: NEO4J (2020)

Uma nova forma de pensar em como recuperar os dados no banco de dados!

Na etapa cinco, exploramos a distância entre os nós e o menor caminho entre dois nós informados. A cada nó passado, denominam-se “hops” – saltos. Relacionar todos filmes e atores partindo de “Kevin Bacon” a uma distância de quatro saltos.

```
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)  
RETURN DISTINCT hollywood
```

Código-fonte 5.13 – Exemplo do uso do comando Match 5

Fonte: NEO4J (2020)

E qual seria o menor caminho no grafo para ligar “Kevin Bacon” e “Meg Ryan”? Ou seja, dados dois nós, estabelecer o menor caminho entre eles.

```
MATCH p=shortestPath(  
      (bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person  
      {name:"Meg Ryan"})  
      )  
RETURN p
```

Código-fonte 5.14 – Exemplo do uso do comando Match e shortestPath

Fonte: NEO4J (2020)

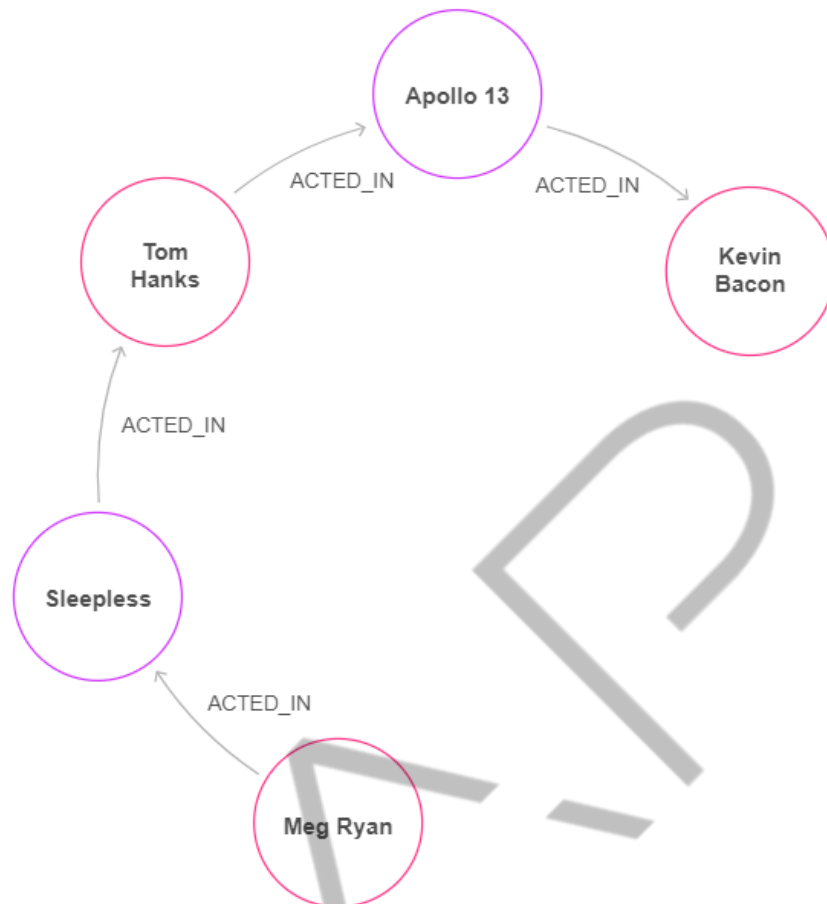


Figura 5.38 – ShortestPath
Fonte: NEO4J (2020)

Na etapa seis já utilizamos comandos para recomendação! Vamos recomendar novos atores para contracenar com “Tom Hanks” a partir daqueles que já contracenaram com ele (vizinhança imediata) e ainda não contracenaram com o Tom.

```

MATCH (tom:Person {name:"Tom Hanks"})
  -[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
  (coActors)
  -[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT
  tom)-[:ACTED_IN]->()<-[:ACTED_IN]-(cocoActors)
  AND tom <> cocoActors
RETURN cocoActors.name AS Recommended, count(*) AS Strength
ORDER BY Strength DESC
  
```

Código-fonte 5.15 – Exemplo do uso do comando Match e Where
Fonte: NEO4J (2020)

5.3.9 Alterando os dados

Seguindo no **CRUD**, vamos focar um pouco, agora, na seção U – **Update**, e para isso, usaremos a cláusula **SET**, que:

- É usada para atualizar rótulos em nós e propriedades em nós e relacionamentos.
- Também pode ser usada com mapas de parâmetros para definir propriedades.
- Definir rótulos em um nó é uma operação idempotente – se você tentar definir um rótulo em um nó que já tenha esse rótulo, nada acontece. As estatísticas da consulta informam se algo precisa ser feito ou não.

Veja alguns exemplos:

Mudando uma propriedade de um nó ou relacionamento:

```
MATCH (n { name: 'Andres' })  
SET n.surname = 'Taylor'  
RETURN n.name, n.surnam
```

O que aconteceria, se essa propriedade já estivesse sido preenchida anteriormente? Faça o teste na base de filmes (tutorial) e veja.

```
MATCH (n { name: 'Andres' })  
SET n.name = NULL RETURN n.name, n.age
```

Removendo uma propriedade:

```
MATCH (at { name: 'Andres' }),(pn { name: 'Peter' })  
SET at = pn  
RETURN at.name, at.age, at.hungry, pn.name, pn.age
```

Copiando propriedades:

Alterando valores a partir de dados agregados:

```
MATCH (n {name: 'John'})-[:FRIEND]-(friend)  
WITH n, count(friend) AS friendsCount  
SET n.friendsCount = friendsCount  
RETURN n.friendsCount
```

Informando mais de uma propriedade ao mesmo tempo:

```
MATCH (n { name: 'Andres' })  
SET n.position = 'Developer', n.surname = 'Taylor'
```

Vários labels em um nó:

```
MATCH (n { name: 'Emil' })  
SET n:Swedish:Bossman  
RETURN n.name, labels(n) AS labels
```

Para mais detalhes, veja documentação on-line:

<<https://neo4j.com/docs/developer-manual/3.4/cypher/clauses/set/>>.

5.3.10 Removendo os dados

Seguindo no **CRUD**, vamos focar um pouco, agora, na seção D – **Delete** e para isso usaremos as cláusulas **DELETE** e **DETACH**:

- A cláusula **DELETE** é usada para excluir elementos gráficos - nós, relacionamentos ou caminhos.
- Lembre-se de que você não pode excluir um nó sem excluir os relacionamentos que iniciam ou terminam no nó. Exclua explicitamente os relacionamentos ou use **DETACH DELETE**.

Veja alguns exemplos:

Removendo um nó:

```
MATCH (n:Person { name: 'UNKNOWN' }) DELETE n
```

Removendo somente os relacionamentos:

```
MATCH (n { name: 'Andres' })-[r:KNOWS]->() DELETE r
```

Removendo todos os nós e seus relacionamentos:

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

Mais detalhes e outros exemplos, podem ser vistos na documentação online Neo4J: <<https://neo4j.com/docs/developer-manual/3.4/cypher/clauses/delete>>.

Porém há uma outra cláusula que trata de remoções, é a cláusula **REMOVE**, usada para remover propriedades e rótulos dos elementos gráficos. Muito usada para remover rótulos de um nó, é uma operação idempotente: se você tentar remover um rótulo de um nó que não tenha esse rótulo, nada acontecerá. As estatísticas da consulta informam se algo precisa ser feito ou não.

Removendo uma propriedade:

```
MATCH (a { name: 'Andres' })  
  
REMOVE a.age  
  
RETURN a.name, a.age
```

O Neo4j não permite armazenar **NULL** em propriedades. Em vez disso, se nenhum valor existir, a propriedade simplesmente não estará lá. Portanto, **REMOVE** é usado para remover um valor de propriedade de um nó ou de um relacionamento.

Removendo labels:

```
MATCH (n { name: 'Peter' })  
  
REMOVE n:German  
  
RETURN n.name, labels(n)
```

Removendo múltiplos labels:

```
MATCH (n { name: 'Peter' })  
  
REMOVE n:German:Swedish  
  
RETURN n.name, labels(n)
```

Mais detalhes, você encontrará aqui:

<<https://neo4j.com/docs/developer-manual/3.4/cypher/clauses/remove/>>.

Tente agora, remover o nó do Wagner Moura. Analise o resultado e corrija possíveis erros. Agora podemos excluir todo grafo – seus nós e relacionamentos.

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

Código-fonte 5.16 – Exemplo do uso do comando Delete

Fonte: NEO4J (2020)

5.3.11 Fixação dos comandos e consultas agregadas

Pratique com os comandos e explicações abaixo:

Contando quanto filmes por ano existem no grafo.

```
MATCH (m:Movie)
RETURN m.year as year, count(*) as total
ORDER BY year DESC;
```

Código-fonte 5.17 – Exemplo do uso da cláusula count (*)

Fonte: NEO4J (2020)

Várias operações de agregação juntas.

```
MATCH (m:Movie)
RETURN m.type,
       count(*) as movie_count,
       min(m.votes) as min_votes,
       max(m.votes) as max_votes,
       sum(m.votes) as sum_of_votes,
       avg(m.votes) as average_votes;
```

Código-fonte 5.18 – Exemplo do uso das cláusulas para agregação

Fonte: NEO4J (2020)

Explorando os metadados e verificando os relacionamentos existentes no grafo.

```
MATCH (m)-[r]->(n)
RETURN labels(m), type(r), labels(n), count(*) as total
ORDER BY total DESC;
```

Código-fonte 5.19 – Exemplo do uso do comando Match e count (*) 1

Fonte: NEO4J (2020)

Os dez atores que mais atuaram juntos.

```
MATCH (p1:Person) -[:ACTED_IN]-> (m:Movie),
       (p2:Person) -[:ACTED_IN]-> (m:Movie)
WHERE p1.person_id <> p2.person_id
RETURN p1.name as name1, p2.name as name2, count(*) as total
ORDER BY total desc, name1, name2
LIMIT 10;
```

Código-fonte 5.20 – Exemplo do uso do comando Match e count (*) 2

Fonte: NEO4J (2020)

A qual distância em termos de número de saltos estão os atores Jennifer Lawrence e Daniel Radcliff.

```
MATCH path=(m:Person {name : 'Jennifer Lawrence'})
          -[:ACTED_IN*6]-
          (n:Person {name : 'Daniel Radcliffe'})
RETURN path;
```

Código-fonte 5.21 – Exemplo do uso do comando Match 6

Fonte: NEO4J (2020)

Número de pessoas associadas ao Kevin Bacon no grafo pela aresta ACTED_IN e a distância total (saltos) entre elas.

```
MATCH path=allshortestpaths(
  m:Person {name : "Kevin Bacon"} )
  -[:ACTED_IN*]- (n:Person))
WHERE n.person_id <> m.person_id
RETURN length(path)/2 as bacon_number, count(distinct
n.person_id) as total
ORDER BY bacon_number;
```

Código-fonte 5.22 – Exemplo do uso da cláusula allshortestpaths

Fonte: NEO4J (2020)

Adicionando uma propriedade tagline no Movie Mystic River.

```
MATCH (m:Movie)
WHERE m.title = "Mystic River"
SET m.tagline = "We bury our sins here, Dave. We wash them
clean."
RETURN m;
```

Código-fonte 5.23 – Exemplo do uso do comando Set 1

Fonte: NEO4J (2020)

Alterando o conteúdo da propriedade released no Movie Mistic River.

```
MATCH (m:Movie)
WHERE m.title = "Mystic River"
SET m.released = 2003
RETURN m;
```

Código-fonte 5.24 – Exemplo do uso do comando Set 2

Fonte: NEO4J (2020)

Alterando propriedades no relacionamento.

```
MATCH (kevin)-[r:ACTED_IN]->(mystic)
WHERE kevin.name="Kevin Bacon"
AND mystic.title="Mystic River"
SET r.roles = ["Sean Devine"]
RETURN r;
```

Código-fonte 5.25 – Exemplo do uso do comando Set 3

Fonte: NEO4J (2020)

Deletando nó e seus relacionamentos.

```
MATCH (emil:Person {name:"Emil Eifrem"})  
OPTIONAL MATCH (emil)-[r]-()  
DELETE emil,r;
```

Código-fonte 5.26 – Exemplo do uso do comando Delete

Fonte: NEO4J (2020)

5.3.12 Módulos e suas funcionalidades

São muitos os módulos para expandir as funcionalidades do banco de dados Neo4j. Destacamos aqui seus plugins APOC, Graph Data Science Library, GraphQL e Neo4j Streams.

5.3.12.1 APOC

APOC significa *Awesome Procedure On Cypher* sendo um conjunto de procedimentos e funções definidos e implementado por usuários do Neo4j. Foi introduzido com o Neo4j 3.0. A biblioteca APOC consiste em muitos (cerca de 450) procedimentos e funções, que podem ser utilizados em muitas tarefas diferentes em áreas como:

- Algoritmos sobre grafos.
- Metadados.
- Índices manuais e índices de relacionamento.
- Pesquisa de texto completo.
- Integração com outros bancos de dados como MongoDB, ElasticSearch, Cassandra e bancos de dados relacionais.
- Expansão de caminho.
- Importar e exportar dados.
- Funções de data e hora.
- Carregamento de XML e JSON de APIs e outros arquivos.

- Tratamento de strings e função de texto.

Na interface gráfica de acesso ao Neo4j no comando de linha, pode-se explorar suas funcionalidades.

```
:play apoc
```

Código-fonte 5.27 – Acessando módulo APOC
Fonte: NEO4J (2020)

Como um plugin pode ser instalado após a criação do banco de dados.

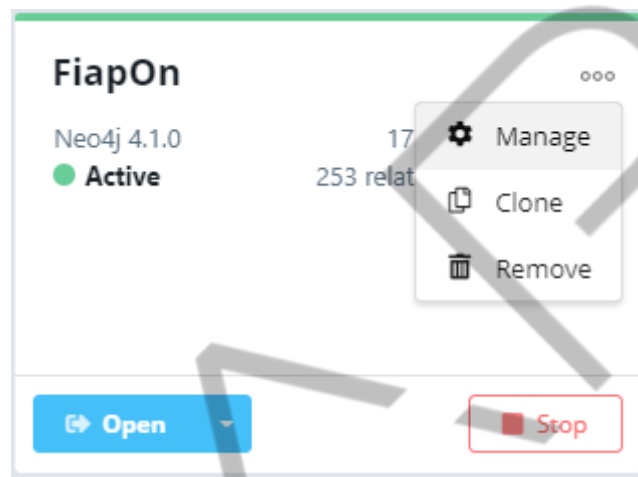


Figura 5.39 – Ativando APOC - Banco de Dados - Manage
Fonte: NEO4J (2020)

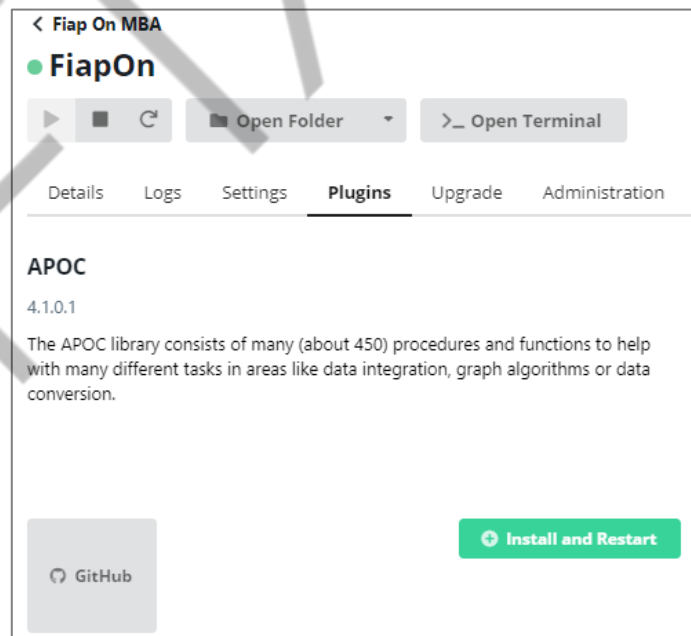


Figura 5.40 – Ativando APOC - Plugins APOC
Fonte: NEO4J (2020)

5.3.12.2 Graph Data Science Library

Uma nova biblioteca para facilitar o trabalho dos cientistas de dados no uso de técnicas de aprendizado de máquina baseadas em grafos para extrair padrões e insights de seus dados e inferir um comportamento com base em dados e estruturas de rede conectadas. Combina um espaço de trabalho de análise de grafos nativo e banco de dados de grafos com algoritmos escaláveis e visualização de grafos para uma experiência confiável e fácil de usar.

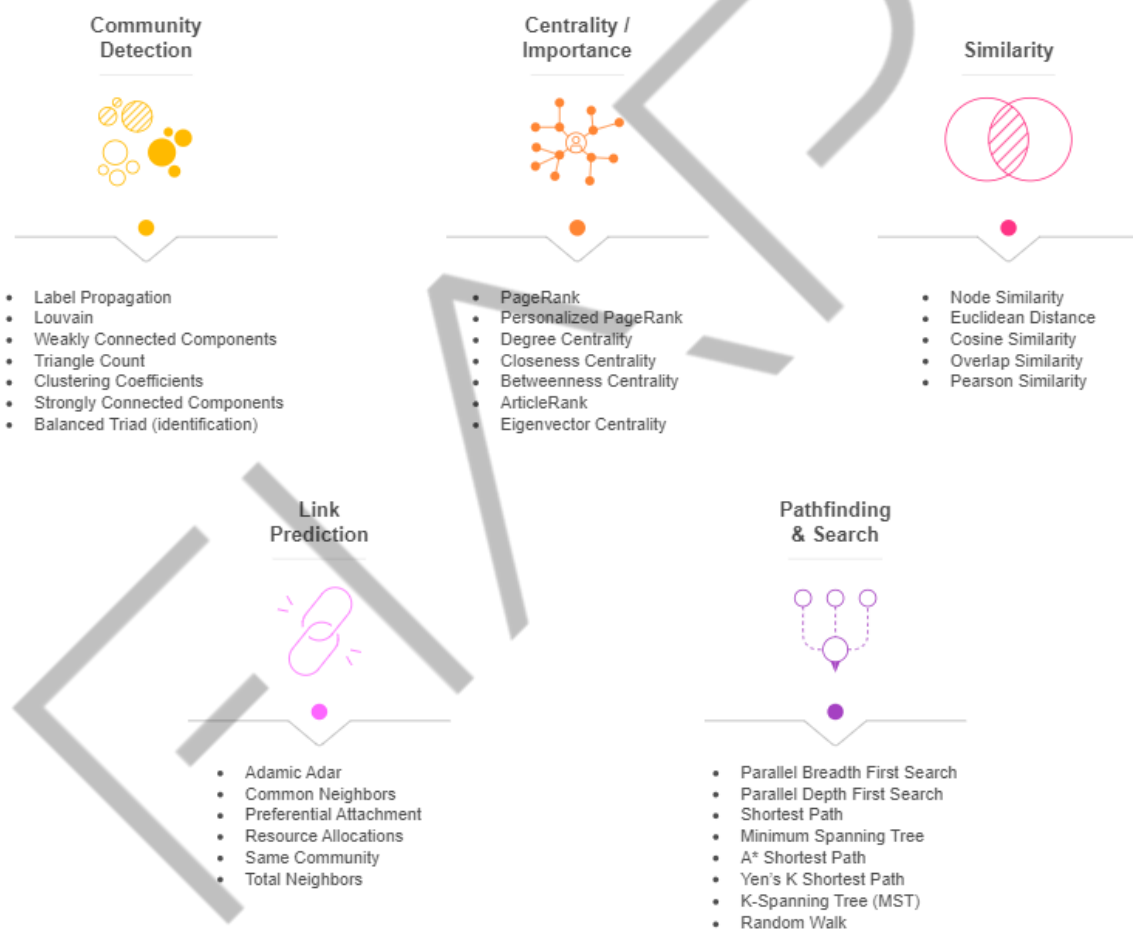


Figura 5.41 – Graph Data Science Library
Fonte: NEO4J (2020)

Possui um conjunto de algoritmos de grafos e um componente de visualização de grafos chamado Bloom. Os recursos são gravados em Java e acessados como procedimentos armazenados. A biblioteca contém um total de 48 algoritmos em cinco categorias principais, que são úteis para diferentes casos de uso:

- Algoritmos de detecção de comunidade, que podem ser usados para detectar subgrafos ou comunidades distintas no grafo maior. Essa classe

de algoritmos é útil para coisas como segmentação de clientes e identificação de pontos em comum entre usuários.

- Algoritmos de centralidade e importância, que informam ao usuário a importância de um determinado nó com base em alguma suposição de importância. PageRank é um exemplo desse tipo de algoritmo.
- Algoritmos de busca de caminhos, que podem calcular o caminho mais curto, mais rápido ou o melhor para chegar entre dois nós A e B no grafo, como o conhecido Dijkstra.
- Algoritmos de similaridade, que podem detectar a proximidade de dois pontos no grafo com base nos nós vizinhos.
- Algoritmos de previsão de link, que preveem a probabilidade de haver uma borda entre um determinado par de nós.

A biblioteca permite que os usuários executem vários algoritmos em seus grafos na memória para realizar o que precisam ou, se não precisam persistir os resultados, podem transmitir os resultados para Python.

Quando os resultados dos algoritmos forem subgrafos materializados mantidos na memória, é melhor rodar em máquinas com bastante RAM. Os algoritmos também sobrecarregam os processadores, de modo que uma configuração robusta da CPU é normalmente desejada se respostas oportunas e rápidas forem importantes.

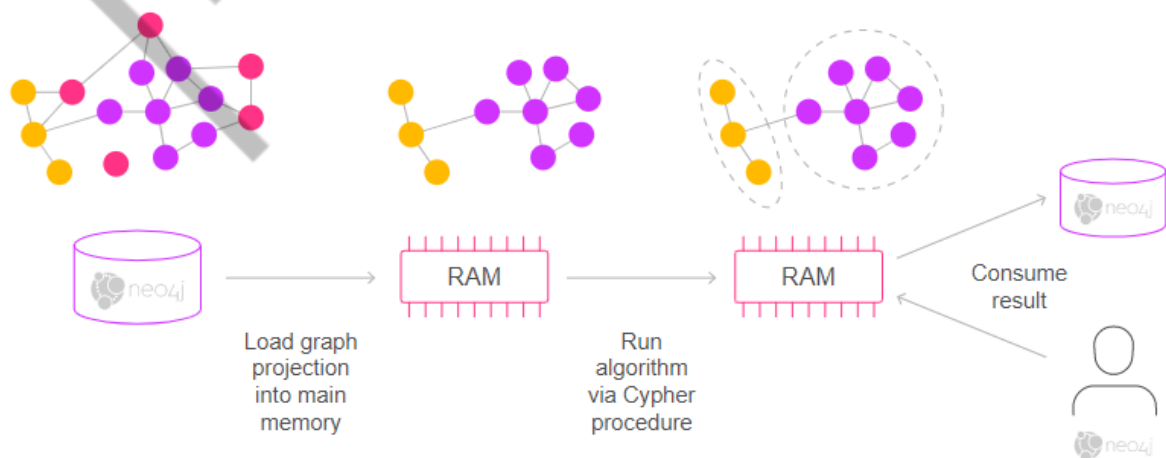


Figura 5.42 – Workflow
Fonte: NEO4J (2020)

Os principais recursos incluem:

- Algoritmos paralelos otimizados que executam dezenas de bilhões de nós e relacionamentos.
- Recursos de produção, como propagação determinística, para consistência para acelerar o teste do modelo.
- Um grafo escalável na memória que é materializado em paralelo e que pode agregar e remodelar de forma flexível o grafo de origem subjacente.
- Um grafo mutável na memória que permite a estratificação das etapas de análise de dados.
- Experiência amigável em ciência de dados com gerenciamento de memória lógica, API intuitiva e extensa documentação e guias.
- Exploração visual de grafos e resultados de algoritmos que podem ser compartilhados entre as equipes de ciência de dados, desenvolvimento e negócios para melhorar a colaboração.

5.3.12.3 GraphQL

Como alternativa ao REST, o GraphQL é uma linguagem de consulta de API desenvolvida pelo Facebook especialmente direcionada a desenvolvedores de aplicativos móveis e de front-end.

O GraphQL obtém um pedaço do grafo, recuperando uma árvore de dados que combina perfeitamente com uma visão de front-end, independentemente de onde esses dados foram extraídos.

Como os serviços GraphQL podem ser implementados em qualquer linguagem, uma Linguagem de Definição de Esquema (SDL), independente da linguagem, é usada para definir os tipos GraphQL.

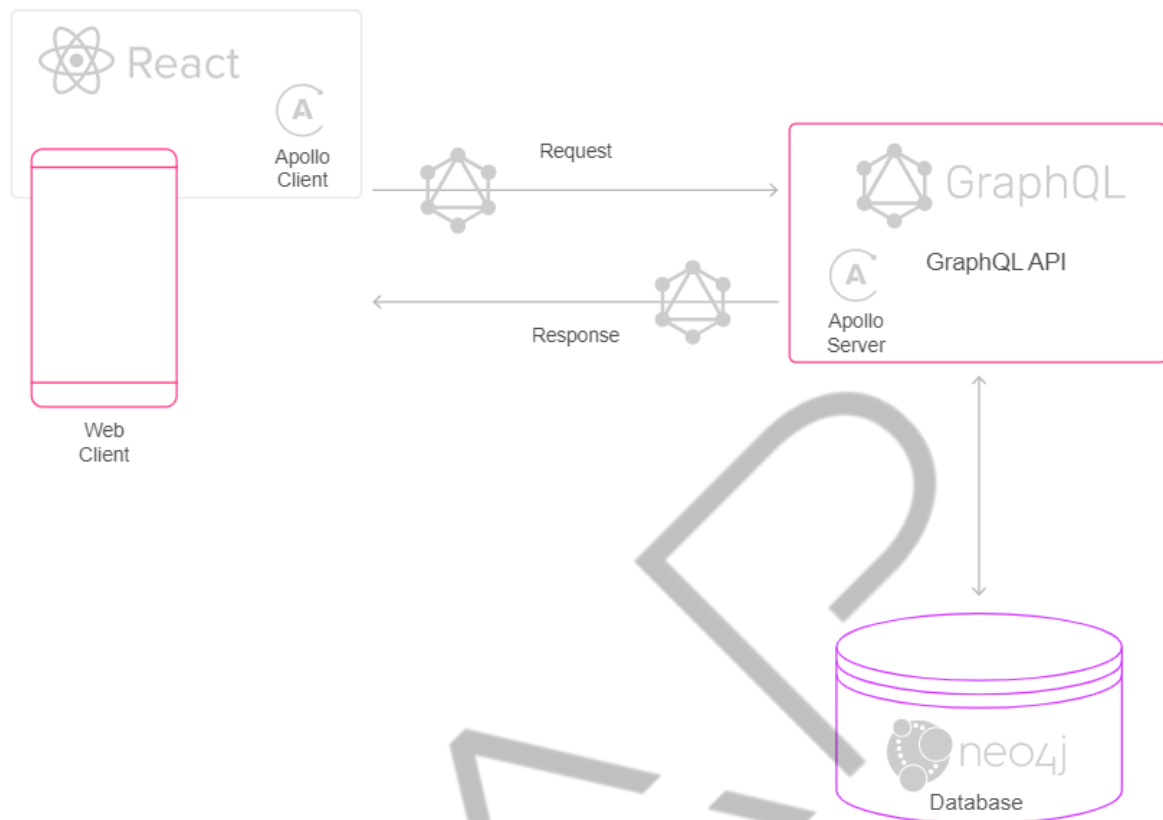


Figura 5.43 – Neo4j GraphQL (1)
Fonte: NEO4J (2020)

O Neo4j-GraphQL é uma extensão do servidor Neo4j que permite transformar consultas do GraphQL em comandos na sintaxe Cypher e executá-las no grafo.

```

{
  moviesByTitle(title: "Matrix Reloaded") {
    title
    similar(limit: 2) {
      title
      year
    }
  }
}
  
```

Código-fonte 5.28 – Consulta exemplo GraphQL
Fonte: NEO4J (2020)

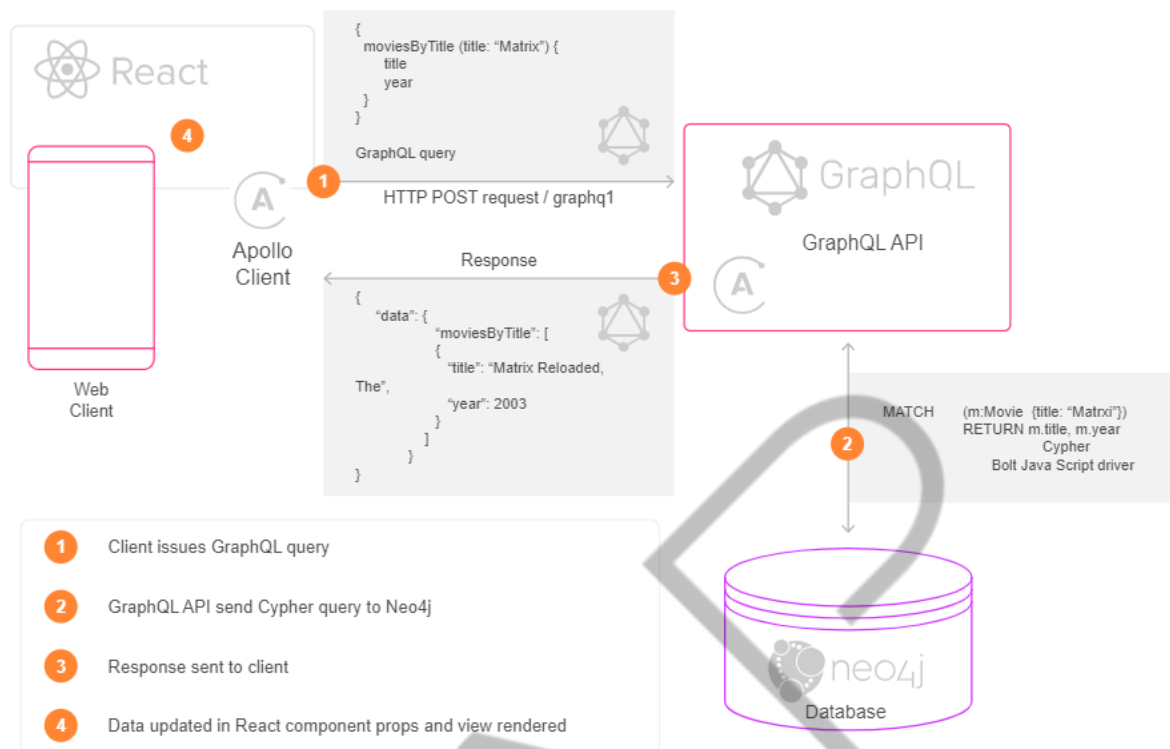


Figura 5.44 – Neo4j GraphQL (2)
Fonte: NEO4J (2020)

```

{
  "data": {
    "moviesByTitle": [
      {
        "title": "Matrix Reloaded, The",
        "similar": [
          {
            "title": "Matrix, The",
            "year": 1999
          },
          {
            "title": "Lord of the Rings: The Fellowship of
the Ring, The",
            "year": 2001
          }
        ]
      }
    ]
  }
}

```

Código-fonte 5.29 – Resultado da consulta
Fonte: NEO4J (2020)

5.3.12.4 Neo4j Streams

O Neo4j Streams Connector para Kafka e Confluent Platform permite tratar informações em tempo real. Os fluxos de eventos são enriquecidos e análises baseadas em grafos são aplicadas.

Kafka é uma plataforma de streaming distribuído capaz de lidar com trilhões de eventos por dia. Confluent Platform é uma plataforma de streaming que permite organizar e gerenciar dados de muitas fontes distintas com um sistema confiável e de alto desempenho. Os usuários do Neo4j podem consumir eventos do Kafka, ingeri-los e transformá-los em grafos.

Além de enviar novos dados de eventos dos canais Kafka para o Neo4j e convertê-los em estruturas de dados gráficos, o Neo4j Streams também ajuda a expor os resultados do processamento no banco de dados de grafos de volta ao Kafka, onde ele pode ser usado para impactar os sistemas downstream.

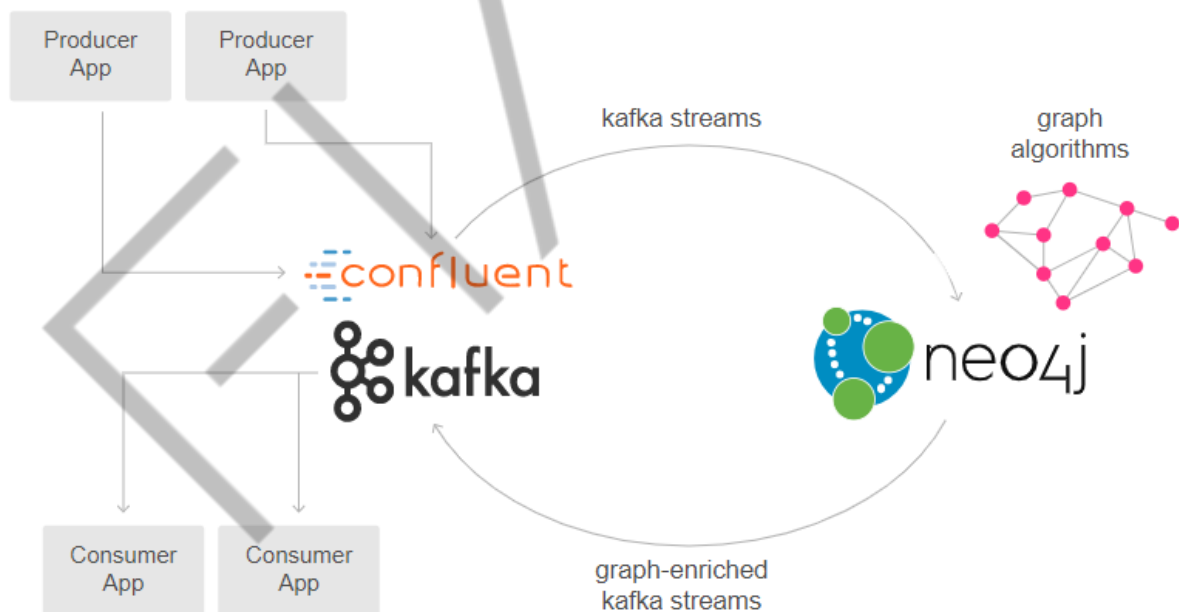


Figura 5.45 – Neo4J Streams
Fonte: NEO4J (2020)

Os clientes podem usar o Neo4j Streams para uma variedade de casos de uso em tempo real, incluindo análise de fraude financeira, grafo do conhecimento e cliente 360.

O Neo4j Streams recebeu a certificação Verified Gold da Confluent, garantindo aos usuários que é totalmente compatível com a API Kafka Connect.

O Neo4j Streams permite gerenciar eventos do Apache Kafka de três maneiras:

- Como uma fonte, em que qualquer alteração no banco de dados será publicada como evento Kafka.
- Como Sink para ingerir qualquer tipo de evento Kafka em seu gráfico.
- Por meio dos procedimentos do Neo4j Streams que permitem produzir/consumir eventos Kafka diretamente do Cypher.

5.3.12.5 Exemplo Aeroportos

Criar um grafo com os aeroportos e suas conexões a partir de um arquivo JSON disponibilizado em <https://r.neo4j.com/airports>. O arquivo é uma lista dos aeroportos com alguns metadados como city, country e localização (*longitude*, *latitude*), bem como designações específicas para o aeroporto como código IATA e uma lista de aeroportos de destino presentes na coluna denominada *destination*.

```
{
  "City": "Postville",
  "DBTZ": "A",
  "Name": "Postville Airport",
  "Country": "Canada",
  "IATA/FAA": "YSO",
  "Longitude": "-59.785278",
  "ICAO": "CCD4",
  "Airport ID": "7252",
  "Latitude": "54.910278",
  "Timezone": "223",
  "DST": "-4",
  "destinations": [
    "5492",
    "188",
    "5502"
  ]
},
```

Figura 5.46 – Subconjunto do arquivo JSON de aeroportos
Fonte: Neo4J (2020)

Podemos, antes de carregar o arquivo JSON, já criarmos indexes para melhorar o desempenho da carga dos nós e relacionamentos. Os comandos devem ser executados individualmente.

```
CREATE CONSTRAINT ON (a:Airport) ASSERT a.id IS UNIQUE;
CREATE INDEX ON :Airport(Name);
CREATE INDEX ON :Airport(iata);
CREATE INDEX ON :Airport(location);
```

Código-fonte 5.30 – Exemplo do uso do comando Create Constraint e Index 1
Fonte: NEO4J (2020)

Agora vamos utilizar algumas procedures do módulo APOC para realizarmos a ingestão dos dados.

```
WITH "https://r.neo4j.com/airports" AS url
CALL apoc.load.jsonArray(url) yield value
MERGE (a:Airport {id:value.`Airport ID`})
SET a += apoc.map.clean(apoc.convert.toMap(value),
    ["Airport
ID","Latitude","Longitude","DST","Timezone",
    "destinations","IATA/FAA"],[",",null]),
    a.iata = value.`IATA/FAA`, a.DST = toInteger(value.DST),
    a.Timezone = toInteger(value.Timezone),
    a.location = point({latitude:toFloat(value.Latitude),
    longitude:
toFloat(value.Longitude)})
WITH *
UNWIND value.destinations as dest
MERGE (b:Airport {id:dest})
MERGE (a)-[:CONNECTS]-(b);
```

Código-fonte 5.31 – Exemplo do uso do comando Create Constraint e Index 2
Fonte: NEO4J (2020)

O resultado serão 3.281 labels adicionados, 3.281 nodes criados, 32.549 propriedades, 19.024 relationships criados.

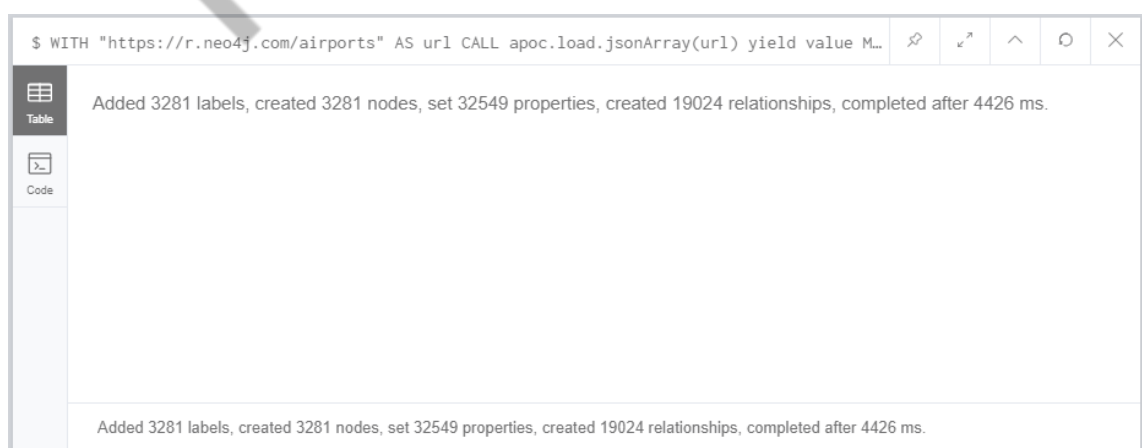


Figura 5.47 – Resultado da ingestão de aeroportos
Fonte: NEO4J (2020)

Visualizar alguns aeroportos e suas conexões aleatoriamente.

```
MATCH p=()-[r:CONNECTS]->()
RETURN p ORDER BY rand() LIMIT 20
```

Código-fonte 5.32 – Exemplo do uso do comando `Match`

Fonte: NEO4J (2020)

Calcular a distância entre os aeroportos pelas coordenadas e armazenar numa propriedade denominada *distance*.

```
MATCH (a:Airport)-[r:CONNECTS]->(b:Airport)
SET r.distance = distance(a.location, b.location);
```

Código-fonte 5.33 – Exemplo do uso do comando para calcular distância

Fonte: NEO4J (2020)

Caminho mais curto entre dois aeroportos.

```
MATCH (a:Airport {iata: 'CHG'}),(b:Airport {iata: 'YYT'})
RETURN shortestPath((a)-[:CONNECTS*]-(b)) as path;
```

Código-fonte 5.34 – Exemplo do uso do comando menor caminho entre aeroportos

Fonte: NEO4J (2020)

Escolha randômica de dois aeroportos e chegar no menor caminho em distância entre eles.

```
MATCH (a:Airport)
WITH apoc.coll.randomItems(collect(a), 2, false) as airports
CALL apoc.algo.dijkstra(airports[0],airports[1],
'CONNECTS','distance')
YIELD path,weight
RETURN path,weight/1000 limit 1;
```

Código-fonte 5.35 – Exemplo do uso do comando para calcular distância

Fonte: NEO4J (2020)

5.3.12.6 Exemplo Graph Data Science Library

Vamos explorar o **Universo Game of Thrones** com os algoritmos Graph Data Science Library. Na interface web do Neo4j , criar uma nova base de dados, inicializá-la e garantir que os plugins estejam instalados. Após isso, clicar no box Graph Data Science Playground.

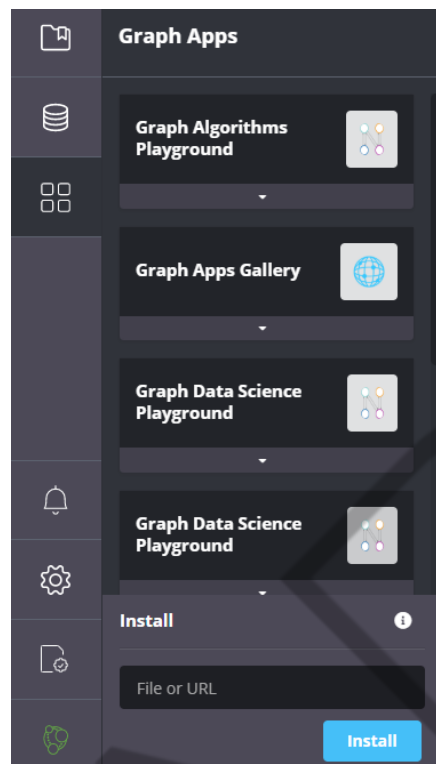


Figura 5.48 – Graph Apps
Fonte: NEO4J (2020)

Um conjunto de algoritmos é apresentado, como vimos no capítulo dos módulos.

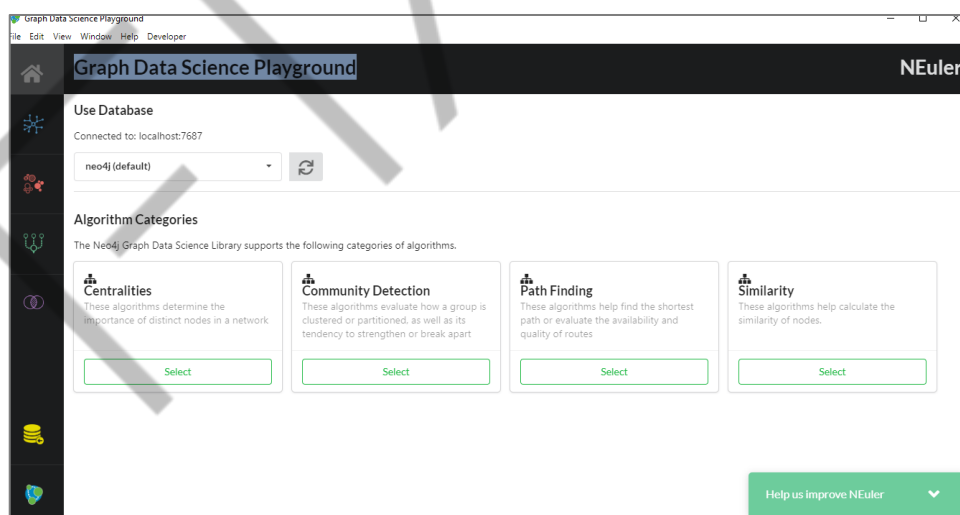


Figura 5.49 – Graph Data Science Playground
Fonte: NEO4J (2020)

No ícone de banco de dados à esquerda, no canto inferior, vamos escolher a base para carregar. Três opções são apresentadas: Game of Thrones, European Roads e Twitter.

Confirmar a carga dos dados para Load Game of Thrones – conjunto de dados contém as interações entre os personagens ao longo das sete primeiras temporadas. Antes disso você deve observar as sintaxes dos comandos!

Game of Thrones

ACHTUNG!
Pressing the 'Yes, load it!' button below will run the following Cypher statements:

```
CREATE CONSTRAINT ON (c:Character) ASSERT c.id IS UNIQUE
```

```
UNWIND range(1,7) AS season
LOAD CSV WITH HEADERS FROM "https://github.com/neo4j-apps/neuler/raw/master/sample-data/got/got-s"
+ season + "-nodes.csv" AS row
MERGE (c:Character {id: row.Id})
ON CREATE SET c.name = row.Label
```

```
UNWIND range(1,7) AS season
LOAD CSV WITH HEADERS FROM "https://github.com/neo4j-apps/neuler/raw/master/sample-data/got/got-s"
+ season + "-edges.csv" AS row
MATCH (source:Character {id: row.Source})
MATCH (target:Character {id: row.Target})
CALL apoc.merge.relationship(source, "INTERACTS_SEASON" + season, {}, {}, target) YIELD rel
SET rel.weight = toInteger(row.Weight)
```

Yes, load it! No, get me outta here!

Figura 5.50 – Carga Game of Thrones
Fonte: NEO4J (2020)

Ao término da carga, podemos explorar os dados e compreender a importância da visualização para as análises de dados. Em Degree podemos observar qual personagem teve o maior número de interações com outros personagens. Pode-se fazer por temporada assinalada na Relationship Type.

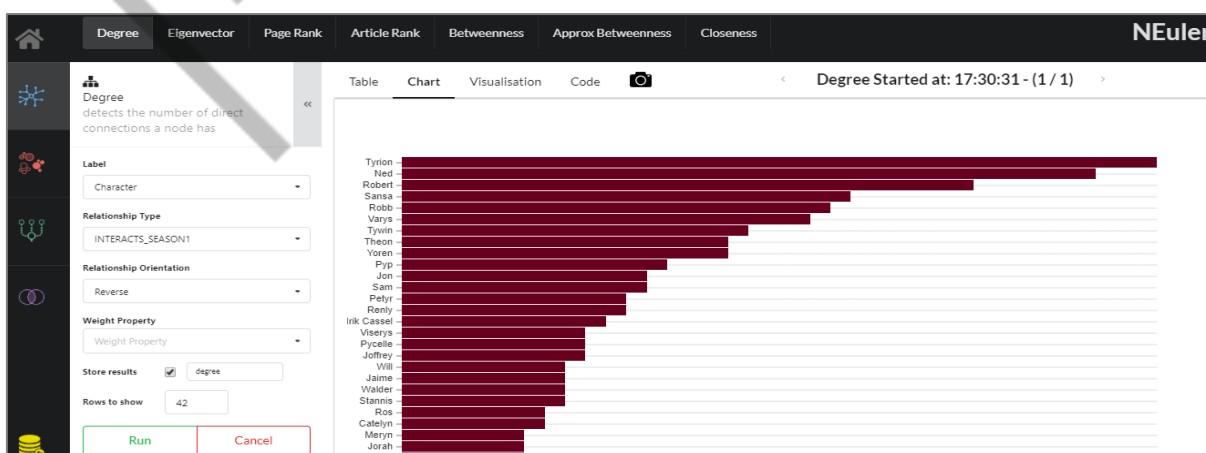


Figura 5.51 – Resultado gráfico da interação dos personagens
Fonte: NEO4J (2020)

Indicar a INTERACTS_SEASON1 para visualizar que na 1ª temporada – Tyrion, Ned e Robert tiveram mais interações. Isso não significa necessariamente que esses são os personagens mais influentes, mas certamente são os que estão falando muito.

Podemos também combinar detecção de comunidade PageRank e centralidade na comunidade com Louvain. O algoritmo de Louvain Modularity detecta comunidades em redes, com base em uma heurística que maximiza os escores de modularidade.

As pontuações de modularidade variam de -1 e 1 como uma medida da densidade de relacionamento dentro das comunidades em relação à densidade de relacionamento de comunidades externas. Se o executarmos na segunda temporada do conjunto de dados de Game of Thrones e visualizarmos o formato de saída, veremos o gráfico abaixo.

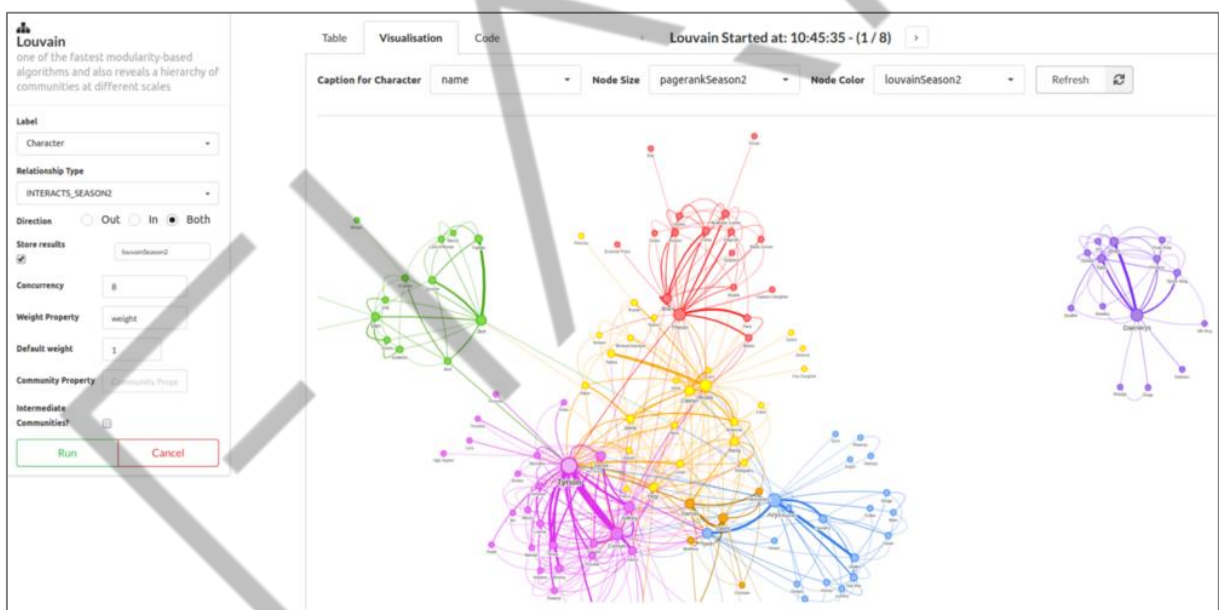


Figura 5.52 – Resultado gráfico do algoritmo Louvain
Fonte: NEO4J (2020)

No cluster roxo superior direito, podemos ver que o grupo Daenerys está desligado por conta própria, desconectado de todos os outros. As pessoas naquele cluster não interagem com mais ninguém. Inicialmente pensamos que deveria haver um problema com os dados ou algoritmo e executamos outro algoritmo de detecção da comunidade, o Connected Components, para confirmar nossas descobertas.

O algoritmo Connected Components é um algoritmo para detecção de comunidade que detecta grupos de usuários com base na existência de algum caminho entre eles. Se executarmos esse algoritmo, veremos a seguinte visualização:

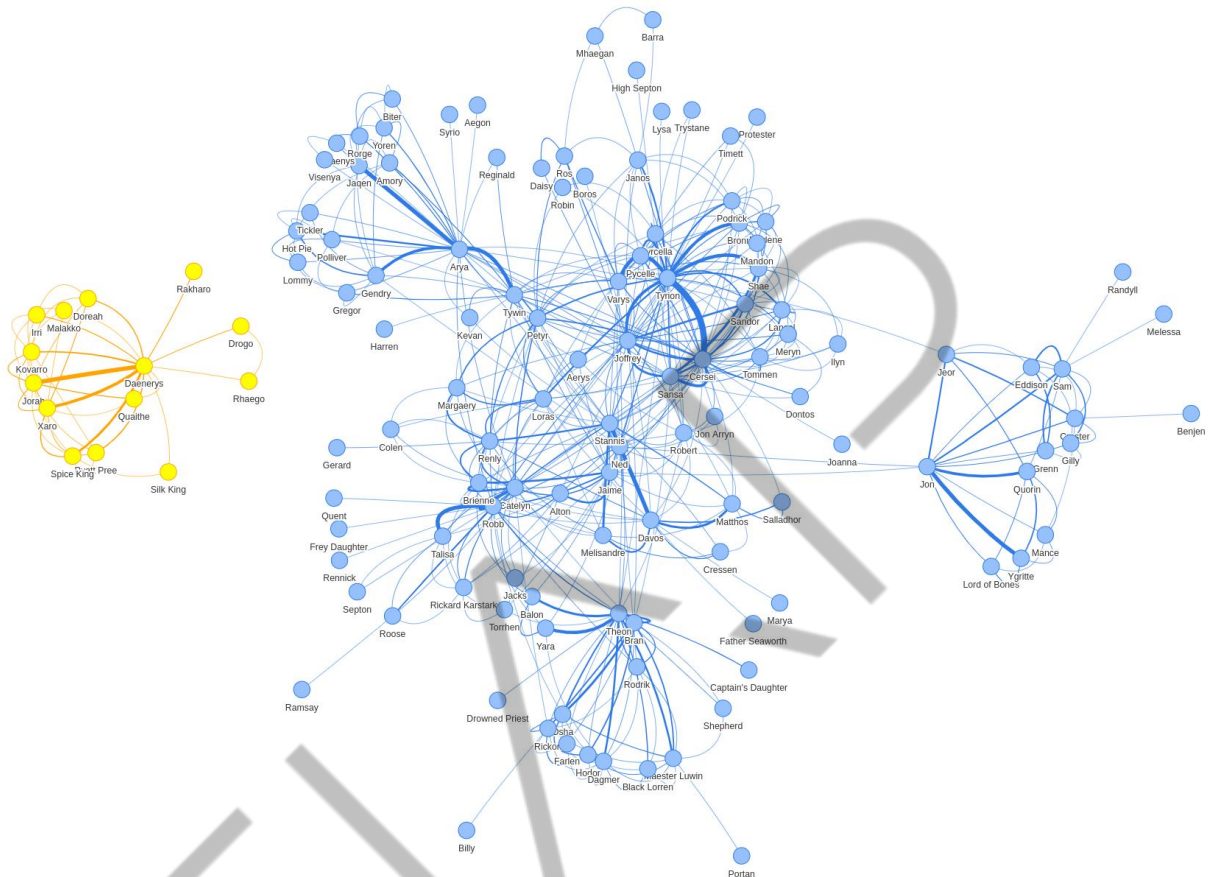


Figura 5.53 – Connected Components
Fonte: NEO4J (2020)

Aqui, temos apenas duas comunidades: a da Daenerys à direita e a grande maioria de outros personagens à esquerda. Isso confirma nossas descobertas do algoritmo Louvain Modularity, e se estendermos nossa memória de volta à segunda temporada, lembraremos que Daenerys estava em uma ilha longe do restante dos personagens principais. Espero que tenham gostado desta rápida passagem pelos algoritmos analíticos! Que tal executá-los na base de Movies?

REFERÊNCIAS

AWS. **AWS Announces General Availability of Amazon Neptune**. 2018. Disponível em: <https://press.aboutamazon.com/news-releases/news-release-details/aws-announces-general-availability-amazon-neptune/> Acesso em: 04 out 2020.

INTRODUCING NEO4J for Graph Data Science, the First Enterprise Graph Framework for Data Scientists. **Cision**, 2020. Disponível em: <https://www.prnewswire.com/news-releases/introducing-neo4j-for-graph-data-science-the-first-enterprise-graph-framework-for-data-scientists-301037233.html>. Acesso em: 28 set. 2020.

HELLER, M. **Amazon Neptune review: A scalable graph database for OLTP**. 2019. Disponível em: https://www.infoworld.com/article/3394860/amazon-neptune-review-a-scalable-graph-database-for-oltp.html#tk.rss_all Acesso em: 25 ago. 2020.

LYON, W. **An Overview of GraphQL**. 2017. Disponível em: <https://dzone.com/refcardz/an-overview-of-graphql?chapter=4/>. Acesso em: 28 set. 2020.

LYON, W. **Fullstack GraphQL Applications with GRANDstack!** 2020. Disponível em: <https://livebook.manning.com/book/fullstack-graphql-applications-with-grandstack/welcome/v-3/5>. Acesso em: 18 jul. 2020.

NEO4J. **Neo4j APOC Library**. 2020. Disponível em: <https://neo4j.com/developer/neo4j-apoc/>. Acesso em: 17 jul. 2020.

NEO4J. **Cypher Basics I**. 2020. Disponível em: <https://neo4j.com/developer/cypher/guide-cypher-basics/>. Acesso em: 17 jul. 2020.

NEO4J. **Neo4j Graph Data Science Library**. 2020. Disponível em: <https://neo4j.com/graph-data-science-library/>. Acesso em: 17 jul. 2020.