Fundamentos do Neo4j e IA Generativa

GraphAcademy - Jan. 2026

Índice

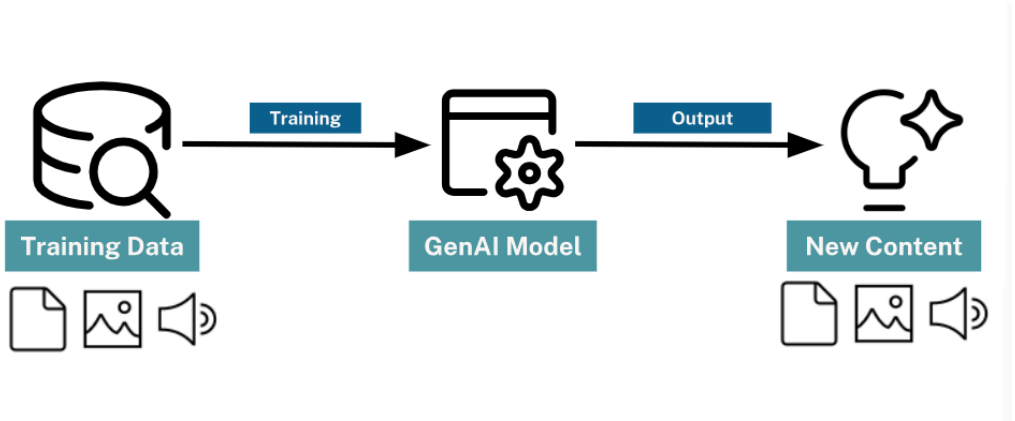
1. IA generativa....3
GenAI....3
Modelos de Linguagem de Grande Porte (LLMs)....3
Instruções....4
Considerações....5
2. Retrieval Augmented Generation (RAG)....7
O que é RAG?....7
Vector RAG....9
Vector Index....13
GraphRAG....17
3. Grafos de conhecimento....21
O que é um grafo de conhecimento?....21
Criando grafos de conhecimento....24
4. Integrando o Neo4j com a IA Generativa....26
GraphRAG para Python....26

1. IA generativa

GenAI

A Inteligência Artificial Generativa (ou GenAI) refere-se a sistemas de inteligência artificial projetados para criar novos conteúdos que se assemelham a dados produzidos por humanos. Os dados podem ser texto, imagens, áudio ou código.

Esses modelos, como o GPT (para texto) ou o DALL-E (para imagens), são treinados em grandes conjuntos de dados e usam padrões aprendidos a partir desses dados para gerar novas saídas.



A IA generativa é amplamente utilizada em aplicações como chatbots, criação de conteúdo, síntese de imagens e geração de código.

Os modelos de IA generativa não são "inteligentes" da mesma forma que os humanos:

1. Eles não entendem nem compreendem o conteúdo que geram.
2. Eles se baseiam em padrões estatísticos e correlações aprendidas a partir de seus dados de treinamento.

Embora os modelos de IA generativa possam produzir resultados coerentes e contextualmente relevantes, eles carecem de compreensão.

Modelos de Linguagem de Grande Porte (LLMs)

Os LLMs são um tipo de modelo de IA generativa projetado para entender e gerar textos semelhantes aos humanos.
Esses modelos são treinados com grandes quantidades de dados textuais e podem executar diversas tarefas, incluindo responder a perguntas, resumir dados e analisar textos.

A resposta gerada por um LLM é uma continuação probabilística das instruções que recebe.
O LLM fornece a resposta mais provável com base nos padrões que aprendeu a partir de seus dados de treinamento.

Caso lhe seja apresentada a instrução:

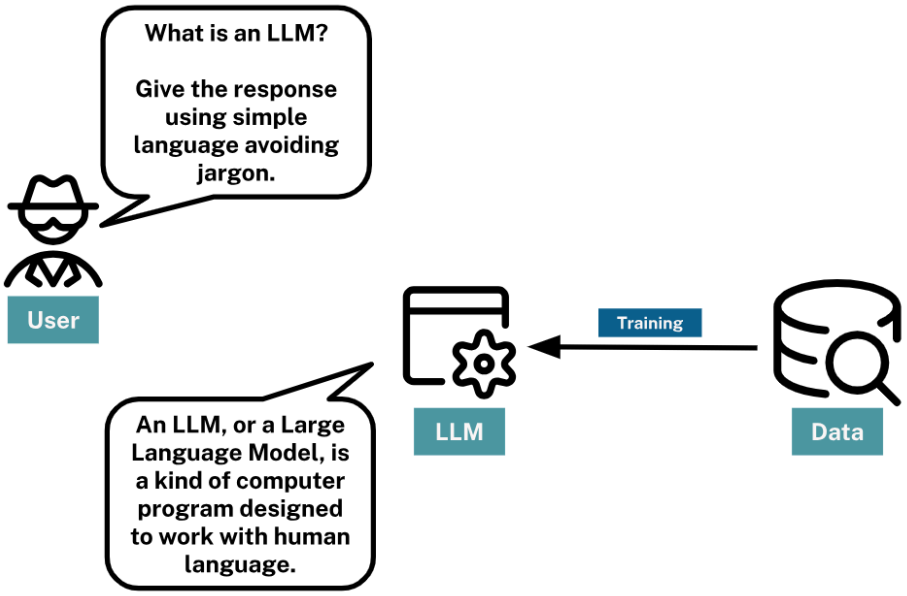
```
"Continue esta sequência - AB C"
```

Uma LLM poderia responder:

```
"DE F"
```

Instruções

Para que um LLM execute uma tarefa, você fornece uma instrução. O enunciado deve especificar suas necessidades e fornecer instruções claras sobre como responder.



A precisão na descrição da tarefa, potencialmente combinada com exemplos ou contexto, garante que o modelo compreenda a intenção e produza resultados relevantes e precisos.

Um exemplo de pergunta pode ser uma questão simples.

```
Qual é a capital do Japão?
```

Ou, poderia ser mais descritivo:

```
Você é um agente de viagens amigável que ajuda um cliente a escolher um pacote de viagem, destino de férias. Seus leitores podem ter o inglês como segunda língua. Portanto, use uma linguagem simples e evite expressões coloquiais. Evite jargões a todo custo.

Fale-me sobre a capital do Japão.
```

O LLM interpretará essas instruções e retornará uma resposta com base nos padrões que aprendeu a partir de seus dados de treinamento.

Considerações

Cuidado

Embora a IA GenAI e os LLMs ofereçam muito potencial, você também deve ser cauteloso.

Em sua essência, os LLMs são máquinas de texto preditivo altamente complexas. Os LLMs não conhecem nem compreendem as informações que produzem; eles simplesmente preveem a próxima palavra em uma sequência.

As palavras são baseadas nos padrões e relações de outros textos presentes nos dados de treinamento.

Acesso a dados

As fontes para esses dados de treinamento são frequentemente a internet, livros e outros textos disponíveis publicamente. Os dados podem ser de qualidade questionável ou até mesmo incorretos.

O treinamento ocorre em um ponto específico no tempo, os dados são estáticos e podem não refletir o estado atual do mundo ou incluir informações privadas.

Ao ser solicitado a fornecer uma resposta relacionada a dados novos ou não presentes no conjunto de treinamento, o LLM pode fornecer uma resposta imprecisa.

Precisão

Os LLMs são projetados para criar textos semelhantes aos humanos e são frequentemente ajustados para serem o mais úteis possível, mesmo que isso signifique ocasionalmente gerar conteúdo enganoso ou sem fundamento, um fenômeno conhecido como **alucinação** .

Por exemplo, ao ser solicitado a *"Descrever a lua"*, um mestre em direito pode responder: *"A lua é feita de queijo"*. Embora seja um ditado comum, não é verdade.

Embora os LLMs possam representar a essência das palavras e frases, eles não possuem uma compreensão genuína ou um julgamento ético do conteúdo.

Temperatura

Os LLMs possuem uma temperatura , que corresponde à quantidade de aleatoriedade que o modelo subjacente deve usar ao gerar o texto.

Quanto maior o valor da temperatura, mais aleatório será o resultado gerado e maior a probabilidade de a resposta conter afirmações falsas.

Uma temperatura mais alta pode ser apropriada ao configurar um LLM para responder com resultados mais diversos e criativos, enquanto uma temperatura mais baixa é necessária quando as respostas devem ser consistentes e precisas.

A influência da temperatura nos resultados depende do modelo, devido a diferenças no projeto e nos dados de treinamento.

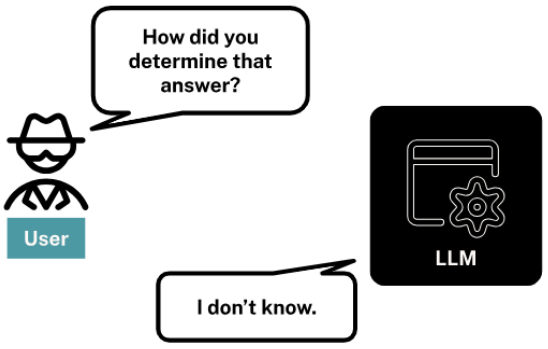
Considere a temperatura correta

Em junho de 2023, um juiz dos EUA sancionou dois advogados americanos por apresentarem um documento jurídico elaborado durante seu mestrado em Direito (LLM) que continha seis citações de casos fictícios.

Transparência

Os modelos GenAI são frequentemente considerados "caixas-pretas" devido à dificuldade em decifrar seus processos de tomada de decisão.

O mestrado em Direito (LLM) também não consegue fornecer as fontes de sua produção nem explicar seu raciocínio.



2. Retrieval Augmented Generation (RAG)

O que é RAG?

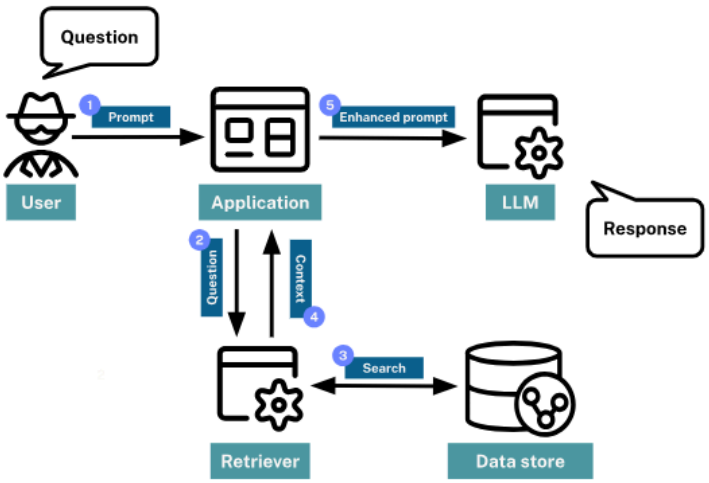
A Geração Aumentada por Recuperação (RAG, na sigla em inglês) é uma abordagem que aprimora as respostas dos LLMs, fornecendo-lhes informações relevantes e atualizadas obtidas de fontes externas.

O RAG ajuda a gerar respostas mais precisas e personalizadas, especialmente quando as informações necessárias não estão presentes nos dados de treinamento do modelo.

O Processo de Geração Aumentada por Recuperação (RAG)

O processo RAG normalmente envolve três etapas principais:

1. Entendendo a consulta do usuário
   O sistema primeiro interpreta a entrada ou pergunta do usuário para determinar quais informações são necessárias.
2. Recuperação de Informação
   Um *mecanismo de recuperação* pesquisa fontes de dados externas (como documentos, bancos de dados ou grafos de conhecimento) para encontrar informações relevantes com base na consulta do usuário.
3. Geração de Respostas
   A informação recuperada é inserida no prompt, e o modelo de linguagem usa esse contexto para gerar uma resposta mais precisa e relevante.



Os sistemas RAG podem fornecer respostas que sejam contextualizadas e baseadas em informações reais e atualizadas.

Se você estiver desenvolvendo um chatbot para uma agência de notícias, pode usar o RAG para obter manchetes ou resultados em tempo real de uma API de notícias.

Quando um usuário pergunta: "Quais são as últimas notícias sobre as Olimpíadas?", o chatbot pode fornecer uma manchete ou um resumo atualizado dos artigos mais recentes, garantindo que a resposta seja oportuna e precisa.

Grounding

O processo de fornecer contexto a um LLM para melhorar a precisão de suas respostas e reduzir a probabilidade de alucinações é conhecido como Grounding.

Retrievers

O mecanismo de recuperação é um componente fundamental do processo RAG. Ele é responsável por buscar e recuperar informações relevantes de fontes de dados externas com base na consulta do usuário.

Um mecanismo de recuperação normalmente recebe uma entrada não estruturada (como uma pergunta ou um comando) e busca dados estruturados que possam fornecer contexto ou respostas.

O Neo4j suporta vários métodos para construir servidores de recuperação de dados, incluindo:

- Pesquisa de texto completo
- Busca vetorial
- Texto para Criptografia

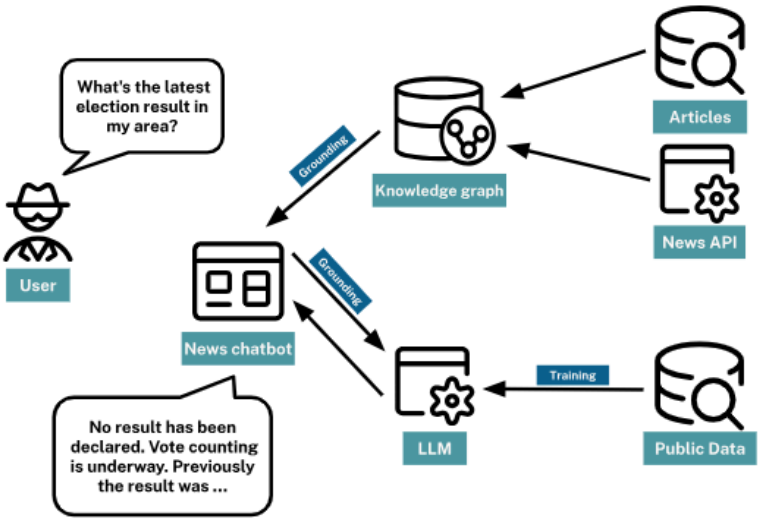
Fontes de dados

As fontes de dados utilizadas no processo RAG podem variar bastante, dependendo da aplicação e do tipo de informação necessária. As fontes de dados comuns incluem:

- Documentos: fontes de dados textuais, como artigos, relatórios ou manuais, que podem ser pesquisadas para encontrar informações relevantes.
- APIs: serviços externos que podem fornecer dados em tempo real ou informações específicas com base em consultas do usuário.
- Grafos de conhecimento: representações gráficas de informações que podem fornecer contexto e relações entre entidades

O chatbot da agência de notícias poderia usar as seguintes fontes de dados:

- Uma API de notícias para recuperar os artigos ou manchetes mais recentes.
- Um grafo de conhecimento para entender as relações entre diferentes tópicos de notícias, como a relação entre eles ou seu contexto histórico. Isso ajudaria o chatbot a fornecer respostas mais detalhadas e contextuais.
- Um banco de dados de documentos para armazenar e recuperar artigos, relatórios ou outros dados textuais que podem ser usados para responder a consultas do usuário.



Vector RAG

Um dos desafios do RAG é entender o que o usuário está solicitando e encontrar as informações corretas para passar para o LLM.

Nesta lição, você aprenderá sobre **busca semântica** e como os índices vetoriais podem ajudá-lo a encontrar informações relevantes a partir da pergunta de um usuário.

Busca Semântica

A busca semântica visa compreender a intenção e o **significado contextual** das frases de busca, em vez de se concentrar em palavras-chave individuais.

A busca tradicional por palavras-chave geralmente depende de palavras-chave de correspondência exata ou de algoritmos baseados em proximidade que encontram palavras semelhantes.

Por exemplo, se você digitar "maçã" em uma busca tradicional, poderá obter predominantemente resultados relacionados à fruta.

No entanto, em uma busca semântica, o mecanismo tenta avaliar o contexto: você está pesquisando sobre a fruta, a empresa de tecnologia ou algo mais?

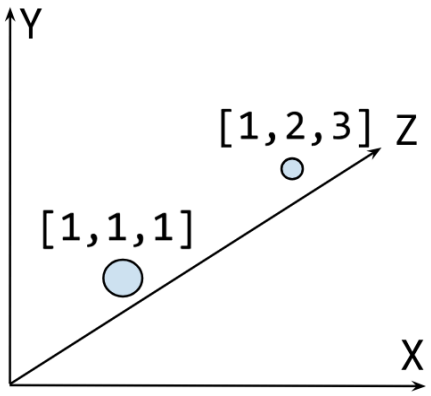
Os resultados são personalizados com base no termo e na intenção percebida.

Vetores

Você pode representar dados como **vetores** para realizar buscas semânticas. Vetores são simplesmente uma **lista de números**. Por exemplo, um vetor [1, 2, 3] é uma lista de três números e pode representar um ponto no espaço tridimensional.

Você pode usar vetores para representar muitos tipos diferentes de dados, incluindo texto, imagens e áudio.

O número de dimensões em um vetor é chamado de **dimensionalidade** do vetor. Um vetor com três números tem dimensionalidade 3. Dimensionalidades mais altas capturam significados mais sutis, mas são computacionalmente mais custosas e, similarmente, dimensionalidades mais baixas são mais rápidas e baratas de calcular, mas oferecem menos nuances.



Embeddings

Ao se referir a vetores no contexto de aprendizado de máquina e NLP (Processamento de Linguagem Natural), o termo "**embedding**" é geralmente usado. Embeddings são traduções numéricas de objetos de dados, como imagens, texto ou áudio, representados como vetores. Dessa forma, os algoritmos de LLM conseguem comparar dois parágrafos de texto diferentes comparando suas representações numéricas.

Cada dimensão em um vetor pode representar um aspecto semântico específico da palavra ou frase. Quando múltiplas dimensões são combinadas, elas podem transmitir o significado geral da palavra ou frase.

Por exemplo, a palavra "maçã" poderia ser representada por um embedding com as seguintes dimensões:

- fruta
- tecnologia
- cor
- gosto
- forma

Quando aplicado em um contexto de busca, o vetor para "maçã" pode ser comparado aos vetores de outras palavras ou frases para determinar os resultados mais relevantes.

Você pode criar embeddings de várias maneiras, mas um dos métodos mais comuns é usar um modelo de embedding .

Por exemplo, a representação gráfica da palavra "maçã" é

0.0077788467, -0.02306925, -0.007360777, -0.027743412, -0.0045747845, 0.01289164, -0.021863015, -0.008587573, 0.01892967, -0.029854324, -0.0027962727, 0.020108491, -0.004530236, 0.009129008,... e assim por diante.

Embora seja possível criar representações vetoriais para palavras individuais, é mais comum incorporar frases ou parágrafos inteiros, já que o significado de uma palavra pode mudar dependendo do contexto.

Por exemplo, a palavra *"bank"* terá um vetor diferente em *"river bank"* do que em *"saving bank"* .

Os sistemas de busca semântica podem usar essas representações contextuais para entender a intenção do usuário.

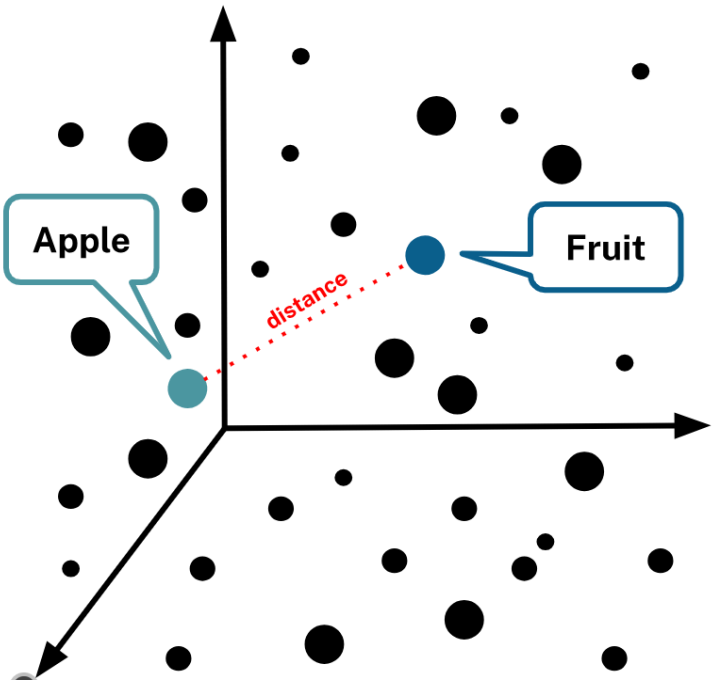
Os embeddings podem representar mais do que apenas texto. Eles também podem representar documentos inteiros, imagens, áudio ou outros tipos de dados.

Como os vetores são usados na busca semântica?

Você pode usar a *distância* ou *o ângulo* entre vetores para avaliar a similaridade semântica entre palavras ou frases.

Palavras com significados ou contextos semelhantes terão vetores próximos uns dos outros, enquanto palavras não relacionadas estarão mais distantes.

Um gráfico tridimensional ilustrando a distância entre vetores. Os vetores representam as palavras "maçã" e "fruta".

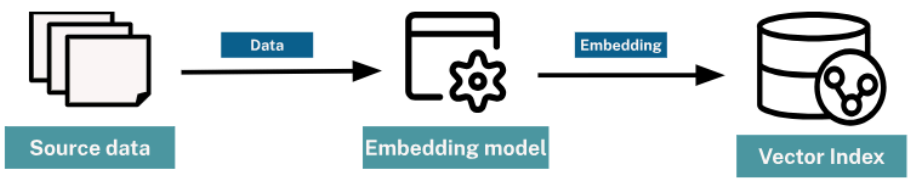


RAG

A busca semântica é empregada em algoritmos RAG baseados em vetores para encontrar resultados contextualmente relevantes para a pergunta de um usuário.

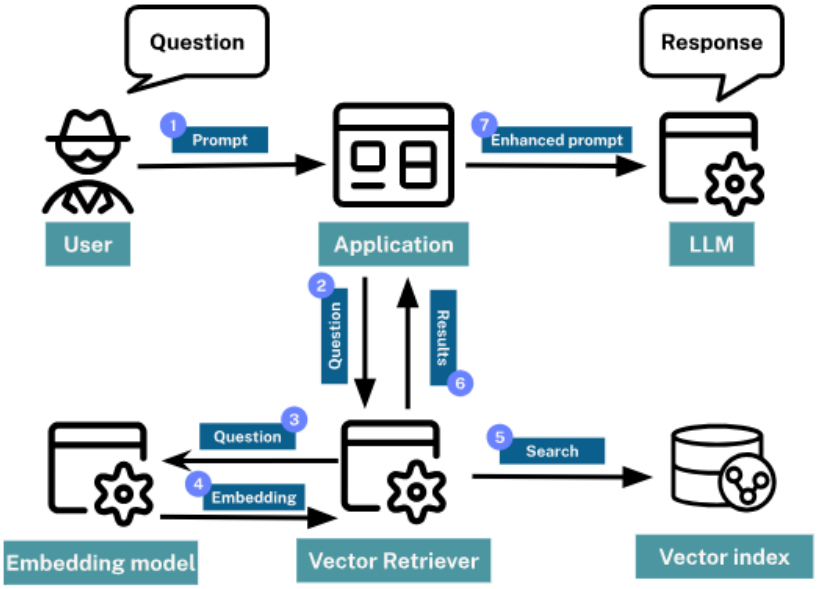
Um modelo de incorporação é usado para criar uma representação vetorial dos dados de origem.

Um diagrama mostrando os dados sendo processados por um modelo de incorporação para criar uma representação vetorial dos dados. Os dados são então armazenados em um índice vetorial.



Quando um usuário envia uma pergunta, o sistema:

1. Cria um embedding da pergunta.
2. Compara o vetor da pergunta com os vetores dos dados indexados.
3. Os resultados são pontuados com base em sua similaridade.
4. Os resultados mais relevantes são usados como contexto para o LLM.



Observação: O Retriever recebe o texto primeiro porque ele é o responsável por gerenciar todo o processo de busca, o que inclui pedir a tradução para o Embedding Model antes de consultar o banco.

Vector Index

Nesta lição, você aprenderá como usar um vector index no Neo4j para comparar embeddings e encontrar dados semelhantes.

Enredos de filmes (movie plots)

Ao se inscrever neste curso, a GraphAcademy criou um ambiente de testes Neo4j com recomendações de filmes. O banco de dados de recomendações contém mais de 9.000 filmes, 15.000 atores e mais de 100.000 avaliações de usuários.

Cada filme tem uma `.plot` propriedade.

```
MATCH (m:Movie {title: "Toy Story"})

RETURN m.title AS title, m.plot AS plot
```

Embedding dos enredos

Foram criados embeddings para 1000 enredos de filmes. O embedding é armazenado na .plotEmbedding propriedade dos Movie nós.

```
MATCH (m:Movie {title: "Toy Story"})
RETURN m.title AS title, m.plot AS plot, m.plotEmbedding
```

A seguinte consulta Cypher retornará os títulos e sinopses dos filmes que possuem conteúdo incorporado:

```
MATCH (m:Movie)
WHERE m.plotEmbedding IS NOT NULL
RETURN m.title, m.plot
```

Um índice vetorial, moviePlots, foi criado para a .plotEmbedding propriedade dos Movie nós.

Você pode usar o moviePlots índice vetorial para encontrar os filmes mais semelhantes comparando os elementos vetoriais (embeddings) dos enredos dos filmes.

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/rec-embed/movie-plot-embeddings-1k.csv'
AS row
MATCH (m:Movie {movieId: row.movieId})
CALL db.create.setNodeVectorProperty(m, 'plotEmbedding',
apoc.convert.fromJsonList(row.embedding));

CREATE VECTOR INDEX moviePlots IF NOT EXISTS
FOR (m:Movie)
ON m.plotEmbedding
OPTIONS {indexConfig: {
 `vector.dimensions`: 1536,
 `vector.similarity_function`: 'cosine'
}};
```

Você pode aprender mais sobre como criar embeddings e índices vetoriais no curso Introdução a Índices Vetoriais e Dados Não Estruturados da GraphAcademy .

Consultando índices vetoriais

Você pode consultar o moviePlots índice usando o db.index.vector.queryNodes() procedimento.

O procedimento retorna o número solicitado de nós vizinhos mais próximos aproximados e sua pontuação de similaridade, ordenados pela pontuação.

```
CALL db.index.vector.queryNodes(
   indexName :: STRING,
   numberOfNearestNeighbours :: INTEGER,
   query :: LIST<FLOAT>
) YIELD node, score
```

O procedimento aceita três parâmetros:

1. `indexName`- O nome do índice do vetor
2. `numberOfNearestNeighbours`- O número de resultados a serem retornados
3. `query`- Uma lista de floats que representam uma incorporação

O procedimento gera dois argumentos:

1. Uma `node` opção que corresponde à consulta.
2. Uma semelhança `score` que varia de `0.0` a `1.0`.

Você pode usar esse procedimento para encontrar o valor de incorporação mais próximo de um determinado valor.

Consultando enredos de filmes semelhantes

Você pode usar o `moviePlots` índice vetorial para encontrar filmes com enredos semelhantes.

Analise este código Cypher antes de executá-lo.

```
MATCH (m:Movie {title: 'Toy Story'})

CALL db.index.vector.queryNodes('moviePlots', 6, m.plotEmbedding)
YIELD node, score

RETURN node.title AS title, node.plot AS plot, score
```

A consulta localiza o nó *"Toy Story"* e utiliza a propriedade para encontrar os enredos mais semelhantes. `Movie.plot Embedding`

O `db.index.vector.queryNodes()` procedimento utiliza o `moviePlot` síndice vetorial para encontrar incorporações semelhantes.

Execute a consulta. O procedimento retorna o número solicitado de nós e sua pontuação de similaridade, ordenados pela pontuação.

Gerar Embeddings

Você pode gerar um novo embedding em Cypher usando a genai.vector.encode função:

```
WITH genai.vector.encode(
   "Text to create embeddings for",
   "OpenAI",
   { token: "sk-..." }) AS embedding
RETURN embedding
```

É necessário possuir uma chave de API.

Você precisará substituir `token: "sk-..."` por uma chave de API da OpenAI .

Gerar um embedding de enredo

Você pode usar o embedding para consultar o índice vetorial e encontrar filmes semelhantes.

Esta consulta cria um vetor de incorporação para o texto *"Uma misteriosa nave espacial aterrissa na Terra"* e o utiliza para consultar o `moviePlots` índice vetorial dos 6 enredos de filmes mais semelhantes.

```
WITH genai.vector.encode(
   "A mysterious spaceship lands Earth",
   "OpenAI",
   { token: "sk-..." }) AS myMoviePlot
CALL db.index.vector.queryNodes('moviePlots', 6, myMoviePlot)
YIELD node, score
RETURN node.title, node.plot, score
```

Considerações

Utilizar embeddings e vetores é relativamente simples e pode gerar resultados rapidamente. A desvantagem dessa abordagem é que ela depende muito dos embeddings e da função de similaridade para produzir resultados válidos.

Essa abordagem também é uma caixa preta. Existem 1536 dimensões; seria impossível determinar como os vetores estão estruturados e como eles influenciam a pontuação de similaridade.

Os filmes retornados parecem semelhantes, mas sem lê-los e compará-los, você não teria como verificar se os resultados estão corretos.

Os vetores funcionam bem para:

- Perguntas contextuais ou baseadas no significado
- Consultas vagas ou imprecisas
- Perguntas amplas ou abertas
- Consultas complexas com múltiplos conceitos

Os vetores são ineficazes para:

- Perguntas altamente específicas ou baseadas em fatos
- Consultas numéricas ou de correspondência exata
- Consultas booleanas ou lógicas
- Consultas ambíguas ou pouco claras sem contexto
- Conhecimento especializado

Na próxima lição, você verá como pode melhorar os resultados usando uma combinação de consultas vetoriais e gráficas.

GraphRAG

GraphRAG (Graph Retrieval Augmented Generation) é uma abordagem que utiliza os pontos fortes dos bancos de dados de grafos para fornecer contexto relevante e útil aos LLMs.

GraphRAG pode ser usado em conjunto com RAG vetorial.

Enquanto o RAG vetorial usa embeddings para encontrar informações contextualmente relevantes, o GraphRAG aprimora esse processo aproveitando os relacionamentos e a estrutura dentro de um grafo.

Benefícios do GraphRAG:

- **Contexto mais rico:** os grafos capturam as relações entre entidades, permitindo a recuperação de informações mais relevantes e conectadas.
- **Precisão aprimorada**: ao combinar a similaridade vetorial com a busca em grafos, os resultados são mais precisos e levam em consideração o contexto.
- **Explicabilidade**: os grafos fornecem caminhos e conexões claros, facilitando a compreensão de por que determinados resultados foram obtidos.
- **Consultas flexíveis**: O GraphRAG suporta consultas complexas, como a combinação de buscas de texto completo, vetor e texto para criptografia.
- **Raciocínio aprimorado**: Os grafos permitem o raciocínio sobre dados, suportando casos de uso avançados, como recomendações e descoberta de conhecimento.

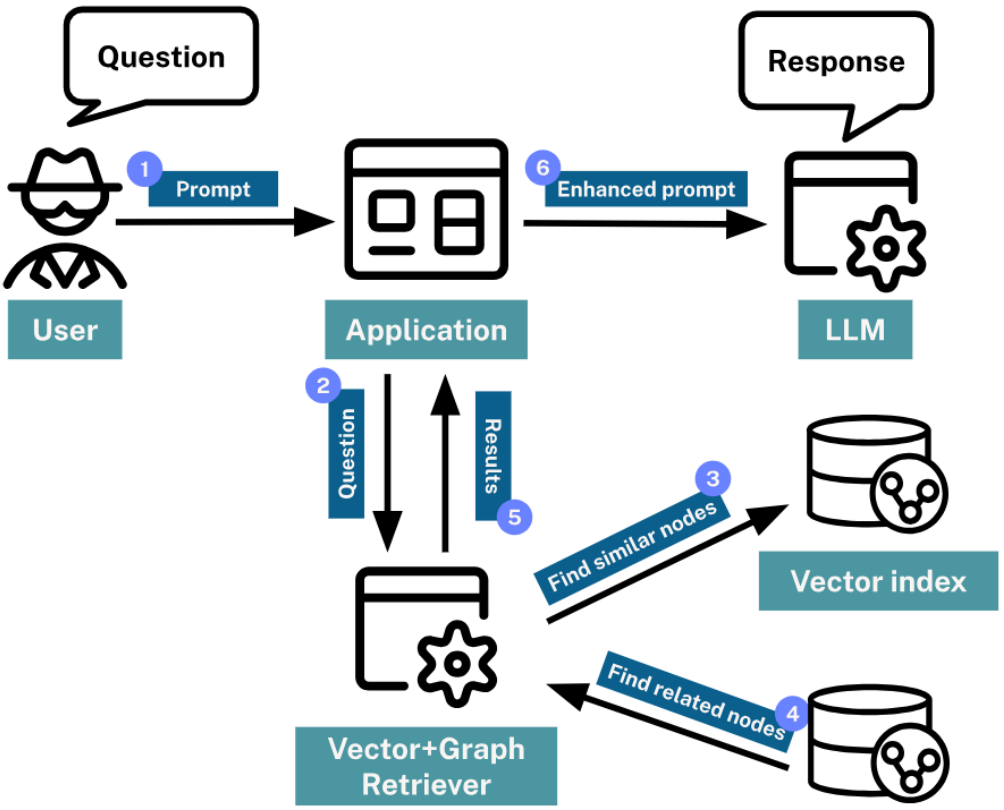
Busca vetorial aprimorada por grafos

Uma abordagem comum para o GraphRAG é usar uma combinação de busca vetorial e percurso em grafos.

Isso permite a recuperação de documentos relevantes com base na similaridade semântica, seguida por uma busca em grafo para encontrar entidades ou conceitos relacionados.

O processo de alto nível é o seguinte:

1. Um usuário envia uma consulta.
2. O sistema utiliza uma busca vetorial para encontrar nós semelhantes à consulta do usuário.
3. Em seguida, o grafo é percorrido para encontrar nós ou entidades relacionadas.
4. As entidades e os relacionamentos são adicionados ao contexto do LLM.
5. Os dados relacionados também podem ser pontuados com base em sua relevância para a consulta do usuário.



Graph Enhanced Search - Enredo de filme

Você pode aprimorar a busca vetorial da trama do filme com o percurso de grafos, por meio de:

- Adicionar atores, diretores ou gêneros relacionados.
- Encontrar outros filmes com temas ou conexões semelhantes.
- Utilizando avaliações de usuários para filtrar ou classificar resultados.

Esta consulta Cypher demonstra como realizar uma busca vetorial por enredos de filmes e, em seguida, percorrer o grafo para recuperar contexto adicional

```
// Search for movie plots using vector search
WITH genai.vector.encode(
    "A mysterious spaceship lands Earth",
    "OpenAI",
    { token: "sk-..." }) AS myMoviePlot
CALL db.index.vector.queryNodes('moviePlots', 6, myMoviePlot)
YIELD node, score

// Traverse the graph to find related actors, genres, and user ratings
MATCH (node)<-[r:RATED]-()
RETURN
  node.title AS title, node.plot AS plot, score AS similarityScore,
  collect { MATCH (node)-[:IN_GENRE]->(g) RETURN g.name } as genres,
  collect { MATCH (node)<-[:ACTED_IN]->(a) RETURN a.name } as actors,
  avg(r.rating) as userRating
ORDER BY userRating DESC
```

Pesquisa de texto completo

A busca por texto completo é outra técnica poderosa que pode ser combinada com a busca aprimorada por grafos para melhorar ainda mais a recuperação de informações.

Embora a busca vetorial seja excelente para encontrar conteúdo semanticamente semelhante, a busca de texto completo permite que os usuários encontrem palavras-chave ou frases específicas em documentos ou nós.

Se o usuário estiver procurando por um filme ou ator pelo nome, a busca por texto completo pode localizar rapidamente essas entidades com base em correspondências exatas de texto.

A pesquisa de texto completo pode ser usada como substituta ou em conjunto com a pesquisa vetorial.

Quando usada em conjunto com a busca vetorial, a busca de texto completo pode refinar os resultados, filtrando conteúdo irrelevante com base em palavras-chave ou frases específicas.

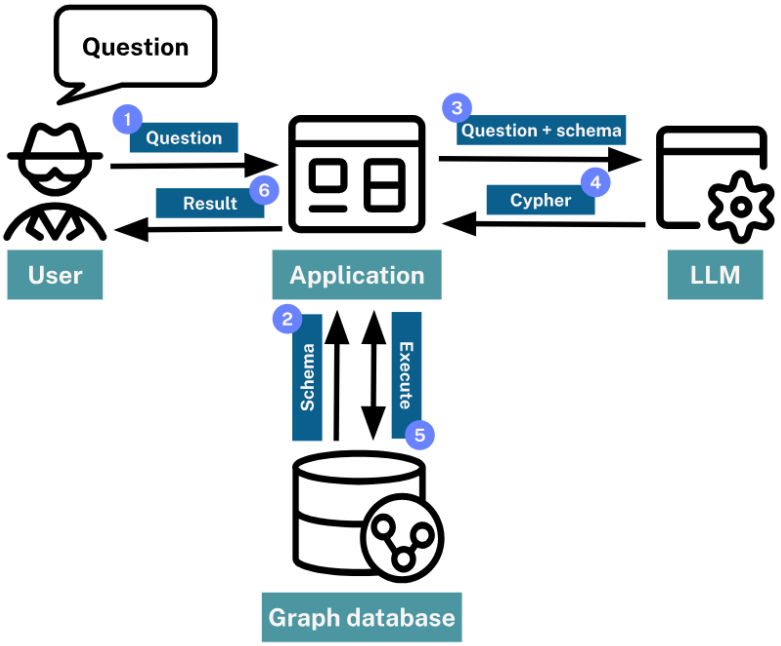
Texto para Cypher

O recurso Texto para Cypher é uma abordagem alternativa no GraphRAG que permite aos usuários expressar suas necessidades de informação em linguagem natural, que é então traduzida automaticamente em consultas Cypher.

Você aproveita o poder dos LLMs para interpretar a intenção do usuário e gerar consultas gráficas precisas, permitindo acesso direto a dados estruturados e relacionamentos dentro do grafo.

Você pode usar a conversão de texto em Cypher para transformar as consultas dos usuários em buscas complexas, agregações ou travessias, tornando a consulta avançada em grafos mais acessível e flexível.

A função Text to Cypher funciona passando a consulta do usuário e o esquema do grafo para um LLM, que gera uma consulta Cypher que pode ser executada no banco de dados de grafos.



3. Grafos de conhecimento

O que é um grafo de conhecimento?

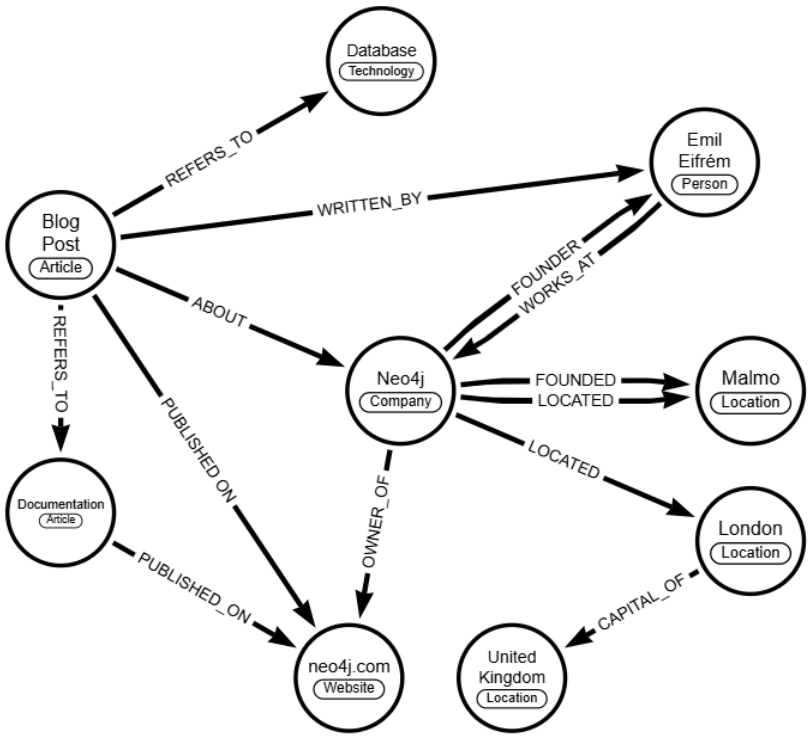
> *Um grafo de conhecimento é uma representação organizada de entidades do mundo real e seus relacionamentos.*

Os grafos de conhecimento fornecem uma maneira estruturada de representar entidades, seus atributos e seus relacionamentos, permitindo uma compreensão abrangente e interconectada das informações.

Os grafos de conhecimento são úteis para aplicações de IA generativa porque fornecem dados estruturados e interconectados que aprimoram o contexto, o raciocínio e a precisão nas respostas geradas.

Os mecanismos de busca normalmente usam grafos de conhecimento para fornecer informações sobre pessoas, lugares e coisas.

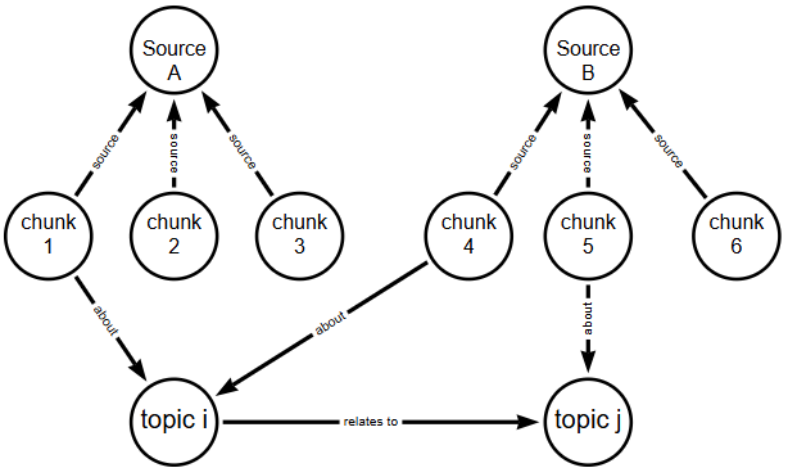
Este grafo de conhecimento poderia representar o Neo4j:



Os grafos de conhecimento podem decompor fontes de informação e integrá-las, permitindo visualizar as relações entre os dados.

Essa integração de diversas fontes proporciona aos grafos de conhecimento uma visão mais holística e facilita consultas complexas, análises e insights.

Os grafos de conhecimento podem se adaptar e evoluir facilmente à medida que crescem, incorporando novas informações e mudanças de estrutura.



O Neo4j é ideal para representar e consultar dados complexos e interconectados em Grafos de Conhecimento. Ao contrário dos bancos de dados relacionais tradicionais, que usam tabelas e linhas, o Neo4j utiliza um modelo baseado em grafos com nós e relacionamentos.

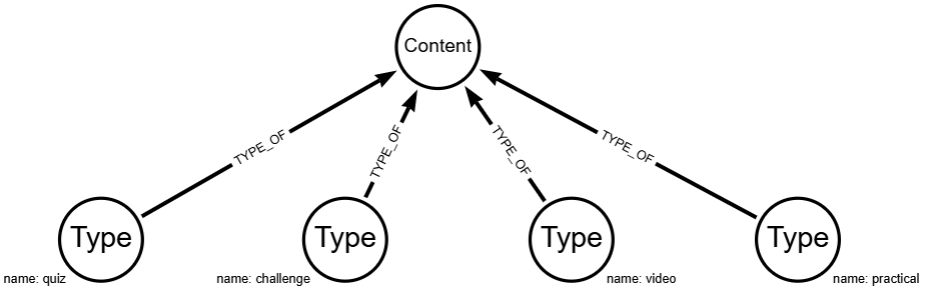
Princípios de organização

Um grafo de conhecimento armazena dados e relacionamentos juntamente com estruturas conhecidas como princípios organizadores.

Os princípios organizadores são as regras ou categorias que regem os dados e lhes conferem estrutura. Esses princípios podem variar desde descrições simples dos dados, como por exemplo, descrever um curso da GraphAcademy como "curso" course → modules → lessons, até um vocabulário complexo que descreve a solução completa.

Os grafos de conhecimento são inerentemente flexíveis, e você pode alterar os princípios de organização à medida que os dados crescem e mudam.

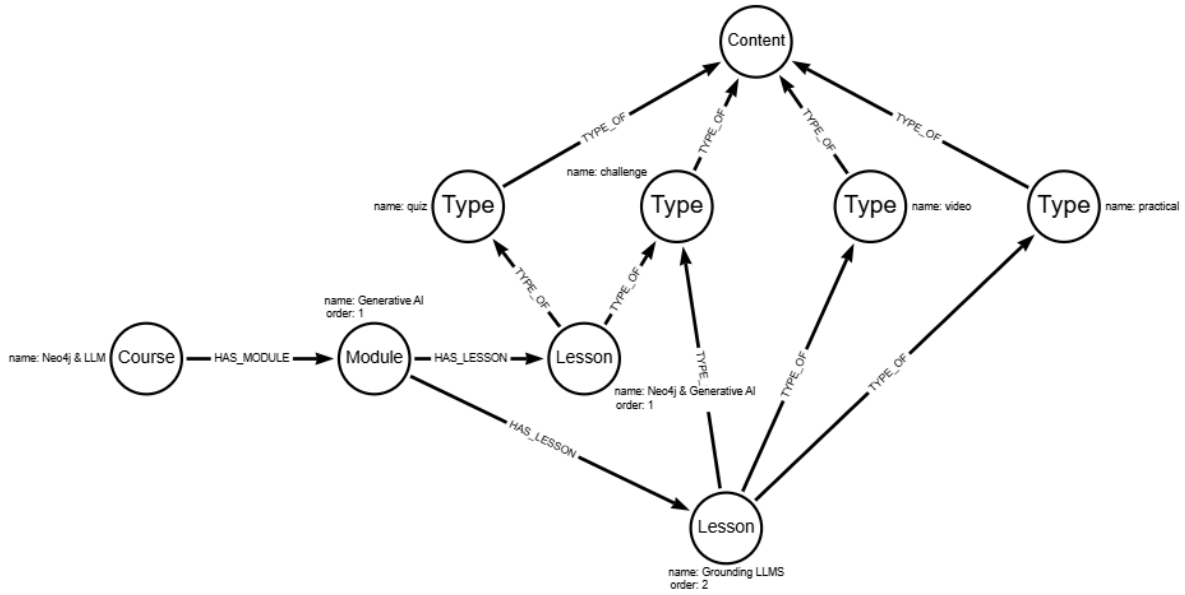
Os princípios organizacionais que descrevem o conteúdo do GraphAcademy poderiam ser assim:



Os princípios organizadores são armazenados como nós no grafo e podem ser armazenados juntamente com os dados propriamente ditos.

Essa integração de princípios organizacionais e dados permite a realização de consultas e análises complexas.

Mapear os princípios organizacionais ao conteúdo da lição no GraphAcademy poderia ser feito da seguinte forma:



Aplicações de IA generativa

Em aplicações de IA generativa, os grafos de conhecimento desempenham um papel crucial ao capturar e organizar informações importantes, específicas de um domínio ou proprietárias da empresa. Eles não se limitam a dados estritamente estruturados — os grafos de conhecimento também podem integrar e representar informações menos organizadas ou não estruturadas.

O GraphRAG pode usar grafos de conhecimento para contexto, formando a base para aplicações que aproveitam dados proprietários ou específicos de um domínio. Ao fundamentar as respostas em um grafo de conhecimento, essas aplicações podem fornecer respostas mais precisas e maior *explicabilidade* , graças ao rico contexto e aos relacionamentos presentes nos dados.

Criando grafos de conhecimento

A forma como você cria um grafo de conhecimento depende do tipo de dados que você possui e de como deseja estruturá-los.

Dados não estruturados

Dados não estruturados, como documentos de texto, páginas da web ou PDFs, podem ser uma rica fonte para grafos de conhecimento.

Criar grafos de conhecimento a partir de dados não estruturados pode ser complexo, envolvendo múltiplas etapas de consulta, limpeza e transformação de dados.

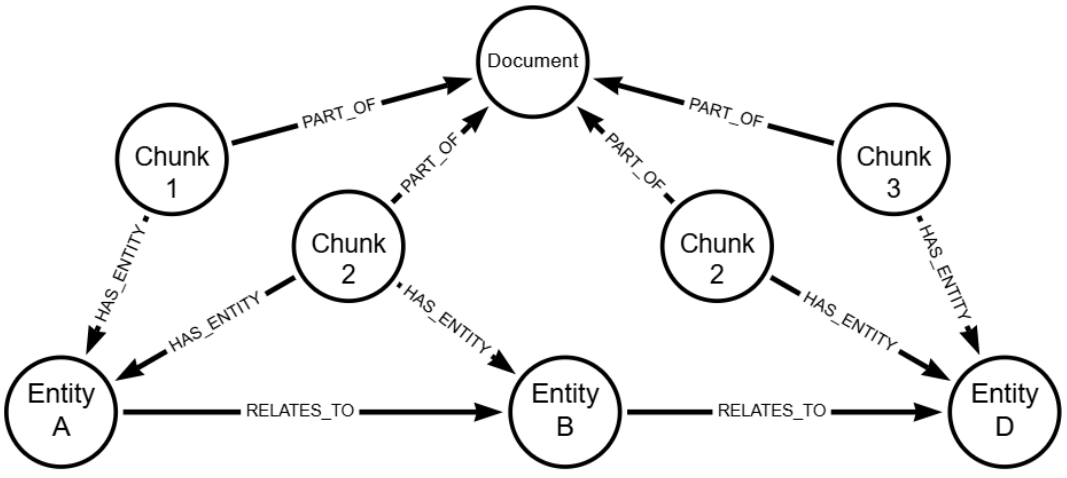
No entanto, você pode usar os recursos de análise de texto dos Modelos de Linguagem de Grande Porte (LLMs) para ajudar a automatizar a criação de grafos de conhecimento.

Normalmente, você seguiria estes passos para construir um grafo de conhecimento a partir de texto não estruturado usando um LLM:

1. Reunir os dados - Os dados podem vir de múltiplas fontes e estar em diferentes formatos.
2. Divida os dados em partes gerenciáveis, ou *blocos* , que o LLM possa processar com eficiência.
3. *Vetorizar* os dados - Dependendo dos seus requisitos para consultar e pesquisar os dados, você pode precisar criar representações vetoriais (embeddings) .
4. Passe os dados para um LLM - Extraia entidades (nós) e relacionamentos dos dados. Você pode fornecer contexto ou restrições adicionais para a extração, como o tipo de entidades ou relacionamentos que deseja extrair.
5. Gere o grafo - Utilize a saída do LLM para criar nós e relações no grafo.

Estrutura de dados não estruturada

Uma estrutura típica para um grafo de conhecimento poderia ser assim:



Você pode aprender mais sobre como construir grafos de conhecimento a partir de dados não estruturados no curso "Building Knowledge Graphs with LLMs" da GraphAcademy .

Dados estruturados

Construir grafos de conhecimento a partir de dados estruturados costuma ser mais simples do que a partir de fontes não estruturadas. Os dados estruturados já estão organizados, o que facilita o mapeamento de nós, relacionamentos e relações diretamente em um grafo.

Para criar um grafo de conhecimento a partir de dados estruturados, normalmente você:

1. Identifique as fontes - As fontes de dados podem ser outros gráficos, bancos de dados relacionais, arquivos CSV, APIs.
2. Analise os dados – Compreenda as entidades, os atributos e os relacionamentos presentes nos seus dados (por exemplo, as linhas de uma tabela podem representar entidades, as colunas podem representar atributos e as chaves estrangeiras podem representar relacionamentos).
3. Defina um esquema de grafo - O esquema deve representar as entidades como nós e relacionamentos, além de definir os princípios organizadores do grafo.
4. Criar o grafo - Transformar e importar os dados para o banco de dados do grafo.

Esse processo permite aproveitar fontes de dados estruturados existentes — como bancos de dados relacionais, arquivos CSV ou APIs — para construir rapidamente um grafo de conhecimento que pode ser consultado e expandido conforme necessário.

Você pode aprender mais sobre como importar dados para o Neo4j no curso Fundamentos de Importação de Dados da GraphAcademy .

4. Integrando o Neo4j com a IA Generativa

GraphRAG para Python

O pacote GraphRAG para Python neo4j-graphrag ( ) permite acessar funções de IA generativa do Neo4j, incluindo:

- Retrievers
- Pipelines GraphRAG
- Construção de grafo de conhecimento

O objetivo é fornecer um pacote completo para desenvolvedores, onde o Neo4j possa garantir compromisso e manutenção a longo prazo, além de lançar rapidamente novos recursos e padrões e métodos de alto desempenho.

Você usará o neo4j-graphrag acote para criar funções de recuperação e implementar aplicações simples que utilizam o GraphRAG para fornecer contexto às consultas LLM.

Você precisa configurar um ambiente de desenvolvimento para executar os exemplos de código e os exercícios.

Vector Retriever

In this lesson, you will create a vector retriever to retrieve relevant data from Neo4j.

A retriever is a component that takes unstructured data (typically a users query) and retrieves relevant data.

You will create a vector retriever that find similar movies based on a movie plot. The retriever will use the `moviePlots` vector index you used to search for similar movies using Cypher.

To find similar movies using a retriever you need to:

1. Connect to a Neo4j database
2. Create an *embedder* to convert users queries into vectors
3. Create a *retriever* that uses the `moviePlots` vector index
4. Use the retriever to search for similar movies using the users query
5. Parse the results

Open the genai-fundamentals/vector_retriever.py file and review the program:

```
import os
from dotenv import load_dotenv
load_dotenv()

from neo4j import GraphDatabase
from neo4j_graphrag.embeddings.openai import OpenAIEmbeddings
from neo4j_graphrag.retrievers import VectorRetriever

# Connect to Neo4j database
driver = GraphDatabase.driver(
    os.getenv("NEO4J_URI"),
    auth=(
        os.getenv("NEO4J_USERNAME"),
        os.getenv("NEO4J_PASSWORD")
    )
)

# Create embedder
embedder = OpenAIEmbeddings(
    model="text-embedding-3-small"
)

# Create retriever
retriever = VectorRetriever(
    driver,
    index_name="moviePlots",
    embedder=embedder,
    return_properties=["title", "plot"],
)

# Search for similar items
result = retriever.search(
    query_text = "Toys coming alive",
    top_k = 5
)

# Parse results
for item in result.items:
    print(item.content, item.metadata["score"])

# CLose the database connection
driver.close()
```

output:

```
(venv) C:\Users\anamoser\genai-fundamentals>python
genai-fundamentals/vector_retriever.py
{'title': 'Toy Story', 'plot': 'Toys coming alive and having adventures with
a boy named Andy.'} 0.8465895652770996
```

RAG Pipeline

You can use a retriever as part of a RAG (Retrieval-Augmented Generation) pipeline to provide context to a LLM.

In this lesson, you will use the vector retriever you created to pass additional context to an LLM allowing it to generate more accurate and relevant responses.

Open the `genai-fundamentals/vector_rag.py` file and review the program:

```python
import os
from dotenv import load_dotenv
load_dotenv()

from neo4j import GraphDatabase
from neo4j_graphrag.embeddings.openai import OpenAIEmbeddings
from neo4j_graphrag.retrievers import VectorRetriever
from neo4j_graphrag.llm import OpenAILLM
from neo4j_graphrag.generation import GraphRAG

# Connect to Neo4j database
driver = GraphDatabase.driver(
    os.getenv("NEO4J_URI"),
    auth=(
        os.getenv("NEO4J_USERNAME"),
        os.getenv("NEO4J_PASSWORD")
    )
)

# Create embedder
embedder = OpenAIEmbeddings(model="text-embedding-3-small")

# Create retriever
retriever = VectorRetriever(
    driver,
    index_name="moviePlots",
    embedder=embedder,
```

```
    return_properties=["title", "plot"],
)

# Create the LLM
llm = OpenAILLM(
    model_name="gpt-4o",
    model_params={
        "temperature": 0.4,
        #"max_tokens": 1000
    }
)

# Create GraphRAG pipeline
rag =  GraphRAG(
    retriever=retriever,
    llm=llm,
)

# Search
query_text = "Find me movies about toys coming alive"

response = rag.search(
    query_text=query_text,
    retriever_config={"top_k":5}
)

print(response.answer)

# CLose the database connection
driver.close()
```

output:

```
(venv) C:\Users\anamoser\genai-fundamentals>python
genai-fundamentals/vector_rag.py
One movie about toys coming alive is "Toy Story," which features toys having
adventures with a boy named Andy.
context: [RetrieverResultItem(content="{'title': 'Toy Story', 'plot': 'Toys
coming alive and having adventures with a boy named Andy.'}",
metadata={'score': 0.7947583198547363, 'nodeLabels': ['Movie'], 'id':
'4:751ff32d-f8de-4caa-a2a9-4baa7d9ff6e8:0'})]
```

Graph-Enhanced Vector Retriever

To take advantage of the relationships in the graph, you can create a retriever that uses both vector search and graph traversal to find relevant data.

The `VectorCypherRetriever` allows you to perform vector searches and then traverse the graph to find related nodes or entities.

Open the `genai_fundamentals/vector_cypher_rag.py` file and review the code:

```
import os
from dotenv import load_dotenv
load_dotenv()

from neo4j import GraphDatabase
from neo4j_graphrag.embeddings.openai import OpenAIEmbeddings
from neo4j_graphrag.llm import OpenAILLM
from neo4j_graphrag.generation import GraphRAG
from neo4j_graphrag.retrievers import VectorCypherRetriever

# Connect to Neo4j database
driver = GraphDatabase.driver(
    os.getenv("NEO4J_URI"),
    auth=(
        os.getenv("NEO4J_USERNAME"),
        os.getenv("NEO4J_PASSWORD")
    )
)

# Create embedder
embedder = OpenAIEmbeddings(model="text-embedding-3-small")

# Define retrieval query
retrieval_query = """
MATCH (node) <-[r:RATED]-()
RETURN
    node.title AS title,
    node.plot AS plot,
    score AS similarityScore,
    collect { MATCH (node)-[:IN_GENRE]->(g) RETURN g.name } as genres,
    collect { MATCH (node)<-[:ACTED_IN]->(a) RETURN a.name } as actors,
    avg(r.rating) as userRating
ORDER BY userRating DESC
"""
```

```
# Create retriever
retriever = VectorCypherRetriever(
    driver,
    index_name="moviePlots",
    retrieval_query=retrieval_query,
    embedder=embedder,
)

#  Create the LLM
llm = OpenAILLM(model_name="gpt-4o")

# Create GraphRAG pipeline
rag = GraphRAG(retriever=retriever, llm=llm)

# Search
query_text = "Find the highest rated action movie about travelling to other
planets"

response = rag.search(
    query_text=query_text,
    retriever_config={"top_k": 5},
    return_context=True
)

print(response.answer)
print("CONTEXT:", response.retriever_result.items)

# Close the database connection
driver.close()
```

output:

```
(venv) C:\Users\anamoser\genai-fundamentals>python
genai-fundamentals/vector_cypher_rag.py
Received notification from DBMS server: {severity: WARNING} {code:
Neo.ClientNotification.Statement.UnknownRelationshipTypeWarning} {category:
UNRECOGNIZED} {title: The provided relationship type is not in the database.}
{description: One of the relationship types in your query is not available in
the database, make sure you didn't misspell it or that the label is available
when you run this statement in your application (the missing relationship
type is: RATED)} {position: line: 2, column: 19, offset: 168} for query:
'CALL db.index.vector.queryNodes($vector_index_name, $top_k *
$effective_search_ratio, $query_vector) YIELD node, score WITH node, score
LIMIT $top_k \nMATCH (node) <-[r:RATED]-()\nRETURN\n    node.title AS
title,\n    node.plot AS plot,\n    score AS similarityScore,\n    collect {
```

```
MATCH (node)-[:IN_GENRE]->(g) RETURN g.name } as genres,\n    collect { MATCH
(node)<-[:ACTED_IN]->(a) RETURN a.name } as actors,\n    avg(r.rating) as
userRating\nORDER BY userRating DESC\n'
Received notification from DBMS server: {severity: WARNING} {code:
Neo.ClientNotification.Statement.UnknownRelationshipTypeWarning} {category:
UNRECOGNIZED} {title: The provided relationship type is not in the database.}
{description: One of the relationship types in your query is not available in
the database, make sure you didn't misspell it or that the label is available
when you run this statement in your application (the missing relationship
type is: IN_GENRE)} {position: line: 7, column: 30, offset: 292} for query:
'CALL db.index.vector.queryNodes($vector_index_name, $top_k *
$effective_search_ratio, $query_vector) YIELD node, score WITH node, score
LIMIT $top_k \nMATCH (node) <-[r:RATED]-()\nRETURN\n    node.title AS
title,\n    node.plot AS plot,\n    score AS similarityScore,\n    collect {
MATCH (node)-[:IN_GENRE]->(g) RETURN g.name } as genres,\n    collect { MATCH
(node)<-[:ACTED_IN]->(a) RETURN a.name } as actors,\n    avg(r.rating) as
userRating\nORDER BY userRating DESC\n'
Received notification from DBMS server: {severity: WARNING} {code:
Neo.ClientNotification.Statement.UnknownRelationshipTypeWarning} {category:
UNRECOGNIZED} {title: The provided relationship type is not in the database.}
{description: One of the relationship types in your query is not available in
the database, make sure you didn't misspell it or that the label is available
when you run this statement in your application (the missing relationship
type is: ACTED_IN)} {position: line: 8, column: 31, offset: 364} for query:
'CALL db.index.vector.queryNodes($vector_index_name, $top_k *
$effective_search_ratio, $query_vector) YIELD node, score WITH node, score
LIMIT $top_k \nMATCH (node) <-[r:RATED]-()\nRETURN\n    node.title AS
title,\n    node.plot AS plot,\n    score AS similarityScore,\n    collect {
MATCH (node)-[:IN_GENRE]->(g) RETURN g.name } as genres,\n    collect { MATCH
(node)<-[:ACTED_IN]->(a) RETURN a.name } as actors,\n    avg(r.rating) as
userRating\nORDER BY userRating DESC\n'
Received notification from DBMS server: {severity: WARNING} {code:
Neo.ClientNotification.Statement.UnknownPropertyKeyWarning} {category:
UNRECOGNIZED} {title: The provided property key is not in the database}
{description: One of the property names in your query is not available in the
database, make sure you didn't misspell it or that the label is available
when you run this statement in your application (the missing property name
is: rating)} {position: line: 9, column: 11, offset: 416} for query: 'CALL
db.index.vector.queryNodes($vector_index_name, $top_k *
$effective_search_ratio, $query_vector) YIELD node, score WITH node, score
LIMIT $top_k \nMATCH (node) <-[r:RATED]-()\nRETURN\n    node.title AS
title,\n    node.plot AS plot,\n    score AS similarityScore,\n    collect {
MATCH (node)-[:IN_GENRE]->(g) RETURN g.name } as genres,\n    collect { MATCH
(node)<-[:ACTED_IN]->(a) RETURN a.name } as actors,\n    avg(r.rating) as
userRating\nORDER BY userRating DESC\n'
```

```
Based on the context provided, a highly rated action movie about traveling to
other planets is "Interstellar." Directed by Christopher Nolan, this film is
renowned for its stunning visuals, compelling storyline, and innovative
depiction of space travel.
CONTEXT: []
```

Context

When you run the code, it will complete a vector search for the provided query and then traverse the graph to find related nodes.

The additional context allows the LLM to generate more accurate responses based on the additional data in the graph.

Transparency

The context is returned after the response, allowing you to see what data was used to generate the response.
This transparency is important for understanding how the LLM arrived at its response and for debugging purposes.

When sent the query *"Find the highest rated action movie about travelling to other planets"*, the GraphRAG pipeline will follow these steps:

1. Perform a vector search for movie plots related to *travelling to other planets*.
2. Run the retrieval query to find related actors, genres, and user ratings.
3. Pass the retrieved data to the LLM to generate a response.

You can expect a response to be based on:

- Travelling to other planets.
- The action genre.
- With the highest user rating (not the vector similarity score).

A typical response might be *"The highest rated action movie about traveling to other planets is "Aliens," with a user rating of 3.92"*

Test the code with different queries relating to movies, actors, and genres, such as:

- *Find a comedy movie about vampires*
- *Who acts in drama movies about romance and love?*
- *What genres are represented about movies where the hero fails his mission?*

Optional challenge

Modify the retrieval query to include the directors of the movies in the context.

Directors can be found using the pattern `(node)←[:DIRECTED]-(director)`.

Try queries relating to directors, such as *"Who has directed movies about weddings?"*

Text to Cypher Retriever

Vector and full text retrievers are great for finding relevant data based on semantic similarity or keyword matching.

To answer more specific questions, you may need to perform more complex queries to find data relating to specific nodes, relationships, or properties.

For example, you want to find:

- The age of an actor.
- Who acted in a movie.
- Movie recommendations based on rating.

Text to Cypher retrievers allow you to convert natural language queries into Cypher queries that can be executed against the graph.

```
[User Query]
"What year was the movie Babe released?"
[Generated Cypher Query]
MATCH (m:Movie)
WHERE m.title = 'Babe'
RETURN m.released
[Cypher Result]
1995
[LLM Response]
"The movie Babe was released in 1995."
```

Open the genai_fundamentals/text2cypher_rag.py file and review the code:

```
import os
from dotenv import load_dotenv
load_dotenv()

from neo4j import GraphDatabase
from neo4j_graphrag.llm import OpenAILLM
from neo4j_graphrag.generation import GraphRAG
from neo4j_graphrag.retrievers import Text2CypherRetriever

# Connect to Neo4j database
driver = GraphDatabase.driver(
    os.getenv("NEO4J_URI"),
    auth=(
        os.getenv("NEO4J_USERNAME"),
        os.getenv("NEO4J_PASSWORD")
    )
)

# Create Cypher LLM
t2c_llm = OpenAILLM(
    model_name="gpt-4o",
    model_params={"temperature": 0}
)

# Specify your own Neo4j schema
neo4j_schema = """
Node properties:
Person {name: STRING, born: INTEGER}
Movie {tagline: STRING, title: STRING, released: INTEGER}
Genre {name: STRING}
User {name: STRING}

Relationship properties:
ACTED_IN {role: STRING}
RATED {rating: INTEGER}

The relationships:
(:Person)-[:ACTED_IN]->(:Movie)
(:Person)-[:DIRECTED]->(:Movie)
(:User)-[:RATED]->(:Movie)
```

```
(:Movie)-[:IN_GENRE]->(:Genre)
"""

# Cypher examples as input/query pairs
examples = [
    "USER INPUT: 'Get user ratings for a movie?' QUERY: MATCH
(u:User)-[r:RATED]->(m:Movie) WHERE m.title = 'Movie Title' RETURN
r.rating"
]

# Build the retriever
retriever = Text2CypherRetriever(
    driver=driver,
    llm=t2c_llm,
    neo4j_schema=neo4j_schema,
    examples=examples,
)

llm = OpenAILLM(model_name="gpt-4o")
rag = GraphRAG(retriever=retriever, llm=llm)

query_text = "Which movies did Hugo Weaving star in?"
query_text = "How many movies are in the Sci-Fi genre?"
query_text = "What is the highest rating for Goodfellas?"
query_text = "What is the averaging user rating for the movie Toy
Story?"
query_text = "What year was the movie Babe released?"

response = rag.search(
    query_text=query_text,
    return_context=True
    )

print(response.answer)
print("CYPHER :", response.retriever_result.metadata["cypher"])
print("CONTEXT:", response.retriever_result.items)

driver.close()
```

output:

```
(venv) C:\Users\anamoser\genai-fundamentals>python
genai-fundamentals/text2cypher_rag.py
```

```
Received notification from DBMS server: {severity: WARNING} {code:
Neo.ClientNotification.Statement.UnknownPropertyKeyWarning} {category:
UNRECOGNIZED} {title: The provided property key is not in the database}
{description: One of the property names in your query is not available in the
database, make sure you didn't misspell it or that the label is available
when you run this statement in your application (the missing property name
is: released)} {position: line: 1, column: 49, offset: 48} for query: "MATCH
(m:Movie) WHERE m.title = 'Babe' RETURN m.released"
1995
CYPHER : MATCH (m:Movie) WHERE m.title = 'Babe' RETURN m.released
CONTEXT: []
```

GenAI Frameworks

A variety of open-source and community-supported frameworks are available to help you integrate Neo4j with generative AI and large language models (LLMs).

These frameworks support use cases such as:

- Retrieval-augmented generation (RAG).
- Agentic workflows.
- Knowledge graph construction.

Typically these frameworks include:

- LLM usage, prompt and output management.
- Embedding model integration .
- Vector and database integration (including Neo4j).
- RAG (Retrieval-Augmented Generation) workflows.
- Agentic workflows and orchestration.
- Monitoring, observability, and deployment tools.

Popular GenAI Frameworks for Neo4j

- LangChain (Python) - A leading open-source framework for building LLM-powered applications, with strong support for Neo4j as a vector store and knowledge graph.
- LangChainJS - The JavaScript/TypeScript version of LangChain, enabling GenAI workflows in Node.js and browser environments.
- LlamaIndex - A data framework for connecting LLMs to external data, with connectors for Neo4j to support RAG and knowledge graph use cases.
- Spring AI - A Spring ecosystem project for integrating AI capabilities into Java applications, including Neo4j support.

- Langchain4j - A Java version of LangChain, supporting Neo4j integration for LLM and RAG workflows.
- Haystack - An open-source framework for building search and question-answering systems, with Neo4j integration for graph-based retrieval.
- Semantic Kernel - A Microsoft open-source orchestration library for AI workflows, supporting Neo4j as a data source.
- DSPy - A framework for programming and optimizing LLM pipelines, with Neo4j connectors.

You can learn more in the Neo4j GenAI Frameworks documentation.

Frameworks support your GenAI application development and help you build complex workflows that integrate Neo4j with LLMs and other AI components.