

```
= Vector Indexes
:order: 2
:type: lesson
:sandbox: true
```

In the last lesson, you learned about vectors and their role in Semantic Search.

In this lesson, you will learn how to create vector embeddings of text content in an existing Neo4j database.

```
== Vectorizing Movie Plots
```

GraphAcademy created a Movie Recommendation Sandbox when you enrolled in this course. The sandbox database contains over 9000 movies, 15000 actors, and over 100000 user ratings.

Each movie has a `.plot` property.

```
.Movie Plot Example
[source,cypher]
MATCH (m:Movie {title: "Toy Story"})
RETURN m.title AS title, m.plot AS plot
```

"A cowboy doll is profoundly threatened and jealous when a new spaceman figure supplants him as top toy in a boy's room."

You can use the vector index to find the most similar movies by converting the plots into vector embeddings and comparing them.

You will use a pre-created [link:https://data.neo4j.com/llm-fundamentals/openai-embeddings.csv](https://data.neo4j.com/llm-fundamentals/openai-embeddings.csv) [CSV file of 1000 movie plot vector embeddings<sup>^</sup>] in this lesson.

The CSV file contains:

- \* ``movieId`` - The ID of the movie
- \* ``embedding`` - The vector embedding of the movie plot generated by OpenAI

```
[source,csv]
----
movieId, embedding
1, [-0.0271058, -0.0242211, 0.0060390322, -0.02437703, ...]
2, [-0.001596838, -0.022397375, 0.0046575777, 0.0019427929, ...]
----
```

[TIP]

.Generating the Embeddings

====

[link:https://platform.openai.com/docs/guides/embeddings/what-are-embeddings](https://platform.openai.com/docs/guides/embeddings/what-are-embeddings) [OpenAI's text-embedding-ada-002 model<sup>^</sup>] was used to create the embeddings. It is a cost-effective model that can generate embeddings for text.

A simple Python script calls the embedding endpoint served by OpenAI. The link:[https://github.com/neo4j-graphacademy/llm-fundamentals/blob/main/plot\\_openai\\_embeddings.py](https://github.com/neo4j-graphacademy/llm-fundamentals/blob/main/plot_openai_embeddings.py)[code^] is available in the link:<https://github.com/neo4j-graphacademy/llm-fundamentals>[github.com/graphacademy/llm-fundamentals^] repository.

Each LLM will provide an embedding in its shape.

====

== Loading Embeddings

The embeddings will be stored as a `.plotEmbedding` property on the `(:Movie)` node.

You will use the `LOAD CSV` command to load the embeddings into the Neo4j Sandbox instance.

The following Cypher loads the embeddings CSV file, performs a `MATCH` query to find the `(:Movie)` node with the corresponding `movieId` property, and then sets the `.plotEmbedding` property on that node.

Review this Cypher statement before running it.

.Loading the Embeddings

[source,cypher]

----

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/llm-fundamentals/openai-embeddings.csv'

AS row

MATCH (m:Movie {movieId: row.movieId})

CALL db.create.setNodeVectorProperty(m, 'plotEmbedding',

apoc.convert.fromJsonList(row.embedding))

RETURN count(\*)

----

The statement:

\* Loads the CSV file

\* Matches the `(:Movie)` node with the corresponding `movieId` property

\* Calls `db.create.setNodeVectorProperty()` procedure to set the `.plotEmbedding` property

\* The procedure also validates that the property is a valid vector

Run the statement to create the Movie embeddings.

Once complete, you can query the database to see the `.plotEmbedding` property on the `(:Movie)` nodes.

[source,cypher]

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

RETURN m.title AS title, m.plot AS plot, m.plotEmbedding

[TIP]

.LOAD CSV and Strings

====

When data is loaded using `LOAD CSV`, it is treated as a string unless specifically cast using a specific function, for example, `toInteger()` or `toFloat()`.

In this case, the embedding is a string representing a JSON list, the statement coerces it into a Cypher `List`  
link:<https://neo4j.com/docs/apoc/current/overview/apoc.convert/apoc.convert.fromJsonList/>[using the `apoc.convert.fromJsonList()` procedure^].

You can

link:<https://graphacademy.neo4j.com/courses/importing-cypher/>[learn how to use the `LOAD CSV` command in the Importing CSV Data into Neo4j course^].

====

== Creating the Vector Index

You will need to create a vector index to search across these embeddings.

You will use the `CREATE VECTOR INDEX` Cypher statement to create the index:

```
.CREATE VECTOR INDEX Syntax  
[source,cypher, role=noplay nocopy]
```

----

```
CREATE VECTOR INDEX [index_name] [IF NOT EXISTS]  
FOR (n:LabelName)  
ON (n.propertyName)  
OPTIONS {" option: value[, ...] }"
```

----

`CREATE VECTOR INDEX` expects the following parameters:

- \* `index\_name` - The name of the index
- \* `LabelName` - The node label on which to index
- \* `propertyName` - The property on which to index
- \* `OPTIONS` - The options for the index, where you can specify:
  - \*\* `vector.dimensions` - The dimension of the embedding e.g. OpenAI embeddings consist of `1536` dimensions.
  - \*\* `vector.similarity\_function` - The similarity function to use when comparing values in this index - this can be `euclidean` or `cosine`.

Review and run the following Cypher to create the vector index:

```
.Create the vector index  
[source,cypher]
```

----

```
CREATE VECTOR INDEX moviePlots IF NOT EXISTS  
FOR (m:Movie)  
ON m.plotEmbedding  
OPTIONS {indexConfig: {
```

```
`vector.dimensions`: 1536,  
`vector.similarity_function`: 'cosine'  
}}  
----
```

Note that the index is called `moviePlots`, it is against the `Movie` label, and it is on the `.plotEmbedding` property. The `vector.dimensions` is `1536` (as used by OpenAI) and the `vector.similarity\_function` is `cosine`. The `IF NOT EXISTS` clause ensures that the statement only creates the index if it does not already exist.

Run the statement to create the index.

[TIP]

.Choosing a Similarity Function

====

Generally, cosine will perform best for text embeddings, but you may want to experiment with other functions.

You can

link:<https://neo4j.com/docs/cypher-manual/current/indexes-for-vector-search/#indexes-vector-similarity>[read more about similarity functions in the documentation^].

Typically, you will choose a similarity function closest to the loss function used when training the embedding model. You should refer to the model's documentation for more information.

====

=== Check the index creation status

The index will be updated asynchronously. You can check the status of the index population using the `SHOW INDEXES` statement:

Check that you created the index successfully using the `SHOW INDEXES` command.

.Show Indexes

[source,cypher]

----

```
SHOW INDEXES YIELD id, name, type, state, populationPercent WHERE type =  
"VECTOR"
```

----

You should see a result similar to the following:

.Show Indexes Result

|===

```
| id | name | type | state | populationPercent
```

```
| 1 | "moviePlots" | "VECTOR" | "ONLINE" | `100.0`
```

|===

Once the `state` is listed as online, the index will be ready to query.

The `populationPercentage` field indicates the proportion of node and property pairing.

When the `populationPercentage` is `100.0`, all the movie embeddings have been indexed.

## == Querying Vector Indexes

You can query the index using the `db.index.vector.queryNodes()` procedure.

The procedure returns the requested number of approximate nearest neighbor nodes and their similarity score, ordered by the score.

```
.db.index.vector.queryNodes Syntax  
[source,cypher,role=nocopy noplay]
```

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

The procedure accepts three parameters:

1. `indexName` - The name of the vector index
2. `numberOfNearestNeighbours` - The number of results to return
3. `query` - A list of floats that represent an embedding

The procedure yields two arguments:

- . A `node` which matches the query
- . A similarity `score` ranging from `0.0` to `1.0`.

```
// TODO - MH comment - should we also show how to search for a user  
question embedding? should we prepare a few in a file?
```

You can use this procedure to find the closest embedding value to a given embedding.

For example, find movies with a similar plot to another.

Review this Cypher before running it.

```
[source,cypher]  
.Similar Plots  
----  
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

----

The query finds the `_Toy Story_`Movie`` node and uses the `.plotEmbedding`` property to find the most similar plots. The ``db.index.vector.queryNodes()`` procedure uses the ``moviePlots`` vector index to find similar embeddings.

Run the query. The procedure returns the requested number of approximate nearest neighbor nodes and their similarity score, ordered by the score.

.Similar Plots Results

|===

| title | plot | score

| "Toy Story" | "A cowboy doll is profoundly threatened and jealous when a new spaceman figure supplants him as top toy in a boy's room." | 1.0

| "Little Rascals, The" | "Alfalfa is wooing Darla and his He-Man-Woman-Hating friends attempt to sabotage the relationship." | 0.9214372634887695

| "NeverEnding Story III, The" | "A young boy must restore order when a group of bullies steal the magical book that acts as a portal between Earth and the imaginary world of Fantasia." | 0.9206198453903198

| "Drop Dead Fred" | "A young woman finds her already unstable life rocked by the presence of a rambunctious imaginary friend from childhood." | 0.9199690818786621

| "E.T. the Extra-Terrestrial" | "A troubled child summons the courage to help a friendly alien escape Earth and return to his home-world." | 0.919100284576416

| "Gumby: The Movie" | "In this offshoot of the 1950s claymation cartoon series, the crazy Blockheads threaten to ruin Gumby's benefit concert by replacing the entire city of Clokeytown with robots." | 0.9180967211723328

|===

The similarity score is between ``0.0`` and ``1.0``, with ``1.0`` being the most similar. Note how the most similar plot is that of the `_Toy Story_` movie itself!

== Considerations

As you can see, this approach is relatively straightforward and can quickly yield results. The downside to this approach is that it relies heavily on the embeddings and similarity function to produce valid results.

This approach is also a black box - with 1536 dimensions, and it would be impossible to determine how the vectors are structured and how they influenced the similarity score.

The movies returned look similar, but without reading and comparing them, you would have no way of verifying that the results are correct.

== Check your understanding

```
include::questions/1-create-index.adoc[leveloffset=+1]
```

```
include::questions/2-query-index.adoc[leveloffset=+1]
```

[.summary]

== Lesson Summary

In this lesson, you learned how to create, populate, and use a Vector index in Neo4j.

In the next lesson, you will learn how to use feedback to improve the suggestions provided by Semantic Search.