Getting Started with Weaviate

Searching for information is an activity that has preoccupied humans for quite a long time. You could say that our compulsion to find things out and communicate them to others is one of the key traits that makes us human. As a species we've come up with various methods to make it easier for us to find things, such as libraries (or libraries of information) catalogs and indexes.

In this tutorial, we will understand what search engines are, how they work in general, and then focus on the **Weaviate** search engine. We will work with a datasets of blog posts and authors to aide our understanding.

Prerequisites

1. Docker.
2. Docker compose.
3. Working Weaviate installation.
4. Python 3.6+.
5. Weaviate Python client library.

What are search engines? How do they work? What problems do they solve?

A search engine is a system used to search for information on the Internet. Search engines are our gateway to 'the collective knowledge of mankind,' and in just 20 years we have gone from querying only text to being able to ask images, audio recordings and video too. This is a wonderful example of how it is possible for the intelligence of the whole human race to be focused by the network into something dramatically more powerful than any one individual can create through their own efforts.

In the early days of the web (the 1990s), the first search engines acted as directories of links to sites on the internet (think the original Yahoo.com website), then with improvements, came search engines that added “weight” to these links to enable them rank and return the most relevant results to search queries, the most popular is “**PageRank**” from Google.

As time went on, with the increase in data (text, audio, images and video) generated by the average computer system user, the need for the ability to query data in natural language for similarity, relevance, classification, recommendation, semantics etc which the traditional search engines were not providing led to the development of “**Vector Search**” engines.

What is a Vector Search Engine?

Traditional search engines are excellent, but are limited to keyword matching and cannot capture context. They work by finding pages based on keywords.

To find a page about cats, for example, you might type in the query 'cats.' A keyword search looks for webpages with that exact string of text anywhere on the page. That turns up the website http://www.elizabethancats.com/ (made up), which has a lot to say about...cats!

However, a vector search engine is a Web search engine that uses a large number of dimensions called “**Vectors**”, to represent the keywords in each document. This results in more precise matches, and better ranking.

In computer science the word vector means a sequence of numbers. So a vector search engine is designed to search for pieces of text that have a specific pattern in them, like three consecutive numbers or an email address. A vector search engine's job is to find other copies of the same text. That way, when you search for something like "123456" you'll find other instances of that number, not just one instance.

Despite being more complex than the simple popular systems of Google searches, these vector search engines give more precise and suitable results. Weaviate is an example of a vector search engine and additionally, a vector database as it stores data and its vector representation within.

How Vector Search Works

In this section I'll give a simple example to illustrate how vector search works and describe how you might use it in practice.

Imagine we have a simple document collection which only contains two words: “Spam” and “Eggs”:

Fig. 1.1 - The Document Collection

The vector search engine will count the number of times each word appears in each document and represent them as a list of numbers as displayed in the image below. This process is called “**Vectorization**”:

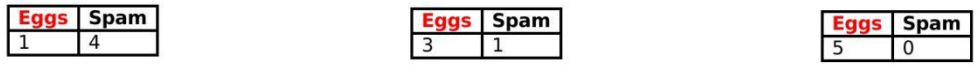

Fig. 1.2 - The Word Frequency Table

Next, the search engine will create an index which is a list of all the words of our vocabulary which at this moment is only contains the words, “Eggs” and “Spam” and note the documents each word appears in:

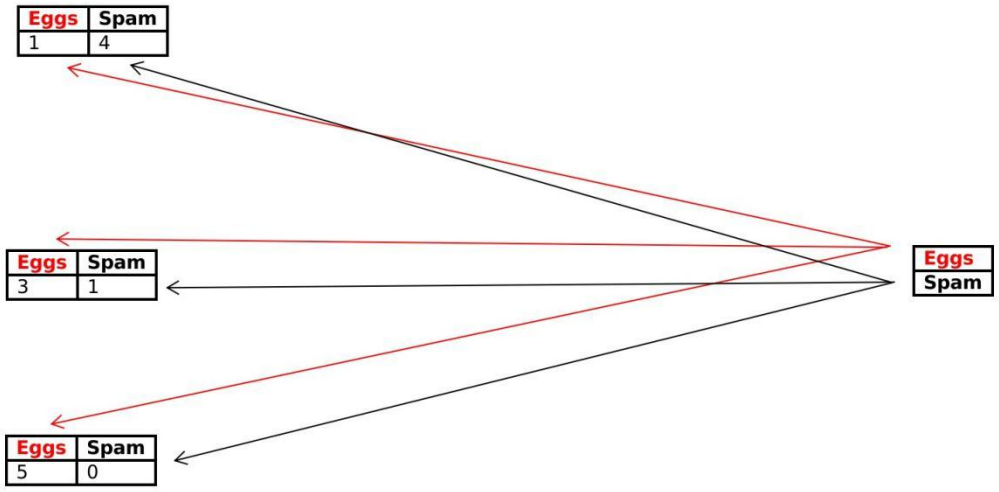

Fig. 1.3 - The Word Index

If a user enters a search query term “Spam” in the vector search engine, the search term is turned also into a vector as seen previously in Fig. 1.2

Search Query

Eggs	Spam
0	1

Fig. 1.4 - The Vectorized Search Query

Next, the search engine will use the index to locate the documents that contain the word “Spam”.

Detail Breakdown: In its most simplistic form, the vector search engine will plot the words in its vocabulary i.e. “Spam” and “Eggs” on the axis of a graph. Since the 3rd document in Fig. 1.3 does not contain the search query “**Spam**”, the search engine can safely ignore it:

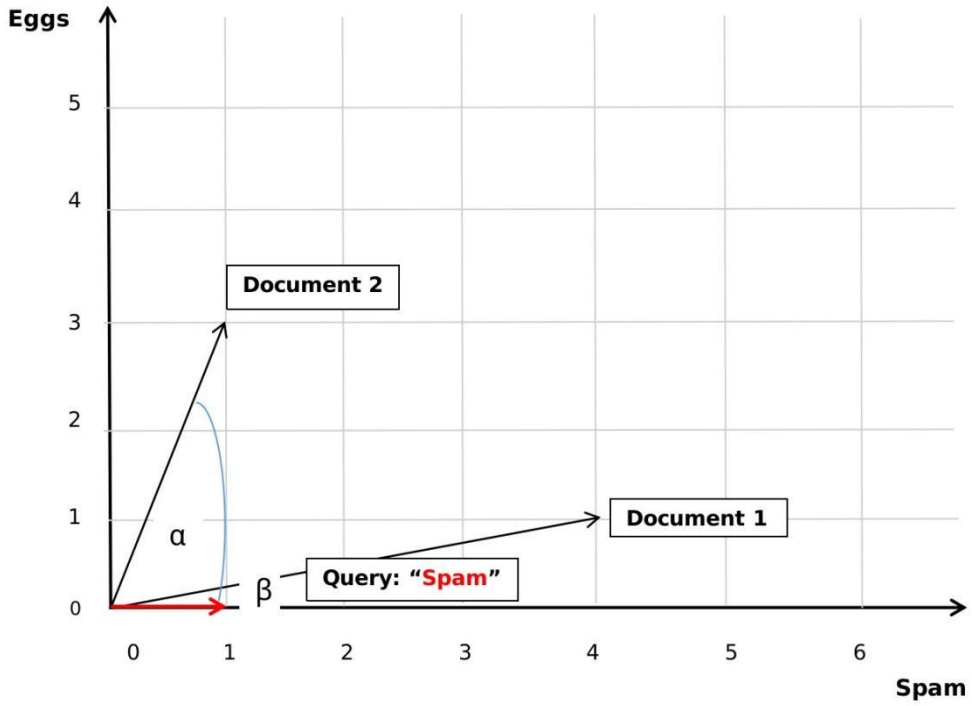

Fig. 1.5 - The Vector Similarity Search

The vector search engine will then measure the angular difference on the graph between the search query vector and each document, denoted by α and β and will return the document with the smallest difference which in this case is Document 1, which has 4 occurrences of “Spam”, as it is more relevant.

This principle can be extended to scenarios where we have documents with more more than two words. As each additional unique word will be added to the vocabulary, a new axis added to the graph and a vector similarity evaluated with the most relevant document returned.

Note: The vector search engine has no limit to the number of axes it can plot on a graph. And at the end of a Vectorization process, similar objects will lie in close to each other on the graph, denoting similarity and relevance.

ELI5: You can compare the process of how a Vector Search engine Vectorizes data to the "**Planogram**" of a supermarket.

A planogram is a diagram that shows how and where specific retail products should be placed on retail shelves or displays in order to increase customer purchases.

If a supermarket were a vector search engine, it would arrange items like "Apples", "Oranges", "Pineapples" on the same aisle because they are "Fruits", and put "Beer", "Booze", "Champagne" on another aisle because they are "Alcohol", while items like "Bacon", "Beef", "Barbecue sauce", would be closer to the "Alcohol" aisle than to the fruit aisle. "Because, what goes better with Beer than beef?"

Its a more efficient arrangement method than if the supermarket was a full-text search search engine which would arrange items alphabetically as: "Apples", "Bacon", "Barbecue sauce", "Beef", "Beer", "Booze", "Champagne", "Oranges" and "Pineapples" on the same aisle.

Now that we understand how a vector search engine works under the hood, lets get our hands dirty working with Weaviate.

What is Weaviate?

Weaviate is an open-source, cloud-native, modular, real-time vector search engine and database where data is stored as vectors in a high-dimensional space.

Weaviate can be used for semantic, image and similarity search, insight derivation, relationship prediction, power recommendation engines, e-commerce search, cybersecurity threat analysis, automated data harmonization, anomaly detection etc.

Before Weaviate can store data, it has to have an idea of how the data is structured. And this structure is known as "**Schema**". For people with relational database experience, schemas in Weaviate are the same thing as

SQL database schemas. You will have to define the schema before you can add data to or query the Weaviate database.

Weaviate Schema

A Weaviate schema is a simple concept that consists of a class, an optional description and at least one property and relationships (which are also properties) between the data objects.

Schemas are very important part of Weaviate as it uses the data schema to automatically place data objects into context and derive meaning.

How to define a Schema

Like we already stated previously, a schema consists of a class, an optional description and at least one property.

A class defines a data object as a noun or verb (in PascalCase notation), while properties (denoted in CamelCase) defines the data objects in each class.

A property consists of at least a “name” and “dataType” with and optional “description”.

The dataType describes the type of data that can be stored in the property. These types include:

string	string
string[]	list of strings
int	int64 (*)
int[]	list of int64 (*)
boolean	boolean
boolean[]	list of booleans
number	float64
number[]	list of float64
date	string
date[]	list of string
text	text
text[]	list of texts
geoCoordinates	string
phoneNumber	string
blob	base64 encoded string
cross reference	string

Also, you can describe a relationship between data objects as a property with the Class name of data object declared as the value of the property.

Defining the Data Schemas

We will be defining two schemas, “BlogPost” and “Author”. The Author schema will have two properties: “name” and “wrotePosts”.

Property	**dataType**
name	string
wrotePosts	BlogPost

```
{
            "class": "Author",
            "description": "The name of blog post author",
            "properties": [
            {
                "name": "name",
                "description": "Authors name",
                "dataType": ["string"]
            },
            {
                “name”: “wrotePosts”,
                “description”: “Posts written by the author”,
                “dataType”: ["BlogPost”]
            }
            ]
}
```

While the BlogPost will have the following properties: “title”, “body”, and “authoredBy”.

Property	dataType
title	string
body	text
authoredBy	string

```
{
		"class": "BlogPost",
		"description": "A blog post",
		"properties": [{
			"name": "title",
			"description": "The blog post title",
			"dataType": ["string"]
		},
		{
			"name": "body",
			"description": "The body content of the blog post",
			"dataType": ["text"]
		},
		{
			"name": "authoredBy",
			"description": "The author of the blog post",
			"dataType": ["Author"]
		}
		]
}
```

Uploading the Data Schemas

We will use the Weaviate Python client to upload the schema to the running Weaviate instance on port 8080:

```
>>> import weaviate
>>> client = weaviate.Client(“http://localhost:8080”)
>>> schema = {
    “classes”: [
                {
                    "class": "Author",
                    "description": "The name of blog post author",
                    "properties": [{
                        "name": "name",
                        "description": "Authors name",
                        "dataType": ["string"]
                    },
                    {
                        “name”: “wrotePosts”,
                        “description”: “Posts written by the author”,
                        “dataType”: [“BlogPost”]
                    }]
                },
                {
                    "class": "BlogPost",
                    "description": "A blog post",
                    "properties": [{
                            "name": "title",
                            "description": "The blog post title",
                            "dataType": ["string"]
                        },
                        {
                            "name": "body",
                            "description": "The body content of the blog post",
                            "dataType": ["text"]
                        },
                        {
                            "name": "authoredBy",
                            "description": "The author of the blog post",
                            "dataType": ["Author"]
                        }
                    ]
                }
  ]
}

>>> client.schema.create(schema)
```

Now that we have uploaded our data schema to Weaviate, the next step is to upload the search data.

Importing Data

Weaviate offers two modes for users to import data: Single-mode and Batch-mode.

In single-mode, you can get, add, update, and delete individual data objects to and from a Weaviate instance. While in batch-mode, you can upload a lot of data objects in bulk.

The advantages of batch over single mode are pretty obvious:
1. **Speed**: It is more performative sending a single bulk upload than multiple single uploads for a large data set.
2. **Time**: Since bulk operations are faster, you will invariably spend less time uploading data.

These factors should guide your choice of operation, however, I will walk through both methods.

The image below shows the signature of a data_object:

```
{
  “class”: <class name>,
  “id”: <uuid>,
  “properties”: {
   “<property-name>”: “<property-value>”,
  }
}
```

Note: All fields except the “id” field are required. Weaviate will generate an id if its not provided.

Lets now add some records using the Single-mode method:

```
>>> import weaviate

>>> from datetime import datetime, timezone

>>> client = weaviate.Client("http://localhost:8080")

>>> # Create a blog post data_object

>>> blog_post_data = {
      "title": "The AI machines",
       "body": "The AI machines are coming. The is not a drill, the machines are coming!",
}

>>> blog_post_uuid = client.data_object.create(
       data_object = blog_post_data,
      class_name = "BlogPost"
)

>>> # Now lets create the author and the cross references between the blog post and
the author.

>>> # Create an author data_object

>>> author_data = {
      "name": "Ben Smith"
}

>>> author_uuid = client.data_object.create(
      data_object = author_data,
      class_name = "Author"
)

>>> # Create the cross references

>>> client.data_object.reference.add(
           from_uuid = author_uuid,
           from_class_name = "Author",
           from_property_name="wrotePosts",
           to_uuid = blog_post_uuid,
           to_class_name = "BlogPost")

>>> client.data_object.reference.add(
             from_uuid = blog_post_uuid,
             from_class_name = "BlogPost",
             from_property_name = "authoredBy",
             to_uuid = author_uuid,
             to_class_name = "Author")
```

Now lets use the batch-mode to create multiple data_objects.

```
>>> import weaviate
>>> import pandas as pd
>>> from weaviate.util import generate_uuid5

>>> client = weaviate.Client(“http://localhost:8080”)

>>> client.batch.configure(
          batch_size=100,
          dynamic=False,
          timeout_retries=3,
          callback=None )

>>> def add_blog_post(batch, blog_post_data):
         blog_post_object = {
         'title': blog_post_data['title'],
         'body': blog_post_data['body'].replace('\n', '')
         }
         blog_post_uuid = blog_post_data[“id”]
         # add article to the object batch request
         batch.add_data_object(
         data_object=article_object,
         class_name='Article',
         uuid=blog_post_uuid)
         return blog_post_uuid

>>> def add_author(batch, name, created_authors):
          if name in created_authors:
               return created_authors[name]
          author_uuid = generate_uuid5(author)

          batch.add_data_object(
          data_object = {“name”: name},
          uuid=author_uuid,
          class_name = “Author”)
          created_authors[name] = author_uuid
          return author_uuid

>>> def add_cross_references(batch, blog_post_uuid,author_uuid):
          batch.add_reference(
          from_object_uuid = author_uuid,
          from_object_class_name = “Author”,
          from_property_name = “wrotePosts”,
          to_object_uuid = blog_post_uuid)

          batch.add_reference(
          from_object_uuid = blog_post_uuid,
          from_object_class_name = “BlogPost”,
          from_property_name = “authoredBy”,
          to_object_uuid = author_uuid)
```

Searching and Retrieving Data

Now that we've imported the data, let's query it by using the query attribute of the client object.

```
>>> import weaviate
>>> from pprint import pprint
>>> client = weaviate.client("http://localhost:8080")
>>> # Lets query the BlogPost data_object
>>> query = """
        {
           Get {
             BlogPost {
               title
               body
               authoredBy {
                 ... on Author {
                   name
                 }
               }
             }
           }
         }
    """

>>>  result = client.query.raw(query)
>>>  pprint(result)

      {'data': {'Get': {'BlogPost': [{'authoredBy': [{'name': 'Ben Smith'}],
           'body': 'The AI machines are coming. The is not a drill, the machines
are coming!',
           'title': 'The AI machines'}]}}}

>>> # Lets query the Author class
>>> query = """
        Get {
          Author {
            name
            wrotePosts {
             ... on BlogPost {
               title
               body
             }
           }
         }
        }
    """
>>> result = client.query.raw(query)
>>> result
{'data': {'Get': {'Author': [{'name': 'Ben Smith',
      'wrotePosts': [{'body': 'The AI machines are coming. The is not a drill, the
machines are coming!',
         'title': 'The AI machines'}]}]}}}
```

There other things you can do with Weaviate like:
1. Filtering
2. Aggregation
3. Semantic search

See **here** for a more detailed introduction.

Conclusion

This tutorial was a quick introduction to Vector search engines in general and Weaviate in particular. We spoke about the history of search engines, the shortcomings of traditional search engines which lead to the creation of Vector search. We then worked through the main functionalities of Weaviate from declaring the data schema, to uploading the schema and importing data. We lastly were introduced to searching and retrieving data from Weaviate.

I will suggest you go through the **developer documentation** and **API reference** to discover more functionality not covered in this tutorial. Also, its accompanying source code can be found **here**.