University of Modena and Reggio Emilia

Department of Engineering Enzo Ferrari

Master's Degree Course in
Computer Engineering

Experimental Analysis of Graph RAG

Thesis supervisor:
Prof. Giovanni Simonini

Candidate:
Bilel Arfaoui

Academic Year 2023/2024

«*No.* »

Hamlet, Act III, Scene III, Line 87

Abstract

This thesis explores Vector Database Management Systems (VDBMS) as a solution for handling unstructured data by representing them as vector embeddings. We examine the creation of embeddings, compare sparse and dense vector representations, and analyze state-of-the-art indexing techniques such as HNSW and IVF. Additionally, we discuss Approximate Nearest Neighbors (ANN) search and its role in scalable querying, along with filtering methods to enhance retrieval accuracy.

Beyond querying, we investigate Multi-Modal search, which integrates multiple data types (text, images, and audio) within a unified vector space. We also explore Retrieval-Augmented Generation (RAG), focusing on optimizing retrieval strategies to improve Large Language Model (LLM) responses. Our final experiment compares different RAG techniques, including knowledge graph integration, to refine retrieved contexts and enhance response reliability.

By evaluating various VDBMS solutions and retrieval methodologies, this thesis aims to provide insights into the evolving landscape of vector-based data management and its applications in modern AI-driven systems.

Contents

List of Figures

Introduction

The exponential growth of unstructured data, such as text, images, and audio, has outpaced the capabilities of traditional relational databases, which are optimized for structured records and ACID compliance. While conventional databases excel at transactional consistency, they struggle with efficient retrieval and similarity search across high-dimensional data spaces. This challenge has led to the rise of **Vector Database Management Systems (VDBMS)**, which represent data as vector embeddings, enabling powerful search and retrieval mechanisms based on similarity rather than exact matching.

VDBMS leverage Approximate Nearest Neighbors (ANN) search to efficiently query large-scale datasets, balancing speed and accuracy. Unlike traditional indexing methods, vector databases employ structures such as **Hierarchical Navigable Small World (HNSW) graphs** and **Inverted File Index (IVF)** to optimize search performance. Additionally, quantization techniques reduce memory and storage footprints while preserving retrieval quality.

Beyond retrieval, the application of VDBMS extends to **Multi-Modal search**, where different data types—such as images, text, and audio—are mapped into a unified vector space for seamless cross-modal querying. Furthermore, the integration of VDBMS with **Retrieval-Augmented Generation (RAG)** has gained traction, enhancing Large Language Model (LLM) responses by retrieving relevant context from vectorized data sources.

This thesis explores the architecture, indexing techniques, and retrieval strategies of modern VDBMS. It evaluates leading implementations such as Weaviate, FAISS, and Milvus, comparing their performance in large-scale search applications. Additionally, we investigate the role of vector databases in **Agentic RAG**, examining strategies to mitigate issues improve retrieval precision through knowledge graphs and optimized prompting techniques.

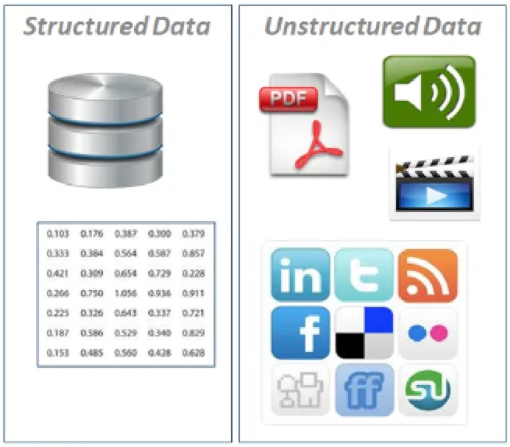
This work explores the capabilities and limitations of Vector Database Management Systems (VDBMS) to provide a comprehensive understanding of their impact on AI-driven search and retrieval. We highlight their role as a foundational technology in modern information systems. Finally, we introduce and analyze GraphRAG, a novel retrieval technique, comparing it to vector-based RAG. We examine their respective applications, advantages, and limitations, illustrating different use cases. As a final experiment, we propose an architecture for join discovery, leveraging knowledge graphs as a knowledge base to link entities and tables, enabling users to query structured data using natural language prompts.

1. Vector Representation of Data

In this chapter, we will explore the field of unstructured data, discussing how it differs from the traditional structured approach and how it plays a crucial role in modern applications. We will examine the process of converting unstructured data into vectorized representation and what it entails. Additionally we will introduce Vector Database Management Systems, exploring their architecture, functionalities and their impact on the market.

1.1 Unstructured Data vs. Structured Data

In the context of data management, data can be classified as structured or unstructured. Structured data refers to information arranged in rows and columns, typically found in relational databases. These follow a defined schema and are used to store records or transactions within a database environment. Meanwhile, unstructured data lack predefined formats and cannot be easily arranged into columns. Examples include text, audio, images, documents, and videos. This characteristic makes efficient indexing, organization, and retrieval challenging.



0.103	0.176	0.387	0.300	0.379
0.333	0.384	0.564	0.587	0.857
0.421	0.309	0.654	0.729	0.228
0.266	0.750	1.056	0.936	0.911
0.225	0.326	0.643	0.337	0.721
0.187	0.586	0.529	0.340	0.829
0.153	0.485	0.560	0.428	0.628

Figure 1: Structured vs. Unstructured data. Source: DeepLearning.AI. [1].

Today's digital services and applications generate vast amounts of data, whose scale suggests that valuable insights can be extracted. If organized effectively, such data can significantly aid stakeholders in long-term decision-making processes.

[1] https://www.deeplearning.ai/short-courses/building-multimodal-search-and-rag/

1.2 From Raw Data to Vector Representation

A vector embedding is a numerical representation of data (such as images, documents, and audio) that encapsulates meaning, characteristics, and associations. This representation enables unstructured data to be interpreted by computational systems, unlocking numerous use cases and applications. The embedding process maps objects and entities into a vector space, where dimensionality is determined by the encoding model.

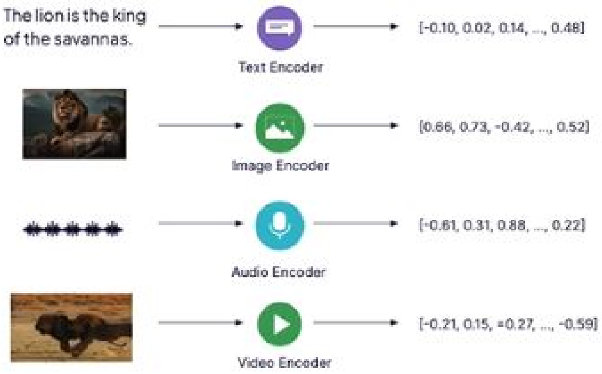


Figure 2: Encoding process. Source: DeepLearning.AI.[2].

Thanks to this structure, machine learning techniques can analyze unstructured data, uncover patterns, and generate insights. These techniques are widely applied in fields such as Natural Language Processing (NLP), Computer Vision, Recommendation Systems, and Search Engines. Each application comes with its own challenges. Another key advantage of mathematical vector representations is their ability to facilitate computational efficiency and scalability in data processing.

1.3 Dense vs. Sparse Vectors

It is important to distinguish between two types of vector embeddings: dense and sparse. Dense embeddings have lower dimensionality, with each value carrying meaningful information, while sparse embeddings have high dimensionality but contain only a few non-zero values. Typically, dense vectors are obtained through deep learning techniques or by applying dimensionality reduction to sparse vectors. Each representation has its own strengths, limitations, and use cases, and they can also be effectively combined in certain applications.

[2] `https://www.deeplearning.ai/short-courses/building-multimodal-search-and-rag/`

1.3.0.1 Sparse Vectors

Sparse embeddings are commonly used in NLP and recommender systems, where data often has high dimensionality but each instance activates only a few features. In a typical sparse vector, each dimension represents a token, and the value indicates that token's importance within a document. This makes sparse vectors particularly advantageous for applications requiring keyword matching. Several methods exist for creating sparse vectors:

- **Bag of Words (BOW)**: Simple and interpretable, but ignores word order and context.
- **One-Hot Encoding**: Clearly maps each token but results in extremely high dimensionality with large vocabularies.
- **TF-IDF**: Highlights unique terms and relevance within a document but does not capture semantic relationships.
- **BM25**: Extends TF-IDF by considering term frequency and document length but requires full corpus statistics in advance.
- **Sparse Neural Embeddings**: Uses neural networks for efficient keyword matching on large datasets, though often less interpretable than traditional methods.

However, sparse vectors struggle to capture nuanced relationships between words. Each dimension may correspond to a word or subword grouping, which aids in interpreting document rankings. Sparse vectors are particularly useful in scenarios with rare keywords or specialized terms that may not appear in standard vocabularies, where general-purpose embeddings might fail to convey the intended meaning. They are highly effective for text search and hybrid search applications.

1.3.0.2 Dense Vectors

Dense embeddings are widely used in NLP and machine learning to capture complex relationships within data. Unlike sparse vectors, dense embeddings represent rich, context-sensitive information in lower-dimensional space, making them ideal for tasks such as semantic search and sentence similarity. While they excel at capturing nuanced meanings, dense embeddings can be computationally intensive and less interpretable. Deep learning techniques are typically used to generate these embeddings, learning from large datasets to uncover intricate patterns. In many cases, dense and sparse vectors can be combined to take advantage of both.

1.4 Vector Database Architecture

Vector databases share several core components that enable efficient storage, retrieval, and querying of vector embeddings:

- **Client Interface**: Provides API or GUI access for querying and managing data.
- **Query Processor**: Handles similarity searches using distance metrics and ranking mechanisms.
- **Vector Storage**: Stores vector embeddings and associated metadata.
- **Indexing Module**: Manages indexing structures to enable efficient retrieval.
- **Embedding Generation Module**: Converts raw unstructured data (e.g., text, images, audio) into high-dimensional vector embeddings using traditional or deep learning techniques.

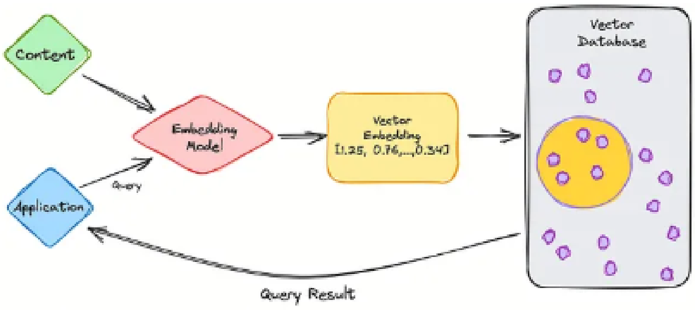


Figure 3: Architecture. Source: Medium.[3].

1.4.1 Querying and Search Efficiency

Vector databases balance speed and accuracy using Approximate Nearest Neighbor (ANN) search techniques. These approaches enable efficient retrieval without performing exhaustive comparisons, making them well-suited for large-scale datasets.

1.4.2 Scalability Considerations

Handling high-dimensional data efficiently requires scalable solutions. VectorDBs achieve scalability through distributed architectures, memory-efficient indexing, and cloud-based deployments. These capabilities make them viable for both on-premise and large-scale cloud environments.

[3] https://medium.com/data-and-beyond/vector-databases-a-beginners-guide-b050cbbe9ca0

1.5 Multimodal Vector Space

One key application of vector DBMSs is the mapping of data in Multimodal Vector Spaces, which refers to the storage and retrieval of data across multiple modalities, such as text, images, audio and video, in a unified embedding space. This means that conceptually related items, regardless of their format (e.g., an image of a lion and the phrase "the king of the jungle"), are placed close to each other in this space.

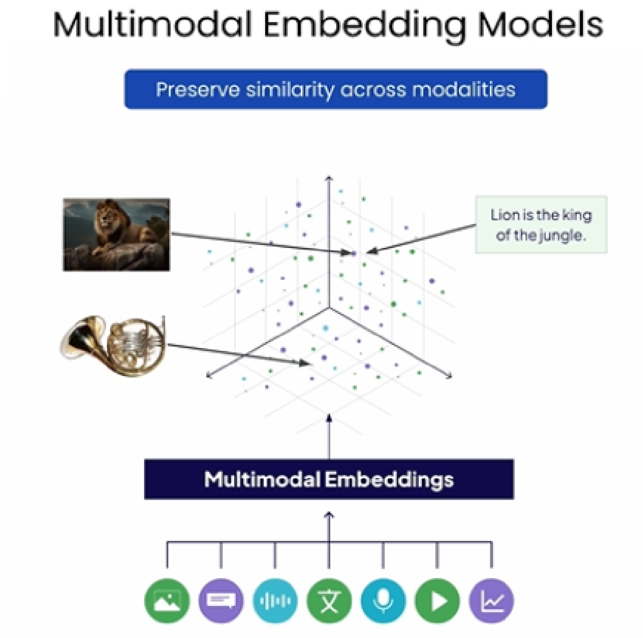

Multimodal Embedding Models
Preserve similarity across modalities
Lion is the king of the jungle.
Multimodal Embeddings

Figure 4: Multimodal Vector Space.

Source: `https://www.deeplearning.ai/short-courses/building-multimodal-search-and-rag/`

The key motivation for multimodal data comes from the fact that humans understand the world, not through a single mode like text alone, but through a combination of sensory inputs. For example, hearing the sound of a cash register can immediately evoke the concept of a store or a purchase. By integrating multiple modalities, we aim to develop systems capable of learning and reasoning across diverse data formats.

1.5.1 Constructing the Multimodal Vector Space

To achieve this, we employ specialized encoders that learn representations for each modality separately and then align them through contrastive learning. The process of constructing a Multimodal vector space consists of the following steps:

1. Train individual embedding models for each modality (e.g., an image encoder for images,

a text encoder for text).

2. Align these models in a shared vector space.
3. Optimize the embeddings to ensure cross-modal similarity.
4. Use contrastive learning to fine-tune the models by distinguishing between positive and negative examples.

1.5.2 Applications of Multimodal Retrieval

Multimodal retrieval has various real-world applications, including:

- **Image-Text Search**: Finding relevant images based on textual queries (e.g., retrieving an image of the Eiffel Tower when searching "Paris landmark").
- **Video Retrieval**: Searching for specific moments in videos using natural language queries.
- **Audio-Visual Understanding**: Associating spoken words with corresponding visual elements in video content.
- **Medical Imaging**: Aligning radiology reports with corresponding X-ray images to enhance diagnostics.

1.6 Key Products and Features

The following are leading vector database solutions available in the market:

- **Milvus:** Open-source, distributed database optimized for scalability and GPU acceleration, widely used in AI analytics and recommendation systems.
- **Pinecone:** Managed service with real-time indexing, automatic scaling, and seamless ML integration, ideal for semantic search and anomaly detection.
- **QDrant:** High-performance, open-source vector search engine supporting hybrid search (vector + metadata), multimodality and efficient memory management.
- **Weaviate:** Similar to QDrant offering flexible filtering and retrieval strategies.

2. Vectorization Techniques

In this chapter, we will explore vectorization techniques in depth, beginning with traditional methods such as One-Hot Encoding, Bag of Words, and TF-IDF. We will then transition into fundamental Deep Learning principles, which lay the groundwork for advanced representations, including word embeddings and transformer-based models, enabling richer, context-aware representations. Furthermore Deep Learning principles will serve as context for advanced applications of Vector Databases in the following chapters.

2.1 Traditional Vectorization Techniques

Traditional vectorization techniques transform textual data into numerical representations, enabling computational processing and analysis. These methods primarily rely on statistical properties of text rather than learned representations from deep learning models. This section explores widely used traditional techniques, including One-Hot Encoding (OHE), Bag of Words (BoW), Term Frequency-Inverse Document Frequency (TF-IDF).

2.1.1 Tokenization

Tokenization is a crucial preprocessing step for both traditional and deep learning-based vectorization. It involves breaking text into smaller units (tokens), with different approaches offering various trade-offs:

- **Byte-level tokenization**: Operates on a small vocabulary, encoding text at the byte level.
- **Word-based tokenization**: Uses full words as tokens, preserving meaning but struggling with unseen words due to a large vocabulary.
- **Subword tokenization**: A compromise between the two, capturing word roots (e.g., NLP, lemmatization) while remaining trainable and statistically driven.

2.1.2 One-Hot Encoding (OHE)

One-Hot Encoding (OHE) is the simplest vectorization technique, representing each unique word in a vocabulary as a binary vector. Given a vocabulary of size V, each word w_i is mapped to a vector of length V where only the corresponding index is set to 1, while all other elements remain 0.

For example, given the vocabulary:

$$\{\text{apple}, \text{banana}, \text{cherry}\}$$

the one-hot encoded vectors are:

$$\text{apple} \to [1, 0, 0], \quad \text{banana} \to [0, 1, 0], \quad \text{cherry} \to [0, 0, 1]$$

Mathematically, for a given word w_i in a vocabulary of size V, its one-hot representation $\mathbf{v}_{w_i}$ is:

$$\mathbf{v}_{w_i} = [0, 0, \dots, 1, \dots, 0, 0] \tag{2.1}$$

where the position of 1 corresponds to the index of the word in the vocabulary.

While simple and interpretable, one-hot encoding suffers from high dimensionality and sparsity, making it inefficient for large vocabularies. Additionally, it does not capture semantic relationships between words (e.g., "cat" and "dog" are equally distant from "fish" in this representation).

2.1.3 Bag of Words (BoW)

The Bag of Words model extends OHE by considering word frequencies in a document rather than binary presence. It represents a document by counting the occurrences of words, ignoring word order and syntax. Formally, let D be a collection of documents and V the vocabulary of unique words across all documents. The BoW representation of a document d is a vector:

$$\mathbf{v}_d = (f_1, f_2, \dots, f_n) \tag{2.2}$$

where f_i is the frequency of word w_i in the document d. While BoW is effective in some applications, it suffers from high dimensionality and lacks semantic understanding.

2.1.4 Term Frequency-Inverse Document Frequency (TF-IDF)

TF-IDF improves upon the BoW model by assigning importance to words based on their frequency across multiple documents. The TF-IDF value for a word w in a document d is given by:

$$\text{TF-IDF}(w, d) = \text{TF}(w, d) \times \text{IDF}(w) \tag{2.3}$$

where:

- **Term Frequency (TF)** measures how frequently a word appears in a document:

  $$\text{TF}(w, d) = \frac{f_w}{\sum_{w' \in d} f_{w'}} \tag{2.4}$$

  where f_w is the count of word w in document d.

- **Inverse Document Frequency (IDF)** reduces the weight of common words appearing in many documents:

  $$\text{IDF}(w) = \log\left(\frac{N}{1 + \text{DF}(w)}\right) \tag{2.5}$$

  where N is the total number of documents, and $\text{DF}(w)$ is the number of documents containing w.

TF-IDF effectively emphasizes informative words while downplaying common ones, making it useful for tasks like document classification and search retrieval.

2.2 Deep Learning

The conversion of unstructured data to vector embedding is an automatic process that leverages Deep Learning techniques by using Neural Networks, which is essentially a computational model inspired by how the human brain processes information, in a nutshell Neural Networks are composed of layers of interconnected "neurons" that perform calculations on the input data.

2.2.1 Neuron

The "Neuron" is the most basic computational unit of the Neural Network, each neuron takes multiple inputs, each with an associated weight (which indicates the importance), computes their weighted sum, adds a constant called bias, and then passes this result through an activation function, which generates the neuron's output.

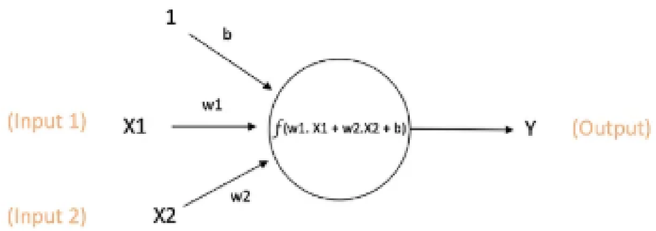


Figure 5: Neuron Source.[1].

The most popular and widely known activation functions in neural networks are:

- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Tanh**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- **ReLU**: $f(x) = \max(0, x)$
- **Softmax**: $\sigma(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

As said before the neural networks is has an architecture that is composed by layers each one with its neurons, they are the following:

- **Input layer**: Which is responsible to receive the input data.
- **Output Layer**: Responsible for producing the final result.
- **Hidden Layer/s**:One or more layers situated between the input and output layers of a neural network, they process and transform the data to help the network learn complex patterns.

The number of neurons in each layer and the way they are interconnected depend entirely on

[1] `https://medium.com/analytics-vidhya/neural-networks-in-a-nutshell-bb013f40197d`

the network's design, which is tailored to suit the specific use case.

2.2.2 Training process

Once the neural network architecture is defined, it needs to be trained, which is an iterative process where each iteration called "epoch" is composed of three main steps: Forward propagation, calculation of loss, Backward propagation. The forward propagation starts with initializing the weights and bias (non zero number) and the calculation of the output is done. Then the network's output is compared to the expected result using a loss function. After calculating the loss function, the last step of the iteration is Backward propagation, which through an optimizer adjusts the parameters (weights and biases). The dataset used for training is typically employed in batches, doing so the network trains on each batch on each epoch.

2.3 Autoencoders

One of the architectures used to create vector embeddings leveraging neural networks, is the Autoencoder, a type of neural network with three main parts: encoder layers (which compress the input), a bottleneck layer (where the compact embedding is stored), and decoder layers (which try to rebuild the original input). Basically the encoder layers gradually reduce the number of neurons, compressing the data until it reaches the bottleneck layer. The decoder layers then expand the data again to reconstruct the original input. The vector embedding of the raw data corresponds to the output of the bottleneck layer, and its dimensionality is determined by the number of neurons in that layer. Since the network compresses and decompresses the data, some information is lost in the process, but with proper training the weights will be adjusted to minimize the loss.

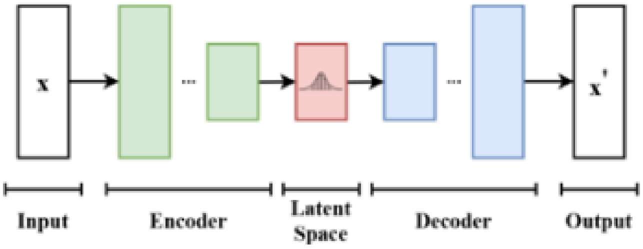


Figure 6: Autoencoder Source.[2].

[2] https://www.deeplearning.ai/short-courses/building-multimodal-search-and-rag/

2.3.1 Example with the MINST dataset

As an example, we use the MNIST dataset, which consists of images of handwritten digits. The neural network architecture, shown in the figure below, is an autoencoder with two layers in both the encoder and decoder. The encoder progressively compresses the data from 256 dimensions to 128 and then to 2 dimensions in the bottleneck layer, while the decoder reverses this process to reconstruct the original input. The network uses dense layers, meaning each neuron is fully connected to the neurons in the following layer.

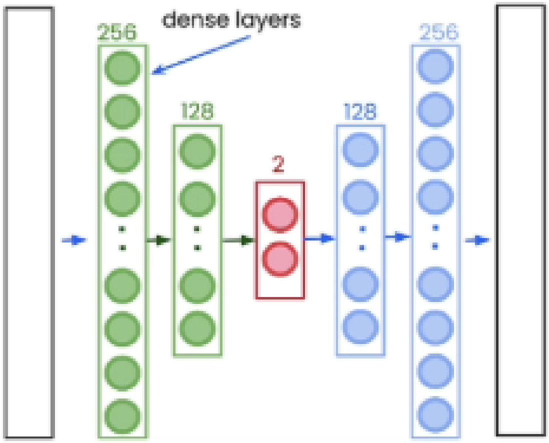


Figure 7: example of AutoEncoder dimensions: Source.[3]

For simplicity and clarity, the images are represented as 2-dimensional vectors, as shown in the plot below.

As expected, some information is lost during the compression process, as the input and output images are not identical. However, by adjusting the weights, the network can minimize this loss and produce outputs closer to the original.

In the context of VectorDBs, the encoder is the more relevant component, as it outputs the embeddings needed for database operations. Embeddings can be produced by pre-trained or use-case specific models in order to capture insights. As shown above, it's easy for a person to recognize that the first two images represent the same entity. Thanks to the mathematical properties of embeddings, we can make this relationship machine-understandable. By using a similarity function, the system can also determine which pairs of images are more similar

[3] `https://www.deeplearning.ai/short-courses/building-multimodal-search-and-rag/`

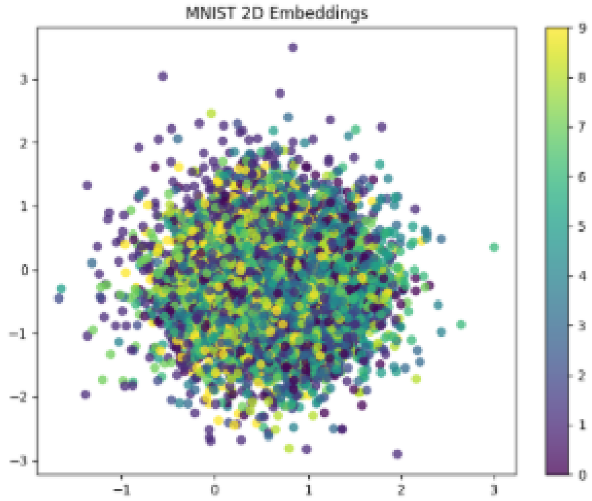


Figure 8: MNIST mapped in 2D: Source.[3]

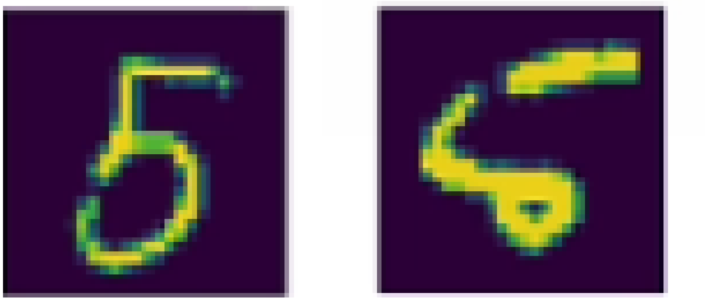


Figure 9: Before and After encoding: Source.[3]

2.4 Evolution to Context aware techniques

While autoencoders provide a compressed representation of the data, and can be trained in an unsupervised way, they still require a training dataset and do not capture contextual relationships. For these reasons, next step mechanisms of that capture the contextual dependencies have been developed such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory Networks (LSTMs), which we will briefly show their functionalities and how they paved the way to Transformers.

2.4.1 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) introduced the idea of modeling sequential dependencies by maintaining a hidden state h_t that evolves over time:

$$h_t = f(W_h h_{t-1} + W_x x_t + b), \tag{2.6}$$

where W_h and W_x are weight matrices, b is a bias term, and f is an activation function (e.g., tanh or ReLU). However, RNNs suffer from the *vanishing gradient problem*, meaning that the gradient the derivative of the loss function, which provides information information of the direction that the model is taking, becomes very small during backpropagation, making it difficult to learn long-range dependencies.

2.4.2 LSTM

LSTMs (Long Short-Term Memory networks) improve upon standard RNNs (Recurrent Neural Networks) by introducing a **memory cell** C_t and **gating mechanisms** that control the flow of information over time. This allows the network to "remember" important information for longer periods and "forget" irrelevant data. At each time step t, the LSTM takes the input x_t and the previous hidden state h_{t-1}, and updates the memory cell C_t and hidden state h_t. The LSTM consists of the following components:

- The **forget gate** f_t decides how much of the previous memory C_{t-1} should be kept.
- The **input gate** i_t controls how much new information $\tilde{C}_t$ should be added to the memory cell.
- The **output gate** o_t decides what part of the memory cell C_t should be output as the current hidden state h_t.

The memory cell is updated as follows:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

where $\tilde{C}_t$ is the candidate memory, computed from the input x_t and previous hidden state h_{t-1}. The hidden state is then computed as:

$$h_t = o_t \odot \tanh(C_t)$$

2.4.3 The Transformer Architecture

Transformers address LSTMs' scalability issues by processing entire sequences simultaneously using self-attention, the core mechanism of the architecture. Self-attention allows the model to interpret a word's meaning based on its context, even when the same word has different meanings. The self-attention mechanism computes attention scores as:

$$A = \frac{QK^T}{\sqrt{d_k}} \tag{2.7}$$

where are the query, key, and value matrices derived from input embeddings. Specifically:

- **Q**(Query): Represents the current word/token for which attention is being computed.
- **K**(Key): Represents all words/tokens in the sequence, determining relevance.
- **V**(Value): Contains the actual token representations used to construct the final weighted output.

These matrices are obtained by linearly transforming the input embeddings using learned weight matrices :

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V. \tag{2.8}$$

The attention scores are normalized using softmax and then we compute the final representation with V:

$$Z = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V. \tag{2.9}$$

Multi-head attention enhances this mechanism by capturing multiple representation aspects, each attention head processes the input independently with learned projections:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V),$$

where W_i^Q, W_i^K, and W_i^V are learned projection matrices for queries, keys, and values, respectively. The outputs of all h heads are concatenated and linearly transformed using another learned weight matrix W^O:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O.$$

Here, W^O ensures that the final output maintains the desired dimensionality for subsequent layers.

2.4.3.1 GPT: Autoregressive Transformer Models

The Generative Pre-trained Transformer (GPT) series, along with models like Mistral and LLaMA 2, follows an **autoregressive** architecture, predicting one token at a time based on prior tokens:

$$P(x_t|x_1, x_2, ..., x_{t-1}) \tag{2.10}$$

GPT uses **causal self-attention**, ensuring tokens only attend to past inputs, making it effective for text generation. It is trained unsupervised on vast text corpora using next-token prediction. During training, a prompt like *"Jack and Jill went over..."* leads the model to assign probabilities to possible next tokens. Training involves guiding initial tokens (e.g., forcing *"The Rock"* for *"The rock at first..."*) and continuing until reaching a token limit or an `[EOS]` token.

[3] `https://www.deeplearning.ai/short-courses/building-multimodal-search-and-rag/`

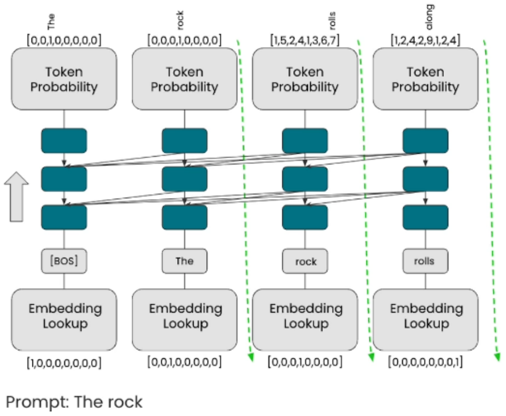


Figure 10: GPT model example. Source: DeepLearning.AI.[3].

2.4.4 Visual Transformers (ViTs)

Visual Transformers extend the principles of self-attention, initially designed for textual data, to computer vision tasks. A Vision Transformer (ViT) processes an image by dividing it into patches, embedding each patch, and applying a transformer model to classify the image based on its patch representations.

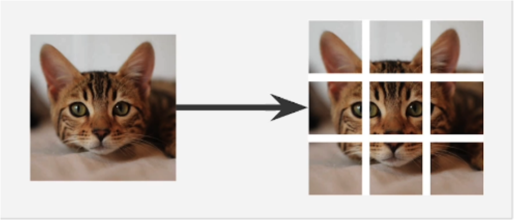


Figure 11: Image patching. Source: DeepLearning.AI.[4]

Key steps in ViT include:

- **Patch Embedding**: An input image of size $H \times W$ is divided into non-overlapping patches of size $P \times P$, flattened, and projected into an embedding space.
- **Positional Encoding**: Added to embeddings to retain spatial information.
- **Multi-Head Self-Attention**: Computes relationships across patches.
- **Classification Head**: Uses a CLS token for final classification.

[4] `https://www.deeplearning.ai/short-courses/building-multimodal-search-and-rag/`

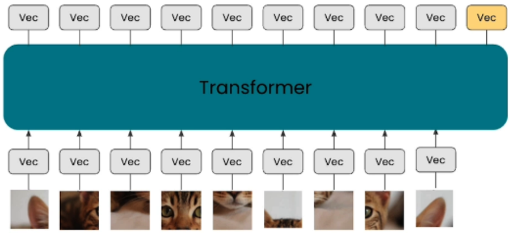


Figure 12: Vision Transformer process. Source: DeepLearning.AI.[4]

2.5 Training LLMs to See Images and Multimodality

Recent advancements in multimodal learning have enabled Large Language Models (LLMs) to not only process text but also understand and generate responses based on visual inputs. One key approach to achieve this is **Visual Instruction Tuning**. In this process, an LLM is trained to take both visual and textual inputs, allowing it to generate appropriate responses to prompts that involve both types of data. For instance, when given an image of a painting and a prompt like "Who drew this painting?", the model can correctly respond with "Vincent Van Gogh."

2.5.1 Multimodal Learning Process

Large Language Models can be trained to process visual and language input, the process starts with patching and embedding the images as shown before. Moreover, the text prompt is also embedded in a vector space. The model then processes these embeddings, representing both visual and textual information. The core of the model's training involves integrating these two types of embeddings—*visual embeddings* (from image patches) and *language embeddings* (from text). This allows the model to consider both image and text inputs when generating a response, making it capable of handling Multimodal requests. The resulting model can handle tasks that require both visual understanding and language generation, such as identifying objects in an image or answering questions about visual content.

2.5.2 Large Multimodal Models (LMMs)

Large Multimodal Models (LMMs) take this integration even further by combining multiple types of data—text, images, audio, and video—into a single model. These models extend the capabilities of traditional LLMs by processing and understanding a wider range of inputs, which enables more sophisticated, context-aware AI systems.

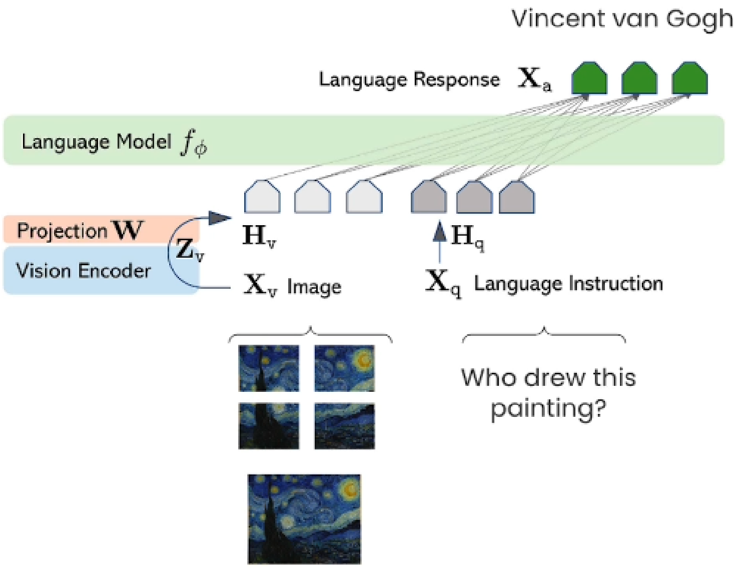


Figure 13: Text and Image input processed. Source: DeepLearning.AI.[4]

2.5.3 Key Components of Multimodal Learning

Multimodal learning is the process of combining and aligning different types of data to produce richer and more meaningful representations. The main components of LMMs include:

- **Modality-Specific Encoders**: Each modality (e.g., text, image, audio) is processed by a dedicated encoder. For example, in CLIP (Contrastive Language-Image Pretraining), distinct encoders handle the text and image inputs, learning a shared embedding space that links the two modalities.
- **Cross-Attention Mechanisms**: These mechanisms allow the model to exchange information between different modalities, enhancing the model's understanding of the data. For example, this helps the model connect visual features with corresponding textual descriptions.
- **Unified Representations**: The model fuses the multimodal inputs into a single vector space, facilitating tasks like generation and retrieval that require understanding and synthesizing information across modalities.

2.5.4 Contrastive Learning for Multimodal Embeddings

Contrastive learning plays a key role in aligning embeddings from different modalities. The goal is to learn a representation where similar data points (positive pairs) are drawn closer together, while dissimilar ones (negative pairs) are pushed further apart.

2.5.4.1 Contrastive Learning Procedure

The contrastive learning process follows these steps:

1. Define an anchor (e.g., an image of a cat).
2. Identify a positive example, which is semantically related to the anchor (e.g., the phrase "a small feline").
3. Identify a negative example, which is unrelated to the anchor (e.g., an image of a car).
4. Train the model to minimize the distance between the anchor and the positive example while maximizing the distance to the negative example.

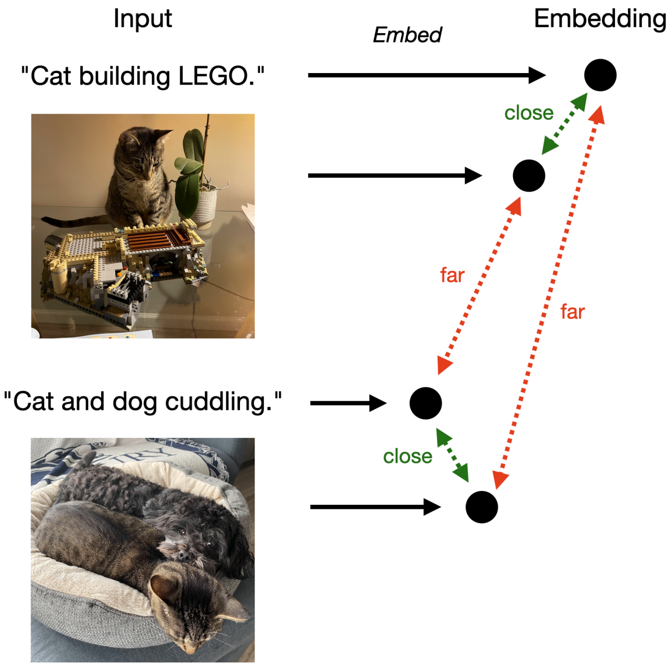


Figure 14: Illustration of Contrastive Learning. Source:v7labs[4].

2.5.4.2 Contrastive Loss Function

The contrastive loss function is formulated as follows:

$$\mathcal{L} = -\log \frac{\exp(\mathrm{sim}(q, k^+))}{\sum_{k \in K} \exp(\mathrm{sim}(q, k))} \tag{2.11}$$

[4] https://www.v7labs.com/blog/contrastive-learning-guide

where:

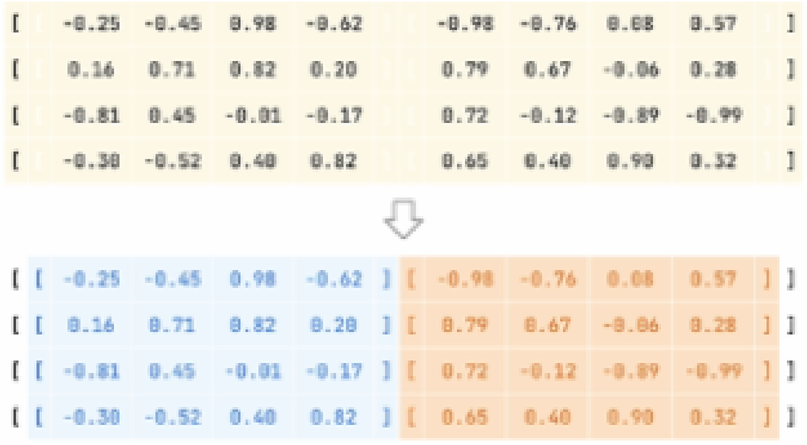
- q is the embedding of an anchor (e.g., an image encoded by a function f).
- k^+ is the embedding of a positive example (e.g., a corresponding text encoded by function g).
- K includes both the positive example and a set of negative examples k^-.
- $\text{sim}(q, k)$ denotes a similarity measure, such as cosine similarity.
- The negative log ensures that the similarity between q and k^+ is maximized while pushing away the negative examples k^-.

3. Quantization Techniques

This chapter explores **vector quantization** techniques, which arise from the need to optimize the memory and disk footprint of **VDBMS**. By reducing storage requirements, these techniques enable more efficient resource usage and higher **queries per second (QPS)**. However, this efficiency comes at the expense of search accuracy, introducing a trade-off between precision and performance. Quantization methods can be broadly categorized into two types: those that reduce dimensionality, such as **Product Quantization**, and those that approximate values, such as **Scalar Quantization** and **Binary Quantization**.

3.1 Product Quantization

Product Quantization (PQ) reduces memory usage by restricting vector representation, making storage and retrieval more efficient. The process begins by dividing the original vector into several smaller sub-vectors of equal size, and the greater the number of sub-vectors, the lower the compression rate, as more data points need to be stored. Each sub-vector represents a separate subspace, where we apply a clustering algorithm associated with its closest centroid.



```
[   -0.25  -0.45  0.98  -0.62     -0.98  -0.76  0.08   0.57    ]
[    0.16   0.71  0.82   0.20      0.79   0.67 -0.06   0.28    ]
[   -0.81   0.45 -0.01  -0.17      0.72  -0.12 -0.89  -0.99    ]
[   -0.30  -0.52  0.40   0.82      0.65   0.40  0.90   0.32    ]

[ [ -0.25  -0.45  0.98  -0.62 ] [ -0.98  -0.76  0.08   0.57  ] ]
[ [  0.16   0.71  0.82   0.20 ] [  0.79   0.67 -0.06   0.28  ] ]
[ [ -0.81   0.45 -0.01  -0.17 ] [  0.72  -0.12 -0.89  -0.99  ] ]
[ [ -0.30  -0.52  0.40   0.82 ] [  0.65   0.40  0.90   0.32  ] ]
```

Figure 15: Product Quantization. Source: DeepLearning.AI.[1]

Codebook: It refers to the collection of all centroids of a specific subspace.

[1] https://learn.deeplearning.ai/courses/retrieval-optimization-from-tokenization-to-vector-quantizati lesson/1/introduction

Formally, the **product quantization** process goes as follows:

1. Decomposing the vector $\mathbf{x}$ into subvectors k: $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k]$, where each subvector $\mathbf{x}_i \in \mathbb{R}^{d_i}$ and $\sum_{i=1}^{k} d_i = d$.
2. Quantizing each sub-vector $\mathbf{x}_i$ independently using a **codebook** $C_i = \{c_{i1}, c_{i2}, \ldots, c_{iM}\}$, where M is the number of centroids for each sub-vector.
3. The quantized vector is then the concatenation of the indices of the chosen codewords: $\mathbf{q} = [q_1, q_2, \ldots, q_k]$, where q_i is the index of the nearest codeword in C_i.

3.1.1 Centroid Generation

As for generating centroids, VDBMSs like Weaviate and Qdrant rely on the widely used encoder algorithm **K-means**. It defines a **K** parameter, which is the number of partitions (also referred to as tiles) we want for our subvector space. The algorithm starts by assigning **K** random centroids within the dataset, then iteratively assigns the closest vectors to them, recalculating the cluster using the mean of the residing vectors. The algorithm stops when the centroids have stabilized or a maximum number of iterations have been reached. Another algorithm used to partition the subvector space is the **Tile encoder**. This algorithm aims to resolve one of the main limitations of K-means, which is the tendency for centroids to become outdated over time as new vectors join the collection. This results in some centroids becoming more populated than others.

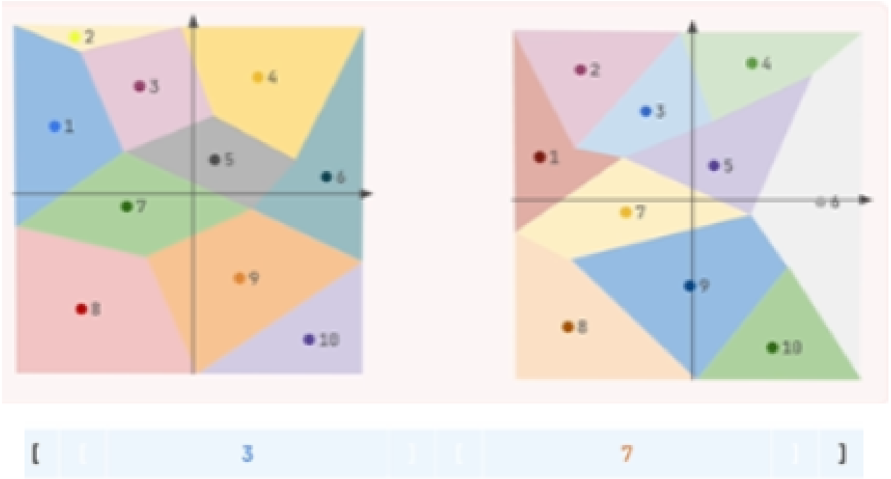
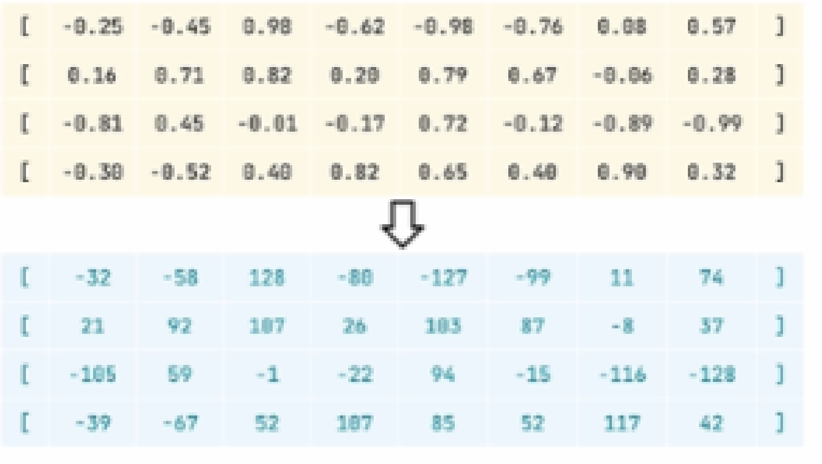


Figure 16: Partitioned Vector Space. Source: DeepLearning.AI.[1]

The Tile encoder addresses this issue by employing the Cumulative Density Function (CDF), denoted as **CDF**(x). Given our data collection, it returns the probability (a value from 0 to 1) that a data point is less than or equal to x. Using the formula code$(x) = \text{CDF}(x) \times c$ (where c is the number of codes/clusters we need), we determine the cluster to which the data point x belongs.

3.2 Scalar Quantization

Scalar Quantization reduces the precision of numerical values by mapping continuous floating-point numbers to a fixed set of discrete levels. Typically, this involves converting each dimension from a 32-bit floating-point representation to an 8-bit integer. The process begins by analyzing the data to determine the range, setting boundaries based on the minimum and maximum values from a training set. The range is then divided into a fixed number of intervals (e.g., 256 for 8-bit quantization), with each value assigned to the nearest bucket. The resulting integer represents the corresponding bucket, enabling a more compact and efficient representation while preserving as much information as possible within the given bit constraint.



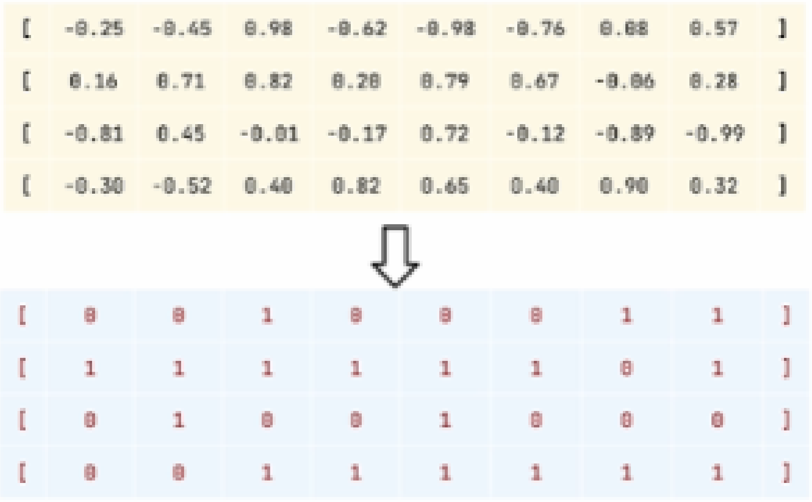
```
[  -0.25  -0.45   0.98  -0.62  -0.98  -0.76   0.08   0.57  ]
[   0.16   0.71   0.82   0.20   0.79   0.67  -0.06   0.28  ]
[  -0.81   0.45  -0.01  -0.17   0.72  -0.12  -0.89  -0.99  ]
[  -0.30  -0.52   0.40   0.82   0.65   0.40   0.90   0.32  ]

[   -32    -58    128    -80   -127    -99     11     74   ]
[    21     92    107     26    103     87     -8     37   ]
[  -105     59     -1    -22     94    -15   -116   -128   ]
[   -39    -67     52    107     85     52    117     42   ]
```

Figure 17: Scalar Quantization. Source: DeepLearning.AI.[1]

3.3 Binary Quantization

Binary Quantization consists of converting a floating point to a binary representation, taking the quantization to the extreme by transitioning from a 32-bit representation to a single bit. By sacrificing a huge amount of information and distance calculation accuracy, **Binary Quantization** excels by having a minor memory and disk footprint. This technique is very susceptible to **oversampling**(vectors with the same compressed representation), while being the fastest.



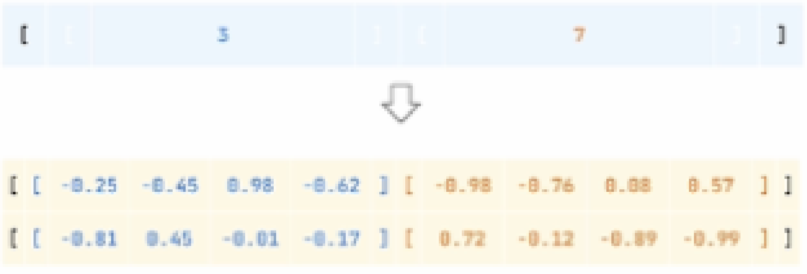
```
[ -0.25 -0.45  0.98 -0.62 -0.98 -0.76  0.08  0.57 ]
[  0.16  0.71  0.82  0.20  0.79  0.67 -0.06  0.28 ]
[ -0.81  0.45 -0.01 -0.17  0.72 -0.12 -0.89 -0.99 ]
[ -0.30 -0.52  0.40  0.82  0.65  0.40  0.90  0.32 ]

[ 0 0 1 0 0 0 1 1 ]
[ 1 1 1 1 1 1 0 1 ]
[ 0 1 0 0 1 0 0 0 ]
[ 0 0 1 1 1 1 1 1 ]
```

Figure 18: Binary Quantization. Source: DeepLearning.AI.[1]

3.4 Overfetching and Rescoring

To mitigate the limitations of vector quantization, vector database management systems (**VDBMSs**) employ **overfetching** and **rescoring**. Due to the lossy nature of quantization, multiple vectors may share the same compressed representation, leading to potential collisions. To address this, databases like Weaviate use **overfetching**, retrieving a larger set of candidates instead of selecting only the top-K results from the quantized index. This ensures that relevant matches are not overlooked. Once these compressed vectors are retrieved, **rescoring** is performed by fetching the original, high-precision vectors from disk. The query is then re-evaluated against this smaller subset, improving accuracy while maintaining search efficiency.



```
[ [ 3 ] [ 7 ] ]

[ [ -0.25 -0.45  0.98 -0.62 ] [ -0.98 -0.76  0.08  0.57 ] ]
[ [ -0.81  0.45 -0.01 -0.17 ] [  0.72 -0.12 -0.89 -0.99 ] ]
```

Figure 19: Example of vectors with the same quantized representation. Source: DeepLearning.AI.[1].

[1] https://learn.deeplearning.ai/courses/retrieval-optimization-from-tokenization-to-vector-quantizati
lesson/1/introduction

3.5 Memory Impact of Quantization in Vector Databases

Quantization optimizes memory usage in large-scale vector databases by reducing precision. Common techniques include:

- **Scalar Quantization (SQ):** Converts 32-bit floating-point values to 8-bit integers, reducing storage size by a factor of 4.
- **Product Quantization (PQ):** Splits vectors into subvectors, assigns each to a centroid, and stores them as small fixed-size codes (e.g., 1-byte indices). This significantly reduces memory usage but increases indexing complexity.
- **Binary Quantization:** Maps floating-point values to a single bit per value, maximizing compression but sacrificing accuracy.

By default, an **OpenAI embedding** with 1,536 dimensions requires approximately **6KB per vector**. Applying **SQ** reduces this to **1.5KB**, while **PQ** achieves **10x or greater compression**, balancing memory efficiency with computational overhead and potential precision loss. The choice of quantization technique depends on the trade-off between storage savings, search latency, and retrieval accuracy.

3.6 Experimentation with the quantization techniques

The Figures below show the disk storage impact and query performance across different embedding formats for the Paper dataset, which consists of 2 million vectors (200 dimensions, float32) evaluated over 100k queries. The experiment has been performed on the VDBMS Weaviate.

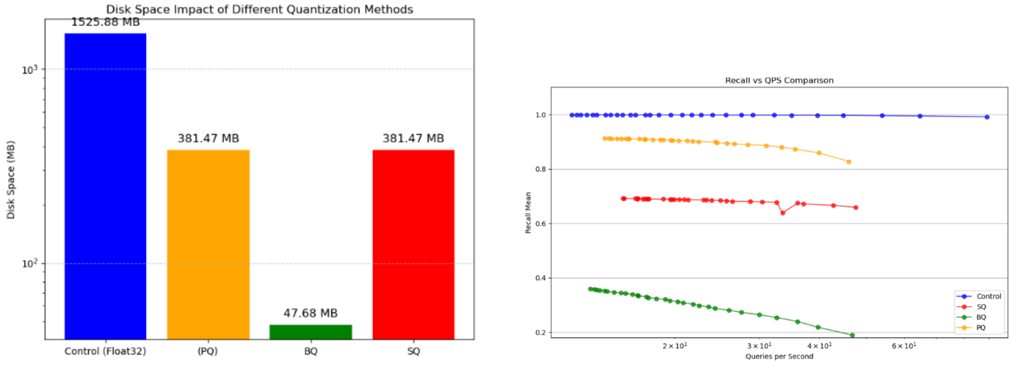


Figure 20: Disk usage and performance difference between the different techniques.

4. Indexes

This chapter will shed light on the indexing aspect of Vector Database Management Systems, specialized data structures that intelligently organize vectors by leveraging their mathematical properties. Their purpose is to optimize the retrieval process that assists search algorithms by improving their performance. There is no index that is tailored for all use cases; each index has been designed with their advantage and disadvantages, the only thing that different approaches have in common is to assist (especially in the large-scale context) reducing the search complexity to sublinear levels.

4.1 Flat Index

The flat index is the simplest approach for vector storage; it involves storing the vectors directly, meaning that queries are always an exact k-nearest neighbors (kNN) search, where the query vector is compared to every other vector in the collection. This approach has limited scalability, as the search speed begins to decline as the vector collection grows. The flat index is best suited for use cases with a manageable collection size, where search accuracy takes priority over search speed.

4.2 LSH with Random Projection

The Locality-Sensitive Hashing (LSH) indexing technique maps high-dimensional vectors to low-dimensional representations by leveraging hash functions that preserve locality and distribution. Vectors that are close in the original space are more likely to share the same hash value. This approach intentionally creates collisions to facilitate efficient retrieval of similar vectors, using hyperplanes as partitioning boundaries.

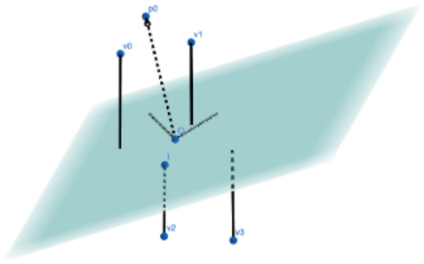


Figure 21: Example of Hyperplane. Source: Sofrwaredoug.[1].

A hyperplane splits the vector space into two regions: a positive side and a negative side. To determine which side a vector resides on, the dot product between the vector and a normal vector (perpendicular to the hyperplane) is computed.

The dot product of two vectors $\vec{a} = (a_1, a_2, \ldots, a_n)$ and $\vec{b} = (b_1, b_2, \ldots, b_n)$ is:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

- If the dot product is **positive**, the vector is assigned a label of **1** (positive side).
- If the dot product is **negative**, the vector is assigned a label of **0** (negative side).

By applying multiple hyperplanes, we create a hashed representation of vectors as binary sequences (composed of 0s and 1s), significantly reducing storage requirements. With N random hyperplanes, each vector is represented as an N-bit binary code indicating on which side of each hyperplane it falls.

To measure similarity between vectors, we compute their **Hamming distance**, which is the number of differing bits between their binary codes.

- A Hamming distance of **0** indicates that both vectors fall into the same region, meaning no hyperplane separated them.
- Larger Hamming distances indicate greater dissimilarity.

Additionally, random projections can be visualized as an **LSH Tree**, where each hyperplane acts as a decision node, branching into two possible paths: positive or negative. Traversing the tree based on these binary splits results in an efficient search structure for approximate nearest-neighbor queries.

4.3 Inverted File Index (IVF)

The **Inverted File Index** (IVF) is an indexing technique designed to enhance Approximate Nearest Neighbor (ANN) search performance. IVF partitions the vector space into centroids and limits searches to specific regions, making it ideal for large-scale, high-dimensional applications.

[1] `https://softwaredoug.com/blog/2023/08/21/implementing-random-projections`

4.3.1 Clustering and Voronoi Cells

A predefined number of centroids is established before clustering, creating an inverted index that associates each vector with its nearest centroid. Once clustering is complete, clusters expand until they form *Voronoi cells*, where each vector belongs to a single cell, defined by the shortest distance to its centroid.

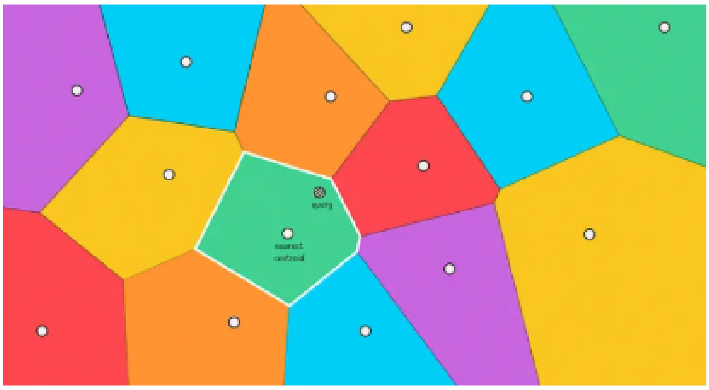


Figure 22: Example of Voronoi cells. Source: Towardsdatascience.[2]

4.3.2 Querying and the Edge Problem

Queries locate the nearest centroids and search for the closest vectors within them. However, the **edge problem** arises when suitable vectors near Voronoi boundaries are ignored. This issue worsens in high dimensions as cell boundaries become less distinct. To mitigate this, querying multiple cells increases recall at the cost of speed.

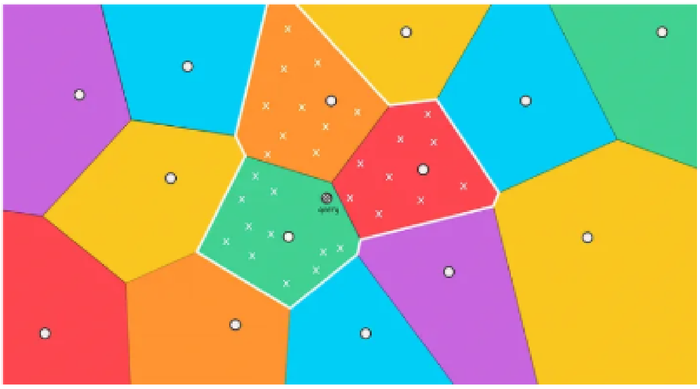


Figure 23: Edge problem. Source: Towardsdatascience.[2]

[2] `https://towardsdatascience.com/similarity-search-knn-inverted-file-index-7cab80cc0e79`

4.3.3 IVF Variants

The basic form of IVF, known as **IVFFlat**, performs a brute-force search within identified cells. Advanced variants such as **IVFPQ** (Product Quantization) and **IVFSQ** (Scalar Quantization) compress vector representations, enhancing scalability and reducing memory usage. These techniques will be covered in the next chapter.

4.4 Navigable Small Worlds

The NSW index consists of a graph that contains both long-range and short-range links. It is based on the “small world” or "six degrees of separation" rule, where each entity is, on average, separated by six degrees of link separation. This structure exhibits a high degree of clustering, where nodes are grouped into tightly-knit communities, yet only a few steps are needed to traverse between two different nodes. Essentially, this allows nodes that are typically "far" from each other in other graph-based structures to reach one another in just a few steps. Each node represents an item and maintains a friend list containing other vertices to which it is connected. This solution is fast but not highly accurate and requires tuning to perform well in ANN search over high-dimensional spaces.

4.4.1 Construction

Unlike some other indexes, the construction of the index is not a deterministic process, at first the dataset is shuffled and then the nodes are sequentially created. The main parameters in this process is **M** which the maximum number of connections a node can have. As nodes are inserted it gets connected to nearest nodes based on a distance metrics like the cosine distance.

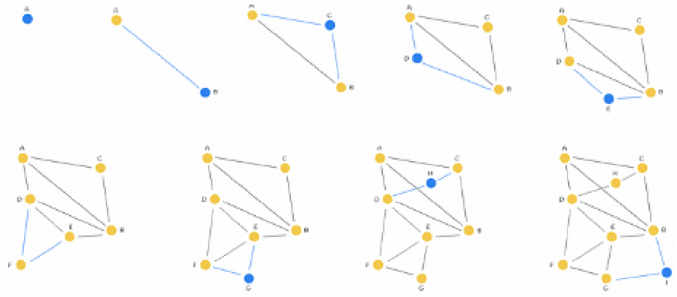


Figure 24: Example of NSW construction with M=2. Source: Towardsdatascience.[2].

[2] https://www.deeplearning.ai/short-courses/vector-databases-embeddings-applications/

4.4.2 Graph traversal

The search begins by selecting a vertex as the entry point. It then performs a greedy search, determining the next vertex to move to by checking the current vertex's friend list and moving to the closest vertex to the query node. The search terminates when no closer node is found. Since this search strategy is inherently greedy, it attempts to reach the global optimal solution by making the best local decision at each step. While this approach is simple and fast, it is susceptible to early stopping, where the search gets trapped in regions of the graph where the local optimum has no better neighboring candidates. To mitigate the risk of getting stuck in dead regions, various strategies can be employed. One such method is **“beam search"**, which maintains a **top-k** list of the best candidates at each step. Another approach is “multi-hop search,” which considers not only direct neighbors but also neighbors of neighbors.

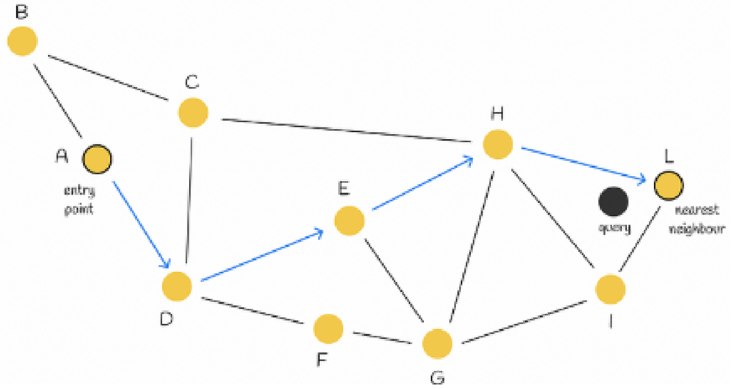


Figure 25: example of NSW routing where A is entry point. Source: DeepLearningAI.[3].

4.5 Hierarchical Navigable Small Worlds

The HNSW index is an evolution of the **Navigable Small World (NSW)** graph, incorporating principles from the probabilistic data structure known as the **skip list**. A skip list consists of multiple layers of ordered linked lists, where each upper layer contains a subset of elements from the layer below. This hierarchical structure enables efficient search by starting from the topmost layer and performing a sequential search.

If the target element is found, the search terminates. Otherwise, upon encountering a greater value, the algorithm descends to the next layer, using the previously visited element as the new entry point.

[3] https://www.deeplearning.ai/short-courses/vector-databases-embeddings-applications/

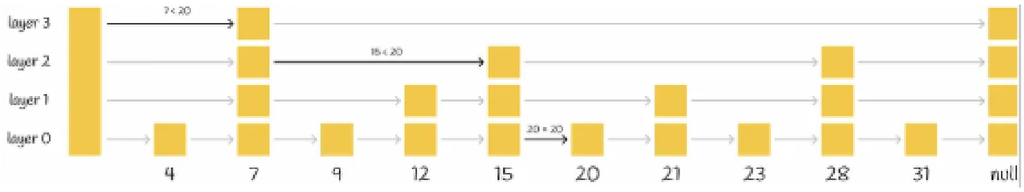


Figure 26: Skip List. Source: Towardsdatascience.[2]

The **HNSW** index follows a similar layered structure, where:

- The **topmost layer** contains a small subset of representative nodes.
- The **in-between layers** contain progressively more nodes.
- The **lowest layer** includes all nodes in the dataset.

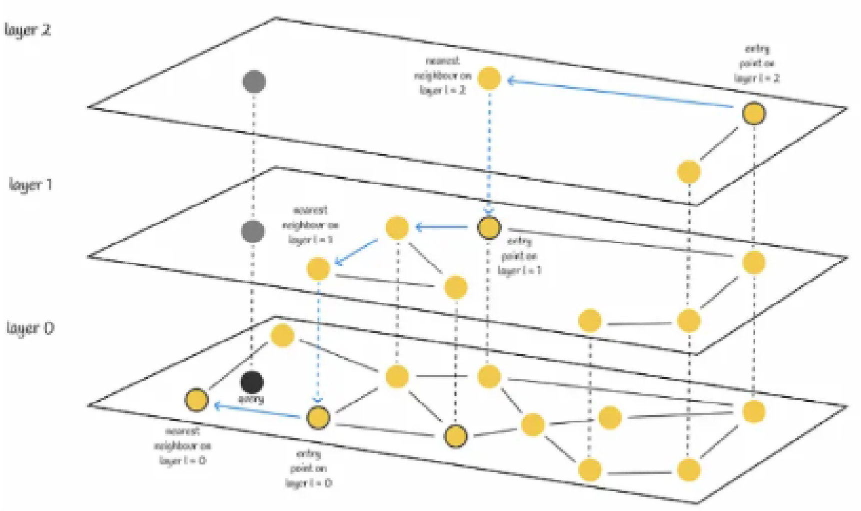

layer 2
nearest neighbour on layer l = 2
entry point on layer l = 2
layer 1
nearest neighbour on layer l = 1
entry point on layer l = 1
layer 0
entry
nearest neighbour on layer l = 0
entry point on layer l = 0

Figure 27: HNSW index. Source: Towardsdatascience.[2]

This multi-layered organization significantly improves retrieval efficiency by reducing the number of comparisons required for nearest neighbor search.

4.5.1 Search Process

The HNSW query process is similar to NSW but introduces a hierarchical search mechanism. After locating the nearest node in a layer, it serves as the entry point for the lower layer.

HNSW graphs follow the skip list principle:

- The **topmost layer** has fewer nodes and connections.
- Lower layers have increasing connectivity until **Layer 0**, which contains all data points.

The search begins at a predefined entry node in the top layer and performs a greedy search.

The algorithm descends to the next layer when it finds a closer node in that layer. This process continues until reaching Layer 0, ensuring all potential nearest neighbors are considered.

HNSW search is controlled by key parameters:

- ef **(exploration factor)**: Determines the number of candidate nodes kept during the search. A higher ef improves recall at the cost of increased latency.
- **Entry Point Selection**: The algorithm selects a fixed or dynamically chosen entry node in the topmost layer to begin the search.

These parameters allow fine-tuning of search accuracy, efficiency, and resource usage in HNSW-based vector searches.

4.5.1.1 Impact of M parameter on HNSW search

The Figures below show the impact of different values of M,performed over the embedding that come from the Tripclick dataset, which consists rhoughly of 800k vectors (768 dimensions, float32) evaluated over 1000 queries. The experiment has been performed on the VDBMS Weaviate.

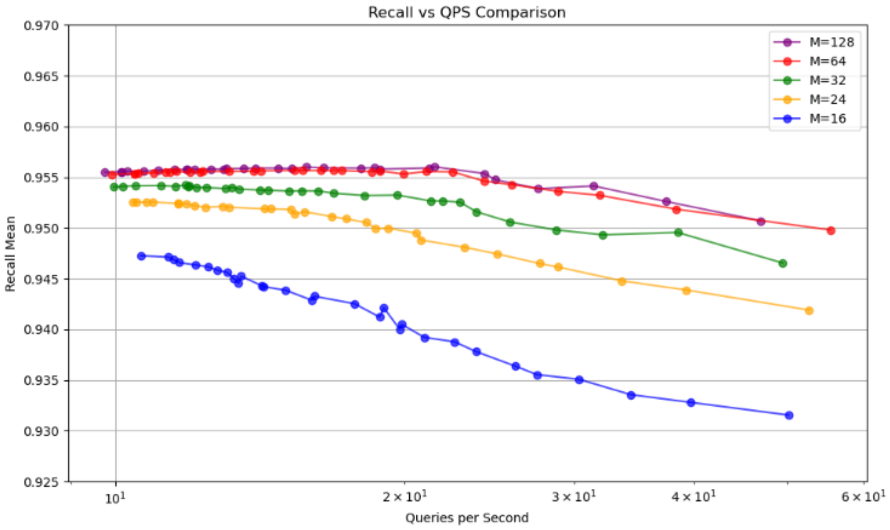


Figure 28: QPS vs Recall between different values of M.

4.5.2 Construction Process of HNSW

The construction of an **HNSW** index follows a hierarchical approach, inspired by skip lists, to optimize approximate nearest neighbor search. The graph is built incrementally, adding one data point at a time while maintaining efficient connectivity across multiple layers.

Key Parameters

The construction process is governed by several key parameters:

- **Maximum Layer ($L_{\max}$):** Determines the highest level a node can be assigned. It is typically chosen using a logarithmic distribution based on the dataset size.
- **Maximum Connections per Layer (M):** Defines the number of bidirectional edges each node can maintain per layer, balancing search efficiency and memory usage.
- **Connectivity Factor (efConstruction):** Controls the number of nearest neighbors considered during graph construction. A higher value results in better graph quality but increases indexing time.
- **Insertion Strategy:** Each new data point is assigned a random layer up to $L_{\max}$ and inserted into the graph by connecting to its M nearest neighbors in each layer.
- **Pruning Strategy:** Redundant edges are removed to ensure efficient navigation while maintaining strong graph connectivity.

Construction Algorithm

1. Assign each new data point a layer up to $L_{\max}$.
2. Insert the point into the topmost layer and connect it to the M nearest neighbors.
3. Repeat the process for each lower layer, ensuring bidirectional edges.
4. Apply pruning to maintain an optimal balance between connectivity and efficiency.

The hierarchical structure ensures that higher layers facilitate fast exploration, while lower layers refine the search for precise nearest neighbors.

4.6 DiskANN

One of the main drawback of graph-based indexes like NSW and HNSW, is the significant memory footprint, which makes querying over large amount of data expensive. While this may not be an issue in distributed and high resources environments, the same cannot be said for single machines with limited memory. The most basic approaches for handling single node environments are the employment of **IVFPQ** or by sharding the dataset with their own in-memory indexes which during the search each result is merged together. These solution suffer from a lower recall and to address this, **DiskANN** provides an SSD resident index with the objective of reducing the memory footprint and optimize disk accesses thanks to the assistance of the **Vamana** algorithm, which differentiates itself from NSW by constructing the graph in order to optimize disk access instead of improved traversal.

4.6.1 Graph Construction Process

Instead of starting with a sparse graph and gradually adding edges (as in NSW), Vamana begins with a dense graph and iteratively prunes unnecessary edges. The process consists of the following steps:

1. **Initialize:** Start with a randomly connected graph with excess edges.
2. **Graph Construction:** For each point p:
   (a) **Find Nearest Neighbors:** Greedy search in the current graph.
   (b) **Prune Edges:** Remove redundant connections while ensuring connectivity.
   (c) **Add Backward Links:** Ensure bidirectional edges for efficient traversal.
3. **Refinement:** Perform two rounds of pruning with different distance thresholds to optimize local and long-range connections.
4. **Disk Optimization:** Store the graph efficiently to minimize random disk access and enable batched reads.

This pruning-based method enables Vamana to optimize the trade-off between **graph diameter**, which influences search depth, and **node degree**, which impacts memory consumption, more efficiently than conventional techniques. DiskANN utilizes Vamana because it builds a **flat graph**, making it more suitable for disk storage compared to hierarchical structures.

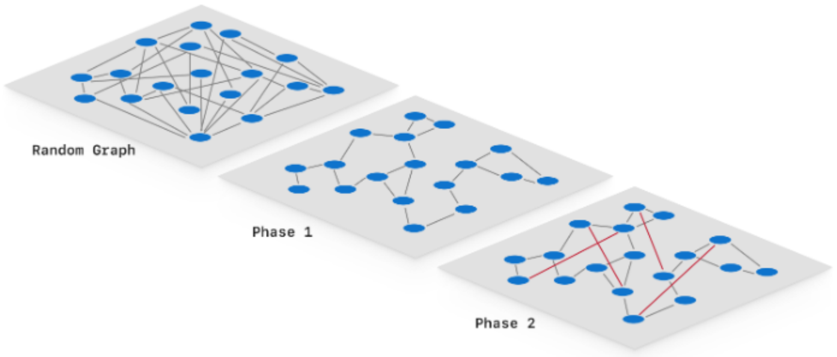


Figure 29: Vamana Graph example. Source: Planetscale.[4].

4.6.2 Optimizations in DiskANN

DiskANN provides some optimization in order to increase querying performance:

- **Caching:** Nodes that are fetched more freqeuntly than others are stored in RAM, in order to reduce SSD use and increase the performance.
- **Compressed vectors and re-ranking:** In order to increase retrieval efficiency, vectors are stored in RAM in their compressed form, in order to do a preliminary comparison. After, a re-ranking process is done with a full precision SSD search.
- **Clustered Indexing:** Partitions data into overlapping clusters using k-means clustering. Each data point belongs to multiple clusters, ensuring connectivity for search algorithms.
- **Beam Search:** Retrieves neighborhood data from SSD in small batches, reducing read latencies by grouping nearby data access requests.

[4] https://planetscale.com/docs/vectors/terminology-and-concepts

5. Query processing

Query processing in a vector database is the process of searching for the most relevant results from a collection of vectors. It starts with a search specification, which includes a similarity score and the type of query; all of these are provided by users through a query interface. After receiving the query, the system processes it by running a series of steps in the vector collection. These steps include simple similarity searches and faster methods that use indexes to improve efficiency. This ensures that the search is handled quickly and returns accurate results.

5.1 Vector similarity search

The goal of a vector similarity search is to efficiently identify vectors that are most similar to a given query vector based on a specified distance or similarity metric. This search process is often used in high-dimensional spaces, where traditional search techniques can become computationally expensive. Several types of vector similarity search techniques exist, each with its trade-offs in terms of accuracy, efficiency, and applicability to different problems.

5.1.1 Distance metrics

Vector similarity search leverages the mathematical properties of vectors to find the most relevant vectors in a database in response to a query. This is done by evaluating the distance metric between the query vector **q** and the vectors **v** in the database. A distance score, often denoted **d(q,v)**, calculates a scalar value of the two **D**-dimensional vectors, where a smaller value indicates greater similarity between the vectors. The vectors with the lowest distance scores are considered the most similar to the query vector.

- **Euclidean Distance:** Measures the straight-line distance between two points in a D-dimensional space.

$$d(a, b) = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}$$

- **Manhattan Distance (L1 Norm):** Calculates the sum of absolute differences between the coordinates of two vectors.

$$d(a,b) = \sum_{i=1}^{D} |a_i - b_i|$$

- **Cosine Similarity:** Computes the cosine of the angle between two vectors, often used as a similarity metric. The corresponding distance is:

$$d(a,b) = 1 - \frac{\sum_{i=1}^{D} a_i b_i}{\sqrt{\sum_{i=1}^{D} a_i^2}\sqrt{\sum_{i=1}^{D} b_i^2}}$$

- **Jaccard Distance:** Used for binary or sparse data, defined as:

$$d(a,b) = 1 - \frac{|a \cap b|}{|a \cup b|}$$

- **Hamming Distance:** Calculates the number of positions where two binary vectors differ.

$$d(a,b) = \sum_{i=1}^{D} \mathbb{I}(a_i \neq b_i)$$

5.1.2 KNN

The K Nearest Neighbor algorithm is the most straightforward approach when it comes to Vector Similarity Search, providing an exact result by confronting the k nearest vectors in the vector space. Although this brute-force algorithm provides perfect recall, it requires a lot of computation resources, and, by consequence, the query runtime increases along as the number of data points increases.

5.1.3 ANN

The Approximate Nearest Neighbor (ANN) algorithm was developed to address the limitations of the k nearest neighbors (kNN) approach, particularly its inefficiency when scaling to large datasets. The primary goal of ANN is to provide near-optimal results while significantly reducing computation time compared to exact kNN searches. Using techniques that approximate nearest neighbors, ANN enables the handling of massive collections of vectors, offering a practical solution for high-dimensional search tasks. The ANN approach also makes use of the vector indexes such as NSW and HNSW that are based on the concept of "six degrees of separation" allowing the algorithm to explore the vector space more efficiently.

5.2 Performance metrics

Assessing the quality of the vector similarity search requires the definition of clear and relevant performance metrics. These metrics provide a standardized way to measure how effectively the search algorithm retrieves the most relevant results from the vector collection, and allows one to find which approach is more suitable for a specific use case.

- **Precision:** Measures the proportion of relevant results among the retrieved results.

$$\text{Precision} = \frac{\text{Number of Relevant Results Retrieved}}{\text{Total Number of Results Retrieved}}$$

- **Recall:** Measures the proportion of relevant results that were successfully retrieved.

$$\text{Recall} = \frac{\text{Number of Relevant Results Retrieved}}{\text{Total Number of Relevant Results in the Database}}$$

- **Queries Per Second (QPS):** Measures the system's throughput by calculating the number of queries processed per second.

$$\text{QPS} = \frac{\text{Number of Queries Processed}}{\text{Total Time (in seconds)}}$$

5.3 Filtering Techniques

Filtering techniques in vector search help refine results based on predefined criteria. These techniques ensure that retrieved documents meet specific conditions, improving relevance and accuracy.

5.3.1 Pre-filtering

Pre-filtering applies constraints before executing the search query. This method reduces the search space, leading to faster retrieval times. Common pre-filtering techniques include:

- Applying metadata filters (e.g., date, category, author).
- Restricting searches to specific document subsets.
- Leveraging user-defined conditions for search optimization.

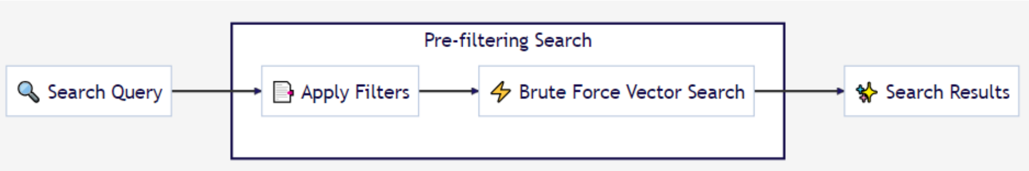


Figure 30: Pre-filtering process. Source: Weaviate[1]

5.3.2 Post-filtering

Post-filtering applies constraints after the search query has been executed. This approach ensures that filtering does not interfere with similarity computations. Common post-filtering techniques include:

- Removing results that do not meet certain thresholds.
- Applying additional business logic to refine search outputs.
- Filtering based on dynamic user preferences after initial retrieval.

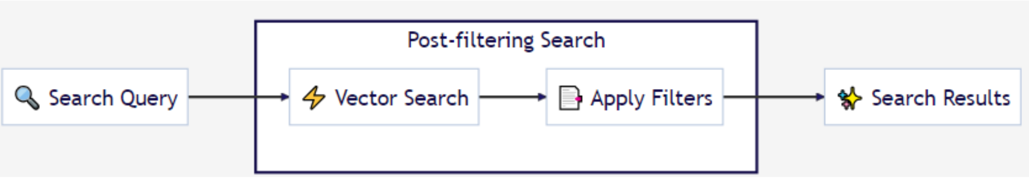


Figure 31: Post-filtering process. Source: Weaviate[1]

[1] `https://weaviate.io/blog/speed-up-filtered-vector-search`

5.3.3 In-filtering

In-filtering integrates filtering conditions directly into the similarity computation process. This technique ensures that relevance scoring inherently considers filtering constraints. Key aspects of in-filtering include:

- Modifying similarity calculations to account for filtering parameters.
- Embedding constraints within vector representations.
- Ensuring efficient and scalable implementation in high-dimensional spaces.

5.4 ACORN

ACORN is a predicate-agnostic hybrid search method that extends HNSW to efficiently handle structured query filters. Unlike traditional pre-filtering and post-filtering methods, ACORN achieves sublinear retrieval times even with high-cardinality predicate sets. It constructs a denser graph index and applies predicate-aware traversal strategies to optimize search performance. By leveraging efficient predicate-subgraph traversal while maintaining the speed advantages of HNSW, ACORN provides a scalable and high-performance alternative to traditional hybrid search strategies.

Key parameters governing ACORN's behavior include:

- γ: Neighbor expansion factor.
- M_β: Compression parameter for neighbor pruning.
- M: Degree bound for traversed nodes.
- e: Fixed entry point to the index.
- s_{min}: Minimum predicate selectivity.

5.4.1 Index Construction

The ACORN-γ index is built using two key modifications to HNSW:

1. **Neighbor List Expansion:** Each node collects $M \cdot \gamma$ approximate nearest neighbors instead of the standard M. This ensures that filtering operations during search do not fragment the graph.
2. **Predicate-Agnostic Pruning:** To counter increased index size, a compression heuristic is applied to bottom-level neighbor lists, reducing memory overhead while maintaining

navigability.

A recommended choice for γ is $1/s_{min}$, ensuring efficient pre-filtering when selectivity drops below s_{min}. This enables ACORN to balance indexing cost and search efficiency.

5.4.2 Search Algorithm

ACORN employs a hierarchical greedy search similar to HNSW but integrates predicate filtering:

1. Search begins at the highest level with a fixed entry point (e).
2. Nodes are evaluated based on both distance and predicate constraints.
3. At each level, the algorithm retrieves filtered neighbors ($N_p^l(v)$), either by:
    - Applying a simple predicate filter (truncating to M neighbors).
    - Expanding two-hop neighbors for enhanced connectivity.
4. The search proceeds hierarchically until reaching the base level, returning the nearest neighbors that satisfy the query.

5.4.3 Complexity Analysis

- **Index Construction:** $O(n \cdot \gamma \cdot \log(n) \cdot \log(\gamma))$, where n is the dataset size.
- **Search Complexity:** $O((d + \gamma) \cdot \log(s \cdot n) + \log(1/s))$, closely approximating HNSW's complexity with minor overhead.

5.4.4 Weaviate's Implementation of ACORN

Weaviate modifies ACORN with:

- **Graph Construction:** Retains HNSW's original pruning logic to reduce resource overhead and support seamless activation without reindexing.
- **Graph Exploration:** Uses two-hop expansion only when the first-hop node fails the filter, optimizing performance in high-filter-density regions.
- **Entry Point Seeding:** Introduces additional entry points at layer zero to improve retrieval efficiency in low-filter-density areas.

5.4.5 Performance Analysis on Weaviate

In the following figures, we present a performance comparison of different filtering strategies. The first figure compares pre-filtering and post-filtering, where post-filtering has been expanded to account for filter selectivity. While its recall improves and approaches that of pre-filtering, this comes at the cost of higher resource usage. The second figure compares the default and ACORN filtering strategies, showing that ACORN achieves slightly better performance with lower resource usage. The study was conducted on the TripClick vector dataset (800k vectors of 768 dimensions) using an HNSW index.

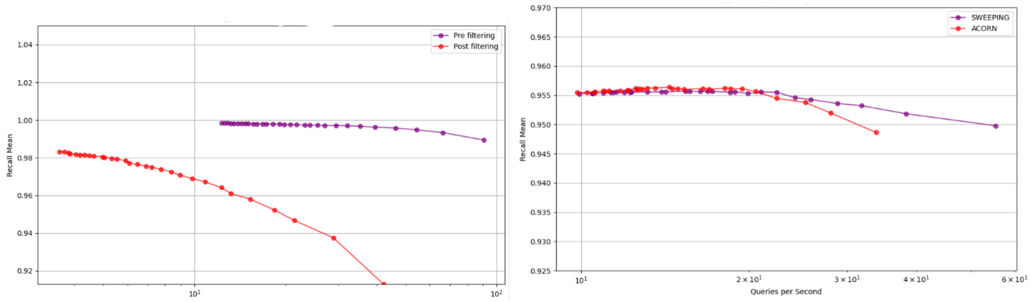


Figure 32: Performance comparison in Weaviate.

5.5 Range Filtering ANN

Range Filtering ANN (RFANN) is used for high-dimensional data where each vector has an associated, ordered attribute. Given a dataset D, a query vector q, a parameter K, and a query range $[x, y]$, RFANN aims to find the approximate K nearest upper neighbors in D whose attributes fall within $[x, y]$. This is valuable in applications like e-commerce, where users filter products by price. However, such searches can be computationally expensive. Common approaches—Post-Filtering, In-Filtering, and Pre-Filtering—each have limitations: Post- and In-Filtering struggle with high-selectivity queries, while Pre-Filtering becomes inefficient with large datasets. Optimizing resource usage and performance remains a key challenge.

5.6 Naive Solution

A simple but impractical approach to solving range-filtering approximate nearest neighbor (RFANN) queries is to build dedicated graph-based indexes for all possible query ranges. However, this does not scale well. Given a collection of n objects, the number of possible query ranges grows to $O(n^2)$.

5.7 IRangeFiltering

IRangeGraph is a novel method designed to handle range-filtering queries efficiently without constructing a massive number of dedicated indexes. Instead of materializing a separate graph for each query range, **IRangeGraph constructs a moderate number of precomputed graphs, called elemental graphs**, which are later used to dynamically build the required index for any given query range.

The key benefits of IRangeGraph are:

- **On-the-fly graph construction**: Instead of storing all possible dedicated graphs, it dynamically assembles a graph from *elemental graphs* corresponding to segments of the data.
- **Efficient querying**: Despite constructing the graph during query execution, the overhead is low, and performance is close to that of fully materialized dedicated graphs, while using significantly less memory.

5.7.1 Constructing Elemental Graphs and Forming Dedicated Graphs

To efficiently construct elemental graphs, **IRangeGraph leverages a segment tree**:

- The segment tree has $O(\log n)$ **layers**, where each node represents a range of objects.
- The **root node** corresponds to the entire dataset.
- Each **child node** represents a sub-range, recursively partitioning the dataset into smaller segments.
- **Leaf nodes** correspond to individual objects (segments of size 1), meaning it contains elements with the same values.
- Each segment stores an **elemental graph**, which serves as a building block for constructing query-specific graphs **irangegraph**.

Each **node** in the segment tree, not just the leaves, has its own elemental graph materialized. This means that an elemental graph is constructed for every segment at every level of the segment tree, leading to a total of $O(n \log n)$ elemental graphs across the entire structure **irangegraph**.

5.7.2 Dedicated Graph Construction

In order to construct on-the-fly the dedicated graph, IRangeGraph utilizes the concept of the **Relative Neighborhood Graph (RNG)**. Which ensures that an edge between two objects exists only if no other object is significantly closer to both of them. This pruning rule helps maintain a sparse yet effective structure for nearest neighbor search.

When a query is issued with a given range $[L, R]$, IRangeGraph constructs the required graph dynamically:

1. Identify the **elemental graphs** that intersect with the query range.
2. Combine relevant edges from these elemental graphs to form the dedicated graph.
3. Apply **RNG pruning rules** to retain only essential edges.

5.7.2.1 Example: Querying a Specific Range

Consider a dataset with values in the range $[0, 4]$ and an issued query for the range $[2, 3]$. The segment tree structure would include:

- A root node covering the full range $[0, 4]$.
- Child nodes covering $[0, 2]$ and $[3, 4]$.
- Further subdivisions where necessary, with leaf nodes at individual values.

For the query $[2, 3]$:

1. The elemental graphs corresponding to segments $[2, 2]$ and $[3, 3]$ are selected since they fully intersect with the query range.
2. Additionally, the segment $[2, 3]$ is selected, as it directly encompasses the range.
3. The relevant edges from these elemental graphs are combined to construct the dedicated graph.
4. The resulting graph enables efficient nearest neighbor search within the range $[2, 3]$.

5.8 Segment Graph for RFANN Search (SeRF)

SeRF addresses the issue of the Naive approach by introducing a *segment graph*, which losslessly compresses these n HNSW indexes into a single structure with a significantly reduced memory footprint. The data structure is refereed as Segment Graph, which organizes the neighbor

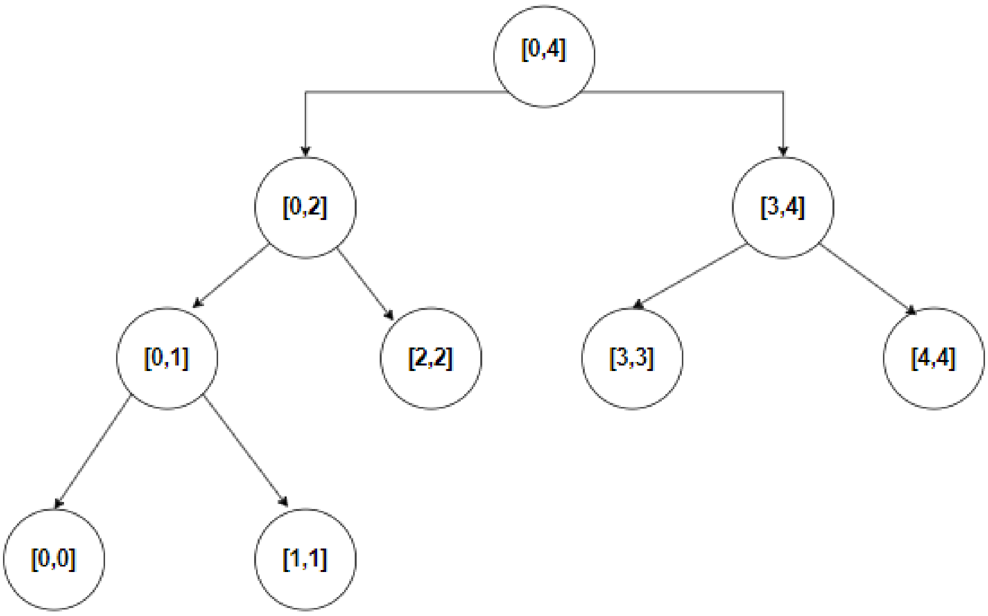


Figure 33: Example of Segment Tree.

relationships dynamically across different ranges, permitting to represent the the connectivity of n HNSW graphs using only $\mathcal{O}(nM)$ space, where M is the maximum degree of nodes in HNSW.

This is achieved through:

- Storing each edge only once, annotated with its active range $[b, e]$, instead of duplicating it across multiple graphs.
- Avoiding the explicit storage of n independent HNSW graphs, leading to a compression factor of approximately $\mathcal{O}(n)$ in the worst case.
- Ensuring efficient query execution by enabling range-based edge retrievals without reconstructing multiple indexes.

5.8.1 Segment Graph construction

The construction follows these key steps:

1. **Sequential HNSW Insertion:** Data points are sorted based on their associated attribute values. The HNSW graph is incrementally built in an ordered manner, meaning that at step x, the index constructed so far contains only the first x points.
2. **Tracking Persistent Neighbor Relationships:** When adding a new point v_x to the

graph, its approximate nearest neighbors are determined using an ANNS search over the existing structure. These neighbors are then pruned to ensure efficient navigation in the HNSW hierarchy. Instead of storing these neighbor connections separately for each step, we record their validity over a *segment range* $[b, e]$, where:

- b represents the insertion step when the edge was first added.
- e represents the step at which the edge was removed due to the pruning process.

3. **Segment-Based Edge Storage:** Each edge (v_i, v_j) in the segment graph is stored as a tuple (v_j, b, e) in the adjacency list of v_i. This means that v_j remains a neighbor of v_i in all query-specific graphs from step b to e. Instead of storing n redundant graphs, this method encodes all necessary information in a single compressed structure.

4. **Edge Pruning and Updates:** When a new node v_x is inserted, it may cause the removal of certain edges due to dominance criteria in HNSW pruning (i.e., when a new node provides a more efficient navigation path). The segment graph updates the validity range of affected edges dynamically, ensuring that removed edges are correctly marked with an endpoint $e = x - 1$.

5.9 Further Compression of the Segment Graph

SeRF compresses multiple HNSW indexes into a single segment graph but is limited to static datasets. The **Dynamic Range-Filtering Approximate Nearest Neighbor Search (DR-FANN)** model enhances adaptability with a *dynamic segment graph*, enabling incremental updates while reducing redundancy and storage overhead.

By integrating a rectangle tree and dynamic segment graph, DRFANN surpasses static RFANN methods (SeRF, iRangeGraph, etc.):

- **Supports Dynamic Data:** Allows unordered vector insertions.
- **Compact Indexing:** Reduces $\mathcal{O}(n^2)$ HNSW graphs to $\mathcal{O}(n \log n)$ edges.
- **Efficient Queries:** Eliminates costly ANN result merging.
- **Fast Updates:** Only $\mathcal{O}(\log n)$ edges are modified per insertion.

5.9.1 Dynamic Segment Graph and Compression

The **dynamic segment graph** represents RFANN query ranges in a single, continuously updated structure. Instead of duplicating HNSW graphs, it assigns each edge a rectangular validity label $(l, r] \times [b, e)$, indicating the attribute range where the edge remains valid.

5.9.2 Graph Construction

As new vectors arrive, the dynamic segment graph is updated as follows:

1. **Rectangle Representation:** Each edge (v_i, v_j) is assigned a validity range $(l, r] \times [b, e)$.
2. **Edge Reuse:** Edges are shared across overlapping query ranges, minimizing storage.
3. **Incremental Updates:** New insertions adjust only $\mathcal{O}(\log n)$ edges, preventing index bloat.

5.9.3 Rectangle Tree for Space Partitioning

The *rectangle tree* hierarchically partitions query ranges, optimizing storage and retrieval:

- Inserting a vector with attribute $\mathbf{x}$ initializes a root node covering $(-\infty, x] \times [x, \infty)$.
- Nearest neighbors are retrieved iteratively, with layer n containing n shared neighbors.
- Nodes represent query ranges $(l, r] \times [b, e]$, reducing redundancy.
- Leaf nodes store approximate nearest neighbors (ANNs) for fast lookup.
- Internal nodes partition the search space, ensuring efficient range-filtered searches.

5.9.4 Example of Insertion in Dynamic Segment Graph

To better understand the how the Dynamic Segment Graph works, we illustrate the process of the insertion of a new vector. Let the existing dataset consist of nine vectors, each with an associated attribute value, and suppose we insert a new vector:

$$v10 \quad \text{with attribute value } 54.$$

1. Identify the K nearest neighbors using the existing segment graph.
2. Initialize an R-tree with root $(-\infty, 54] \times [54, \infty)$, representing v_{10}'s rectangle.
3. Partition the range space iteratively using the K nearest neighbors.
4. Update the **rectangle tree** to incorporate the new attribute ranges.
5. Add edges upon reaching the child nodes.

Vector	Distance to v_{10}	Attribute
v_1	2	28
v_2	18	36
v_3	17	68
v_4	8	37
v_5	4	43
v_6	11	56
v_7	9	57
v_8	13	66
v_9	10	35

Table 1: Dataset before inserting v_{10}.

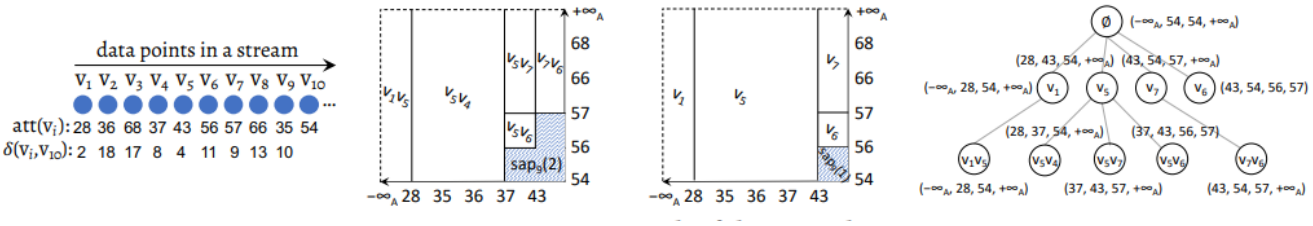


Figure 34: Rectangle Tree and range space partitioning. Source: Miaoqiao[2]

[2] https://miaoqiao.github.io/paper/VLDB25_TR.pdf

6. RAG

This chapter explores Retrieval-Augmented Generation (RAG), which enhances Large Language Models (LLMs) by integrating retrieval from Vector Databases (VDBMSs) to mitigate their limitations. We discuss alternative RAG approaches, including Agentic RAG, which leverages AI agents for iterative reasoning and refinement.

6.1 Retrieval-Augmented Generation

Large Language Models (LLMs) are trained on vast datasets, but they face challenges when queried about information beyond their training data. In such cases, they may either admit a lack of knowledge or, worse, generate inaccurate responses—a phenomenon known as hallucination.

LLMs, despite their capabilities, have two significant limitations:

- They can confidently produce incorrect or outdated information (hallucinations).
- They may not have been trained on the specific data required for a given query.

Retrieval-Augmented Generation (RAG) addresses this issue by supplementing LLMs with relevant external data. Instead of relying solely on pre-trained knowledge, RAG retrieves pertinent documents and incorporates them into the prompt, ensuring more accurate and context-aware responses. This approach is sometimes referred to as generative search or in-context learning.

RAG mitigates these issues through a two-step process:

1. **Retrieval** – Relevant information is fetched based on a query.
2. **Generation** – The LLM is then prompted with both the retrieved data and the user's query, allowing it to generate a response using up-to-date and relevant information.

By leveraging retrieval-based augmentation, RAG enhances the accuracy and reliability of LLM outputs, reducing reliance on potentially outdated or incorrect pre-trained knowledge.

6.2 Agentic Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) enhances Large Language Models (LLMs) by incorporating external knowledge, improving accuracy, and reducing hallucinations. Instead of relying solely on the static knowledge encoded during pretraining, RAG retrieves relevant information from external sources to provide more accurate, up-to-date, and contextually appropriate responses.

However, traditional RAG approaches have several limitations:

- **Static retrieval and generation process:** The retrieval step typically happens only once before the generation phase, which means there is no opportunity for iterative refinement based on context or response quality.
- **Limited to a single external knowledge source:** Most RAG implementations retrieve information from a single vector database (VDB), but some use cases require multiple sources, such as APIs, structured databases, or web-based knowledge repositories.
- **Lack of reasoning over retrieved information:** The retrieved context is used as-is without further validation, filtering, or iterative reasoning, potentially leading to the propagation of incorrect or incomplete information.

6.2.1 Introducing Agentic RAG

Agentic RAG extends the traditional RAG paradigm by introducing AI agents—autonomous LLM-based systems that perform specialized tasks while leveraging dynamic retrieval, reasoning, and tool use. Unlike standard RAG, which retrieves and applies context in a single step, agentic systems operate iteratively, refining their approach and responses through multiple retrieval and reasoning cycles.

In an agentic framework, each agent has a specific role, allowing for modular execution where tasks are divided and optimized at different stages. The agents operate in a structured, step-by-step manner, where each stage builds upon previous results, ensuring continuous improvement in response quality.

The key components of an AI agent include:

- **LLM:** The core reasoning and generation engine responsible for processing information and making decisions.

- **Task:** A predefined objective or role assigned to the agent, guiding its execution strategy.
- **Long-term memory:** A persistent storage system, often implemented using vector databases (VDBMs), enabling the agent to retain learned information over multiple interactions.
- **Short-term memory:** A temporary state tracker that maintains context throughout execution, enabling dynamic workflow adjustments and information flow between agent interactions.
- **Tools:** External utilities such as APIs, databases, search engines, code execution environments, or computational functions that extend the agent's capabilities beyond text-based generation.

By leveraging these components, agents can actively retrieve, analyze, and refine information rather than passively relying on a single retrieval step. This iterative approach results in more contextually accurate and reliable responses.

6.2.2 Agentic RAG and Agent-Based Frameworks

Agentic RAG integrates agent-based frameworks with RAG, creating a dynamic pipeline that iteratively retrieves, reasons, and refines information. This approach addresses the limitations of static retrieval by enabling LLMs to interact with external tools, break down complex tasks, and continuously adapt based on intermediate results.

Unlike traditional RAG, which retrieves data once and directly uses it for generation, Agentic RAG incorporates multiple retrieval cycles, validation mechanisms, and step-by-step task execution. The agent can refine the retrieved information, cross-check responses, and adapt its approach dynamically.

Key characteristics of Agentic RAG include:

- **Autonomous execution:** Agents execute tasks independently, continuously assessing and refining their output based on intermediate feedback.
- **Multi-step decision-making:** Instead of retrieving information once, agents can perform multiple queries, validate responses, apply logical reasoning, and adjust their strategy as needed.
- **Tool integration:** Agents leverage APIs, databases, and computational tools to dynamically fetch, analyze, and process external data beyond what is stored in their training

corpus.

- **Dynamic adaptation:** Agents can modify their retrieval strategy based on contextual cues, ensuring more relevant and precise results.
- **Progressive refinement:** The agent iteratively improves responses by retrieving additional data, analyzing inconsistencies, and updating its understanding.

6.3 Agentic Frameworks

Agentic frameworks provide structured methodologies for developing AI agents that integrate reasoning, retrieval, and action execution. These frameworks enable the creation of more intelligent and autonomous LLM-powered systems by allowing agents to dynamically interact with tools, retrieve external knowledge, and refine their responses iteratively. By leveraging these frameworks, developers can design AI agents that retrieve information more accurately, adapt dynamically to new inputs, and maintain context over extended interactions.

6.3.1 ReAct Framework

One widely adopted framework for agent-based reasoning is the ReAct framework, which stands for **Reason + Act**. A ReAct agent processes sequential, multi-step queries while maintaining state by combining reasoning, tool use, and query planning into a cohesive workflow.

The ReAct process follows an iterative cycle:

- **Thought:** Upon receiving a query, the agent reasons about the next action.
- **Action:** The agent executes the decided action, such as querying a database, calling an API, or retrieving relevant documents.
- **Observation:** The agent evaluates the output of the action and determines the next step.

This cycle repeats until the agent reaches a final response. The iterative nature of ReAct allows it to refine answers dynamically, validate retrieved information, and adjust its approach based on real-time observations.

6.3.2 LangChain and LangGraph

Several frameworks have emerged to facilitate the development of agentic systems, with **LangChain** and **LangGraph** being two of the most prominent. While both are designed to enable LLM-powered applications, they differ in their execution models and capabilities.

6.3.2.1 LangChain: Modular and Sequential Execution

LangChain provides a modular architecture for integrating LLMs with various external tools, databases, and APIs. It offers structured workflows for chaining multiple components, such as prompt templates, memory modules, and tool interfaces.

Key characteristics of LangChain:

- **Sequential Execution:** LangChain primarily follows a linear pipeline, where tasks are executed in a predefined order.
- **Memory Management:** It supports both short-term and long-term memory, enabling persistent context handling across interactions.
- **Tool Integration:** LangChain allows agents to interact with APIs, databases, and custom functions, expanding their ability to fetch and process information.
- **Pre-built Components:** Developers can leverage pre-configured templates for common use cases, simplifying implementation.

LangChain is well-suited for structured LLM applications, such as chatbots, document retrieval systems, and structured automation pipelines, where execution follows a predictable sequence.

6.3.2.2 LangGraph: Graph-Based Execution for Agentic Systems

LangGraph extends LangChain by introducing a **graph-based execution model**, allowing agents to perform dynamic, non-linear reasoning and iterative refinement. Unlike LangChain's sequential structure, LangGraph enables flexible workflows where execution can be cyclic, conditional, or parallel.

Key differences of LangGraph:

- **Graph-Based Workflow:** Instead of a linear pipeline, LangGraph represents tasks as nodes and execution logic as edges, enabling more complex task decomposition.
- **Cyclic Execution:** Supports iterative reasoning, allowing agents to refine responses through multiple retrieval and validation steps.
- **Conditional Logic:** Agents can make real-time decisions based on retrieved information, enabling adaptive responses.
- **Multi-Agent Coordination:** Facilitates communication between multiple agents, each

specialized for different tasks, improving efficiency in complex workflows.

LangGraph is particularly useful for applications requiring multiple retrieval steps, adaptive decision-making, and real-time processing, such as dynamic research assistants, financial analysis bots, and multi-agent orchestration systems.

6.3.3 Example of Document Retrieval with LangGraph

As previously mentioned, Agentic RAG helps overcome the limitations of static RAG. One such improvement can be observed in the following experiment. The initial setup consists of a vector database containing PubMed documents indexed using an HNSW index. Each document includes a title, abstract, and tags, all of which are embedded using the multilingual Ollama model. The challenge arises during querying: when searching for documents related to *breast cancer*, the vector database embeds the query, retrieves the top K matching documents, and forwards them to the LLM for response generation. However, if an irrelevant but semantically similar document (such as documents about mouth cancer or cancer in general) is included in the retrieved set, the LLM may produce suboptimal results due to the presence of non-relevant context.

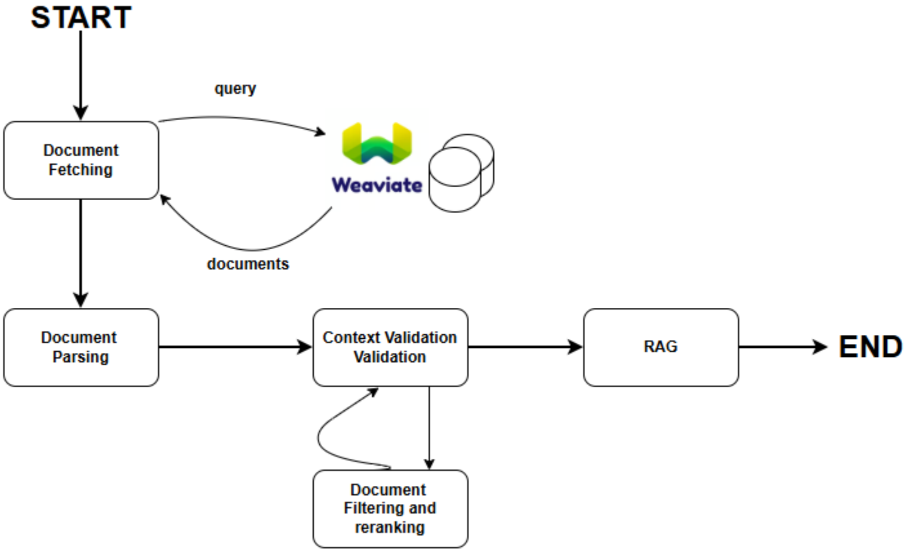


Figure 35: LangGraph Workflow for Document Retrieval.

LangGraph addresses this issue by introducing a structured, step-by-step retrieval process where each agent performs a distinct task. These tasks include:

- Fetching documents from the vector store
- Parsing the vector database output

- Validating the retrieved context using an LLM
- Correcting and re-ranking the context
- Providing the refined context to the LLM

This structured approach significantly enhances the RAG workflow by ensuring more accurate results while also standardizing the retrieval process.

7. GraphRAG

This chapter introduces the concept of Retrieval-Augmented Generation (RAG) in conjunction with Knowledge Graphs, which serve as structured data representations that capture complex relationships between entities. Knowledge Graphs are particularly advantageous due to their ability to synthesize information and reveal intricate connections between data points, making them an effective tool for enhancing retrieval-based AI models. We begin by exploring the applications of Graph RAG, demonstrating how it improves information retrieval by leveraging structured data. Following this, we delve into the core experiment of this thesis: the implementation of an Agentic RAG system that utilizes a Knowledge Graph to represent relational database schemas. This innovative approach enables efficient graph traversal for SQL query generation, enhancing both the explainability and accuracy of the generated queries.

7.1 Knowledge Graphs and GraphRAG

Another RAG approach that has been lately gaining traction employs the graph based data structures known as Knowledge Graphs. These structures are particularly useful for representing entities as nodes and the relationships between them with edges. These graphs are widely

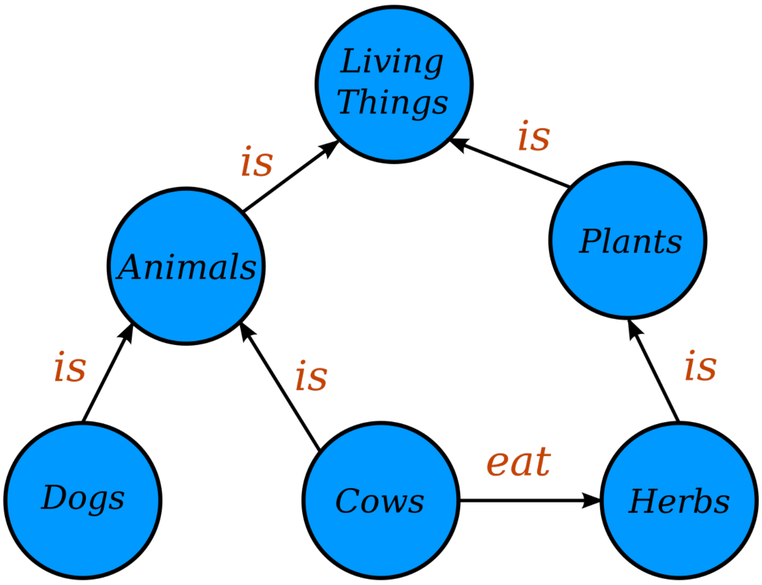


Figure 36: Example of Knowledge Graph. Source: Wikipedia[1]

used in AI-driven applications to enhance contextual understanding and reasoning by encoding explicit relationships between concepts. By structuring information in this way, knowledge graphs enable more sophisticated retrieval mechanisms beyond simple keyword or embedding-based searches.

7.1.1 GraphRAG: A Knowledge Graph-Driven RAG Approach

In the context of Retrieval-Augmented Generation (RAG), knowledge graphs provide an alternative to traditional vector-based retrieval by making us of the more meaningful connections between concepts. While Vector based RAG systems rely on dense vector embeddings to find semantically similar text snippets, while being an effective technique is struggles with the sense making tasks that require understanding the relationships between entities across an entire dataset.

7.1.2 Key Differences Between GraphRAG and Vector RAG

The primary distinction between GraphRAG and Vector RAG lies in their retrieval mechanisms and capacity for global reasoning:

- **Retrieval Mechanism:** Vector RAG uses embedding similarity to find relevant item, whereas GraphRAG navigates structured relationships within a knowledge graph.
- **Sensemaking Ability:** GraphRAG excels at answering global questions that require synthesizing information across multiple documents, while Vector RAG is better suited for fact-based, localized retrieval.
- **Scalability and Efficiency:** While GraphRAG requires an initial indexing phase to construct the knowledge graph, it can efficiently retrieve high-quality summaries for complex queries without retrieving an excessive number of items.

7.1.3 Examples of Graph RAG applications with LangChain

With LangChain, we can provide a robust framework for integrating LLMs with our graph database by dynamically generating Cypher queries based on user questions. LangChain promotes seamless interaction between LLMs and graphs through the following components:

- **Graph Database:** A structured representation of knowledge, such as Neo4j, where entities and relationships are stored.
- **Prompt Engineering:** A carefully crafted prompt template to guide the LLM in gen-

[1] `https://en.wikipedia.org/wiki/Knowledge_graph`

erating precise Cypher queries.

- **Query Execution:** The generated Cypher query is executed against the Knowledge Graph to retrieve relevant information.
- **Response Synthesis:** The retrieved data is processed and formatted into a natural language response by the LLM.

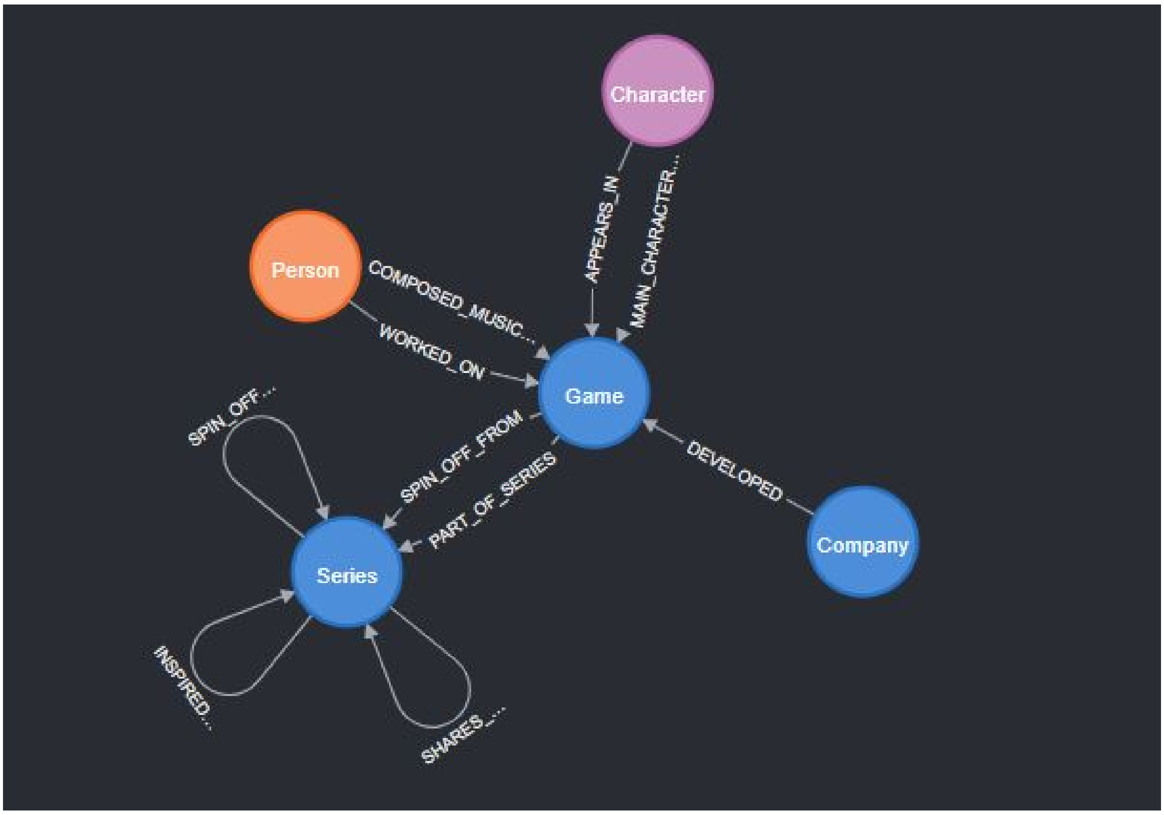


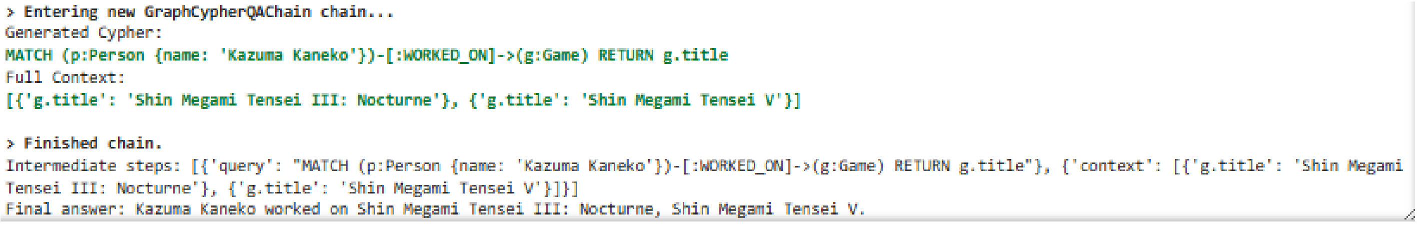
Figure 37: KG schema visualized with APOC

In the figure above, we present a schema of our graph, which describes the entities related to game companies, the series produced, the people involved in game development, and even the characters. This serves as a heterogeneous knowledge base containing different types of entities connected to one another.

For example, we can ask the LLM, *"Which games has Kazuma Kaneko worked on?"* The chain responds with: *"Kazuma Kaneko worked on Shin Megami Tensei III: Nocturne, Shin Megami Tensei V."*

In the figure below, we illustrate the steps taken by LangChain:

Another angle worth exploring is the one about processing unstructured data into a Knowledge Graph. This way we can get the advantages of structured data such as finding complex rela-

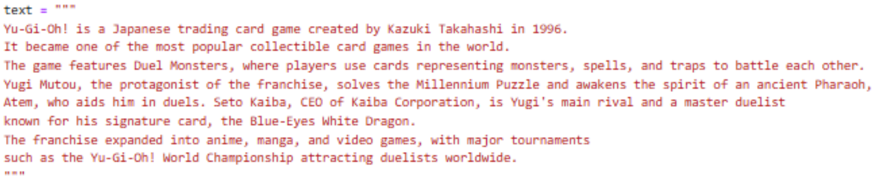


```
> Entering new GraphCypherQAChain chain...
Generated Cypher:
MATCH (p:Person {name: 'Kazuma Kaneko'})-[:WORKED_ON]->(g:Game) RETURN g.title
Full Context:
[{'g.title': 'Shin Megami Tensei III: Nocturne'}, {'g.title': 'Shin Megami Tensei V'}]

> Finished chain.
Intermediate steps: [{'query': "MATCH (p:Person {name: 'Kazuma Kaneko'})-[:WORKED_ON]->(g:Game) RETURN g.title"}, {'context': [{'g.title': 'Shin Megami
Tensei III: Nocturne'}, {'g.title': 'Shin Megami Tensei V'}]}]
Final answer: Kazuma Kaneko worked on Shin Megami Tensei III: Nocturne, Shin Megami Tensei V.
```

Figure 38: LangChain steps

tionships and patterns between entities, by starting from just a corpus of unstructured data. To construct this Knowledge Graph, LangChain leverages the use of an LLM that parses the text and categorizes the entities and their relationships. In the images below we can see the text corpus and the resulting schema of our generated Knowledge Graph.



```
text = """
Yu-Gi-Oh! is a Japanese trading card game created by Kazuki Takahashi in 1996.
It became one of the most popular collectible card games in the world.
The game features Duel Monsters, where players use cards representing monsters, spells, and traps to battle each other.
Yugi Mutou, the protagonist of the franchise, solves the Millennium Puzzle and awakens the spirit of an ancient Pharaoh,
Atem, who aids him in duels. Seto Kaiba, CEO of Kaiba Corporation, is Yugi's main rival and a master duelist
known for his signature card, the Blue-Eyes White Dragon.
The franchise expanded into anime, manga, and video games, with major tournaments
such as the Yu-Gi-Oh! World Championship attracting duelists worldwide.
"""
```

Figure 39: Text corpus

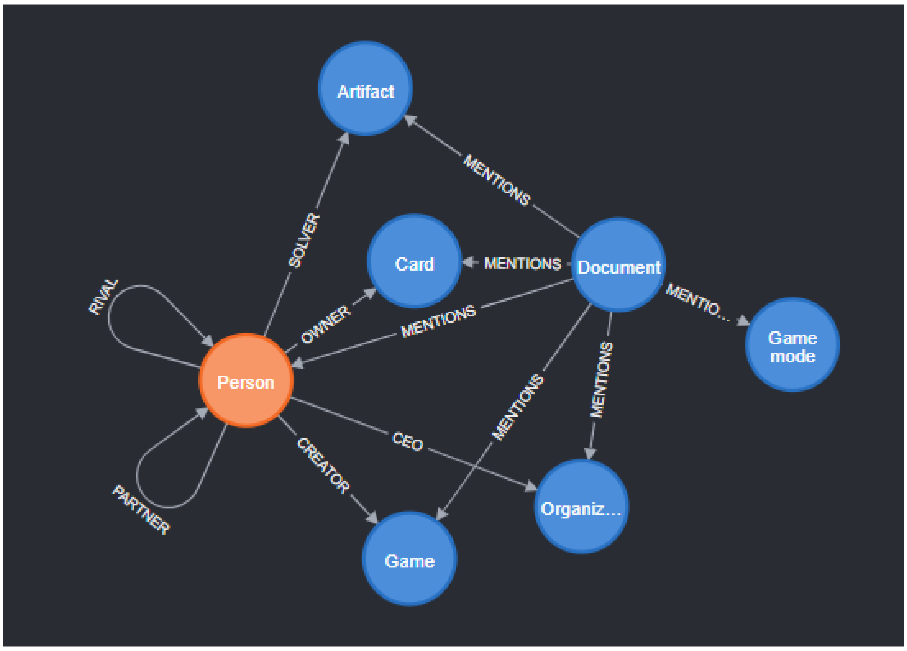


Figure 40: Generated Knowledge Graph

As demonstrated, LangChain effectively generates precise Cypher queries and retrieves relevant context from the Knowledge Graph to construct accurate responses. It is capable also of constructing Knowledge Graphs from unstructured data such as documents or corpus of text.

However, while this approach is beneficial, it has limitations in handling more complex queries, and the still experimental nature of graph building dedicated LLMs, make it less suitable for production environments. A more adaptable solution for implementing Graph RAG is to utilize LangGraph, which enables greater control over workflow execution and allows for the incorporation of additional steps to improve response accuracy and handling of intricate queries.

7.2 Experimental SQL generation with Knowledge Graphs

In this final section, we present an experiment that integrates the theoretical principles discussed throughout this thesis. The experiment explores the use of Knowledge Graphs in generating SQL queries from natural language prompts. The workflow is built around LangGraph, which systematically processes natural language input to generate SQL queries. The process begins by constructing a Knowledge Graph from the schema of two CSV tables retrieved from the OpenData platform, which we will discuss shortly. This schema is represented as a Knowledge Graph in Neo4j, providing a structured understanding of the data relationships. LangGraph leverages a Large Language Model (LLM) to generate a Cypher traversal query, establishing connections between the two tables. The generated query then undergoes validation, ensuring it meets syntax requirements and other adequacy criteria before final execution.

7.2.1 Workflow

The general workflow of the architecture follows these steps:

1. **Schema Analysis and Knowledge Graph (KG) Building**: The CSV files are loaded, and a knowledge graph is constructed to represent the schema of the tables.
2. **Vectorization and Storage**: Column labels are vectorized and stored in a vector database, such as Weaviate, using Ollama.
3. **Entity Matching**: A two-step matching process is performed:
    - *Label-Based Search*: Using cosine similarity to find relevant columns.
    - *Value-Based Matching*: A customized scoring function evaluates whether columns can be joined based on their values.
4. **Schema Filtering**: To improve Cypher query generation accuracy, schema elements irrelevant to the natural language question are filtered out. Only tables and relationships that exceed a predefined threshold are retained.

5. **Question Enrichment**: The natural language question is enriched with schema labels, reducing ambiguity and improving the mapping between the question and the schema for Cypher query generation.
6. **Cypher Query Generation**: A Cypher query is generated based on the enriched question and filtered schema.
7. **SQL Generation**: Finally, an SQL query is generated from the Cypher query.

7.2.2 Motivations

Our experiment addresses the challenge of querying structured databases using natural language. Traditional database queries require structured syntax, such as SQL, which must adhere to the database schema and its constraints. However, users may not have full knowledge of the database structure—only a general understanding of the domain. Even with SQL expertise, writing accurate queries can be time-consuming and costly.

To overcome these limitations, we propose a scalable framework that enables users to query databases using natural language while maintaining accuracy comparable to that of a data expert. This approach simplifies database interactions, reduces the need for deep schema knowledge, and enhances efficiency.

7.2.3 Knowledge Graph Construction

The **Knowledge Graph** serves as a structured data representation that captures entities and their relationships. The first step in the process is constructing the Knowledge Graph, where **tables and columns** are represented as **nodes**, and their relationships are defined by **edges**. Specifically, edges emphasize the association between columns and the tables they belong to.

Additionally, column nodes include metadata such as:

- **Uniqueness** – to determine whether a column serves as an identifier.
- **Data type** – to ensure strict matching between columns of the same type, with the exception of cases where **float and integer** types may be considered compatible.

In the **Neo4j** graph, the possible relationships include:

- **"HAS COLUMN"** – an edge connecting a **table node** to its **column nodes**.
- **"SAME ATTRIBUTE"** – an edge linking **column nodes** that share the same label and metadata. However, if two columns match perfectly, instead of creating a relationship, they are **collapsed into a single node** to maintain data integrity.

By leveraging this **Knowledge Graph**, we enable traversals between columns that effectively represent **joins** between relational database tables.

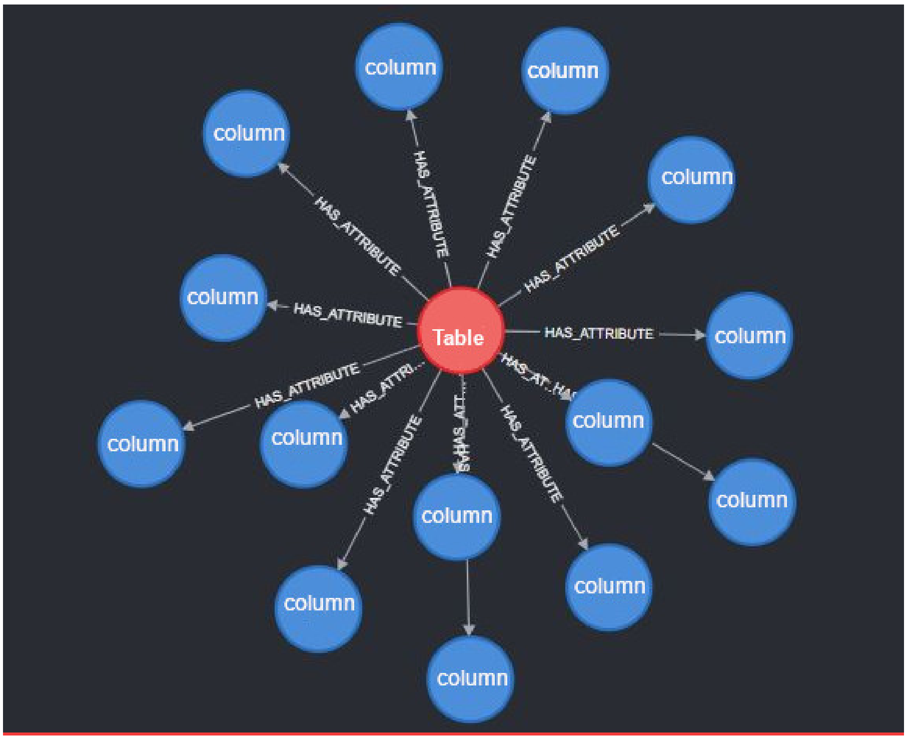


Figure 41: Example of generated Knowledge Graph

7.2.4 Entity Matching

After constructing the Knowledge Graph, we also vectorize the column labels and store them in a Vector Database. For this, we use **Weaviate**, chosen for its lightweight processing, flexibility in vectorization, and ease of integration across platforms—thanks to its official **Docker** container. Weaviate supports multiple vectorization methods, and for this use case, we employ an **Ollama** container running the **snowflake-arctic-embed2** model, which provides fast and efficient lightweight vectorization. To store the vector embeddings, we use **Weaviate's default flat index**, as the dataset contains only a few hundred embeddings. In this case, a flat search offers similar performance to **HNSW**, making it a practical choice for simplicity and efficiency.

Once the vector store is built, entity matching begins. Each column label undergoes a **vector search**, retrieving candidates with a similarity score above a predefined threshold. If a match meets this criterion, we perform a **value-based scoring** step to determine whether the columns are suitable for joining. A good join key should:

1. **Have similar content** – The values should be closely related or directly matching.

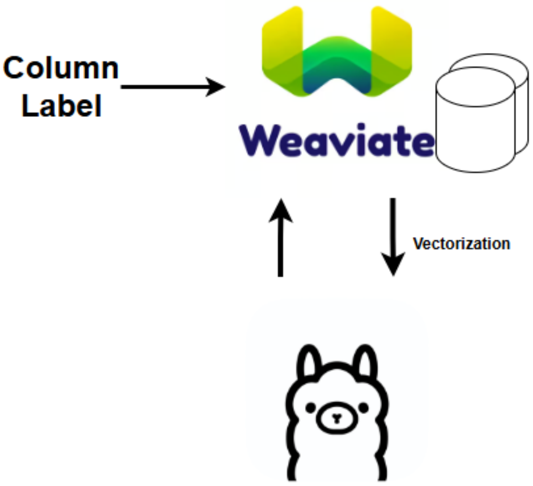


Figure 42: Column label vectorization

2. **Exhibit diversity** – Columns with excessive repetition (e.g., "Yes"/"No" or a single repeated category) provide little value in a join.
3. **Share overlapping unique values** – Columns with entirely different value distributions are less useful for meaningful joins.

The scoring process is based on a **TF-IDF vectorizer** provided by the `scikit-learn` library. The steps involved are as follows:

1. The values within each column are vectorized using the TF-IDF method.
2. A similarity matrix is computed, and the diagonal elements are extracted to calculate the mean similarity score.
3. A penalty factor is applied to columns with a low diversity of unique values, reducing their overall score.
4. The adjusted similarity score is compared against a predefined threshold to determine if the columns should be considered a match.

If the similarity score exceeds the threshold, the Knowledge Graph is updated accordingly. The matched columns are linked via an edge labeled **"SAME ATTRIBUTE"**, which stores the computed similarity score as an edge property.

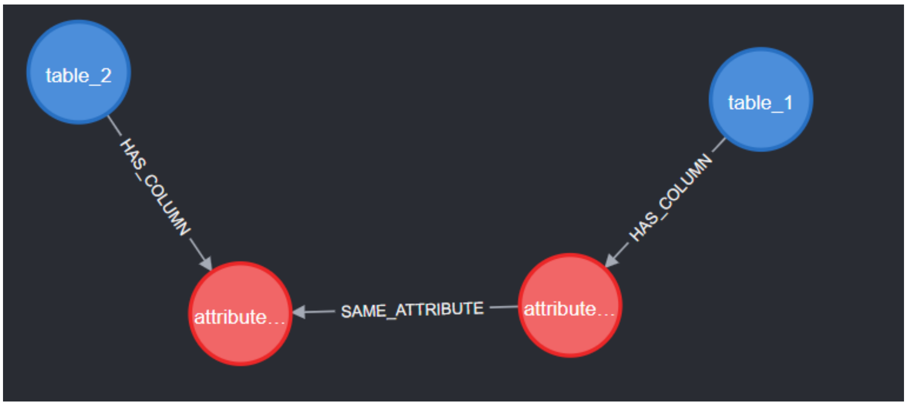


Figure 43: Example of matching nodes in the Knowledge Graph

7.2.5 Schema Filtering and Question Enrichment

After building the Knowledge Graph and Matching the entities, to enhance the accuracy of Cypher query generation, we introduce a two-step process: *Schema Filtering* and *Question Enrichment*.

7.2.5.1 Schema Filtering

In this step, we refine the schema by filtering out elements that are not relevant to the given natural language question. Only tables and their traversals that meet a predefined relevance threshold are retained. This reduces noise and ensures that the generated Cypher query is based on the most pertinent schema components.The schema provides also similarity scores between nodes, this allows us to discern which column is needed to ensure the join statement.

7.2.5.2 Question Enrichment

After schema filtering, the natural language question is enriched by incorporating schema labels and relationships between nodes. This step strengthens the connection between the question and the schema, reducing ambiguity and improving the precision of Cypher query generation. The expected resulting enriched prompt will contain the labels of the entities while still maintaining the sense of the original prompt.

7.2.6 Cypher Generation with LangGraph Sub-Graphs

For Cypher query generation, we leverage LangGraph's support for sub-graphs, enabling the construction of larger and more complex graphs in a modular manner.

The *Cypher Generation* sub-graph begins with the provided enriched prompt and schema,

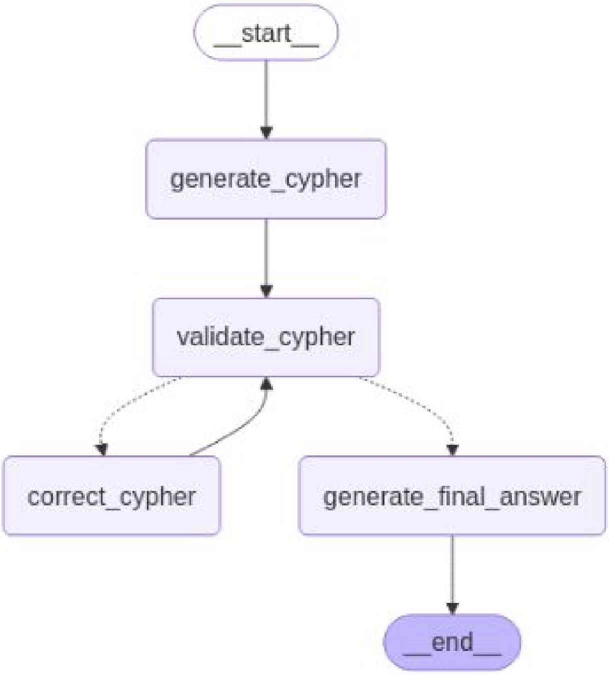


Figure 44: Cypher Generation

generating the initial Cypher traversal. Once the traversal is created, an iterative process of validation and correction is employed to ensure the correctness of the query.

To verify syntax accuracy, the query is executed using the `EXPLAIN` clause, which helps detect errors. Additionally, the process includes validating the existence of specific edges to ensure the correct generation of the Cypher query. Once validated, the final query is output to the main graph for further processing.

7.2.7 SQL Translation from Cypher Traversal

Once the Cypher query is generated, it provides a traversal that links table nodes through common neighbors, effectively representing the joining columns. This structured traversal serves as a bridge between the graph representation and relational databases, facilitating a seamless translation into SQL.

The SQL translation process closely mirrors the Cypher generation workflow, ensuring correctness through an iterative validation approach. Since the Cypher query already encodes the necessary relationships and joins, the translation primarily involves mapping graph-based traversals into equivalent SQL join operations.

To guarantee syntactic accuracy, an iterative validation process is applied, similar to the validation performed during Cypher generation. This includes executing preliminary checks and leveraging SQL execution plans to detect potential syntax errors or logical inconsistencies. Any detected issues prompt refinements to the query before finalizing the SQL output.

This structured approach ensures that the generated SQL query accurately reflects the intended data retrieval logic while maintaining correctness and efficiency.

7.2.8 Execution over UK and Canadian OpenData

After detailing the entire process, this section demonstrates its application to OpenData provided by Canada and the UK using SPARQL and CKAN. The UK dataset involves combining two CSV tables: one containing indicators of knee pain and the other related to varicose vein conditions. Once the Knowledge Graph (KG) is constructed, we observe that the provided code achieves the highest similarity score when joining tables within the filtered schema.

7.2.8.0.1 Nodes Representing Columns:

- **Procedure**
  - `data_type`: string
  - `is_unique`: `False`
  - `value`
  - `similarity`: 0.9566
- **Year**
  - `data_type`: string
  - `is_unique`: `False`
  - `value`
  - `similarity`: 0
- **AgeBand**
  - `data_type`: string
  - `is_unique`: `False`
  - `value`
  - `similarity`: -2.6154
- **ProviderCode**

  - `data_type`: string
  - `is_unique`: `False`
  - `value`
  - `similarity`: 3.2149

- **PostOpQReadmitted**
  - `data_type`: int
  - `is_unique`: `False`
  - `value`
  - `similarity`: 1.9546

- **VaricoseVeinPostOpQLeftFrontCount**
  - `data_type`: int
  - `is_unique`: `False`
  - `value`

- **KneeReplacementPostOpQReadmitted**
  - `data_type`: int
  - `is_unique`: `False`
  - `value`

- **KneeReplacementOKSPostOpQPredicted**
  - `data_type`: float
  - `is_unique`: `False`
  - `value`

7.2.8.0.2 Question Refinement:

Initially, the query posed was: *"What is the average count of varicose vein post-op questions left front for knee replacement patients readmitted to hospital, by age band and provider code?"*

The refined question, incorporating schema labels, is:

"What is the average `VaricoseVeinPostOpQLeftFrontCount` for `Procedure` 'Knee Replace-

ment' patients with ***`PostOpQReadmitted`*** *greater than 0, grouped by* ***`AgeBand`*** *and* ***`ProviderCode`****?"*

This enrichment ensures greater specificity by aligning the query with structured schema elements.

7.2.8.0.3 Traversal and SQL Conversion: The Cypher traversal identifies the most optimal path between tables based on similarity scores, determining that the best route is through the `ProviderCode` node. The Cypher Traversal goes as follows:

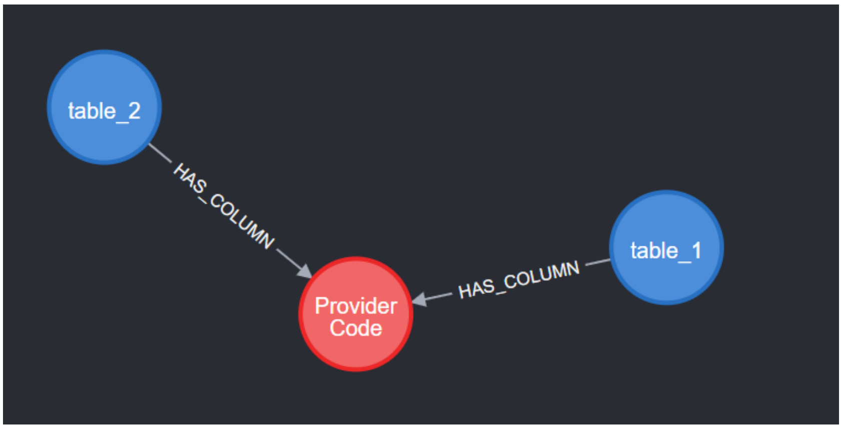


Figure 45: Generated Traversal

This traversal is then converted into an equivalent SQL query:

```
SELECT T1.AgeBand, T1.ProviderCode,
       AVG(T1.VaricoseVeinPostOpQLeftFrontCount) AS average_count
FROM Table1 T1
JOIN Table2 T2
ON T1.ProviderCode = T2.ProviderCode AND T1.AgeBand = T2.AgeBand
WHERE T2.PostOpQReadmitted IN (1, ’yes’, TRUE, ’Yes’)
      AND T1.PostOpQReadmitted IN (1, ’yes’, TRUE, ’Yes’)
GROUP BY T1.AgeBand, T1.ProviderCode;
```

7.2.8.0.4 Application to Canadian OpenData: For the query: *"Get analysis types, values, and product types for Canadian samples"*

The refined question becomes: *"What are the* ***`AnalysistypeTypedanalyse`****,* ***`ValuetextValeurtextuelle`****, and* ***`ProducttypeTypedeproduit`*** *for samples from* ***`CountryoforiginPaysdorigine`*** *Canada?"*

This results in the following SQL query:

```
SELECT DISTINCT
    t1.AnalysistypeTypedanalyse AS analysis_type,
    t1.ValuetextValeurtextuelle AS value,
    t1.ProducttypeTypedeproduit AS product_type,
    t2.AnalysistypeTypedanalyse AS analysis_type2,
    t2.ValuetextValeurtextuelle AS value2,
    t2.ProducttypeTypedeproduit AS product_type2
FROM Table1 t1
INNER JOIN Table2 t2
ON t1.CountryoforiginPaysdorigine = t2.CountryoforiginPaysdorigine
WHERE LOWER(t1.CountryoforiginPaysdorigine) = LOWER('Canada')
      AND LOWER(t2.CountryoforiginPaysdorigine) = LOWER('Canada');
```

These examples illustrate how our methodology effectively refines user queries and generates precise SQL queries by leveraging Knowledge Graph-based enrichment.

7.3 Join Discovery with a Knowledge Graph of N Tables

Building upon the previously introduced job pipeline, we extend the process from joining two tables to handling a more complex use case involving N tables. The goal is to determine the necessary relational tables required for a given query.

7.3.1 Metadata Integration in the KG construction

In this augmentation we keep the Knowledge Graph construction process mostly unchanged, aside from an additional step before the construction, which is fetching a sample of each CSV and instruct the LLM or a human operator to provide a brief description of what the relational table represents.This process is simple and straightforward making the difference between the performances of a human operator and a LLM minimal.

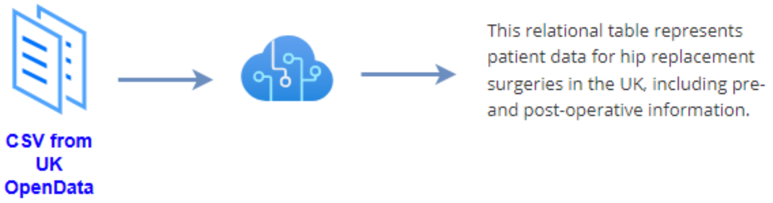

CSV from UK OpenData
This relational table represents patient data for hip replacement surgeries in the UK, including pre- and post-operative information.

Figure 46: Example of an LLM's description of a Medical CSV

While the graph construction process remains almost unchanged, the primary modification occurs in the querying phase.

7.3.2 Clustering in Knowledge Graph Representation

As the Knowledge Graph representation grows in complexity, clustering becomes essential. This preliminary phase organizes the graph into clusters of N nodes, where each node represents a relational table. Nodes are connected if they share one or more common elements. By providing the LLM with the schema of the clustered Knowledge Graph, we can efficiently identify the most relevant nodes for a given query. This approach significantly reduces the graph's size while maintaining performance comparable to the previous case involving only two tables. This approach allows for the system to scale very well as it greatly reduces the surface of the Knowledge Graph to explore, and can be stored in a persistence layer for later use, and gets reprocessed when a new CSV is fetched in our database still keeping a linear growth of complexity.

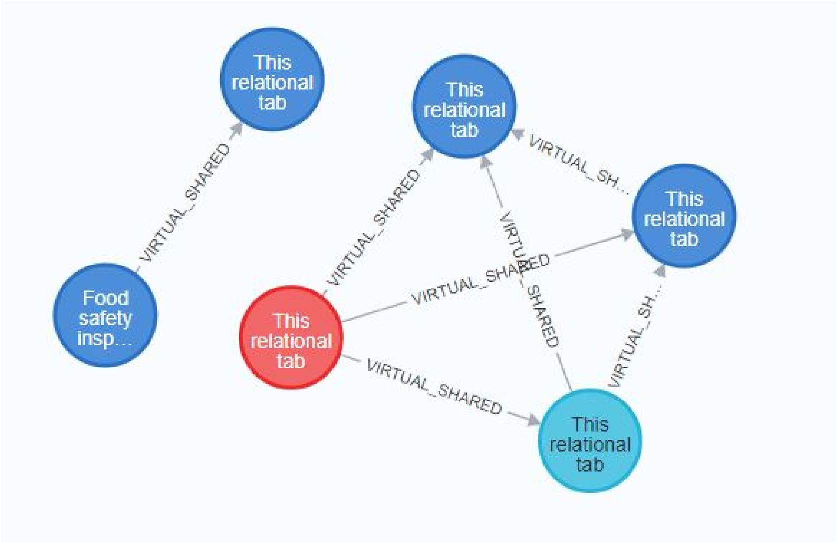


Figure 47: Virtual Graph

7.4 Potential Improvements

While the experiment establishes a standardized pipeline for generating structured SQL queries, several improvements can enhance its efficiency and adaptability:

- **Parallelization of LLM Prompting**: Implementing parallel processing can significantly speed up query generation by handling multiple prompts simultaneously.
- **Federated Graph**: Using federated graph may allow to parallelize our system and scale

it horizontally, with replication and also by keeping unrelated tables separated, removing unnecessary complexity from the workflow.

- **Automatic Language Detection**: Since the data sources primarily use English, querying non-English sources introduces additional complexity for the LLM in understanding both the prompt and schema. To address this, an automated language detection mechanism should be implemented. This process would:

  1. Identify the language of the Open Data source.
  2. Translate the user prompt into the detected language.
  3. Generate the query in the appropriate language.
  4. Translate the final result back into the original language for consistency.

- **Index vector store**: As the vector store increases in size the employment of specialized scalable indexes like HNSW may become more suitable, and it's worth to consider to use ACORN filtering strategy due to the correlation of the object and the querying predicate, also vector quantization may be useful in future uses.

- **Automatically generated labels**: One challenge that must be tackled is the presence of automatically generated labels, which reduce significantly the sense making capabilities of the Knowledge Graph, for future applications this challenge is one of the main difficulties that must be dealt with. One possible approach is to employ an Hybrid matching between label and value based matching, between columns of matching data to improve robustness of the matching. Another possible approach is to employ a human operator to assess matching when the uncertainty is high during the matching task.

- **Security and query sanitation**: Security when it comes from fetching data from outside sources is always a concern to consider head on, API key must be stored in secure locations and not be put into the code or even an environment variable, if we're using containers the use of secrets is warranted. We make sure that the generated queries are sanitized and not executing an injection or other unauthorized operations. We analyze the data in order to assess that it's not tampered with, we validate and clean incoming data.

- **Keyword Extraction with CKAN Integration**: The system begins by extracting high-level keywords from the user's natural language query. These keywords are used to query the CKAN open data portal via its API, leveraging available metadata for real-time dataset retrieval without prior ingestion or full indexing. If the initial keyword set is insufficient, the system iteratively refines the keywords to better match the query's intent. Retrieved tables and metadata are stored in a structured dictionary for use in downstream processing.

These enhancements would improve both the performance and versatility of the system when handling diverse datasets.

7.5 Conclusions

In this study, we explored the integration of Knowledge Graphs into Retrieval-Augmented Generation (RAG) to enhance the generation of SQL queries from natural language prompts. By leveraging structured relationships within a Knowledge Graph, GraphRAG provides a robust alternative to traditional vector-based retrieval methods, enabling more effective sensemaking and complex query generation. Unlike purely embedding-based retrieval approaches, which rely on semantic similarity and may struggle with structural dependencies, our framework introduces explicit relational reasoning, leading to more accurate and contextually relevant SQL query construction.

Our experimental workflow demonstrated the feasibility of using LangGraph and Neo4j to systematically translate user queries into Cypher and SQL, ensuring both accuracy and efficiency. We implemented a hybrid retrieval strategy that combines entity matching through vector embeddings with value-based scoring, enabling more refined schema selection and join identification. This approach enhances the system's ability to handle ambiguous or incomplete user queries by leveraging explicit graph relationships to infer missing connections. Furthermore, the multi-stage query refinement process allows for incremental improvements, ensuring that the generated SQL statements align closely with the intended query semantics.

The evaluation of our framework over UK and Canadian OpenData datasets highlighted its effectiveness in translating natural language into precise SQL queries. We observed that the structured representation within the Knowledge Graph significantly improved the model's ability to generate meaningful queries in complex database schemas, particularly in scenarios where conventional keyword-based retrieval failed to capture intricate table relationships. The introduction of Cypher-based retrieval prior to SQL generation facilitated improved schema sensemaking, reducing the occurrence of incorrect table joins and mismatched column references.

While the current implementation effectively bridges natural language and structured data retrieval, several areas for future improvement remain. One key enhancement would be the parallelization of LLM prompting to reduce processing latency while maintaining high accuracy. Additionally, incorporating automatic language detection for multilingual query support could broaden the system's applicability across diverse datasets. Further optimization of schema filtering techniques, including dynamic weighting of database elements based on query context, could further improve retrieval efficiency. Another promising avenue for exploration is the use of reinforcement learning or feedback-driven fine-tuning to iteratively enhance query generation based on user corrections and real-world usage patterns.

Overall, this work contributes to the growing field of knowledge-driven query generation, demonstrating the potential of combining Knowledge Graphs and LLMs to simplify database interactions while maintaining expert-level accuracy. By integrating structured knowledge into the retrieval-augmented generation process, we enable more intelligent query formulation that extends beyond simple keyword matching. As AI-driven database querying continues to evolve, the fusion of structured and unstructured retrieval approaches, as demonstrated in this study, offers a promising path toward more intuitive and accessible data exploration tools.

Bibliography

References for Chapter 1

[1] DeepLearning.AI, *Vector Databases and Embeddings Applications*, 2024. Available: https://www.deeplearning.ai/short-courses/vector-databases-embeddings-applications/.

[2] DeepLearning.AI, *Retrieval Optimization: From Tokenization to Vector Quantization*, 2024. Available: https://www.deeplearning.ai/short-courses/retrieval-optimization-from-tokenization-to-vector-quantization/.

[3] Elastic, *What is Vector Embedding?*, 2024. Available: https://www.elastic.co/what-is/vector-embedding.

[4] Pinecone, *Dense Vector Embeddings in NLP*, 2024. Available: https://www.pinecone.io/learn/series/nlp/dense-vector-embeddings-nlp/.

[5] LinkedIn, *Understanding Differences Between Encoding and Embedding*, 2024. Available: https://www.linkedin.com/pulse/understanding-differences-between-encoding-embedding-mba-ms-phd/.

[6] LevelUp, *6 Resources to Master Vector Databases and Building a Vector Storage*, 2024. Available: https://levelup.gitconnected.com/6-resources-to-master-vector-databases-building-a-vector-storage-8d94ca1e3897.

[7] Tech Padawan Chronicles, *Vector Search: Unlocking the Power of Unstructured Data*, 2024. Available: https://medium.com/tech-padawan-chronicles/vector-search-unlocking-the-power-of-unstructured-data-ecf7c8ce91f6.

[8] UBIAI NLP, *What is Unstructured Data? What Issues Can It Bring to Deep Learning Models in 2024?*, 2024. Available: https://medium.com/ubiai-nlp/what-is-unstructured-data-what-issues-can-bring-to-the-deep-learning-model-in-2024-

References for Chapter 2

[9] MyScale, *Understanding Vector Indexing: A Comprehensive Guide*, 2024. Available: https://medium.com/@myscale/understanding-vector-indexing-a-comprehensive-guide-d1abe36ccd3c.

[10] Freedium, *Overview of a Neural Network's Learning Process*, 2024. Available: https://freedium.cfd/https://medium.com/data-science-365/overview-of-a-neural-networks-learning-process-61690a502fa.

[11] DeepLearning.AI, *Vector Databases and Embeddings Applications*, 2024. Available: https://www.deeplearning.ai/short-courses/vector-databases-embeddings-applications/.

[12] Anish Nama, *Exploring the Power of Encoder-Decoder Models: Pros, Cons, and Applications*, 2024. Available: https://medium.com/@anishnama20/exploring-the-power-of-encoder-decoder-models-pros-cons-and-applications-8bfbe2e66e

[13] Analytics Vidhya, *Neural Networks in a Nutshell*, 2024. Available: https://medium.com/analytics-vidhya/neural-networks-in-a-nutshell-bb013f40197d.

[14] BuiltIn, *Transformer Neural Network*, 2024. Available: https://builtin.com/artificial-intelligence/transformer-neural-network.

[15] DataCamp, *The Top 5 Vector Databases*, 2024. Available: https://www.datacamp.com/blog/the-top-5-vector-databases.

[16] Florian Schroff, Dmitry Kalenichenko, and James Philbin, *FaceNet: A Unified Embedding for Face Recognition and Clustering*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015. Available: https://arxiv.org/abs/1503.03832.

[17] Jeff Johnson, Matthijs Douze, and Hervé Jégou, *Billion-scale similarity search with GPUs*, arXiv preprint arXiv:1702.08734, 2017. Available: https://arxiv.org/abs/1702.08734.

[18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeffrey Dean, *Efficient Estimation of Word Representations in Vector Space*, arXiv preprint arXiv:1301.3781, 2013. Available: https://arxiv.org/abs/1301.3781.

[19] DeepLearning.ai, "Building Multimodal Search and RAG", https://www.deeplearning.

ai/short-courses/building-multimodal-search-and-rag/.

[20] KX Systems, "Guide to Multimodal RAG for Images and Text", https://medium.com/kx-systems/guide-to-multimodal-rag-for-images-and-text-10dab36e3117.

[21] OpenAI Cookbook, "Custom Image Embedding Search", https://cookbook.openai.com/examples/custom_image_embedding_search.

[22] Langchain Blog, "Semi-Structured Multi-Modal RAG", https://blog.langchain.dev/semi-structured-multi-modal-rag/.

[23] Kinomoto, "Convert Images to Embeddings for RAG", https://medium.com/kinomoto-mag/convert-images-to-embeddings-for-rag-40447d16aaf2.

[24] Data and Beyond, "Vector Databases: A Beginner's Guide", https://medium.com/data-and-beyond/vector-databases-a-beginners-guide-b050cbbe9ca0.

[25] arXiv, "Understanding Multimodal Retrieval from an Embedding Perspective", https://arxiv.org/abs/2310.14021.

References for Chapter 3

[26] Pinecone. "Product Quantization." https://www.pinecone.io/learn/series/faiss/product-quantization/.

[27] Towards Data Science. "Similarity Search Product Quantization." https://towardsdatascience.com/similarity-search-product-quantization-b2a1a6397701.

[28] DeepLearning.AI. "Retrieval Optimization: From Tokenization to Vector Quantization." https://learn.deeplearning.ai/courses/retrieval-optimization-from-tokenization-to-vector-quantization/lesson/1/introduction.

[29] Weaviate. "Product Quantization in Vector Databases." https://weaviate.io/developers/weaviate/concepts/vector-quantization#product-quantization.

[30] Qdrant. "Understanding Product Quantization." https://qdrant.tech/articles/product-quantization/.

[31] Weaviate. "ANN Algorithms: Tiles Encoder." https://weaviate.io/blog/ann-algorithms-tiles-enocoder.

[32] Wikipedia. "Quantile." https://en.wikipedia.org/wiki/Quantile.

[33] Wikipedia. "Cumulative Distribution Function." https://en.wikipedia.org/wiki/Cumulative_distribution_function.

[34] Timescale. "HNSW vs. DiskANN: A Comparison." https://www.timescale.com/learn/hnsw-vs-diskann.

[35] Milvus. "DiskANN: An Efficient ANN Index." https://milvus.io/blog/2021-09-24-diskann.md.

References for Chapter 4

[36] Myscale Team. "Understanding Vector Indexing: A Comprehensive Guide." Medium, 2023. https://medium.com/@myscale/understanding-vector-indexing-a-comprehensive-guide-d1abe36ccd3c

[37] Oracle. "Understanding Hierarchical Navigable Small World Indexes." Oracle Database 23c Documentation. https://docs.oracle.com/en/database/oracle/oracle-database/23/vecse/understand-hierarchical-navigable-small-world-indexes.html

[38] Weaviate. "Flat Vector Index." https://weaviate.io/developers/academy/py/vector_index/flat

[39] Pinecone. "Vector Indexes." https://www.pinecone.io/learn/series/faiss/vector-indexes/

[40] G. Mehta. "Understanding Vector DBs Indexing Algorithms." Medium, 2023. https://gagan-mehta.medium.com/understanding-vector-dbs-indexing-algorithms-ce187dca69c2

[41] KDB.ai. "Indexing Basics." https://kdb.ai/learning-hub/articles/indexing-basics/

[42] Pinecone. "Locality-Sensitive Hashing (LSH)." https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/

[43] Pinecone. "Locality-Sensitive Hashing Using Random Projection." https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing-random-project

[44] D. Turnbull. "Implementing Random Projections." SoftwareDoug, 2023. https://softwaredoug.com/blog/2023/08/21/implementing-random-projections

[45] S. Jain. "Similarity Search: kNN Inverted File Index." To-

wards Data Science, 2023. https://towardsdatascience.com/similarity-search-knn-inverted-file-index-7cab80cc0e79

[46] Facebook Research. "Comparison with LSH." FAISS Wiki. https://github.com/facebookresearch/faiss/wiki/Comparison-with-LSH

[47] Pinecone. "Hierarchical Navigable Small Worlds (HNSW)." https://www.pinecone.io/learn/series/faiss/hnsw/

[48] Y. Barash. "Similarity Search Part 4: Hierarchical Navigable Small World (HNSW)." Towards Data Science, 2023. https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37

[49] DataStax. "Hierarchical Navigable Small Worlds (HNSW) Guide." https://www.datastax.com/guides/hierarchical-navigable-small-worlds

[50] DeepLearning.AI. "NSW Graph Overview." https://www.deeplearning.ai/short-courses/vector-databases-embeddings-applications/

[51] M. Aumüller, E. Bernhardsson, A. Faithfull. "ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms." arXiv preprint, 2017. https://arxiv.org/abs/1708.0979

[52] Y. Malkov, D. Yashunin. "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs." IEEE TPAMI, 2020. https://arxiv.org/abs/1603.09320

[53] A. Subramanya, M. Shrivastava, P. Pande, et al. "DiskANN: Fast Accurate Billion-Point Nearest Neighbor Search on a Single Node." Advances in Neural Information Processing Systems (NeurIPS), 2019. https://arxiv.org/abs/1907.03688

[54] A. Jayaram Subramanya, P. Pande, A. Shrivastava. "Vamana: Scalable Nearest Neighbor Graph Construction for High-Dimensional Data." International Conference on Machine Learning (ICML), 2019. https://arxiv.org/abs/1906.12142

[55] Weaviate. "Generative RAG Starter Guide." Available at: https://weaviate.io/developers/weaviate/starter-guides/generative.

References for Chapter 5

[56] "Latest Research on RAG." Available at: https://arxiv.org/pdf/2404.16130.

[57] LangChain. "Introduction to LangChain." Available at: https://python.langchain.com/docs/introduction/.

[58] LangGraph. "Introduction to LangGraph." Available at: https://langchain-ai.github.io/langgraph/tutorials/introduction/.

[59] Weaviate. "What is Agentic RAG?" Available at: https://weaviate.io/blog/what-is-agentic-rag.

[60] Neo4j. "Knowledge Graph vs Vector RAG." Available at: https://neo4j.com/blog/developer/knowledge-graph-vs-vector-rag/.

Acknowledgments

I would like to express my deepest gratitude to my thesis supervisor, Giovanni Simonini, for his invaluable guidance and support throughout the course of my research. His expertise and insightful feedback not only helped shape this thesis but also introduced me to fascinating fields in AI and Data Science. While this research was undoubtedly challenging, his mentorship made it an enriching and enjoyable journey.

I am profoundly and immeasurably grateful to my mother, father, and dear brother for their unwavering support and encouragement throughout my academic career. Their endless sacrifices, unconditional love, and steadfast belief in me have been my greatest source of strength. In times of doubt and difficulty, their guidance and reassurance lifted me up, providing me with the courage to keep pushing forward. Words cannot truly capture how deeply I appreciate their presence in my life, for they have been my foundation, my motivation, and my greatest blessing.

A heartfelt thank you to my friends who embarked on this academic journey alongside me: Arman Arnautović, Daniele Manicardi, Gianluca Mancusi, Vipul Kumar, and Marco Mezzina. The shared struggles, insights, and collaborative efforts were invaluable throughout my undergraduate years. I wouldn't have been able to access the Master's program without them, and for that, I am deeply grateful. I cherish our friendship and the many precious moments we have shared together.

To my incredible group of friends—Samuele Carretti, Loredana Hasani, Valeria Iacomini, Matteo Caccetta, and Francesco Tahiri—thank you for providing a community filled with shared passions and a drive for self-expression. Your companionship has been a source of emotional support, especially in the hardest of times, and I am deeply appreciative of the bond we share. Beyond that, the joy we have found in our shared passions and the unforgettable experiences we have created together have been truly invaluable. I am grateful for each and every moment we have spent together.

A profound thank you to all my friends who have been a welcoming presence in my life, making

every shared moment a joy to experience: Adela Arnautović, Davide Federzoni, Lorenzo Solmi, Marian Dediu, Roberto Terrasi, Alessio Zella, Matteo Vanzini, Marco Difonzo, Matteo Ballotta, Ionut Sanduc, Sara Morandi, Axl Nava, and Manuel Caminiti. Your friendship has enriched my life in countless ways.

I extend my deepest gratitude to my colleagues and friends at the UNIMORE Department of Engineering Enzo Ferrari for their invaluable support and the privilege of working alongside them. Their talent, dedication, and camaraderie have been a constant source of inspiration, shaping both my academic and professional growth. The insightful discussions and shared experiences have made this journey not only rewarding but profoundly formative.

A special thank you to Davide Palma, Sara Pederzoli, Sara Ferrari, Diego Mercoliano, Emiliano Maccaferri, Giacomo Salici, Angelo Mozzillo, Paolo Attardi, Michele Giarletta, Vincenzo Lapadula, Felicia Puzone, Riccardo Santi, Vincenzo Macellaro, Andrea Agguzzoli, Marco Savarese, Carmenia Basile, Francesca Palazzo, Giovanni Malaguti, Biagio Grimolizzi, Stefano Politanó, Marco Pirrelli, Antonio Solida, Elisa Tomisani, Luca Catalano, Valeria Coloca, Manuel Valbusa, Carmine Zaccagnino, Michele Mosca, Giuseppe Orefice, Aslam Kalam, Marco Francesco Sette, Felice Schena, Adeel Aslam, Luca Corrado, Antonio De Blasi, and Gianmarco Lusvardi. Whether our paths have already diverged or will in time, I will always cherish the moments we shared. Every challenge, lesson, and connection has contributed to my growth, and I am grateful for each one.

No matter where life takes you, know that you have my sincere support in all your endeavors. I wish you happiness and success in every pursuit. Though our journeys may lead us in different directions, my appreciation remains steadfast, and perhaps one day, our paths will cross again in unexpected and meaningful ways.

With heartfelt gratitude and best wishes for the future, I thank you all.