Vector databases (Part 2): Understanding their internals

July 9, 2023 15-minute read
general • databases
vector-db

Background 背景

This is the second post in a series on vector databases. As mentioned in part 1 of this series, there’s been a lot of marketing (and unfortunately, hype) related to vector databases in the first half of 2023, and if you’re reading this, you’re likely curious what vector databases are actually doing under the hood, and how search functionality is built on top of efficient vector storage.

这是关于矢量数据库系列文章的第二篇。如本系列第 1 部分所述，2023 年上半年有很多与矢量数据库相关的营销（不幸的是，炒作），如果您正在阅读本文，您可能会好奇矢量数据库实际上在做什么，以及如何在高效的矢量存储之上构建搜索功能。

Why is everybody talking about vector databases these days?

为什么现在每个人都在谈论矢量数据库?

Before going deeper into what vector DBs are, what can explain this frenzy of activity and investment in this space?

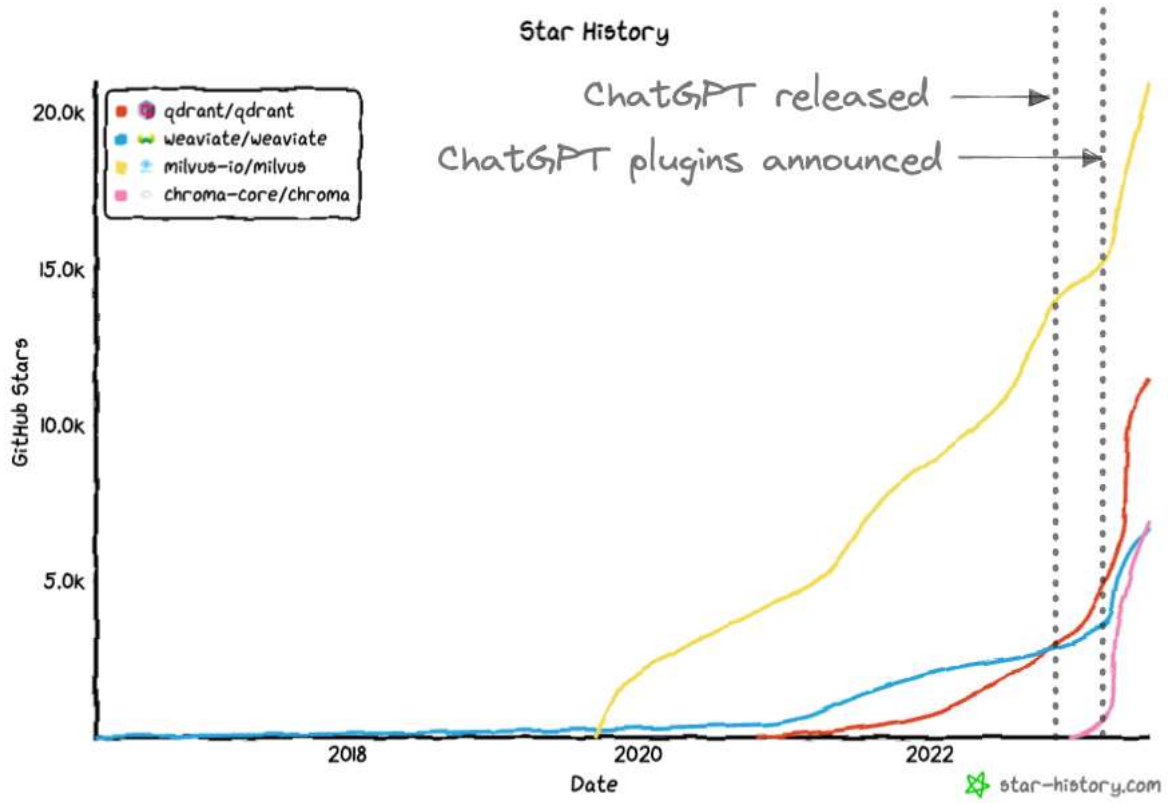
在深入探讨什么是向量数据库之前，什么可以解释这个领域的活动和投资狂潮？

The age of Large Language Models (LLMs)

大型语言模型（LLM）的时代

In November 2022, an early demo of ChatGPT (which is OpenAI's interface to GPT 3.5 and above) was released, following which it quickly became the fastest growing application in history, gaining a million users in just 5 days 🤯! Indeed, if you look at the ✨ history on GitHub for some of the major open-source vector database repos, it's clear that there were sharp spikes in the star counts for some of them *after* the release of ChatGPT in November 2022 and the subsequent release of ChatGPT plugins in March 2023. These factors, and their associated posts in websites like Hacker News and popular media[1], are a large part of why there's been such a lot of activity in this space.

2022 年 11 月，发布了 ChatGPT（这是 OpenAI 与 GPT 3.5 及更高版本的接口）的早期演示，随后它迅速成为历史上增长最快的应用程序，在短短 5 天内🤯获得了一百万用户！事实上，如果你在 GitHub 上查看✨一些主要的开源矢量数据库存储库的历史，很明显，在 2022 年 11 月发布 ChatGPT 和随后于 2023 年 3 月发布 ChatGPT 插件之后，其中一些的星数急剧上升。这些因素，以及它们在Hacker News和流行媒体[1] 等网站上的相关帖子，是为什么在这个领域有这么多活动的很大一部分原因。



Made with ❤️ by star-history.com

由 ❤️ star-history.com 制作

The problem with relying on LLMs

依赖法学硕士的问题

LLMs are *generative*, meaning that they produce meaningful, coherent text in a sequential manner based on a user prompt. However, when using LLMs to answer a human's questions, they often produce irrelevant or factually incorrect results.

LLM是生成性的，这意味着它们根据用户提示以顺序方式生成有意义，连贯的文本。然而，当使用LLM回答人类的问题时，它们经常会产生不相关或事实不正确的结果。

- An LLM often *hallucinates*, i.e., it fabricates information, such as pointing users to URLs or making up numbers that don't exist

  LLM经常产生幻觉，即它捏造信息，例如将用户指向URL或编造不存在的数字

- LLMs learn/memorize a compressed version of their training data, and although they learn quite well, they don't do so *perfectly* – some information is always "lost" in a model's internal representation of the data

  LLM学习/记忆其训练数据的压缩版本，尽管他们学得很好，但他们做得并不完美 - 一些信息总是在模型的内部数据表示中“丢失”

- An LLM cannot know facts that occurred after its training was completed

  法学硕士无法知道其培训完成后发生的事实

Vector databases help address these problems, by functioning as the underlying storage layer that can be efficiently queried by an LLM to retrieve facts. Unlike traditional databases, vector DBs specialize in natively representing data as vectors. As a result, we can now build applications with an LLM sitting on top of a vector storage layer that contains recent, up-to-date, factual data (well past the LLM's training date) and use them to "ground" the model, alleviating the hallucination problem.

矢量数据库通过充当底层存储层来帮助解决这些问题，LLM 可以有效地查询该层以检索事实。与传统数据库不同，向量数据库专门将数据本机表示为向量。因此，我们现在可以使用位于矢量存储层之上的LLM构建应用程序，该存储层包含最近的，最新的，事实数据（远远超过LLM的训练日期），并使用它们来“接地”模型，从而缓解幻觉问题。

Although vector databases (e.g., Vespa, Weaviate, Milvus) have existed well before LLMs, since the release of ChatGPT, the open-source community, as well as marketing teams at the vector DB vendors quickly realized their potential in mainstream use cases like search & retrieval in combination with high-quality text generation. This explains the absolute bonanza of VC funding in the world of vector databases!

尽管矢量数据库（例如Vespa，Weaviate，Milvus）早在LLM之前就已经存在，但自ChatGPT发布以来，开源社区以及矢量数据库供应商的营销团队很快意识到它们在主流用例中的潜力，如搜索和检索与高质量文本生成相结合。这解释了矢量数据库领域风险投资的绝对富矿！

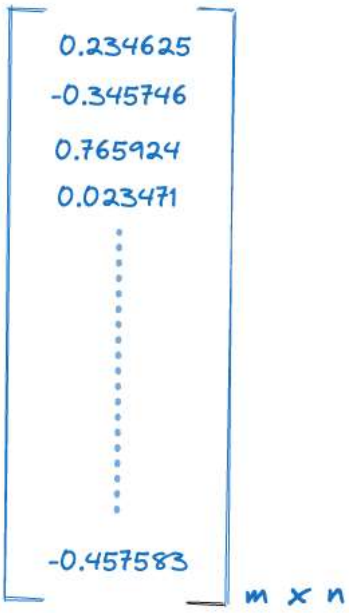
What are embeddings? 什么是嵌入?

A vector database stores not only the original data (which could be images, audio or text), but also its encoded form: *embeddings*. These embeddings are essentially lists of numbers (i.e., vectors) that store contextual representations of the data. Intuitively, when we refer to an "embedding", we are talking about a **compressed**, low-dimensional representation of data (images, text, audio) that actually exists in higher dimensions.

矢量数据库不仅存储原始数据（可以是图像、音频或文本），还存储其编码形式：嵌入。这些嵌入本质上是存储数据上下文表示的数字列表（即向量）。直观地说，当我们提到“嵌入”时，我们谈论的是实际存在于更高维度的数据（图像、文本、音频）的压缩、低维表示。

Within the storage layer, the database stacks m vectors, each representing a data point using n dimensions, for a total size of $m \times n$. The stacks are typically partitioned via sharding for query performance reasons.

在存储层中，数据库堆叠向量，每个 m 向量使用维度表示 n 一个数据点，总大小为 $m \times n$.出于查询性能原因，堆栈通常通过分片进行分区。



How are embeddings generated?
嵌入是如何生成的?

The transformer revolution in NLP[2] has provided engineers with ample means to generate these compressed representations, or embeddings very efficiently, and at scale.

NLP [2] 中的转换器革命为工程师提供了充足的方法来生成这些压缩表示，或者非常有效和大规模地嵌入。

- A popular method is via the open source library `sentence-transformers`, available via the Hugging Face model hub or directly from the source repo

  一种流行的方法是通过开源库 `sentence-transformers` ，可通过拥抱面部模型中心或直接从源存储库获得

- Another (more expensive) method is to use one of many API services:

  另一种（更昂贵）方法是使用许多 API 服务之一：

  - OpenAI embeddings API OpenAI 嵌入 API

- Cohere embeddings API 凝聚嵌入接口

It's important to keep in mind that the lower the dimensionality of the underlying vectors, the more compact the representation is in embedding space, which can affect downstream task quality. Sentence Transformers (sbert) provides embedding models with a dimension n in the range of 384, 512 and 768, and the models are completely free and open-source. OpenAI and Cohere embeddings, which require a paid API call to generate them, can be considered higher quality due to a dimensionality of a few thousand. One reason it makes sense to use a paid API to generate embeddings is if your data is multilingual (Cohere is known to possess high-quality multilingual embedding models that are known to perform better than open source variants).

请务必记住，基础向量的维数越低，嵌入空间中的表示就越紧凑，这可能会影响下游任务质量。句子转换器（sbert）提供了维度在384、512和768 n 范围内的嵌入模型，并且这些模型是完全免费和开源的。OpenAI 和 Cohere 嵌入需要付费 API 调用来生成它们，由于维度为几千，可以被认为是更高的质量。使用付费 API 生成嵌入的一个原因是您的数据是多语言的（众所周知，Cohere 拥有高质量的多语言嵌入模型，已知这些模型的性能优于开源变体）。

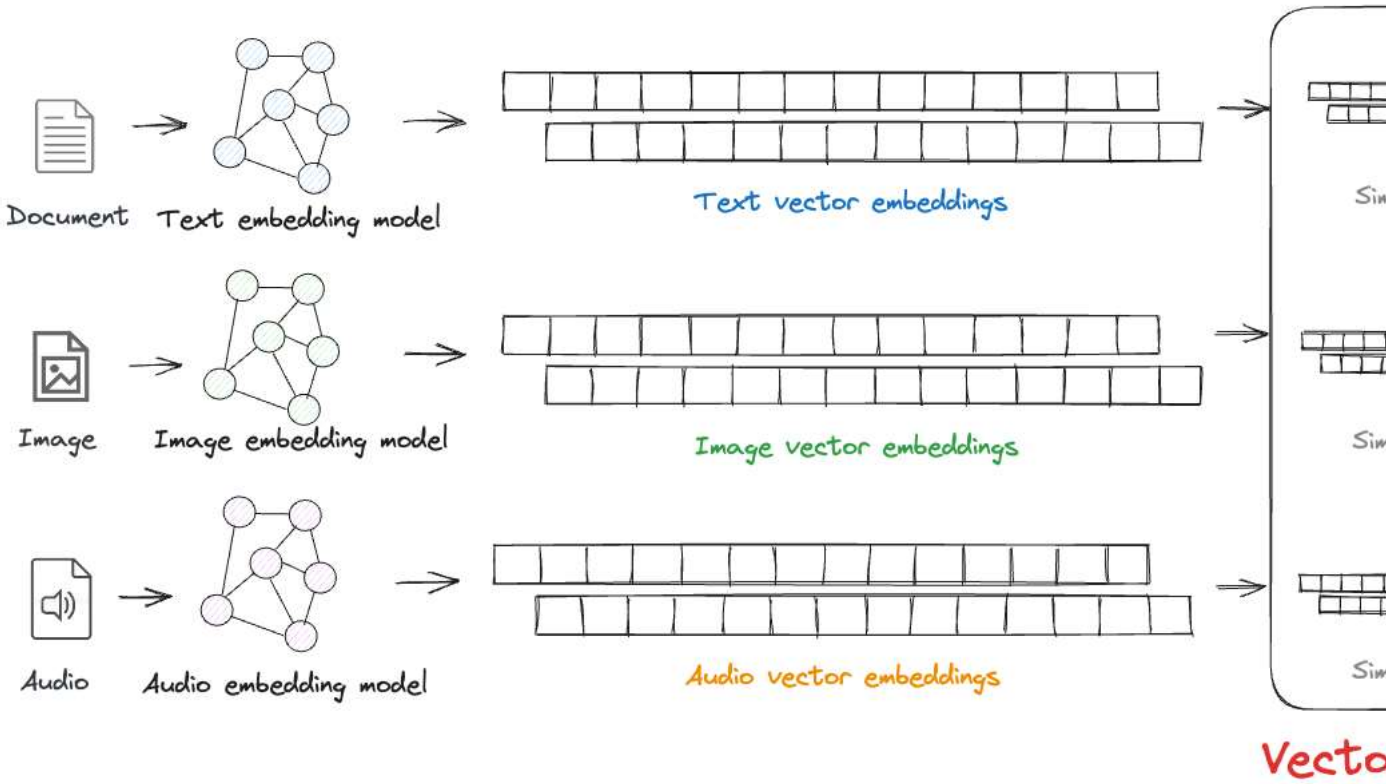
Note 注意
The choice of embedding model is typically a trade-off between quality and cost. In most cases, for textual data in English, the open-source `sentence-transformers` model can be utilized as is for text that isn't too long (~300-400 words for text sequences). It's possible to deal with text that's longer than the context length of `sentence-transformers`, which requires more external tools, but that topic's for another post!

嵌入模型的选择通常是质量和成本之间的权衡。在大多数情况下，对于英语文本数据，可以像使用不太长的文本一样使用开源 `sentence-transformers` 模型（文本序列为 ~300-400 个单词）。可以处理比 的 `sentence-transformers` 上下文长度更长的文本，这需要更多的外部工具，但该主题是另一篇文章！

Storing the embeddings in vector databases
将嵌入存储在向量数据库中

Because of their amenability to operating in embedding space, vector databases are proving to be very useful for *semantic or similarity-based search* for multiple forms of data (text, image, audio). In semantic search, the input query sent by the user (typically in natural language) is translated into vector form, in the same embedding space as the data itself, so that the top-k results that are most similar to the input query are returned. A visualization of this is shown below.

由于矢量数据库在嵌入空间中操作的适应性，它们被证明对于基于语义或相似性的多种形式的数据（文本、图像、音频）搜索非常有用。在语义搜索中，用户发送的输入查询（通常使用自然语言）被翻译成向量形式，与数据本身位于相同的嵌入空间中，以便返回与输入查询最相似的top-k结果。下面显示了对此的可视化。



How is similarity computed?
相似性是如何计算的?

The various vector databases offer different metrics to compute similarity, but for text, the following two metrics are most commonly used:

各种矢量数据库提供不同的指标来计算相似性，但对于文本，最常用的是以下两个指标：

- **Dot product**: This produces a non-normalized value of an arbitrary magnitude

  点积：这将产生任意量级的非归一化值

- **Cosine distance**: This produces a normalized value (between -1 and 1)

  余弦距离：这将产生一个规范化值（介于 -1 和 1 之间）

An example of measuring cosine distance

测量余弦距离的示例

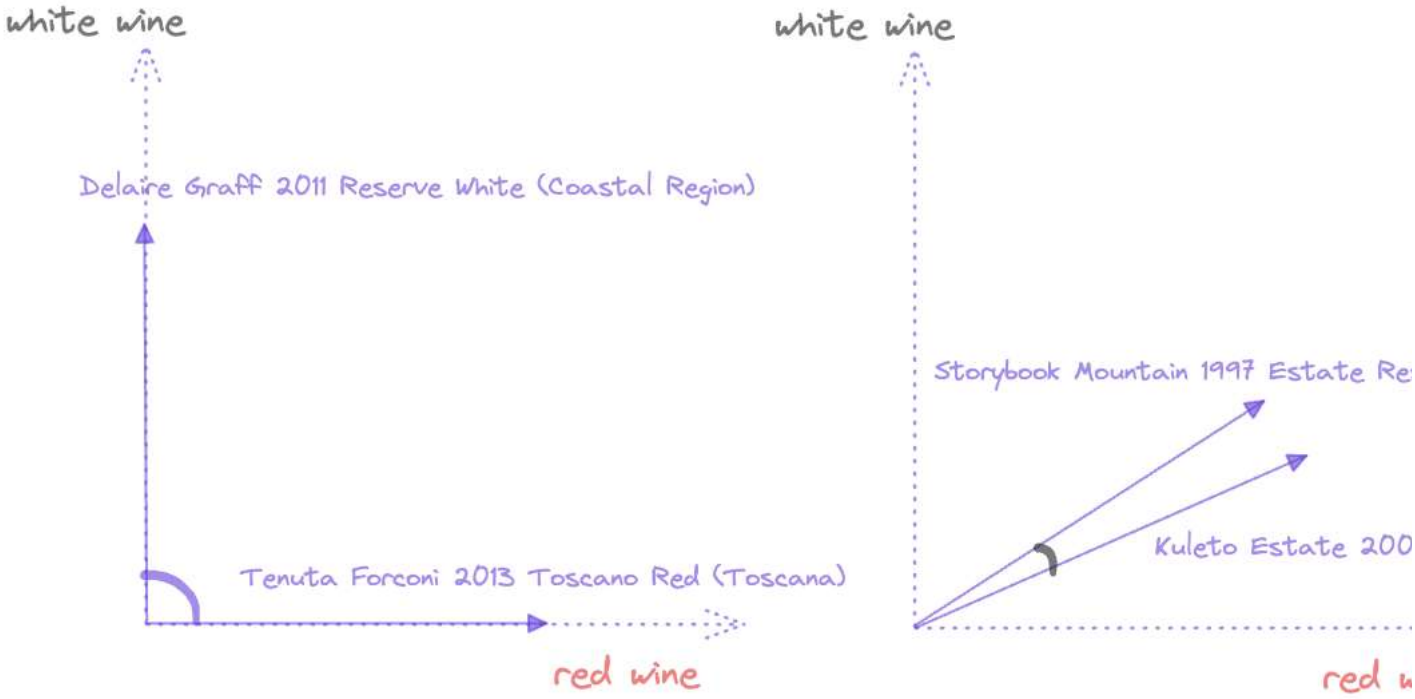
Consider a simplified example where we vectorize the titles of red and white wines in 2-D space, where the horizontal axis represents red wines and the vertical axis represents white wines. In this space, points that are closer together would represent wines that share similar words or concepts, and points that are further apart do not have that much in common. The cosine distance is defined as the cosine of the angle between the lines connecting the position of each wine in the embedding space to the origin.

考虑一个简化的示例，我们在二维空间中矢量化红葡萄酒和白葡萄酒的标题，其中横轴表示红葡萄酒，垂直轴表示白葡萄酒。在这个空间中，距离较近的点将代表具有相似单词或概念的葡萄酒，而相距较远的点则没有那么多共同点。余弦距离定义为连接嵌入空间中每种葡萄酒位置与原产地的线之间角度的余弦。

$$cos\theta = \frac{a^T \cdot b}{|a| \cdot |b|}$$

A visualization will make this more intuitive.

可视化将使这更加直观。



On the left, the two wines (the Reserve White and the Toscana Red) have very little in common, both in terms of vocabulary and concepts, so they have a cosine distance approaching zero (they are orthogonal in the vector space). On the right, the two Zinfandels from Napa Valley have a lot more in common, so they have a much larger cosine similarity closer to 1. The limiting case here would be a cosine distance of 1; a sentence is always perfectly similar to itself.

在左边，两种葡萄酒（储备白葡萄酒和托斯卡纳红葡萄酒）在词汇和概念方面几乎没有共同点，因此它们的余弦距离接近零（它们在向量空间中正交）。在右边，来自纳帕谷的两个仙粉黛有更多的共同点，所以它们的余弦相似性要大得多，接近1。这里的极限情况是余弦距离为 1;一个句子总是与它自己完全相似。

Of course, in a real situation, the actual data exists in higher dimensional vector space (and has many more axes than just the variety of wine) and cannot be visualized on a 2-D plane, but the same principle of cosine similarity applies.

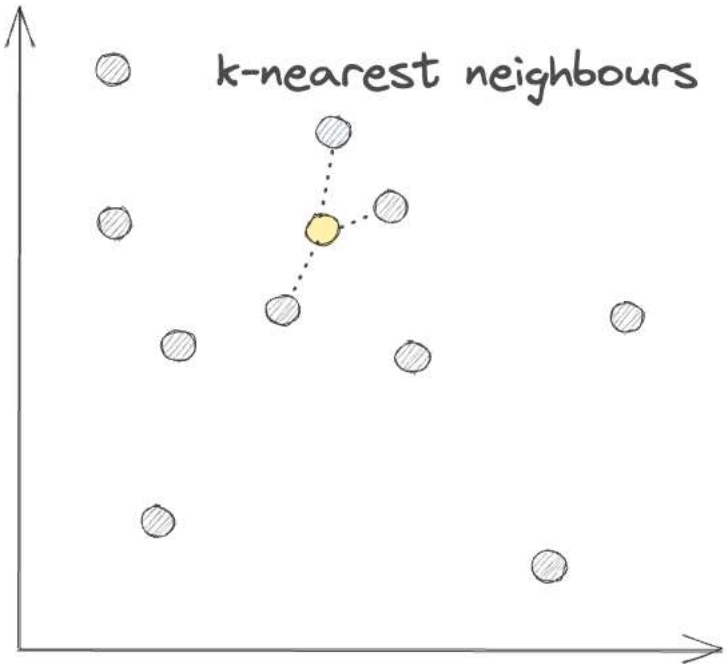
当然，在真实情况下，实际数据存在于更高维的向量空间中（并且具有比葡萄酒种类更多的轴），并且无法在二维平面上可视化，但相同的余弦相似性原理适用。

Scalable nearest neighbour search

可扩展的最近邻搜索

Once the vectors are generated and stored, when a user submits a search query, the goal of similarity search is to provide the top-k most similar vectors to the input query's own vector. Once again, we can visualize this in a simplified 2-D space.

生成并存储向量后，当用户提交搜索查询时，相似性搜索的目标是提供与输入查询自己的向量最相似的前 k 个向量。再一次，我们可以在简化的2D空间中将其可视化。



The most naive way to do this would be to compare the query vector with each and every vector in the database, with the so-called k-nearest neighbour (kNN) method. However, **this quickly becomes way too expensive** as we scale to millions (or billions) of data points, as the number of comparisons required keeps increasing linearly with the data.

最天真的方法是使用所谓的k-最近邻（kNN）方法将查询向量与数据库中的每个向量进行比较。然而，当我们扩展到数百万（或数十亿）个数据点时，这很快就会变得过于昂贵，因为所需的比较数量随着数据的线性增加而增加。

Approximate nearest neighbours (ANN)

近似最近邻 （ANN）

Every existing vector database focuses on making search highly efficient, regardless of the size of the dataset, via a class of algorithms called **Approximate Nearest Neighbour** (ANN) search. Instead of performing an exhaustive comparison between every vector in the database, an approximate search is performed for nearest neighbours, resulting in some loss of accuracy in the result (the *truly* nearest neighbour may not always be returned), but a massive performance gain is possible using ANN algorithms.

每个现有的矢量数据库都专注于通过一类称为近似最近邻（ANN）搜索的算法使搜索高效，无论数据集的大小如何。不是在数据库中的每个向量之间进行详尽的比较，而是对最近邻执行近似搜索，导致结果的准确性有所下降（真正的最近邻可能并不总是返回），但使用 ANN 算法可以大幅提高性能。

Indexing 索引

Data is stored in a vector database via *indexing*, which refers to the act of creating data structures called indexes that allow efficient lookup for vectors by rapidly narrowing down on the search space. The embedding models used typically stores vectors with a dimensionality of the order of 10^2 or 10^3, and ANN algorithms attempt to capture the actual complexity of the data as efficiently as possible in time and space.

数据通过索引存储在向量数据库中，索引是指创建称为索引的数据结构的行为，该索引允许通过快速缩小搜索空间来有效地查找向量。使用的嵌入模型通常存储维度为 10^2 or 10^3 的向量，ANN 算法试图在时间和空间上尽可能高效地捕获数据的实际复杂性。

There are many indexing algorithms used in the various vector DBs available, and their details are out of the scope of this post (I'll be studying these in future posts). But for reference, some of them are listed below.

在各种可用的向量数据库中使用了许多索引算法，它们的细节超出了本文的范围（我将在以后的文章中研究这些算法）。但作为参考，下面列出了其中一些。

- Inverted File Index (IVF)

  倒置文件索引 （IVF）
- Hierarchical Navigable Small World (HNSW) graphs

  分层可导航小世界 （HNSW） 图

- Vamana (utilized in the DiskANN implementation)

  Vamana（用于DiskANN实现）

In a nutshell, the state-of-the-art in indexing is achieved by newer algorithms like HNSW and Vamana, but only a few database vendors offer the DiskANN implementation of Vamana (as of 2023):

简而言之，索引方面最先进的技术是通过 HNSW 和 Vamana 等较新的算法实现的，但只有少数数据库供应商提供 Vamana 的 DiskANN 实现（截至 2023 年）：

- Milvus 米尔沃斯
- Weaviate (work in progress…)

  编织（正在进行中...

- LanceDB (work in progress…)

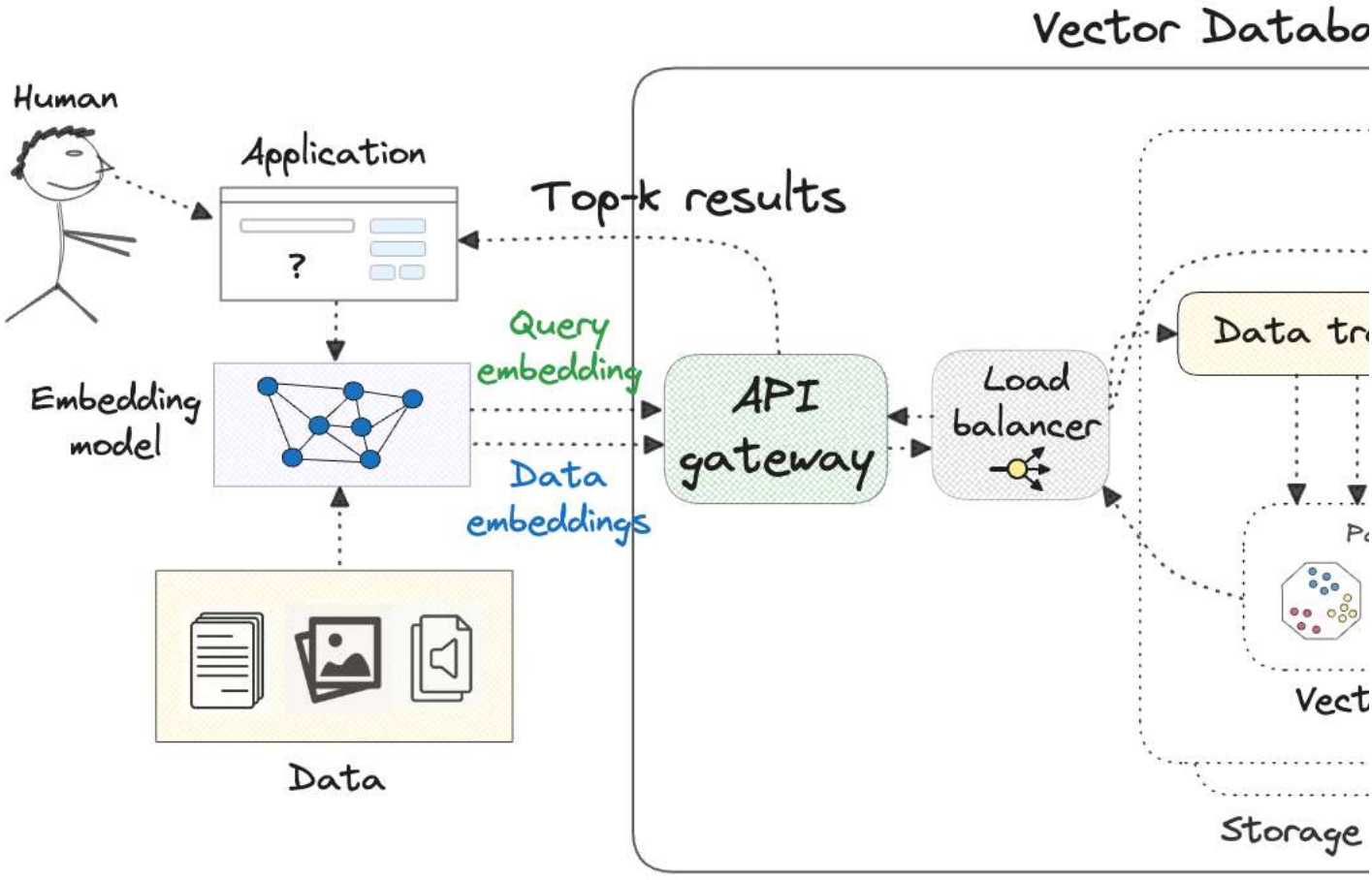
  LanceDB（正在进行中...）

Putting it all together

将一切整合在一起

We can combine all the above ideas to form a mental picture of what a vector database actually *is*.

我们可以结合上述所有想法来形成矢量数据库实际上是什么的心理图景。



The specifics of how each database vendor achieves scalability (via Kubernetes, sharding, streaming and so on) are not important for practitioners – it's up to each vendor to architect the system considering the trade-offs between latency, cost and scalability.

每个数据库供应商如何实现可扩展性（通过 Kubernetes、分片、流等）的细节对于从业者来说并不重要——每个供应商都要考虑延迟、成本和可扩展性之间的权衡来构建系统。

Storage layer and data ingestion

存储层和数据引入

- The data that sits somewhere (locally or on the cloud) is passed to an embedding model, converted to vector form and ingested into the storage layer of a vector DB via the API gateway

位于某个位置（本地或云上）的数据被传递到嵌入模型，转换为矢量形式，并通过 API 网关摄取到矢量数据库的存储层

- The data is indexed, during which it's partitioned/sharded for scalability and faster lookups

  数据被索引，在此期间对其进行分区/分片以实现可扩展性和更快的查找

- The query engine is tightly integrated with the storage layer, to allow for rapid retrieval of nearest neighbours via the database's ANN implementation

  查询引擎与存储层紧密集成，允许通过数据库的ANN实现快速检索最近的邻居

Application layer 应用层

- A user sends a query via an application's UI to the embedding model, which converts the input query to a vector that lies in the same embedding space as the data

  用户通过应用程序的 UI 将查询发送到嵌入模型，该模型将输入查询转换为与数据位于同一嵌入空间中的向量

- The vectorized query is sent to the query engine via the API gateway

  矢量化查询通过 API 网关发送到查询引擎

  - Multiple incoming queries are handled asynchronously, and the top-k results are sent back to the user

    异步处理多个传入查询，并将 top-k 结果发送回用户

Extending vector databases to serve other functions
扩展矢量数据库以提供其他功能

The use case above shows how vector DBs enable semantic search at a scale that was not possible several years ago, unless you were a big tech company with massive resources. However, this is just the tip of the iceberg: vector DBs are used to power a host of downstream functions.

上面的用例显示了矢量数据库如何以几年前不可能的规模实现语义搜索，除非您是一家拥有大量资源的大型科技公司。然而，这只是冰山一角：矢量数据库用于为许多下游功能提供支持。

Hybrid search systems 混合搜索系统

In his excellent review post[3], Colin Harman describes how a lot of companies, due to the plethora of vector DB marketing material out there today, experience "tunnel vision" when it comes to the search & retrieval landscape. As practitioners, we have to remember that vector databases are not the panacea of search – they are very good at *semantic* search, but in many cases, traditional keyword search can yield more relevant results and increased user satisfaction[4]. Why is that? It's largely to do with the fact that ranking based on metrics like cosine similarity causes results that have a higher similarity score to appear above partial matches that may contain specific input keywords, reducing their relevance to the end user.

在他出色的评论文章中 [3]，Colin Harman描述了由于当今大量的矢量数据库营销材料，许多公司在搜索和检索领域如何经历“隧道视野”。作为从业者，我们必须记住，矢量数据库不是搜索的灵丹妙药——它们非常擅长语义搜索，但在许多情况下，传统的关键字搜索可以产生更相关的结果并提高用户满意度 [4]。为什么？这在很大程度上与以下事实有关：基于余弦相似性等指标的排名会导致具有较高相似性分数的结果显示在可能包含特定输入关键字的部分匹配项之上，从而降低它们与最终用户的相关性。

However, pure keyword search also has its own limitations – in case the user enters a term that is semantically similar to the stored data (but is not exact), potentially useful and relevant results are not returned. As a result of this trade-off, real-world use cases for search & retrieval demand a combination of keyword and vector searches, **of which vector databases form a key component** (because they house the embeddings, enabling semantic similarity search and are able to scale to very large datasets).

但是，纯关键字搜索也有其自身的局限性 - 如果用户输入的术语在语义上与存储的数据相似（但不准确），则不会返回可能有用且相关的结果。由于这种权衡，搜索和检索的实际用例需要关键字和矢量搜索的组合，其中矢量数据库构成了关键组件（因为它们容纳嵌入，支持语义相似性搜索，并且能够扩展到非常大的数据集）。

To summarize the points above:

总结以上几点：

- **Keyword search**: Finds relevant, useful results when the user *knows* what they're looking for and expects results that match exact phrases in their search terms. Does **not** require vector databases.

  关键字搜索：当用户知道要查找的内容并期望结果与其搜索字词中的短语完全匹配时，查找相关且有用的结果。不需要矢量数据库。

- **Vector search**: Finds relevant results when the user *doesn't* know what exactly they're looking for. Requires a vector database.

  矢量搜索：当用户不知道他们到底要查找什么时，查找相关结果。需要矢量数据库。

- **Hybrid keyword + vector search**: Typically combines candidate results from full-text keyword and vector searches and re-ranks them using cross-encoder models[5] (see below). Requires both a document database and a vector database.

  混合关键字 + 向量搜索：通常将全文关键字和矢量搜索的候选结果组合在一起，并使用交叉编码器模型 [5] 对其进行重新排名（见下文）。需要文档数据库和矢量数据库。

This can be effectively visualized per the diagram below:

这可以通过下图有效地可视化：

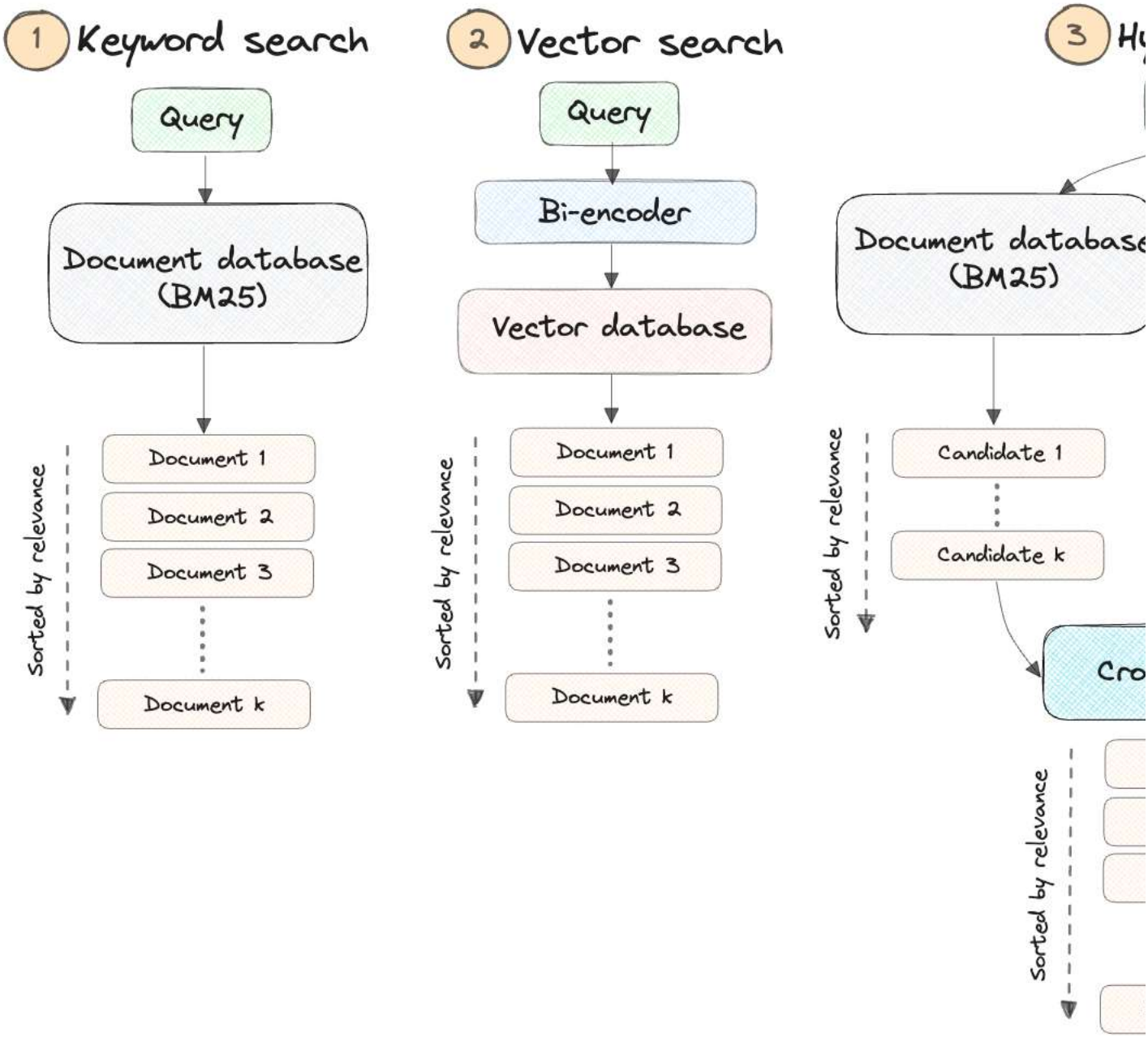


Diagram inspired by Qdrant blog post

受Qdrant博客文章启发的图表

BM25[6] is the most common indexing algorithm used for keyword search in certain databases (e.g., Elasticsearch, Opensearch, MongoDB). It produces a *sparse* vector, by considering keyword term frequencies in relation to their inverse document frequency (IDF). In contrast, vector databases typically encode and store text in embeddings represented by *dense* vectors (none of the terms in the vector are zero, unlike BM25), and this is typically done via a bi-encoder model like BERT, that produces a sentence embedding for a pair of documents, that can then be compared to produce a cosine similarity score.

BM25 [6] 是某些数据库（例如Elasticsearch，Opensearch，MongoDB）中用于关键字搜索的最常见索引算法。它通过考虑关键字术语频率与其反向文档频率 （IDF） 的关系来生成稀疏向量。相比之下，矢量数据库通常在由密集向量表示的嵌入中编码和存储文本（与 BM25 不同，向量中的所有项都不为零），这通常通过像 BERT 这样的双编码器模型完成，该模型为一对文档生成句子嵌入，然后可以进行比较以产生余弦相似性分数。

Understanding the difference between bi-encoders and cross-encoders

了解双编码器和交叉编码器之间的区别

To effectively perform hybrid search, it becomes necessary to combine search result candidates obtained via BM25 (keyword search) and cosine similarity (vector search), which requires a *cross-encoder*. This is a downstream step, as shown in the image below, that allows two sentences to be simultaneously passed to an encoder model like BERT. Unlike the bi-encoder that's used to produce sentence embeddings, a cross-encoder doesn't produce embeddings, but rather, it allows us to classify a pair of sentences by assigning them a score between 0 and 1, via a softmax layer. This is termed ***re-ranking***, and is a very powerful approach to obtaining results that combine the

best of both worlds (keyword + vector search).

为了有效地执行混合搜索，有必要将通过BM25（关键字搜索）和余弦相似性（矢量搜索）获得的候选搜索结果结合起来，这需要交叉编码器。这是一个下游步骤，如下图所示，它允许两个句子同时传递给像BERT这样的编码器模型。与用于生成句子嵌入的双编码器不同，交叉编码器不会产生嵌入，而是允许我们通过softmax层为一对句子分配0到1之间的分数来对一对句子进行分类。这被称为重新排名，是一种非常强大的方法，可以获得结合两全其美的结果（关键字+矢量搜索）。

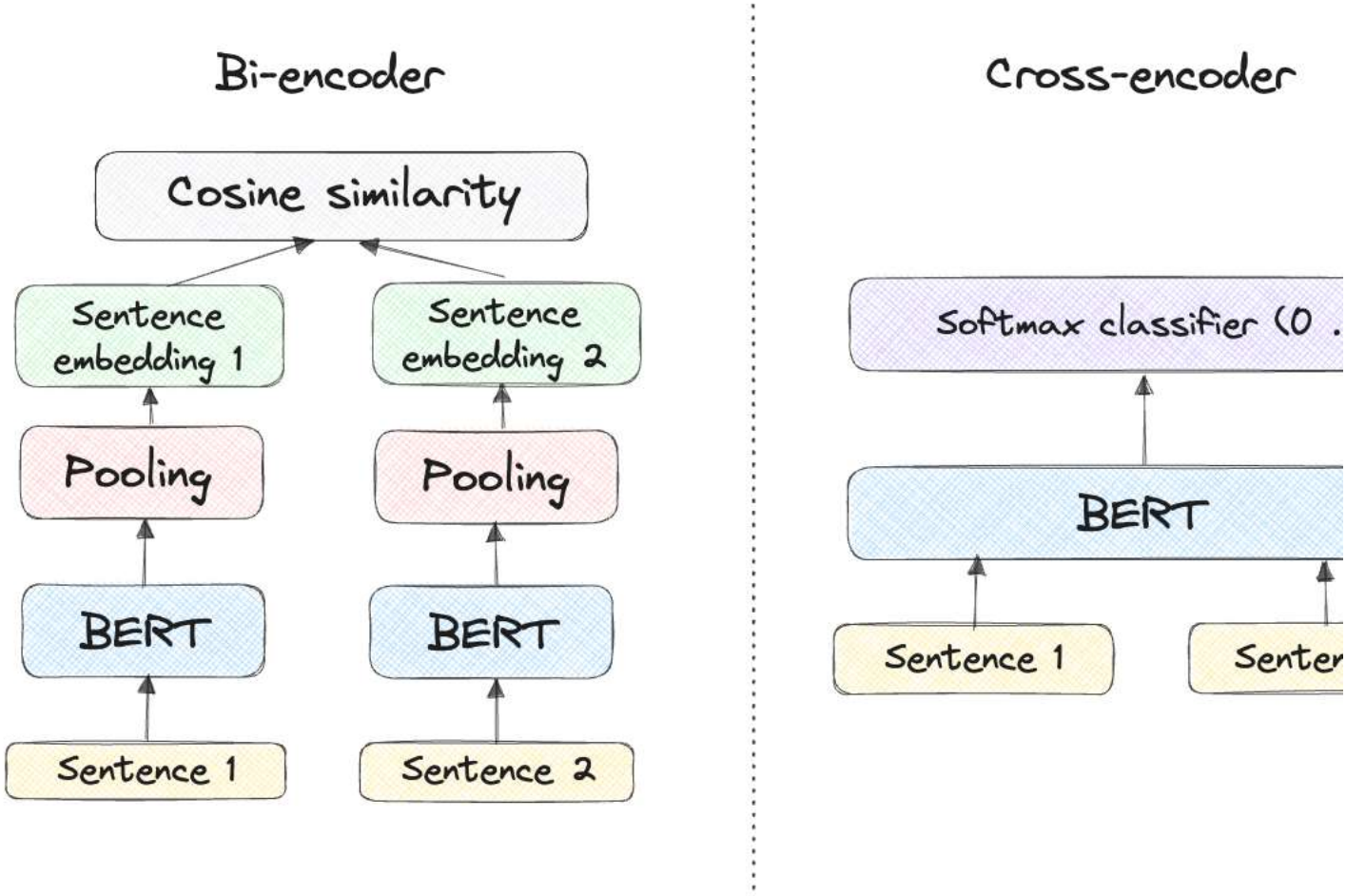


Diagram inspired by Sentence transformers docs

受句子转换器文档启发的图表

Info 信息

It should be noted that re-ranking via cross-encoders is an expensive step, as it requires the use of a transformer model during query time. This approach is used when the quality of search is critical to a use case, and requires more compute resources (typically, GPUs) and tuning time to ensure that the application is performing as intended.

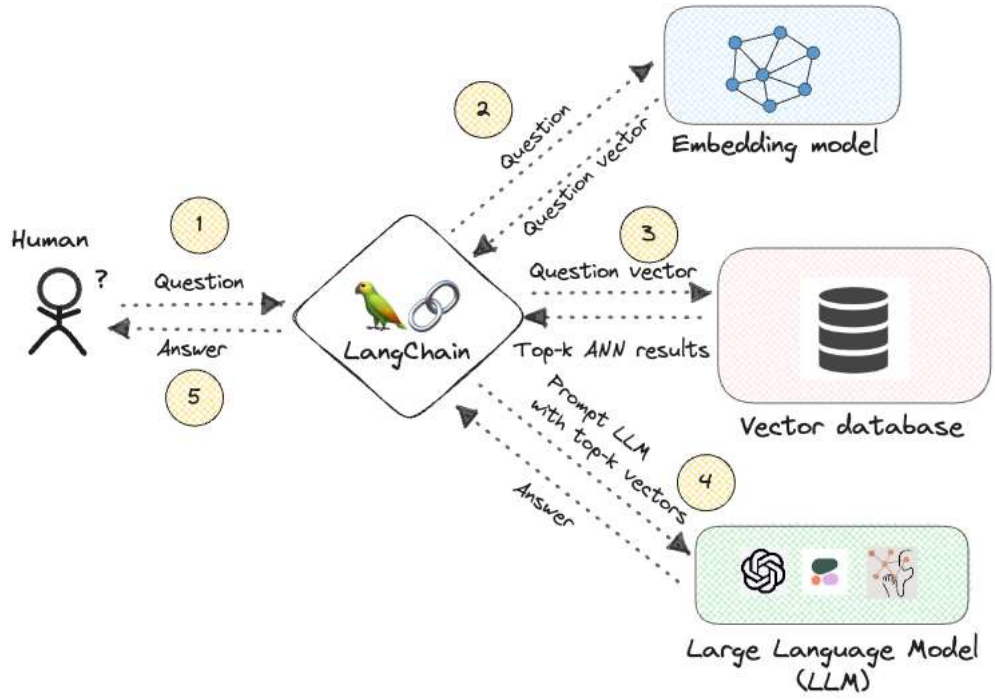
应该注意的是，通过交叉编码器重新排名是一个昂贵的步骤，因为它需要在查询时使用转换器模型。当搜索质量对用例至关重要，并且需要更多的计算资源（通常是 GPU）和调优时间来确保应用程序按预期执行时，将使用此方法。

Generative QA: “Chatting with your data”

生成式 QA： “与数据聊天”

With the advent of powerful LLMs like GPT-4, we can effectively integrate the user experience of an application with clean, factual information stored in a vector DB, allowing the user to query their data via natural language. Because question-answering can involve more than just information retrieval (it may require parts of the data to be analyzed, not just queried), including an agent-based framework like LangChain[7] in between the application UI and the vector DB can be much more powerful than just plugging into the vector DB directly.

随着像 GPT-4 这样强大的 LLM 的出现，我们可以有效地将应用程序的用户体验与存储在矢量数据库中的干净、真实的信息集成在一起，允许用户通过自然语言查询他们的数据。因为问答可能不仅仅涉及信息检索（它可能需要分析部分数据，而不仅仅是查询），在应用程序 UI 和矢量数据库之间包括一个基于代理的框架，如 LangChain [7]，可能比直接插入矢量数据库要强大得多。



Because vector DBs store the data to be queried as embeddings, and the LLM also encodes the knowledge within it as embeddings, they are a natural pairing when it comes to generative QA applications. The vector DB functions as a knowledge base, and the LLM can query a subset of the data directly in the embedding space. This can be done using the following approach:

由于向量数据库将要查询的数据存储为嵌入，并且LLM还将其中的知识编码为嵌入，因此在生成QA应用程序方面，它们是自然配对。向量数据库用作知识库，LLM 可以直接在嵌入空间中查询数据的子集。这可以使用以下方法完成：

1. Human asks a question in natural language via the UI

   人类通过UI用自然语言提问

2. The question's text is passed to an embedding model (bi-encoder), which then returns a sentence embedding vector

   问题的文本被传递给嵌入模型（双编码器），然后返回句子嵌入向量

3. The question vector is passed to the vector DB, which returns the top-k most similar results, obtained via an ANN search

   问题向量被传递给向量数据库，后者返回通过ANN搜索获得的top-k最相似的结果。

   - This step is **crucial**, as it massively narrows down the search space for the LLM, used in the next step

     这一步至关重要，因为它大大缩小了下一步中使用的LLM的搜索空间。

4. An LLM prompt is constructed (based on the developer's predefined template), converted to an embedding, and passed to the LLM

   构建 LLM 提示符（基于开发人员的预定义模板），转换为嵌入，并传递给 LLM

   - A framework like LangChain makes it convenient to perform this step, as the prompt can be dynamically constructed and the LLM's native embedding module called without the developer having to write a lot of custom code for each workflow

     像LangChain这样的框架可以方便地执行此步骤，因为可以动态构造提示并调用LLM的本机嵌入模块，而无需开发人员为每个工作流编写大量自定义代码

5. The LLM searches the top-k results for the information, and produces an answer to the question

   LLM 在前 k 个结果中搜索信息，并生成问题的答案

6. The answer is sent back to the human

   答案被发回给人类

The above workflow can be extended in many ways – for example, if the user's question involved some arithmetic on numbers obtained from the database (which LLMs are notoriously bad at), a LangChain agent could determine, first, that arithmetic is required. It would then pass the numerical information extracted from the top-k results to a calculator API, perform the calculations, and then send the answer back to the user. With such composable workflows, it's easy to see how powerful, generative QA chat interfaces are enabled by vector DBs.

上述工作流程可以通过多种方式进行扩展 - 例如，如果用户的问题涉及从数据库中获得的数字的一些算术（LLM是出了名的不擅长），LangChain代理可以确定，首先，需要算术。然后，它将将从top-k结果中提取的数字信息传递给计算器API，执行计算，然后将答案发送回用户。通过这种可组合的工作流，很容易看出矢量数据库如何启用强大的生成式 QA 聊天界面。

Conclusions 结论

There are many other useful applications that can be built combining the inherent power of LLMs and vector DBs. However, it's important to understand some the underlying limitations of vector DBs:

结合LLM和矢量数据库的固有功能，可以构建许多其他有用的应用程序。但是，了解矢量数据库的一些潜在限制非常重要：

- They do not necessarily prioritize exact keyword phrase matches for relevance in search applications.

  它们不一定会优先考虑与搜索应用程序中的相关性匹配的确切关键字短语。
- The data being stored and queried in vector databases must fit inside the maximum sequence length of the embedding model used (for BERT-like models, this is no longer than a few hundred words). Currently, the best way to do so is to utilize frameworks like LangChain and LlamaIndex[8] to chunk or squash the data into a fixed-sized vector that fits into the underlying model's context.

  在矢量数据库中存储和查询的数据必须符合所用嵌入模型的最大序列长度（对于类似BERT的模型，这不超过几百个单词）。目前，最好的方法是利用LangChain和LlamaIndex [8] 等框架将数据分块或压缩到适合底层模型上下文的固定大小的向量中。

Vector databases are incredibly powerful, but the bottom line, per Harman[3], applies to all of them:

矢量数据库非常强大，但根据 Harman [3] 的说法，底线适用于所有数据库：

> If you're not very knowledgeable about vector databases, and in general, the search and information retrieval space, the following materials **should NOT** be your primary sources of information for any kind of decision-making on choosing your tech stack:
>
> 如果您对矢量数据库以及搜索和信息检索空间不是很了解，那么以下材料不应成为您在选择技术堆栈时做出任何决策的主要信息来源：
>
> - Infrastructure stacks from super-smart VCs
>
>   来自超级智能风投的基础设施堆栈
> - Tutorials from popular LLM application frameworks
>
>   来自流行的LLM应用程序框架的教程

Just like in any other domain, clearly defining the business case and studying the tools at hand allows you to combine them effectively to solve real-world problems. In that regard, I hope this series on vector DBs was helpful so far!

就像在任何其他领域一样，清楚地定义业务案例并研究手头的工具可以让您有效地组合它们来解决现实世界的问题。在这方面，我希望这个关于矢量数据库的系列到目前为止有所帮助！

Other posts in this series

本系列中的其他帖子

- Vector databases (Part 1): What makes each one different?

  矢量数据库（第 1 部分）：是什么让每个数据库与众不同？
- Vector databases (Part 3): Not all indexes are created equal

  向量数据库（第 3 部分）：并非所有索引都是平等的
- Vector databases (Part 4): Analyzing the trade-offs

  向量数据库（第 4 部分）：分析权衡取舍

---

1. The rise of vector databases, Forbes ↩

   矢量数据库的兴起，福布斯 ↩

2. Revolution in NLP is changing the way companies understand text, TechCrunch ↩

   NLP的革命正在改变公司理解文本的方式，TechCrunch ↩

3. Beware tunnel vision in AI retrieval: Colin Harman, Substack ↩ ↩

   当心AI检索中的隧道视觉：科林·哈曼，子堆栈 ↩↩

4. Vector search for clinical decisions, Haystack US 2023 ↩

   临床决策的载体搜索，Haystack US 2023 ↩

5. On hybrid search, Qdrant blog ↩

   关于混合搜索，Qdrant 博客 ↩

6. Okapi BM25, Wikipedia ↩ 霍加披BM25，维基百↩科

7. LangChain docs, python.langchain.com ↩

   语言链文档，python.langchain.com ↩

8. LlamaIndex docs ↩ 美洲驼索引文档 ↩