Vector database management systems: Fundamental concepts, use-cases, and current challenges

Preprint · September 2023

DOI: 10.13140/RG.2.2.10751.79529

1 author:

Toni Taipalus
University of Jyväskylä
31 PUBLICATIONS **247** CITATIONS

SEE PROFILE

Vector database management systems: Fundamental concepts, use-cases, and current challenges

Toni Taipalus

Abstract

Vector database management systems have emerged as an important component in modern data management, driven by the growing importance for the need to computationally describe rich data such as texts, images and video in various domains such as recommender systems, similarity search, and chatbots. These data descriptions are captured as numerical vectors that are computationally inexpensive to store and compare. However, the unique characteristics of vectorized data, including high dimensionality and sparsity, demand specialized solutions for efficient storage, retrieval, and processing. This study provides an accessible introduction to the fundamental concepts, use-cases, and current challenges associated with vector database management systems, offering an overview for researchers and practitioners seeking to explore this burgeoning technology aimed to facilitate effective vector data management.

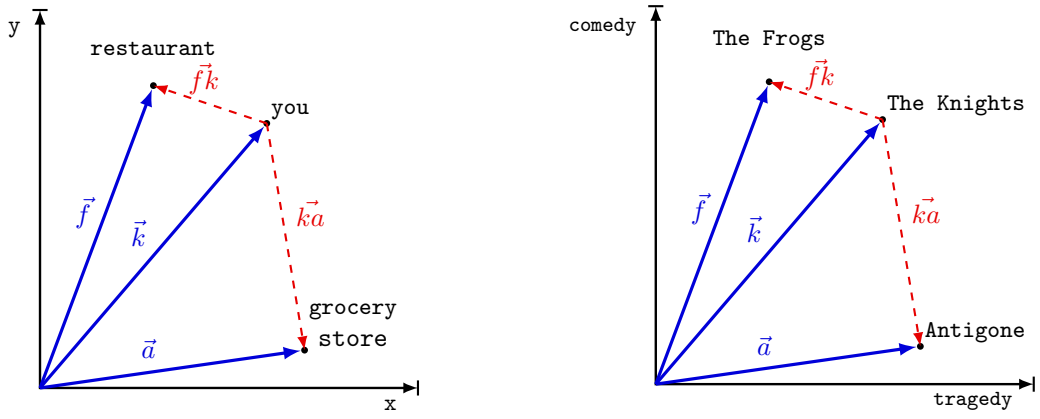
Keywords: vector, database, feature, challenge, neural network, deep learning

1. Introduction

It is increasingly common that rich, unstructured data such as large texts, images and video are not only stored, but given semantics through a process called *vectorization* [1] which captures the features of the data object in a cost-effectively processed numerical vector such as $\vec{k} = [6, 7]$. The vectors are n-dimensional, and consist of natural, real, or complex numbers, where one number represents a feature or a part of a feature. The features that form a vector can range from simple, such as the number of actors in a stage play, to complex, such as textures identified in an image by a neural network [2], where number 3 may correspond to texture of human skin, while number 10 may correspond to the texture of a cat's fur. In contrast to traditional data models such as relational, where queries often take forms such as "*find the orders of a specific user*" or "*find the products that are on sale*", vector queries typically search for *similar* vectors using one or several query vectors. That is, queries take forms such as "*find ten most similar images of cats that look like the cat in this image*" or "*find the most suitable restaurants for me given my current position*".

Managing vector data has gained increased popularity, partly due to applications such as reverse image search, recommender systems, and chatbots, and this trend is on the rise [3]. Consequently, efficient management of data requires a dedicated database management system (DBMS). A vector DBMS (VDBMS) is not strictly a requirement for any business domain, as vectors can be stored and queried without a dedicated DBMS, similarly to relational or document data can be stored and queried without a relational DBMS. The DBMS, however, in all cases, facilitates data management that is *feasible*, freeing development resources towards other business domain critical tasks by providing ready-made features such as transaction and access control, automated database scalability, and query optimization. Additionally, increasingly complex business domains require increasingly complex features such as vector similarity search complemented by metadata filters, as well as searching with multiple query vectors [1], and efficient ways to manage access control and concurrent transactions.

This study aims to provide an easily accessible description of fundamental concepts behind VDBMSs (Section 2) without focusing on the intricacies of a single product, an overview of current VDBMS products and their features (Section 3), explanations behind some popular use-cases such as image similarity search and long-term memory for chatbots (Section 4), and some of the current challenges related to VDBMSs (Section 5). This work assumes that the reader is familiar with fundamentals of some other type of database management system (e.g., relational), and does not detail the mathematics of vectors, or algorithms behind vector search or vector index creation.



(a) Closeness in geospatial locations

(b) Closeness in genres

Figure 1: Simple examples of applications of two-dimensional vectors

2. Vectors and vector database management systems

2.1. Vectors as data representations

Perhaps one of the most intuitive use-case for vector data is in geospatial applications [e.g., 4]. Two-dimensional points such as the location of the end-user and points-of-interest may be represented as vectors, and the closest points-of-interest may be calculated with simple and well-understood operations. For example, by calculating the distance between the end-user ("*you*" in Fig. 1a) and points-of-interest, the length of vectors ($\vec{fk}$ and $\vec{ka}$, i.e., distance) can be compared, and the closest point-of-interest found. If we consider the vector for the end-user as $\vec{k} = [6, 7]$, the vector for the restaurant as $\vec{f} = [3, 8]$ and the vector for the grocery store as $\vec{a} = [7, 1]$, we can calculate the similarity or closeness of the vectors by, e.g., Euclidean distance or cosine similarity.

In addition to coordinates, other types of data can be represented as vectors. For example, instead of coordinates, Fig. 1b shows Greek plays mapped along how comic and tragic they are. By examining closeness based on these two dimensions, we can, e.g., calculate that the play *The Knights* is closer to *The Frogs* than *Antigone*, that is, the vector $\vec{fk}$ is shorter than the vector $\vec{ka}$. The vector for *Antigone* can be represented as $\vec{a} = [7, 1]$, where the first component represents the amount of tragedy, and the second component the amount of comedy. Closeness of the vectors is not the only way to measure similarity.

The aforementioned are examples of very low-dimensional vectors. By increasing the dimensions of vectors (say, by adding z coordinates, or another genre, *drama*), vectors can capture increasingly rich data. If *Antigone*'s drama amounts to 6, the vector for *Antigone* in three-dimensional space is $\vec{a} = [7, 1, 6]$. Furthermore, a high-dimensional vector may have thousands or millions of dimensions, making the visualizations of such vectors unfeasible, and the data unreadable for a human. Such high-dimensional vectors can be used to represent more complex data such as text, image, audio and video features. From data-representation perspective, this separates vector databases from relational and NoSQL databases, in which data objects are often human-readable, contextualized numbers, text strings, and time.

2.2. Vector database management systems

A vector database management system is *a specialized type of database management system that focuses primarily on the efficient management of high-dimensional vector data.* Similarly to other types of database management systems (such as relational, document, and graph), this definition requires that a VDBMS is a piece of software that can manage data. Data management includes but is not limited to data querying and manipulation, collection of metadata, indexing, access control, backups, support for scalability, and interfaces with other systems such as database drivers, programming languages, frameworks, and operating systems. Furthermore, a VDBMS focuses on the management of vector data. There are several DBMSs that offer support for multiple data models (e.g., PostgreSQL supports relational, document and object-oriented

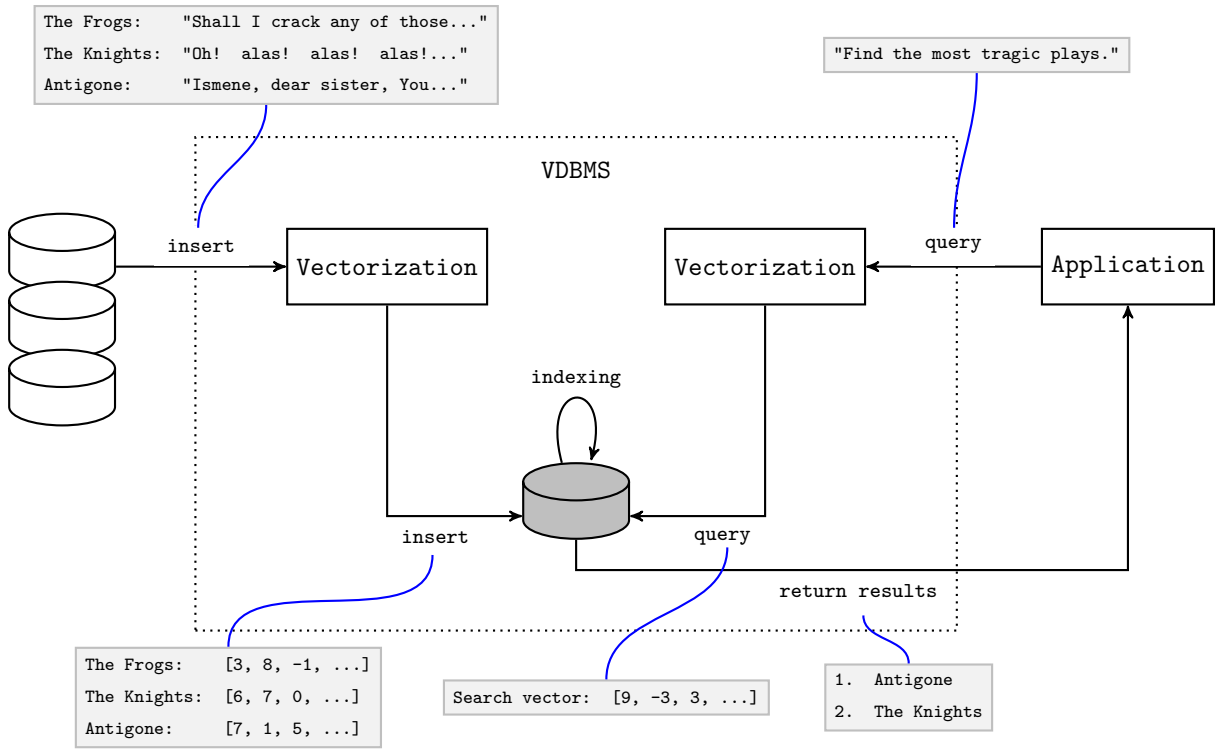


Figure 2: A simplified view of a database system illustrating the flow and transformation of information to and from the vector database; the vectorization process transforms information into vectors which can be quickly compared with each other; it is worth noting that the natural language query depicted here requires data additional to the actual plays

data models, and Redis supports key-value and vector data models), yet the primary focus of such systems is typically on one data model. It has been noted that such systems miss optimization opportunities for vector data, and may lack features such as the use multiple query vectors [1]. Finally, a VDBMS focuses on the management of high-dimensional vectors. Systems focusing on, e.g., two or three-dimensional geospatial data management are not considered VDBMSs in this context.

VDBMSs typically support similarity search through indexing methods that enable rapid and accurate searching of similar vectors, i.e., search for vectors that closely resemble a given query vector based on specific distance metrics such as Euclidean distance or cosine similarity. This capability is particularly valuable in various applications where finding similar vectors is crucial, such as image or text retrieval systems. VDBMSs also offer support for vector operations, allowing users to perform mathematical computations on vectors. These operations may include arithmetic calculations, statistical analysis, or transformations to manipulate the vectors. In colloquial language, the term *vector database* is sometimes used as a synonym for a VDBMS despite the fact that a VDBMS is a piece of software, yet a vector database is a collection of data. It is also worth noting that despite their popularity, we – among others [1] – do not consider algorithms or libraries such as Facebook's FAISS library [5] VDBMSs, as they do not provide many of the functionalities described above.

2.3. Database system architecture

A database system consists of one or several database management systems, databases, and software applications. Fig. 2 shows a simplified flow of information from traditional data sources (depicted on the left-hand side, e.g., relational databases) to the vector database (gray). Continuing with the example of Greek plays, the human-readable texts of the plays are *vectorized*, i.e., transformed into high-dimensional vector representations in a way that captures meaningful relationships or patterns. The outcome of the vectorization process, i.e., the vector, is often called a *vector embedding* or a *feature vector*. In addition to the vector itself, VDBMSs typically store a vector identifier, some vector metadata, and possibly the data that the vector represents. For example, for the play *Antigone*, a VDBMS may store an unique identifier, a

Index type	Characteristics	Use-cases	Advantages	Disadvantages
Product Quantization	Divides vectors into smaller parts	Image search	Reduces dimensionality	Lossy compression may reduce accuracy
Locality-Sensitive Hashing	Hashes similar vectors to same buckets	Near-duplicate detection	Enables approximate similarity search	Requires parameter tuning
Hierarchical Navigable Small World	Creates a hierarchical graph	Recommendation systems, text search	Fast neighborhood exploration	Complex index structure, space overhead
R-trees	Hierarchical structure with bounding boxes	Spatial data (geospatial indexing)	Efficient range queries, updates	Slower nearest-neighbor searches
KD-trees	Binary tree partitioning along dimensions	Machine learning, clustering	Balanced tree structure, good for low dims	Inefficient in high dimensions, complex build
Random Projection	Projects high-dimensional data randomly	Text classification, clustering	Fast indexing, good for high dimensions	May lose information, requires tuning

Table 1: Comparison of some vector indexing techniques

vector embedding of the text of the play that contains numerical data about the amount of tragedy, comedy and drama in *Antigone*, metadata such as type of work ("*play*") and country of origin ("*Greece*"), and the play itself in plain text. The play itself is often referred to as the *payload* of the vector. As another example, YouTube utilizes metadata such as user and video language and time since video was last watched in providing personalized recommendations [6].

In natural language processing, words and phrases are vectorized into vectors in such a way that similar words have similar vector representations. Similarity can mean different things depending on the context, e.g., words may sound similar (*walking* and *talking*), or words may mean similar things (*walking* and *running*) in different contexts. The amount of tragedy in a play may depend on the number of tragic words in the play, a sentiment analysis assessing the tone of the play, or topic modeling which identifies key themes in the play. This process helps algorithms understand and work with the data more effectively. *Word2vec* [7], *FastText*, and *Doc2vec* [8] are examples of techniques that create vector embeddings for words in natural language.

After the data objects have been vectorized and stored in the vector database, the data are indexed to enable faster queries, as with effectively all data models [9]. As vector queries are almost always approximations, one of the primary trade-offs between different indexing algorithms are accuracy and speed. Some popular algorithms are *Product Quantization* [e.g., 10, 11], which divides high-dimensional vectors into smaller parts and summarizes each part separately, reducing dimensionality and storage space requirements, but losing some accuracy, *Locality-Sensitive Hashing* [e.g., 12], which hashes similar vectors to the same buckets, enabling approximate similarity search, and *Hierarchical Navigable Small World* [e.g., 13, 14], which creates a hierarchical graph with fast neighborhood exploration by building a small world network. Other algorithms include *R-trees* [15], *KD-trees* [e.g., 16], and *Random Projection* [e.g., 17]. Table 1 summarizes various vector index types. It is worth noting that the choice of indexing algorithm depends on the data characteristics, dimensionality, and search requirements. Index creation is typically computationally expensive.

Similarly to inserting data into the vector database, queries in natural language or human-readable values in computer language queries must be vectorized before the VDBMS can assess vector similarity. The vectorization may happen in the application program or the VDBMS (the latter case is depicted in Fig. 2, yet the former case is more typical). Vectorization can be done in multiple ways depending on the data and the purpose of the vectorization. Despite the fact that feature vectors of Greek plays and feature vectors of images of cats may look similar (i.e., both are "lists" or numbers), their values represent different things.

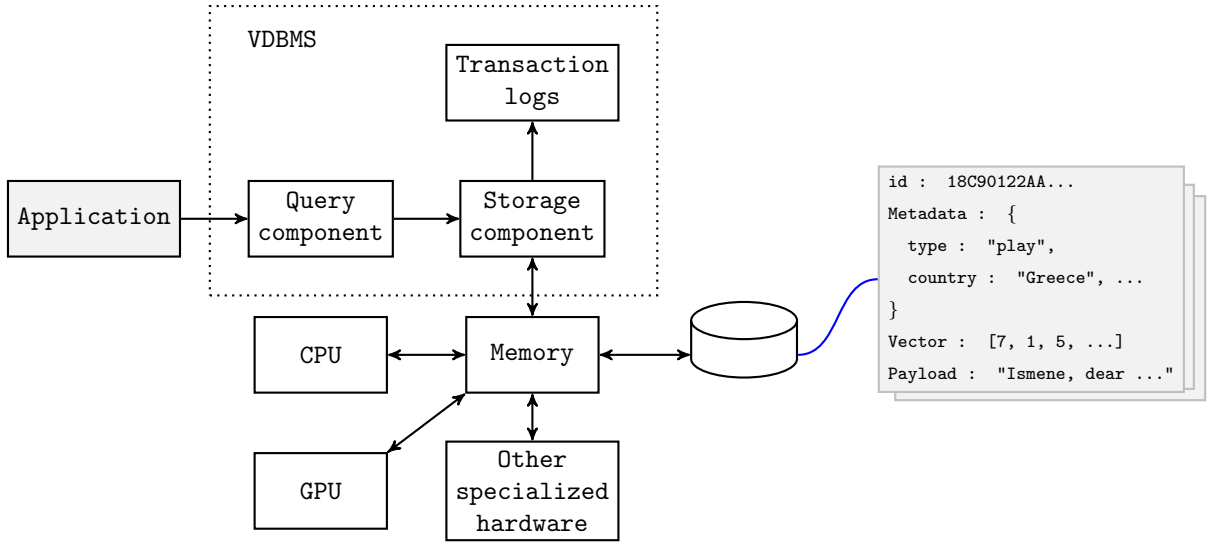


Figure 3: A generalized overview of VDBMS components; the arrows represent the flow of information from the software application through the VDBMS to the physical database; the database represents persistent storage device, contrary to Fig. 2, where the database here represent the logical database structure maintained by the VDBMS; the righ-hand side shows an example of the stored data object consisting of metadata, the vector, and vector payload

As mentioned earlier, vector similarity or closeness may be assessed using several methods such as *Jaccard similarity*, which measures the similarity between two sets (i.e., vectors) by comparing their shared elements to the total number of distinct elements in both sets, *Euclidean distance* (L2), which measures the straight-line distance between two points in a space with multiple dimensions, *dot product*, which computes the sum of the products of corresponding elements in two vectors, or *cosine similarity*, which measures the cosine of the angle between two vectors, indicating how similar their directions are regardless of their magnitudes. The choice of method depends on the context and the specific characteristics of the data. For more in-depth, mathematical explanations, Wang *et al.* [18] provide accessible overview of several indexing and search methods mentioned above.

From a developer perspective, queries in VDBMSs are more closely related to simple document or key-value store queries than to complex queries in relational databases. Instead of retrieving documents based on document identifiers as in many NoSQL systems, vectors are retrieved using one or several query vectors. Despite this similarity in queries, the query execution internals differ, since VDBMS queries typically search for nearest neighbor vectors instead of exact matches. Fig. 4 illustrates some basic queries in three VDBMSs and in PostgreSQL with the *pgvector* extension. Instead of searching for Greek plays where the amount of tragedy is high, as one probably would with a relational query, a vector query may retrieve Greek plays which are similar to a particular play in terms of tragedy, comedy, drama, author, publication year, etc.

In addition to the vectors themselves, queries may utilize metadata to, e.g., limit the number of vectors to compare. For example, if the end-user is requesting data on Greek plays, and the database contains metadata for language and type of media, the vector similarity search may be limited to Greek plays rather than all written art originating from all countries. While vectors are indexed using different vector indices, metadata may be indexed using more traditional techniques such as B^+*-trees* to support range queries. Queries that utilize both a query vector and metadata filters are called *hybrid queries*. If a VDBMS does not provide the means for hybrid queries, metadata-based searches may be implemented separately as part of a broader architecture. Fig. 3 provides a generalized (i.e., not product-specific) overview of VDBMS components. These components are in principle similar to components in other types of DBMSs: the query component parses the queries and other statements from the software application, checks user access rights on data object level, optimizes the query, and passes the query to the storage component. The storage component logs the transaction if the VDBMS utilizes transaction logs such as Write Ahead Logging, manages transaction locks if applicable, and retrieves or stores the data the application has requested with the help of different buffers, memory, CPU and GPU, and possibly other specialized hardware such as Field-Programmable Gate Arrays

```python
results = collection.search(
  data=[[3, 8, -1, ...]],
  anns_field="text",
  param=search_params,
  limit=2,
  expr="country like \"Greece\" && type like \"play\"",
  output_fields=['title'],
  consistency_level="Strong"
)
```

(a) Query in Milvus

```python
payload = {
  "filter": {
    "country": "Greece",
    "type": "play"
  },
  "includeValues": True,
  "includeMetadata": True,
  "topK": 2,
  "vector": [3, 8, -1, ...]
}
response = requests.post(url, json=payload)
```

(b) Query in Pinecone

```python
collection.query(
  query_embeddings=[[3, 8, -1, ...],
  n_results=2,
  where={"country": "Greece", "type": "play"}
)
```

(c) Query in Chroma

```
  SELECT *
    FROM plays
   WHERE country = 'Greece'
     AND type = 'play'
ORDER BY embedding <-> '[3,8,-1, ...]'
   LIMIT 2;
```

(d) Query in PostgreSQL (pgvector)

Figure 4: Hybrid queries in different VDBMSs using Python, and in PostgreSQL using SQL

Table 2: VDBMS features; example use-cases are based on a product's documentation's use-case examples as of August 2023

	License	First release	Querying with metadata
Pinecone	Proprietary	2021	rich expressions
Chroma	Apache 2.0	2023	rich expressions
Milvus	Apache 2.0	2019	rich expressions
Weaviate	BSD 3-clause / proprietary	2019	supported
Qdrant	Apache 2.0 / proprietary	2022	rich expressions
Deep Lake	Apache 2.0 / proprietary	2019	rich expressions
	Integration	**Querying**	**Example use-cases**
Pinecone	OpenAI, LangChain, others	Java, Python, C#, several others	chatbots, image search
Chroma	LangChain, LlamaIndex	JavaScript, Python, Ruby, others	chatbots
Milvus	OpenAI, LangChain, others	Java, Python, Go, Node.js	chatbots, image/audio/video search
Weaviate	OpenAI, Cohere, PaLM	Java, JavaScript, Python, Go, GraphQL	chatbots, image search
Qdrant	OpenAI, LangChain, others	Python, JavaScript, Go, Rust	chatbots, image search
Deep Lake	LlamaIndex, LangChain	Python, SQL-like TQL	image search

or tensor processing units.

3. Products and features

At the time of writing, DB-Engines [19] lists seven VDBMSs: Pinecone, Chroma, Milvus, Weaviate, Vald, Qdrant and Deep Lake. However, since Vald primarily focuses on similarity search and lacks features such as access control and integrations to other technologies, we considered Vald a vector search engine rather than a VDBMSs as defined in Section 2.1. Several of these products are designed from the ground up to utilize different types of processing units or devices, and multi-GPU and CPU parallelism in a coordinated manner [1]. The VDBMSs typically implement several index and search methods (such as Euclidean distance), and the optimizer component selects the most suitable search method depending on the characteristics of the data and the query, similarly to the optimizer in relational DBMSs.

In addition to the VDBMSs mentioned above, there are also several DBMSs with multiple data models, vectors being one of them, several vector extensions to other DBMSs such as PostgreSQL, MongoDB, Cassandra, Redis and SingleStore, and as vector database-enabling libraries for programming languages, such as Thistle for Rust [20]. Table 2 lists some features of these six VDBMSs. Similarly to NoSQL systems, we expect VDBMSs to develop rapidly in terms of features, new products, and community support.

4. Use-cases

4.1. Similarity search in general

As explained in Section 2.1, there are many use-cases for vector data. Effectively all data objects that can be vectorized in a meaningful way may be used in approximate similarity search, which is the basis for almost all vector database retrieval operations. Although the next subsections focus on some popular use-cases for vector databases, it is worth noting that this is not an exhaustive list. For example, vectors are used in storing and comparing molecular structures [21] and rentable apartments [22], automated black-and-white image colorization [23], facial expression recognition [24], tracking digital image assets [25], and recommender systems [26].

4.2. Image and video similarity search

In a similar fashion as Greek plays, images can be vectorized, yet the process is typically more complex and involves image normalization in terms of size and pixel values, and feature extraction prior to vectorization. Feature extraction, which is typically external to the VDBMS, can involve passing the images – one at a time – through a convolutional neural network. The process extracts increasingly abstract features from the image, starting from simple features such as the presence of vertical and horizontal edges and simple shapes [e.g., 27], to textures such as fur, foliage and water. These features are vectorized and used for similarity search. Images with similar vector representations are likely to be visually similar in terms of the captured features. Similarly to Fig. 4, Fig. 5a shows that after a set of images has been vectorized, similar process can be used for reverse image search (i.e., searching for images with image input). Once similar vectors have been found, the VDBMS returns the vector payloads to the application.

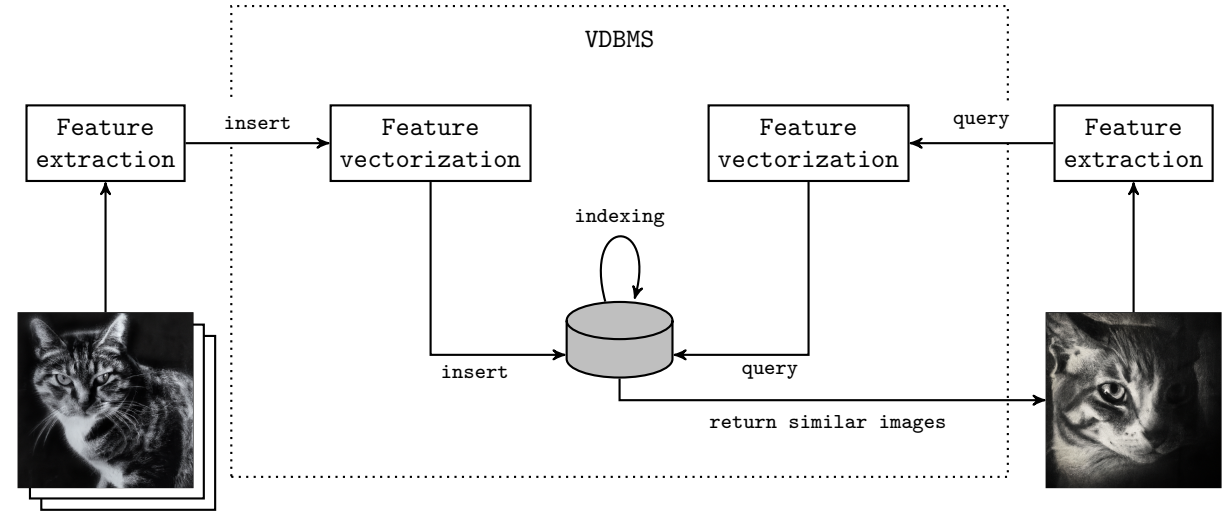
In the context of video vectorization, videos are typically broken down into individual frames, although just a representative subset of frames may be considered. Similarly to stand-alone images, features are extracted from the individual frames and vectorized into a feature vector representing its content. Additionally, temporal information is often needed to further understand the contents of the video. For example, one video of a Greek play may develop from comedy to tragedy, while another may do the opposite. Without temporal information about the order of the frames, it is not possible to tell one from another in this regard. The process results in a sequence of feature vectors which can be combined. In other words, the sequence of vectors may be considered a three-dimensional tensor, where dimensions represent frames, features, and time. This tensor can be stored as a flattened vector in the vector database. It is worth noting that the *dimension* of the tensor here is a different concept than the *dimension* of the vector.

4.3. Voice recognition

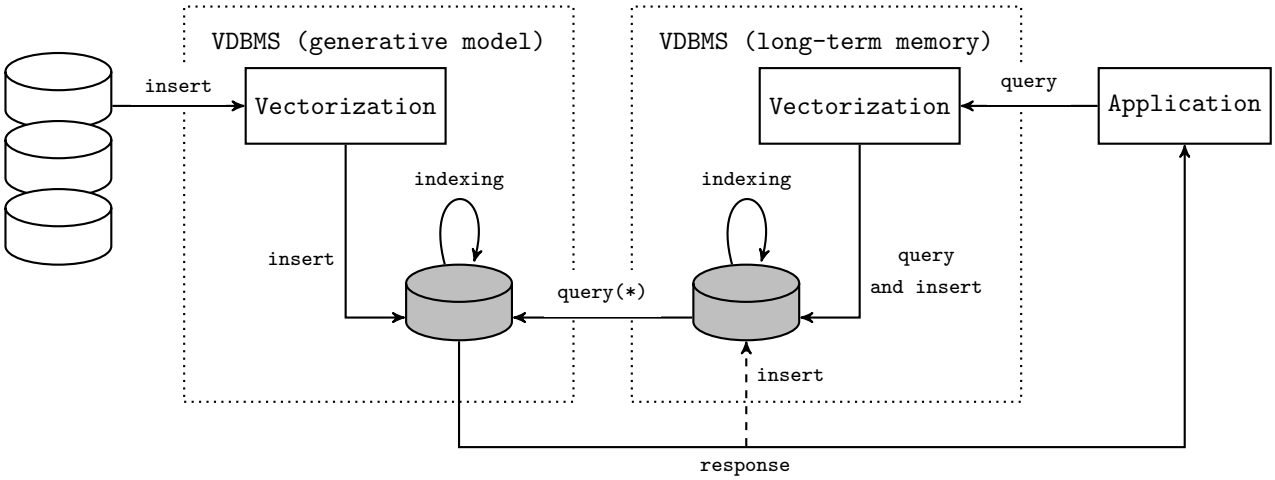
Voice recognition using vectors works in a similar fashion to video vectorization and search. If the audio is analog format, it is digitized and divided into short frames, each of which represent a segment of the audio. Each frame is normalized, filtered and transformed with various techniques, and finally stored as a feature vector [28]. The whole audio is therefore a sequence of feature vectors, all together representing a spoken word or sentence, a song, or some other type of audio. If voice recognition is used in user authentication, similar process may be applied to a spoken keyphrase, and the vectorized spoken keyphrase compared with vectorized recordings. On the other hand, if voice recognition is used in a conversational agent, the sequences of vectors can be used as an input for, e.g., neural networks to recognize and classify spoken words, and to respond accordingly in text or synthethized voice using a generative model such as ChatGPT. The examples of returning similar-looking cats and using voice recognition to authenticate users serve as opposing prime examples of tolerance in similarity search. While several images of similar cats may be returned with relatively low tolerance, authenticating an user by their voice requires high tolerance.

4.4. Chatbots and long-term memory

VDBMs can be used for long-term memory of chatbots or other generative models. Illustrated on the left-hand side of Fig. 5b, a large dataset is first used to train a model capable of imitating understanding and producing natural language to a certain degree. A VDBMS can be used to store and index the model, although this can be done in other ways as well. Generative models have limitations when it comes to remembering past conversations or context, and currently, several technical limitations contribute to this



(a) By vectorizing images (left-hand side) by their features, reverse image search can be used to find similar images (right-hand side)



(b) A VDBMS can be used for long-term storage of chatbot conversations, which can be then queried and used as additional context for generative models; the query with an asterisk contains both the original user prompt as well as similar conversations, which are both used as query vectors for the generative model

Figure 5: Uses-cases for VDBMSs in the domains of image similarity search and chatbots; note how here all the VDBMSs handle the vectorization of data – this is not usually the case

challenge. For example, several models can only consider a limited amount of preceding text when generating a response. Consequently, they cannot recall detailed information from long conversations [29]. Generative models do not have a built-in memory of past interactions, and generate responses based on the immediate context provided in the input. Once a conversation becomes too lengthy or complex, the model's ability to reference earlier parts of the conversation diminishes. Furthermore, generative models are trained on large datasets, but they lack the ability to distinguish between factual information and user-specific interactions. This can lead to instances where the model provides inconsistent or incorrect information based on the training data [cf. e.g., 30].

To counter these limitations, a VDBMS may be used as a long-term memory in such use-cases. As illustrated on the right-hand side of Fig. 5b, when an end-user prompts (i.e., submits a query to) a chatbot, the natural language query is vectorized and used as a query vector for a long-term memory vector database to find top-k similar conversations. The query (i.e., the user prompt in vectorized and natural language form) is also stored in the long-term memory vector database. Next, the original user prompt as well as k similar past conversations are used as query vectors for the generative model (marked with an asterisk

in Fig. 5b). The generative model then generates a response, which is inserted in the long-term memory database in vectorized and natural language form, and returned to the application (i.e., the chatbot user interface). This approach not only allows the chatbot to remember past conversations, but also enables personalized information, conversation sequence encoding, and timestamps through vector metadata, and potentially reduces the use of computational resources without the need to retrain or fine-tune the generative model.

In summary, the previous subsections illustrate different use-cases for VDBMSs, yet it can be seen that the function of the VDBMS is rather uniform regardless of the use-case. That is, from a transaction processing perspective, the VDBMS stores, indexes and retrieves vectors, and domain-specific processes such as image feature extraction are carried out in other parts of the system.

5. Current challenges

5.1. Balancing between speed and accuracy

As most queries in VDBMSs operate by searching approximate nearest neighbors, balancing between query response time and the accuracy of the results is a trade-off largely dictated by the business domain. Some vector index types such as *Product Quantization* save storage space and speed up queries by abstracting and aggregating information with the cost of accuracy, while other index types such as *R-trees* are lossless. Lossless indices are preferred when exact similarity measurements are critical, while lossy indices are used when approximate similarity searches are acceptable, and there is a need to reduce storage and computation costs.

The challenge in choosing between speed and accuracy is two-fold. First, compared to many other data models, the concept of query accuracy plays a significantly larger role. Although many NoSQL data models forsake data integrity for eventual consistency, the effects of such design principles for the end-user are relatively small compared to inaccurate vector searches. On the other hand, VDBMSs disregard many challenges related to other data models, such as the complexity of querying in relational databases, and respective challenges and complexities in logical database design in both relational and NoSQL data models. Second, the trade-offs between speed and accuracy are emphasized in especially large datasets where both speed and query accuracy are critical. For example, natural language operated decision support systems which use large corporate datasets or stock market data need to provide decisions fast, but without returning inaccurate or untrue results. One possible solution for ensuring both speed and accuracy is utilizing several indices for the same vectors, yet this approach naturally requires more storage capacity.

5.2. Growing dimensionality and sparsity

The growing dimensionality of vectors is a challenge. As the needs of the domain grow, it is natural that the vectorized data need more features. For example, it is reasonable to assume that a vector database of Greek plays will soon require more insights on the plays besides the amount of comedy and tragedy. This leads to increased storage requirements and computational complexity.

Increased dimensionality also impacts similarity search, as the notion of proximity becomes less reliable in high-dimensional spaces. Euclidean distance, which is commonly used in low-dimensional spaces, becomes less reliable in high-dimensional spaces due to the concentration of points around the surface of the space. That is, in high-dimensional spaces, the volume of space grows exponentially with new dimensions, while the possible number of vectors typically does not. This results in vectors naturally concentrating close to the hypersurface of the space, as that is where the majority of the space is. Because vectors are concentrated near the surface, the distances between the vectors tend to be more similar in high-dimensional space [31]. Developing effective distance measures that can capture the true similarity or dissimilarity between high-dimensional vectors is an ongoing research challenge.

Another challenge is the increased sparsity of high-dimensional vectors. As the number of dimensions grows, the available space becomes more sparsely populated, meaning that data points are spread out across the vector. For example, if a vector database consisting of feature vectors of images of cats is extended to cover images of other animals as well, vector dimensions associated with cats (or mammals, or chordates, etc.) do not contribute significantly to the overall structure of vectors depicting other animals. This sparsity complicates indexing and retrieval, as methods designed for denser data struggle to efficiently represent and

query sparse data. The challenges associated with sparse data have been addressed in, e.g., the column-family data model, but not in the degree that is required with high-dimensional sparse vectors.

5.3. Achieving general maturity

DBMSs are typically large and complex pieces of software. It follows that there are several aspects to DBMSs that evolve and mature over time, and because VDBMSs are relatively novel systems (cf. Table 2), considerations such as their stability, reliability, and optimization are subject to even drastic development. In comparison, even mature relational DBMSs still receive critical bug fixes [32].

Maturity is not a goal in on itself. Decades of development and testing have likely addressed many bugs and stability issues, making them more reliable for mission-critical applications. Over time, DBMSs tend to accumulate a rich set of features and functionalities. They often support a wide range of data models, query languages, and storage options, allowing them to cater to diverse use-cases. This is not necessarily the case with more novel VDBMSs. Additionally, a mature DBMS typically has a large and active user community, which can be valuable for getting support, finding resources such as online tutorials, and leveraging third-party extensions and integrations.

Information security is a challenge that is not limited to business domains of VDBMSs. Due to their common use-cases, a vector database may contain sensitive information such as conversations intended to be private, biometric data, risk assessment profiles, and geospatial intelligence data. While many similar use-cases are common in relational databases as well, older DBMSs have had time to identify and address more security vulnerabilities, and usually have robust security features and practices in place.

In summary, there are several open challenges regarding VDBMSs, some of which are related to algorithms such as the need for novel index structures, some to software such as the availability of certain features in VDBMSs, and some to social aspects such as the maturity and availability of online support. In the future, we expect the demand for vector databases to grow. Consequently, we expect VDBMS vendors to focus on developing and applying new algorithms for vector indices, as well as making high-dimensional vectors more human-readable through visualizations. Additionally, as data-intensive computational models are computationally expensive to retrain, we expect that VDBMS vendors will try to address this by implementing features for incremental learning, i.e., cost-effective fine-tuning of computational models.

6. Conclusion

Vector database is growing a data model intended for storing vectors which describe rich data in high-dimensional vectors. This study provided an overview of fundamental concepts behind vector databases and vector database management systems, such as different types of vector similarity comparison types, different vector index types, and the principal software components in a VDBMS. Additionally, this study described some VDBMs and their features, as well as some popular use-cases for vector data such as chatbots and image similarity search. Finally, this study discussed some of the current challenges associated with VDBMSs such as high-dimensionality and sparsity of vector data, and the relative novelty of VDBMS products and the implications therein.

Acknowledgements

Bitmap images of cats in Fig. 5 were generated with DALL-E.

References

[1] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, C. Xie, Milvus: A purpose-built vector data management system, in: Proceedings of the 2021 International Conference on Management of Data, ACM, 2021. URL: https://doi.org/10.1145%2F3448016.3457550. doi:10.1145/3448016.3457550.

[2] R. Gasser, L. Rossetto, S. Heller, H. Schuldt, Cottontail DB: An open source database system for multimedia retrieval and analysis, in: Proceedings of the 28th ACM International Conference on Multimedia, ACM, 2020. URL: https://doi.org/10.1145%2F3394171.3414538. doi:10.1145/3394171.3414538.

[3] F. Li, Modernization of databases in the cloud era: Building databases that run like Legos, Proceedings of the VLDB Endowment 16 (2023) 4140–4151.

[4] G. Touya, I. Lokhat, Deep learning for enrichment of vector spatial databases, ACM Transactions on Spatial Algorithms and Systems 6 (2020) 1–21.

[5] J. Johnson, M. Douze, H. Jegou, Billion-scale similarity search with GPUs, IEEE Transactions on Big Data 7 (2021) 535–547.

[6] P. Covington, J. Adams, E. Sargin, Deep neural networks for YouTube recommendations, in: Proceedings of the 10th ACM Conference on Recommender Systems, ACM, 2016. URL: https://doi.org/10.1145%2F2959100.2959190. doi:10.1145/2959100.2959190.

[7] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in: Y. Bengio, Y. LeCun (Eds.), 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, 2013. URL: http://arxiv.org/abs/1301.3781.

[8] Q. V. Le, T. Mikolov, Distributed representations of sentences and documents, in: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014, volume 32 of *JMLR Workshop and Conference Proceedings*, 2014, pp. 1188–1196. URL: http://proceedings.mlr.press/v32/le14.html.

[9] T. Kraska, A. Beutel, E. H. Chi, J. Dean, N. Polyzotis, The case for learned index structures, in: Proceedings of the 2018 International Conference on Management of Data, ACM, 2018. URL: https://doi.org/10.1145%2F3183713.3196909. doi:10.1145/3183713.3196909.

[10] T. Ge, K. He, Q. Ke, J. Sun, Optimized product quantization for approximate nearest neighbor search, in: 2013 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2013. URL: https://doi.org/10.1109%2Fcvpr.2013.379. doi:10.1109/cvpr.2013.379.

[11] H. Jégou, M. Douze, C. Schmid, Product quantization for nearest neighbor search, IEEE Transactions on Pattern Analysis and Machine Intelligence 33 (2011) 117–128.

[12] B. Zheng, X. Zhao, L. Weng, N. Q. V. Hung, H. Liu, C. S. Jensen, PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search, Proceedings of the VLDB Endowment 13 (2020) 643–655.

[13] W. Zhao, S. Tan, P. Li, SONG: Approximate nearest neighbor search on GPU, in: 2020 IEEE 36th International Conference on Data Engineering (ICDE), IEEE, 2020. URL: https://doi.org/10.1109%2Ficde48307.2020.00094. doi:10.1109/icde48307.2020.00094.

[14] Y. A. Malkov, D. A. Yashunin, Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, IEEE Transactions on Pattern Analysis and Machine Intelligence 42 (2020) 824–836.

[15] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD international conference on Management of data, 1984, pp. 47–57.

[16] C. Silpa-Anan, R. Hartley, Optimised KD-trees for fast image descriptor matching, in: 2008 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2008. URL: https://doi.org/10.1109%2Fcvpr.2008.4587638. doi:10.1109/cvpr.2008.4587638.

[17] S. Dasgupta, Y. Freund, Random projection trees and low dimensional manifolds, in: Proceedings of the fortieth annual ACM symposium on Theory of computing, ACM, 2008. URL: https://doi.org/10.1145%2F1374376.1374452. doi:10.1145/1374376.1374452.

[18] J. Wang, W. Liu, S. Kumar, S.-F. Chang, Learning to hash for indexing big data - a survey, Proceedings of the IEEE 104 (2016) 34–57.

[19] DB-Engines, vector database management systems, https://db-engines.com/en/ranking/vector+dbms, 2023. Accessed: 2023-08-30.

[20] B. Windsor, K. Choi, Thistle: A vector database in Rust, 2023. arXiv:2303.16780.

[21] A. C. Mater, M. L. Coote, Deep learning in chemistry, Journal of Chemical Information and Modeling 59 (2019) 2545–2559.

[22] M. Grbovic, H. Cheng, Real-time personalization using embeddings for search ranking at Airbnb, in: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 311–320. doi:10.1145/3219819.3219885.

[23] F. Baldassarre, D. G. Morín, L. Rodés-Guirao, Deep koalarization: Image colorization using CNNs and Inception-ResNet-v2, arXiv preprint arXiv:1712.03400 (2017).

[24] S. Bashyal, G. K. Venayagamoorthy, Recognition of facial expressions using Gabor wavelets and learning vector quantization, Engineering Applications of Artificial Intelligence 21 (2008) 1056–1064.

[25] S. Sahoo, N. Paul, A. Shah, A. Hornback, S. Chava, The universal NFT vector database: A scaleable vector database for NFT similarity matching, arXiv preprint arXiv:2303.12998 (2023).

[26] D. Shankar, S. Narumanchi, H. Ananya, P. Kompalli, K. Chaudhury, Deep learning based large scale visual recommendation and search for e-commerce, arXiv preprint arXiv:1703.02344 (2017).

[27] A. Herbulot, S. Jehan-Besson, S. Duffner, M. Barlaud, G. Aubert, Segmentation of vectorial image features using shape gradients and information measures, Journal of Mathematical Imaging and Vision 25 (2006) 365–386.

[28] G. Venayagamoorthy, V. Moonasar, K. Sandrasegaran, Voice recognition using neural networks, in: Proceedings of the 1998 South African Symposium on Communications and Signal Processing-COMSIG '98 (Cat. No. 98EX214), 1998, pp. 29–32. doi:10.1109/COMSIG.1998.736916.

[29] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, D. Metzler, Long range arena: A benchmark for efficient transformers, arXiv preprint arXiv:2011.04006 (2020).

[30] M. Zhang, O. Press, W. Merrill, A. Liu, N. A. Smith, How language model hallucinations can snowball, 2023. arXiv:2305.13534.

[31] P. Indyk, R. Motwani, Approximate nearest neighbors, in: Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98, ACM Press, 1998. URL: https://doi.org/10.1145%2F276698.276876. doi:10.1145/276698.276876.

[32] Oracle critical patch update advisory - january 2023, https://www.oracle.com/security-alerts/cpujan2023.html#AppendixDB, 2023. Accessed: 2023-09-15.