An introduction to the vector database.

@Milvus.io

5/25/2022

Getting Started with Vector Databases - What is a Vector Database?

Vector databases from 1000 feet

In the previous tutorial, we took a quick look at the ever-increasing amount of data that is being generated on a daily basis. We then covered how these bits of data can be split into structured/semi-structured and unstructured data, the differences between them, and how modern machine learning can be used to understand unstructured data through embeddings. Finally, we briefly touched upon unstructured data processing via ANN search. Through all of this information, it's now clear that the ever-increasing amount of unstructured data requires a paradigm shift and a new category of database management system - the vector database.

Armed with this knowledge, it's now clear what vector databases are used for: searching across images, video, text, audio, and other forms of unstructured data via their *content* rather than keywords or tags (which are often input manually by users or curators). When combined with powerful machine learning models, vector databases such as Milvus have the ability to revolutionize e-commerce solutions, recommendation systems, computer security, pharmaceuticals, and many other industries.

As mentioned in the previous section, a vector database is a fully managed, no-frills solution for storing, indexing, and searching across a massive dataset of unstructured data. A vector database does this by leveraging the power of embeddings from machine learning models. In concert with the underlying technology, usability is also an incredibly important attribute for vector databases. Specifically, a mature vector database should implement the following features (many of these features overlap with those of databases for structured/semi-structured data):

- Scalability and tunability: As the number of unstructured data elements stored in a vector database grows into the hundreds of millions or billions, horizontal scaling across multiple nodes becomes paramount. Furthermore, differences in insert rate, query rate, and underlying hardware may result in different application needs, making overall system tunability a mandatory feature for vector databases. Milvus achieves this via a cloud-native architecture, maintaining multiple service and worker nodes behind a load balancer. Internal object storage and message passing is implemented with other cloud-native, distributed tools, allowing for easy scaling across the entire system.
- Multi-tenancy and data isolation: Supporting multiple users is an obvious feature for all database systems. Parallel to this notion is data isolation - the idea that any inserts, deletes, or queries made to one collection in a database should be invisible to the rest of the system unless the collection owner explicitly wishes to share the information. Milvus implements this through the concept of *collections*, which we'll dive deeper into in a future tutorial.
- A complete suite of APIs: A database without a full suite of APIs and SDKs is, frankly speaking, not a real database. Milvus maintains Python, Node, Go, and Java SDKs for communicating with

and administering a Milvus database.

- An intuitive user interface/administrative console: User interfaces can help significantly reduce the learning curve associated with vector databases. These interfaces also expose new vector database features and tools that would otherwise be inaccessible. Zilliz has open-sourced an efficient and intuitive web-based GUI for Milvus called *Attu*.

In the next two sections, we'll compare vector databases versus vector search libraries and vector search plugins, respectively.

Vector databases versus vector search libraries

A common misconception is that vector databases are merely wrappers around ANN search indices. A vector database is, at its core, a full-fledged solution for unstructured data. As we've already seen in the previous section, this means that user-specific features present in today's database management systems for structured/semi-structured data - cloud-nativity, multi-tenancy, scalability, etc - should also be attributes for a mature vector database as well.

On ther other hand, projects such as FAISS, ScaNN, and HNSW are lightweight *ANN libraries* rather than managed solutions. The intention of these libraries is to aid in the construction of vector indices - data structures designed to significantly speed up nearest neighbor search for multi-dimensional vectors[1]. If your dataset is small and limited, these libraries can prove to be sufficient for unstructured data processing, even for systems running in production. However, as dataset sizes increase and more users are onboarded, the problem of scale becomes increasingly difficult to solve.

[1] We'll go over vector indices in more detail in an upcoming tutorial, so stay tuned.

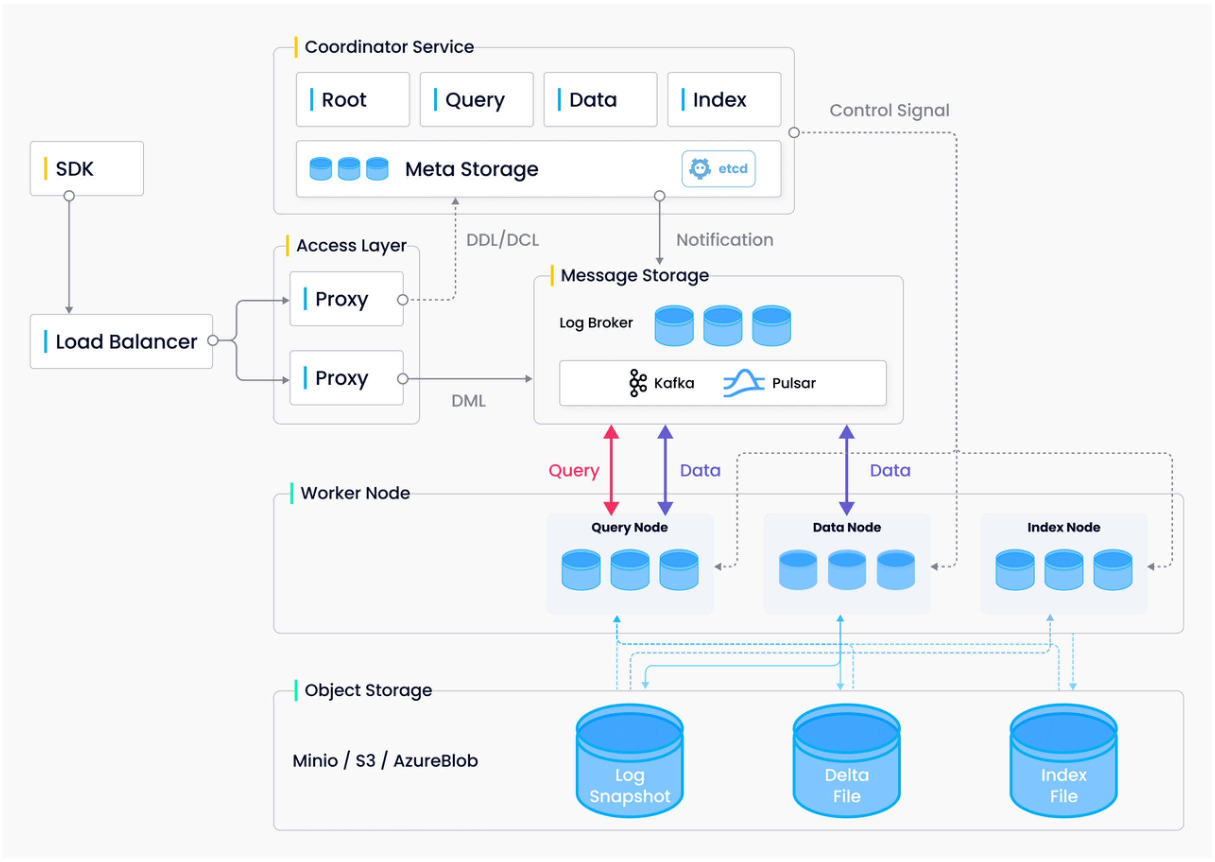

Figure 1: High-level overview of Milvus's architecture. We will dive into each component in detail in future articles.

Another key difference between vector databases and vector search libraries is the layer of abstraction with which they operate - vector databases are full-fledged services, while ANN libraries are meant to be integrated into whatever application is being developed. In this sense, ANN libraries are one of the many components that vector databases are built on top of, similar to how Elasticsearch is built on top of Apache Lucene. To give an example of why this abstraction is so important, let's take the inserting a new unstructured data element into a vector database. In Milvus, this can be done as follows:

```python
from pymilvus import Collection
collection = Collection('book')
mr = collection.insert(data)
```

With a library such as FAISS or ScaNN, there is unfortunately no easy way of doing this without manually re-creating the entire index at certain checkpoints. Even if there was, vector search libraries still lack scalability and multi-tenancy, two of the most important vector database features (from the previous section).

Vector plugins for traditional databases

An increasing number of traditional databases and search systems such as Clickhouse and Elasticsearch are including built-in vector search plugins. Elasticsearch 8.0, for example, includes vector insertion and ANN search functionality that can be called via restful API endpoints.

The problem with vector search plugins should be clear as night and day - these solutions do not take a full-stack approach to embedding management and vector search. Instead, these plugins are meant to be enhancements on top of existing architectures. These vector search plugins are therefore limited and unoptimized.

Let's go back to the list of features that a vector database should implement (from the first section). Vector search plugins are missing two of these features - tunability and user-friendly APIs/SDKs. To illustrate this, we'll continue to use Elasticsearch's ANN engine as an example; other vector search plugins operate very similarly so we won't go too much further into detail.

Elasticsearch supports vector storage via the `dense_vector` data field type and allows for querying via the `_knn_search` endpoint:

```
PUT index
{
 "mappings": {
   "properties": {
     "image-vector": {
       "type": "dense_vector",
       "dims": 128,
       "index": true,
       "similarity": "l2_norm"
     }
   }
 }
}

PUT index/_doc
{
 "image-vector": [0.12, 1.34, ...]
}
```

```
GET index/_knn_search
{
 "knn": {
   "field": "image-vector",
   "query_vector": [-0.5, 9.4, ...],
   "k": 10,
   "num_candidates": 100
 }
}
```

Elasticsearch's ANN plugin supports only the Hierarchical Navigable Small World index (HNSW) with Euclidean distance as a similarity metric. This is a good start, but let's compare it to Milvus, a full-fledged vector database. Using `pymilvus`:

```
>>> field1 = FieldSchema(name='id', dtype=DataType.INT64, description='
    int64', is_primary=True)
>>> field2 = FieldSchema(name='embedding', dtype=DataType.FLOAT_VECTOR,
     description='embedding', dim=128, is_primary=False)
>>> schema = CollectionSchema(fields=[field1, field2], description='
    hello world collection')
>>> collection = Collection(name='my_collection', data=None, schema=
    schema)
>>> index_params = {
        'index_type': 'IVF_FLAT',
        'params': {'nlist': 1024},
        "metric_type": 'L2'}
>>> collection.create_index('embedding', index_params)
```

```
>>> search_param = {
        'data': vector,
        'anns_field': 'embedding',
        'param': {'metric_type': 'L2', 'params': {'nprobe': 16}},
        'limit': 10,
        'expr': 'id_field > 0'
    }
>>> results = collection.search(**search_param)
```

While both Elasticsearch and Milvus have methods for creating indexes, inserting embedding vectors, and performing nearest neighbor search, it's clear from these examples that Milvus has a more intuitive vector search API (better user-facing API) and richer index plus distance metric support (better tunability). Both of these are due to the fact that Milvus was built from the ground-up as a vector database. Milvus also plans to support more vector indices and allow for querying via SQL-like statements in the future, further improving both tunability and usability.

Technical challenges

Earlier in this tutorial, we listed the desired features a vector database should implement, before comparing vector databases to vector search libraries and vector search plugins, respectively. Armed with this knowledge, let's briefly go over some high-level technical challenges associated with modern vector databases. In future tutorials, we'll provide an overview of how Milvus tackles each of these, in addition to how these technical decisions improve Milvus' performance over other open-source vector databases.

Like traditional databases, vector databases are composed of a number of evolving components. Roughly speaking, these can be broken down into the storage, the index, the service. Although these three components are tightly integrated[2], companies such as Snowflake have shown the broader storage industry that hybrid database architectures are arguably superior to the traditional "shared storage" cloud database models. Thus, the first technical challenge associated with vector databases is *designing a flexible and scalable data model*.

Given a data model, querying and indexing is the next important component. The compute-heavy nature of machine learning and multi-layer neural networks has allowed GPUs, NPUs/TPUs, FPGAs, and other general purpose compute hardware to flourish. Vector indexing and querying is also compute-heavy, operating at maximum speed and efficiency when run on accelerators. This diverse set of compute resources gives way to the second main technical challenge, *developing a heterogeneous computing architecture*.

The third primary technical challenge associated with vector database development ties closely into the API and user interface bullet points mentioned in the first section. While a new category of database necessitates a new architecture in order to extract maximal performance at minimal cost, the majority of vector database users are still acclimcated traditional CRUD operations (e.g. `INSERT`, `SELECT`, `UPDATE`, and `DELETE` in SQL). Therefore, the final primary challenge is *developing a set of APIs and GUIs that leverage existing user interface conventions* while maintaining compatibility with the underlying architecture.

Note how each of the three components corresponds to a primary technical challenge. With that being said, there is no one-size-fits-all architecture for vector databases. The best vector databases will fulfill all of these technical challenges by focusing on delivering the features mentioned in the first section.

Wrapping up

In this tutorial, we took a quick tour of vector databases. Specifically, we looked at 1) what features go into a mature vector database, 2) how a vector database differs from vector search libraries, 3) how a vector database differs from vector search plugins in traditional databases or search systems, and 4) the key challenges associated with building a vector database.

This tutorial is not meant to be a deep dive into vector databases, nor is it meant to show how vector databases can be used in applications. Rather, the goal is to provide an overview of vector databases. This is where your journey truly begins!

In the next tutorial, we'll provide an *introduction to Milvus*, the world's most popular open-source vector database:

- We'll provide a brief history of Milvus, including the most important question - where does the name come from!

[2] Updating the storage component, for example, will impact how the vector indices are built in addition to how the user-facing services implement reads, writes, and deletes.

- We'll cover how Milvus 1.0 differs from Milvus 2.0 and where the future of Milvus lies.
- We'll discuss the differences between Milvus and other vector databases such as Google Vertex AI's Matching Engine.
- We'll briefly go over some common vector database applications.

See you in the next tutorial.