JaguarDB

Jaguar Vector Database

Artificial Intelligence and Database

Artificial intelligence (AI) often relies on vector databases for various tasks such as natural language processing, information retrieval, recommendation systems, and similarity matching. The use of vector databases is particularly relevant in the context of machine learning models that leverage embeddings, which are numerical representations of data elements in a continuous vector space.

Here are a few reasons why AI may benefit from vector databases:

Efficient Representation

Vectors provide a compact and efficient representation of complex data structures. By transforming data elements into vectors with thousands of dimensions, AI systems can work with numerical representations that are more amenable to mathematical operations and analysis.

Similarity and Distance Metrics

Vector databases enable the computation of similarity or distance metrics between vectors, such as cosine similarity or Euclidean distance. These metrics are fundamental for tasks like similarity matching, nearest neighbor searches, and clustering, which are essential in recommendation systems, content retrieval, and data exploration.

Information Retrieval

Vector databases allow AI systems to index and retrieve relevant information efficiently. By converting textual or multimedia content into vectors, it becomes possible to organize and search through large volumes of data quickly. For instance, in a search engine, a vector representation of documents enables fast retrieval of relevant documents based on user queries.

Recommendation Systems

Vector databases facilitate the creation of recommendation systems by capturing user preferences and item characteristics. Embedding user interactions and item features into vectors allows AI models to measure the similarity between users and items, thereby generating personalized recommendations based on similar user-item pairs.

Semantic Understanding

Vector representations can capture semantic relationships between words or concepts. Techniques like word embeddings (e.g., Word2Vec or GloVe) map words into vectors, where similar words are located closer in the vector space. This enables AI models to understand the context, meaning, and semantic relationships between words, which is vital for natural language understanding and generation tasks.

Vector databases provide a framework for organizing, manipulating, and querying data in AI systems. By leveraging vector representations and associated operations, AI models can effectively process and analyze complex information, leading to improved performance in various applications.

Why Jaguar Vector Database

In the fields of generative AI, the exponential growth of data is inevitable. From voluminous vector data to vast collections of photos and videos, the potential for information generation knows no bounds. However, efficiently managing this diverse and ever-expanding data landscape poses a significant challenge for traditional database and storage systems. AI-generated data can quickly accumulate and consume significant storage space. Storing and managing this massive amount of data requires robust and scalable infrastructure. Organizations need to invest in adequate storage solutions, such as cloud storage or distributed file systems, to accommodate the growing data volumes.

Traditional databases rely on consistent hashing techniques, which, unfortunately, lead to excessive data migration. During the constant expansion of data systems, incremental scaling operations often require data migration for almost every piece of data and impose substantial costs on the system. These costs manifest in various forms, including increased power consumption, hardware wear and tear, and degraded performance.

The innovative ZeroMove technique is employed in JaguarDB that offers a revolutionary solution. In contrast to the consistent hashing algorithm, which requires data migration when scaling out the system, ZeroMove enables scaling without the need to move data between computers. Data is intelligently tagged with encoded identifiers to facilitate efficient host location. These encoded identifiers serve as unique markers that enable swift and accurate retrieval of data within the system. Our approach ensures that data remains in the host where it is hashed, thereby increasing availability, and improving system performance.

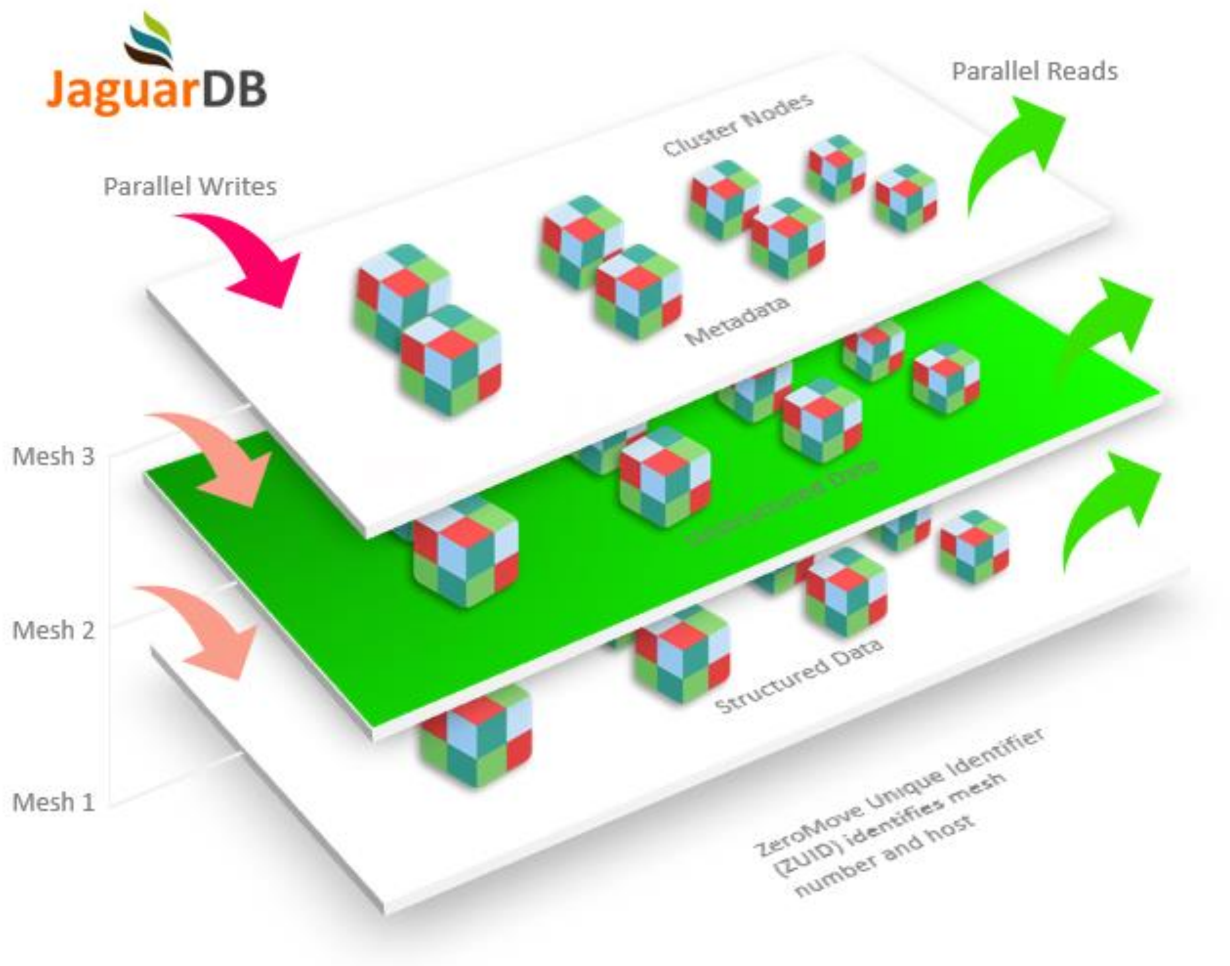
Thanks to the revolutionary and unique ZeroMove technology, our vector database offers the following key benefits:

Scalability

Reducing the amount of data migration of a distributed system can significantly improve the system's scalability by minimizing the disruption and potential risks associated with migrating large amounts of data. The benefit of zero data migration is more pronounced when data replication strategies are implemented.

Simplification

Data migration can be a complex and time-consuming process, especially in large-scale distributed database systems. By avoiding data migration, the system is simplified, and the potential for errors and downtime associated with data migration is reduced.

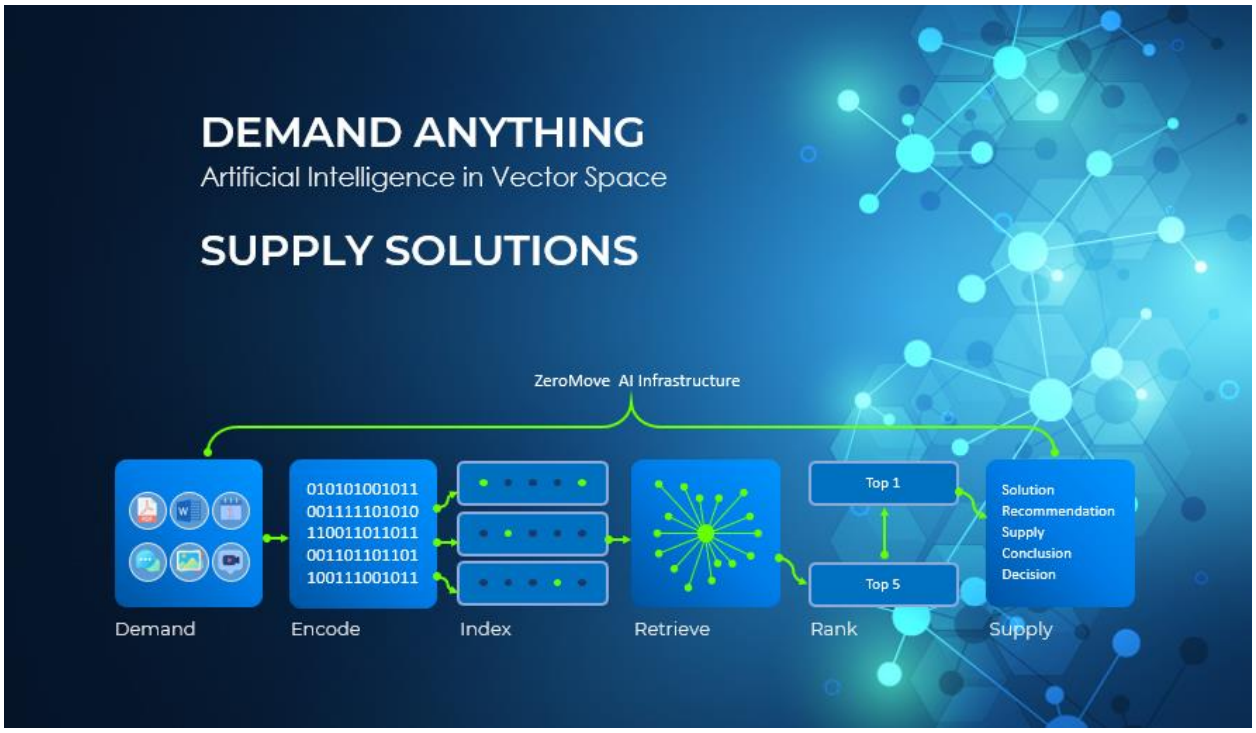


Enhanced Consistency

Data consistency refers to the correctness of one data in relation to another data. It can be a challenge in distributed systems, especially during data migration. By avoiding data migration, the system can potentially maintain better data consistency.

Cost Reduction

Data migration can be expensive, especially if it is big and requires a lot of resources and time. By avoiding data migration, the system can potentially save on equipment, network, and administrative costs.



Use Cases of Vector Databases

Vector databases have a wide range of applications across various industries. Here are some notable use cases:

Recommendation Systems

Vector databases can be used in recommendation systems to provide personalized recommendations to users. By representing items and user preferences as vectors, the database can efficiently compute similarities and make accurate recommendations.

Collaborative Filtering: Collaborative filtering is a popular recommendation technique that analyzes user behavior and item interactions. Vector databases can store user-item interaction data, such as ratings, preferences (size, color, brand, weight, height, quality, price, accessory), or purchase history, as vectors. By comparing the vectors of different users or items, the database can identify similar patterns and make recommendations based on the preferences of similar users or items.

Content-Based Filtering: Content-based filtering focuses on the characteristics and attributes of items to make recommendations. Vector databases can store item features or attributes as vectors, such as genre, keywords, or textual descriptions. By analyzing the similarity between item vectors, the database can suggest items with similar features to those previously liked or interacted with by the user.

Contextual Recommendations: Vector databases can incorporate contextual information to enhance recommendations. Contextual factors such as time, location, season, device, or user demographics can be represented as additional dimensions in the vectors. By considering contextual vectors along with user and item vectors, the database can generate context-aware recommendations that align with specific situations or user contexts.

Multi-Domain Recommendations: Vector databases can support recommendations across multiple domains or types of items. By representing items from different domains as vectors, the database can provide cross-domain recommendations. For example, it can suggest movies based on the user's preferences in music or vice versa.

Image and Video Search

Vector databases can power image and video search engines by representing visual features of media files as vectors. This enables fast and accurate similarity search, content-based retrieval, and visual recommendation systems.

Vector databases can be used to perform content-based image retrieval (CBIR). Images are represented as high-dimensional feature vectors extracted from various visual descriptors such as color, texture, shape, or deep learning-based features. When a user submits a query image, its features are compared against the feature vectors in the database to find visually similar images.

Vector databases can power recommendation systems based on visual content. By representing images as vectors, the database can quickly identify visually related items and provide personalized recommendations. This is particularly useful in e-commerce, where users can discover visually similar products or related visual content.

Vector databases enable efficient video similarity search, allowing users to find videos that are visually similar to a given query video. Videos can be represented using features such as frame-level descriptors or temporal embeddings. The database can then perform similarity calculations and retrieve videos with similar visual content or style. Vector databases can support reverse image search capabilities, where users can submit an image as a query and retrieve similar images from the database. This is useful for applications like finding the original source of an image, identifying visually similar images across the web, or searching for visually related content.

Vector databases can aid in content filtering by analyzing visual features of images or videos. This can be applied in scenarios like moderating user-generated content, identifying explicit or inappropriate visuals, or ensuring compliance with content policies.

Natural Language Processing (NLP)

Vector databases find application in NLP tasks such as document similarity, sentiment analysis, and semantic search. Textual data can be transformed into high-dimensional vector representations, enabling efficient indexing and retrieval.

Document Similarity and Clustering: Vector databases can be used to measure the similarity between documents. Textual data is transformed into vector representations, such as word embeddings or document embeddings. By comparing the vectors of different documents, the database can identify similar content, cluster related documents, and enable efficient document search.

Sentiment Analysis: Vector databases can aid in sentiment analysis, where the sentiment or emotion expressed in text is determined. Textual data is transformed into vectors, and sentiment analysis algorithms can be applied to analyze the sentiment associated with different vectors. This can be useful in social media monitoring, customer feedback analysis, or brand reputation management.

Semantic Search: Vector databases enable semantic search capabilities, allowing users to find documents or passages related to specific concepts rather than just keyword matches. By representing text as vectors, the database can perform similarity calculations and retrieve documents with similar semantic meaning, even if the wording differs.

Machine Translation: Vector databases can support machine translation systems by storing vector representations of words, phrases, or sentences in different languages. By comparing the vectors of source and target language segments, the database can assist in finding the most appropriate translation equivalents, improving the quality and efficiency of machine translation.

Question Answering: Vector databases can aid in question answering systems, where natural language questions are answered based on a collection of documents or knowledge bases. By transforming the text into vectors and using techniques like semantic matching, the database can identify relevant passages or documents that provide answers to specific questions.

Fraud Detection

Vector databases can be used in fraud detection systems to analyze patterns and anomalies. By representing user behavior or transaction data as vectors, the database can quickly identify suspicious activities or detect fraudulent patterns.

Anomaly Detection: Vector databases can be utilized for anomaly detection in fraud detection systems. By representing user behavior or transaction data as vectors, the database can establish normal patterns or profiles based on historical data. Any deviations or anomalies from these patterns can be quickly identified, signaling potential fraudulent activities.

Network Analysis: Vector databases can aid in fraud detection by analyzing connections and relationships between entities. By representing entities such as individuals, accounts, or devices as vectors, the database can identify fraudulent networks or organized fraudulent activities through link analysis, graph algorithms, and clustering techniques.

Real-time Monitoring: Vector databases can support real-time monitoring and detection of fraudulent activities. By continuously updating and analyzing vectors representing ongoing transactions or user interactions, the database can quickly identify and flag potential fraud in real-time, enabling timely intervention and prevention.

Multi-channel Fraud Detection: Vector databases can facilitate fraud detection across multiple channels, such as online transactions, mobile applications, or call centers. By integrating data from various sources and representing them as vectors, the database can perform cross-channel analysis to identify fraudulent activities that span multiple channels.

Historical Analysis and Trend Identification: Vector databases can store and analyze historical fraud data, enabling the identification of long-term trends, evolving fraud patterns, and emerging threats. By representing historical fraud incidents as vectors, the database can uncover patterns and behaviors that may not be immediately apparent, assisting in proactive fraud prevention measures.

Genome Analysis

Vector databases can be utilized in genomics research for DNA sequence analysis, variant calling, and genetic similarity comparisons. By representing genetic information as vectors, researchers can perform efficient searches and comparisons on large genomic datasets.

DNA Sequence Analysis: Vector databases can store and analyze DNA sequences, which are typically represented as strings of nucleotides (A, T, C, G). By representing DNA sequences as vectors, the database can efficiently process and search for specific patterns, motifs, or variations within the genome. This is crucial for tasks such as gene discovery, functional annotation, and identification of genetic variations associated with diseases.

Variant Calling: Variant calling is the process of identifying genetic variations or mutations within an individual's genome. Vector databases can store genomic data in a structured and efficient manner, allowing for the comparison and analysis of genomic variations across different individuals or populations. This aids in identifying disease-causing mutations, understanding genetic diversity, and facilitating precision medicine.

Comparative Genomics: Vector databases enable comparative genomics, where the genomes of different species or individuals are compared to identify similarities and differences. By representing genomes as vectors, the database can perform efficient similarity calculations, phylogenetic analyses, and identification of conserved regions or genes across species.

Personalized Medicine: Vector databases can support personalized medicine by integrating genomic data with clinical information. By representing patient genomes as vectors and combining them with relevant clinical data, the database can aid in identifying genetic markers for disease predisposition, predicting treatment response, and guiding personalized therapeutic approaches.

These are just a few examples of how vector databases can be applied in various domains. The flexibility and efficiency of vector representations make them suitable for a wide range of data-intensive applications, enabling fast and accurate analysis and retrieval of information.

Green Technology

By leveraging ZeroMove™ technology, JaguarDB contributes to reducing carbon footprint and minimizing energy consumption. This aligns with the growing awareness and efforts to promote sustainability across various industries.

Saving Energy: ZeroMove technology could save $140 billion worth of power consumption in the next ten years.

Less CO2 Emission: The ZeroMove groundbreaking technology could facilitate a reduction of approximately 620 billion pounds of CO2 emissions.

Competitors

JaguarDB, founded in 2013, has dedicated the past ten years to active development and rigorous testing, establishing itself as a robust and reliable database solution. In recent years, the market has witnessed the emergence of several competitors focused solely on handling vector data, known as vector databases. They have garnered attention within the AI community due to their specialized capabilities. However, they often operate in isolation or offer limited scalability, relying on traditional consistent hashing mechanisms as described earlier. Consequently, when faced with the colossal volumes of data generated in AI applications, these databases fail to provide the scalability required to support large-scale AI deployments effectively.

Moreover, JaguarDB distinguishes itself by addressing the scalability challenge inherent in large-scale AI applications. By leveraging innovative technologies and advanced algorithms, JaguarDB offers a highly scalable solution capable of handling massive amounts of data. Its unique ZeroMove architecture enables seamless horizontal scaling, allowing organizations to accommodate growing AI workloads effortlessly.

An AI data-lake is equally crucial for AI applications, as media data like images and videos tend to occupy more space compared to structured data. The ZeroMove technology is particularly potent when it comes to efficiently scaling AI data systems. This scalability, coupled with its extensive development and testing history, positions JaguarDB as a reliable choice for enterprises seeking to harness the full potential of AI while maintaining robust and scalable data storage and retrieval capabilities.

JaguarDB Features

JaguarDB is not just a distributed vector database; it is a comprehensive solution that goes beyond vector data management. While it excels at handling vector data, it also seamlessly processes non-vector data within a fully integrated framework.

JaguarDB stores high-dimensional vectors with the state-of-the-art HNSW graph index store. HNSW, short for Hierarchical Navigable Small World, is a data structure and algorithm used for approximate nearest neighbor search in high-dimensional spaces. It is designed to efficiently find data points that are close to a given query point in a high-dimensional space, without exhaustively searching through all data points.

HNSW creates a hierarchical structure of data points that forms a graph. Each level of the hierarchy is a different graph that represents the data points at different levels of detail. HNSW maintains a "small world" property, which means that even though the graph is not fully connected like a traditional graph, it is still possible to navigate from one node to another through a relatively small number of edges. HNSW constructs the hierarchical graph in a way that ensures data points are connected to nearby points, enabling efficient traversal of the graph to find approximate nearest neighbors.

When searching for the nearest neighbors of a query point, HNSW uses the hierarchical structure to quickly navigate through the graph, starting from coarse levels and refining the search as it descends deeper into the hierarchy. HNSW focuses on approximate search rather than exact search. It sacrifices perfect accuracy for improved search efficiency, which is valuable when dealing with high-dimensional data.

JaguarDB brings forth an advanced capability that enables users to engage in KNN similarity searches using a wide spectrum of distance metrics, which include fundamental measures like Euclidean and Manhattan distances, along with specialized metrics such as cosine, Jaccard, Hamming, and Minkowski distances. This diversity empowers users to tailor their similarity searches based on the specific characteristics of their data and the intricacies of their analytical requirements. Whether dealing with spatial relationships, binary patterns, or various dimensions of data, JaguarDB accommodates a versatile selection of distance metrics to ensure proper similarity computation.

JaguarDB offers the flexibility in hybrid search, or multimodal search, that combines multiple types of search techniques or data representations to optimize the search process for different types of queries. This approach is particularly useful in applications that handle heterogeneous data, where the data may include both vector-based embeddings and traditional structured or unstructured data.

The groundbreaking ZeroMove technique is a pivotal feature within JaguarDB, delivering a transformative solution. Diverging from conventional consistent hashing algorithms that demand data migration during system expansion, ZeroMove empowers seamless scalability devoid of the necessity to transfer data across machines. ZeroMove technology aligns particularly well with vector index stores, as the process of removing vectors from such stores might necessitate a comprehensive index reconstruction when data is relocated.

Time series data in JaguarDB refers to a sequence of data points that are ordered based on time intervals. Users of JaguarDB can collect data over successive time periods and let it automatically aggregate data over multiple time windows for real-time analysis, prediction, and decision-making in AI applications. Time series data often come from mobile targets, sensors, devices, financial markets, weather stations, social media, and more.

Geospatial data support of JaguarDB is instrumental in applications such as environmental monitoring, disaster response, agriculture, and natural resource management. Geospatial data often exhibits in the form of vectors such as ling strings and polygons. AI models equipped with geospatial insights can predict the spread of wildfires, monitor deforestation, optimize irrigation strategies, and assess the impact of climate change. These applications hinge on the AI system's ability to process and analyze geospatial data, allowing for timely and informed interventions.

JaguarDB provides data lake capability which represents a powerful feature that integrates storage capabilities directly into the database system, offering a seamless and unified solution for managing, analyzing, and retrieving both structured and unstructured data. This integration brings efficiency, flexibility, and scalability to the storage and processing of diverse data types within the JaguarDB environment.

Fault tolerance is of paramount significance in vector databases used in AI applications due to its role in ensuring system reliability, availability, and consistent performance, even in the face of unexpected errors or failures. This is particularly crucial in AI applications where accurate and timely data retrieval and processing are essential. JaguarDB offers tolerance of machine failures and network disconnections to ensure high availability of the vector database system.

Vector data can be replicated with multiple copies, a maximum of three, in JaguarDB to ensure data availability, reliability, and fault tolerance. It creates and maintains duplicate copies of data across multiple nodes. This redundancy is crucial for the effectiveness of AI systems.

Example of Using Jaguar Vector Database

The subsequent Python example illustrates the integration of JaguarDB into AI applications for the benefit of software engineers and data scientists. In this demonstration, the focus lies on the seamless storage of textual data, the creation of embeddings, and the execution of similarity searches within the text data corpus. The process entails identifying texts that closely correspond to a given query text. Notably, this operation is solely reliant on vector embeddings, rendering the inclusion of explicit keywords or search cues unnecessary.

The following Python code connects to JaguarDB instance:

```
    jag = jaguarpy.Jaguar()

    host = "127.0.0.1"
    port = sys.argv[1]
    user = "admin-api-key"
    database = "vdb"

    rc = jag.connect( host, port, user, database )
    print ("Connected to JaguarDB server" )
```

Next, a store containing vector column and other related data is created:

```
  jag.execute("create store  textvec ( key: zid uuid, value: v vector(1024,
'cosine_fraction_short'), text char(2048), source char(32) )")
```

In this statement, the "zid" field stands as an automatically generated unique identifier. The "v" field represents a vector, comprising two primary elements: an integer vector ID and an array of vector components. Notably, the dimension of the vector is set at 1024. The inclusion of "cosine" within the string "cosine_fraction_short" signifies the intention to employ the cosine distance metric for similarity searches conducted on the vector. The term "fraction" alludes to the anticipated fractional-format input data. It's worth noting that JaguarDB vector storage implements distinct quantization levels. Specifically, the short quantization mode leverages 16-bit quantization techniques to efficiently store vector data. There is no limit on the number of vectors in a store. Multiple vectors can be created on the same store, to capture various types of vectors for the same object. The “text” field can store text data for an object, with a maximum capacity of 2048 bytes. The field source indicates source place where the text was imported from.

Then we can create an index, connecting the integer vector ID of a vector to the unique “zid” field for search of other attributes of an object:

```
    jag.execute("create index  textvec_idx on  textvec(v, zid)")
```

With JaguarDB, users can store various types of vectors, such as feature vectors and embedding vectors. An embedding vector, often simply referred to as an "embedding", is a mathematical representation of a discrete item, such as a word, phrase, image, or any other entity, in a continuous vector space. This technique is commonly used in various fields, including natural language processing (NLP), computer vision, recommendation systems, and more. The primary idea behind embedding vectors is to capture semantic relationships between items by placing similar items closer together in the vector space.

In this example, we use the "BAAI/bge-large-en" pre-trained embedding model to generate embeddings for the text data. A pre-trained embedding model is a machine learning model that has been trained on a large dataset to create meaningful representations (embeddings) of items in a continuous vector space. These embeddings capture semantic relationships and contextual information about the items. Pre-training involves training the model on a specific task, such as language modeling or image classification, with the goal of learning general features and patterns from the data. These learned features can then be fine-tuned or used as-is for various downstream tasks. Pre-trained embedding models are especially popular in natural language processing (NLP) and computer vision. The model "BAA/bge-large-en" requires a dimension of 1024 on the vectors, which was specified in the statement when we created the store and the vector field.

```
    model = SentenceTransformer('BAAI/bge-large-en')
```

There are some simple required steps to setup and use the model. They are described in the github project github.com/fserv/jaguardb, in embedding ➔ text ➔ baai-bge-large ➔ README.md.

Next, we store a group of text data in the store:

```
text = "Human impact on the environment (or anthropogenic environmental
impact) refers to changes to biophysical environments and to ecosystems,
biodiversity, and natural resources caused directly or indirectly by humans."
    zuid1 = storeText( jag, model, text, “wiki” )

    text = "a group of people involved in persistent interpersonal
relationships, or a large social grouping sharing the same geographical or
social territory, typically subject to the same political authority and
dominant cultural expectations. Human societies are characterized by patterns
of relationships (social relations) between individuals who share a
distinctive culture and institutions; a given society may be described as the
total of such relationships among its constituent members."
    zuid2 = storeText( jag, model, text, “wiki”  )

    text = "In 1768, Astley, a skilled equestrian, began performing
exhibitions of trick horse riding in an open field called Ha'Penny Hatch on
the south side of the Thames River, England. In 1770, he hired acrobats,
tightrope walkers, jugglers and a clown to fill in the pauses between the
equestrian demonstrations and thus chanced on the format which was later
named a circus.  Performances developed significantly over the next fifty
years, with large-scale theatrical battle reenactments becoming a significant
feature. "
    zuid3 = storeText( jag, model, text, “wiki”   )
    text = "Astley had a genius for trick riding. He saw that trick riders
received the most attention from the crowds in Islington. He had an idea for
opening a riding school in London in which he could also conduct shows of
acrobatic riding skill. In 1768, Astley performed in an open field in what is
now the Waterloo area of London, behind the present site of St John's Church.
Astley added a clown to his shows to amuse the spectators between equestrian
sequences, moving to fenced premises just south of Westminster Bridge, where
he opened his riding school from 1769 onwards and expanded the content of his
shows. He taught riding in the mornings and performed his feats of
horsemanship in the afternoons."
    zuid4 = storeText( jag, model, text, “wiki”   )

    text = "After the Amphitheatre was rebuilt again after the third fire, it
was said to be very grand.  The external walls were 148 feet long which was
larger than anything else at the time in London.  The interior of the
Amphitheatre was designed with a proscenium stage surrounded by boxes and
galleries for spectators. The general structure of the interior was
octagonal. The pit used for the entertainers and riders became a standardised
43 feet in diameter, with the circular enclosure surrounded by a painted four
foot barrier. Astley's original circus was 62 ft (~19 m) in diameter, and
later he settled it at 42 ft (~13 m), which has been an international
standard for circuses since."
    zuid5 = storeText( jag, model, text, “google”   )


    text = "According to the Big Bang theory, the energy and matter initially
present have become less dense as the universe expanded. Afte
r an initial accelerated expansion called the inflationary epoch at around
10−32 seconds, and the separation of the four known fundamental forces, the
universe gradually cooled and continued to expand, allowing the first
subatomic particles and simple atoms to form. Dark matter gradually gathered,
forming a foam-like structure of filaments and voids under the influence of
gravity. Giant clouds of hydrogen and helium were gradually drawn to the
places where dark matter was most dense, forming the first galaxies, stars,
and everything else seen today."
    zuid6 = storeText( jag, model, text, “wiki”   )

    text = "By comparison, general relativity did not appear to be as useful,
beyond making minor corrections to predictions of Newtonian gravitation
theory. It seemed to offer little potential for experimental test, as most of
its assertions were on an astronomical scale. Its mathematics seemed
difficult and fully understandable only by a small number of people. Around
1960, general relativity became central to physics and astronomy. New
mathematical techniques to apply to general relativity streamlined
calculations and made its concepts more easily visualized. As astronomical
phenomena were discovered, such as quasars (1963), the 3-kelvin microwave
background radiation (1965), pulsars (1967), and the first black hole
candidates (1981), the theory explained their attributes, and measurement of
them further confirmed the theory."
    zuid7 = storeText( jag, model, text, “imf”   )

    text = "In astronomy, the magnitude of a gravitational redshift is often
expressed as the velocity that would create an equivalent shift through the
relativistic Doppler effect. In such units, the 2 ppm sunlight redshift
corresponds to a 633 m/s receding velocity, roughly of the same magnitude as
convective motions in the sun, thus complicating the measurement. The GPS
```

```
satellite gravitational blueshift velocity equivalent is less than 0.2 m/s,
which is negligible compared to the actual Doppler shift resulting from its
orbital velocity."
    zuid8 = storeText( jag, model, text, “wiki”   )

    text = "Turn on the sprinkler system. In order to locate the break or
leak in the sprinkler system, you need to run water through it. Turn on the
sprinkler system to activate the flow of water. Allow the water to run for
about 2 minutes before you check the lines. Do this in the daytime, when
you'll have an easier time spotting the leak. If your sprinkler system is
separated into zones, activate the zones one at a time so you can identify
the break or leak more easily."
    zuid9 = storeText( jag, model, text, “wiki”   )

    text = "Check for water bubbling up from the soil. If you see a pool of
water or water coming from the soil, then there’s a leak in the sprinkler
line buried underneath. Mark the general location of the leak or break so you
can identify it when the water is turned off. Place an item like a shovel or
a rock on the ground near the leak. Turn off the sprinkler system after
you’ve found the leak. If you’ve found the signs of a leak and located the
region where the line is leaking or broken, turn off the water so you can
repair the line. Use the shut-off valve in the control box to stop the flow
of water through the system."
    zuid10 = storeText( jag, model, text, “wiki”   )

    text = "In fact, Antarctica is such a good spot for meteorite hunters
that crews of scientists visit every year, searching for these otherworldly
rocks, driving around the surface until they spot a lone dark rock on an
otherwise unbroken expanse of white. However, you don’t always have to travel
to the other side of the world to find a meteorite. Sometimes meteorites will
come to you. Keep an eye open for local reports of brilliant fireballs
lighting your region’s sky. Debris from such displays scatters across the
ground and sometimes hits structures or vehicles. Watch for information about
fireballs in your area on the websites of the American Meteor Society or the
International Meteor Organization."
    zuid11 = storeText( jag, model, text, “wiki”   )

    text = "Most tornadoes are found in the Great Plains of the central
United States – an ideal environment for the formation of severe
thunderstorms. In this area, known as Tornado Alley, storms are caused when
dry cold air moving south from Canada meets warm moist air traveling north
from the Gulf of Mexico. Tornadoes can form at any time of year, but most
occur in the spring and summer months along with thunderstorms.  May and June
are usually the peak months for tornadoes. The Great Plains are conducive to
the type of thunderstorms (supercells) that spawn tornadoes. It is in this
region that cool, dry air in the upper levels of the atmosphere caps warm,
humid surface air. This situation leads to a very unstable atmosphere and the
development of severe thunderstorms."
   zuid12 = storeText( jag, model, text, “google”   )
```

The function storeTex is implemented with the following program:

```
def storeText(jag, model, text, src):
    sentences = [ text ]
    embeddings = model.encode(sentences, normalize_embeddings=False)
    comma_str = ",".join( [str(x) for x in embeddings[0] ])

    istr = "insert into textvec values ('" + comma_str + "', '" + text +
"',’” + src + “’)"
    jag.execute( istr )
    return jag.getLastUuid()
```

Now we have a query and get similar texts from database:

```
  queryText = "More recently, that focus has shifted eastward by 400 to 500
miles. In the past decade or so tornadoes have become prevalent in eastern
Missouri and Arkansas, western Tennessee and Kentucky, and northern
Mississippi and Alabama—a new region of concentrated storms. Tornado activity
in early 2023 epitomized the trend."

    K = 3;

    retrieveTopK( jag, model, queryText, K )
```

Then we can have another query and get similar texts from database:

```
    queryText = "Think of designing a landscape for the bare lot surrounding
your new home as an adventure in creativity. Perhaps your property needs only
a few small, easily doable projects to make it more attractive. Either way,
it's important to consider how each change will relate to the big picture.
Stand back from time to time to see the entire landscape and how each part
fits into it."

    K = 3;

   retrieveTopK( jag, model, queryText, K )
```

The full listing of Python3 programs is shown below.

```
def searchSimilarTexts(jag, model, queryText, K):

    sentences = [ queryText ]

    embeddings = model.encode(sentences, normalize_embeddings=False)

    comma_separated_str = ",".join( [str(x) for x in embeddings[0] ])
    qstr = "select similarity(v, '" + comma_separated_str
    qstr += "', 'topk=" + str(K) + ",type=cosine_fraction_short')"
    qstr += " from  textvec"

    jag.query( qstr )

    jsonstr = ''
    while jag.reply():
        jsonstr = jag.jsonString()

    return jsonstr



def getTextByVID(jag, vid):
    qstr =" select zid from test.textvec.textvec_idx where v='" + vid + "'"
    zid = ''
    jag.query( qstr )
    while jag.reply():
        zid = jag.getValue("zid")

    qstr = "select text from textvec where zid='" + zid + "'"
    jag.query( qstr )
    txt = ''
    while jag.reply():
        txt = jag.getValue("text")

    return txt

def retrieveTopK( jag, model, queryText, K ):
    print("Query: " + queryText )
    json_str = searchSimilarTexts( jag, model, queryText, K )
    json_obj = json.loads(json_str)
    i = 0;
    print("\n")
    print("Retrieved similar texts: ")
    for rec in json_obj:
        dat = rec[str(i)]
        print("\n")
        print("Rank: " + str(i+1))
        vid = dat["id"]
        print("Vector ID: " + vid )
        print("Distance: " + dat["distance"] )
        txt = getTextByVID( jag, vid )
        print("Text: " + txt )
        i += 1

    print("\n\n")
```

Furthermore, extending beyond text embeddings, the capability exists to generate image and video embeddings. These embeddings serve as efficient tools for rapid image and video searches using vector-based techniques. This advancement empowers users to swiftly locate relevant images and videos by exploiting the inherent characteristics captured within the embedding vectors. As a result, the need for intricate keyword-based searches or complex metadata is significantly reduced, enhancing the speed and accuracy of the search process.

Integrating Vector Search and Exact Search

In various application scenarios, there arises a need for users to perform targeted queries on a dataset, ensuring that the retrieved data records not only adhere to certain criteria but also exhibit a certain level of similarity to a provided data sample. This intricate task demands the identification of vectors that are both closely related and satisfy specific prerequisites. With the innovative capabilities of JaguarDB, this complex process can be streamlined into a single step. Through the integration of similarity search alongside selective criteria, JaguarDB facilitates the discovery of nearest neighbors that fulfill predefined qualifications. This advanced functionality empowers users to seamlessly locate a subset of data records and subsequently assess their likeness to a reference vector, resulting in the assignment of similarity rankings. By encompassing both the aspects of similarity and tailored selection, this approach significantly mitigates the potential for inaccuracies, making it particularly well-suited for environments characterized by stringent business requirements.

JaguarDB's unique amalgamation of similarity-based search and tailored qualification selection brings unprecedented efficiency to the intricate task of querying and comparison. Once a cohort of relevant data records is extracted, their alignment with a given vector is precisely evaluated, generating a hierarchy of similarity rankings. This integrated approach is instrumental in refining the matching process, ensuring that data records not only exhibit the desired attributes but also possess a designated degree of resemblance to a reference sample. This holistic functionality carries substantial benefits, especially in high-stakes scenarios where precision is paramount. By converging the twin challenges of similarity and criterion-based filtering, JaguarDB effectively minimizes the potential for inaccuracies, offering a robust solution for industries demanding precise data retrieval and analysis. Through this innovative approach, JaguarDB empowers users to navigate the complexities of data exploration with enhanced accuracy and confidence, establishing itself as a pivotal tool in the pursuit of data-driven excellence.

The following similarity search statement is extended with the “where clause” to filter the nearest neighbors of the input query vector:

```
select
similarity(v, 'QUERY_VECTOR',
'topk=K,type=DISTANCE_INPUT_QUANTIZATION')
from STORE
where attribute1 = … and attribute2 = …;
```

For example:

```
        select similarity(v, '0.1, 0.2, 0.3, 0.4, 0.5, 0.3, 0.1',
        'topk=100,type=manhatten_fraction_float')
        from vectab
        where customer_region='NE' and marriage_status='single';
```

In this illustrative scenario, the foremost consideration involves the establishment of a topK records subset, containing a specified count of 100 records, which are inspected to see if they match the criteria given by the where predicates. The intersection of the two sets of records is returned to the user.

JaguarDB Programming API

JaguarDB offers a comprehensive set of application programming interfaces (APIs) tailored to various development needs. These APIs can be seamlessly employed within the jql.bin client terminal or seamlessly integrated into programming languages such as Java, Python, Go, and Node.js. This flexibility empowers developers to interact with JaguarDB using their preferred environment, ensuring a smooth and versatile development experience.

Creating a Store for Vectors

```
create store STORE (
    key: …KEY…,
    value: VECCOL vector(dimension,'DISTANCE_INPUT_QUANTIZATION'),
    …other_fields…
)
```

The symbol "VECCOL" designates the name of the vector column, while "dimension" denotes the count of components within a vector. Standard dimensions often include values like 768, 1024, 1536, etc. The string "DISTANCE_INPUT_QUANTIZATION" is a vector definition that serves to specify the nature of the distance, input data type, and level of quantization employed in the vector storage and search of similarity between vectors. This comprehensive approach accommodates various distance types, which encompass:

Euclidean Distance

The Euclidean distance, also known as the L2 distance or the Euclidean norm, is a measure of the straight-line distance between two points in a multi-dimensional space. It's commonly used to quantify the similarity between vectors.

$$\mathrm{dist} = \sqrt{\sum_{i=1}^{n} (A_i - B_i)^2}$$

Cosine Distance

Cosine distance is a measure used to quantify the dissimilarity between two vectors in a multi-dimensional space. Unlike the Euclidean distance that measures the direct geometric distance between vectors, the cosine distance focuses on the angle between the vectors.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2} \sqrt{\sum\limits_{i=1}^{n} B_i^2}}$$

InnerProduct

Inner product similarity is useful for similarity search in scenarios where the magnitudes of vectors are important in addition to their directions.

$$\vec{A} \cdot \vec{B} = \sum_{i=0}^{n} A_i B_i$$

Manhatten Distance

Manhattan distance is a distance metric between two points in a multi-dimensional vector space. It is the sum of absolute difference between the measures in all dimensions of two points.

$$\mathrm{dist} = \sum_{\mathrm{i}=1}^{n} |\mathrm{A_i} - \mathrm{B_i}|$$

Chebyshev Distance

Chebyshev distance is a metric defined on a vector space where the distance between two vectors is the greatest of their differences along any coordinate dimension.

$$\mathrm{dist} = \mathrm{max_i}(|\mathrm{A_i} - \mathrm{B_i}|)$$

Hamming Distance

The Hamming distance between two vectors is the number of positions at which the corresponding components are different.

$$\mathrm{dist} = \sum_{i=1}^{n} \Delta(\mathrm{A_i},\ \mathrm{B_i})$$

Jeccard Distance

The Jeccard distance between two vectors is computed by taking the ratio of Intersection over Union of the two vectors.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Minkowski Half

In general, the Minkowski distance of order p is given by:

$$dist = \left(\sum_{i-1}^{n} |A_i - B_i|^p \right)^{1/p}$$

In JaguarDB, Minkowski Half distance refers to the Minkowski distance where p = 0.5.

The input type in JaguarDB refers to the expected data format in the input vectors. There are two input types: fraction and whole. JaguarDB excels not only in managing vector embeddings but also in handling a diverse range of feature vectors. These vectors can include various types and forms, whether they are normalized or unnormalized, presented in fractional or full original formats. This versatility underscores JaguarDB's capability to accommodate a wide array of data formats.

Fraction Input Format

Each component of a vector is in the range of [-1.0, +1.0], inclusive. An example of a such a vector would be: "0.1, 0.02, -0.04, -0.5, 0.12, 0.53".

Whole Input Format

Components of a vector are not limited to the range of [-1.0, +1.0]. They can be in any range. However, they could be trimmed and converted to the range that is required by the quantization level as described below.

Quantization Level

There are three quantization levels in JaguarDB: byte, short, and float. The process of quantizing input vectors yields efficient memory utilization within the system. While storing a float number demands 4 bytes, employing fewer bytes for storing vector components can yield substantial memory savings. When components are stored as signed integers, memory savings can reach 50%, while utilizing only a single byte for vector components can result in an impressive 75% reduction in memory usage. This approach is termed "short quantization level" for the utilization of signed integers and "byte quantization level" for the use of a single byte. The quantization of input vectors aligns with the level specified by the user during vector creation, optimizing memory consumption while maintaining data integrity.

With byte (8-bit) quantization level, the number of quantized hyper cubes in a 1024-dimensional hyperspace is 256^{1024} which is already a large number and vector distribution would be sparse. With a short (16-bit) quantization level, the number of available hypercubes is even larger. In rare application scenarios, the vectors could be densely populated around clusters. A 16-bit quantization may provide higher resolution of differentiating vectors than an 8-bit quantization. It is a trade-off between storage size and accuracy in searching nearest neighbors.

Multiple Search Types

During the creation of a vector store, the second argument within the "vector()" field description, or key definition, offers the flexibility to incorporate multiple instances of "DISTANCE_INPUT_QUANTIZATION". For instance, it can appear as a series of "cosine_fraction_byte, hamming_whole_short". This allows users to specify multiple distance types and quantization levels, albeit limited to a single input type for the same distance and quantization level. Notably, distinct vector data stores are managed for each unique combination of the three types, ensuring the effective organization of data based on these parameters.

List of Key Definitions

Key Definition	Distance	Input (component x)	Quantization
euclidean_fraction_short	Euclidean	-1.0 <= x <= +1.0	16-bit integer
euclidean_fraction_byte	Euclidean	-1.0 <= x <= +1.0	8-bit integer
euclidean_whole_short	Euclidean	-32767 <= x <= 32767	16-bit integer

euclidean_whole_byte	Euclidean	-127 <= x <= 127	8-bit integer
cosine_fraction_short	Cosine	-1.0 <= x <= +1.0	16-bit integer
cosine_fraction_byte	Cosine	-1.0 <= x <= +1.0	8-bit integer
cosine_whole_short	Cosine	-32767 <= x <= 32767	16-bit integer
cosine_whole_byte	Cosine	-127 <= x <= 127	8-bit integer
innerproduct_fraction_short	Inner Product	-1.0 <= x <= +1.0	16-bit integer
innerproduct_fraction_byte	Inner Product	-1.0 <= x <= +1.0	8-bit integer
innerproduct_whole_short	Inner Product	-32767 <= x <= 32767	16-bit integer
innerproduct_whole_byte	Inner Product	-127 <= x <= 127	8-bit integer
manhatten_fraction_short	Manhatten	-1.0 <= x <= +1.0	16-bit integer
manhatten_fraction_byte	Manhatten	-1.0 <= x <= +1.0	8-bit integer
manhatten_whole_short	Manhatten	-32767 <= x <= 32767	16-bit integer
manhatten_whole_byte	Manhatten	-127 <= x <= 127	8-bit integer
hamming_fraction_short	Hamming	-1.0 <= x <= +1.0	16-bit integer
hamming_fraction_byte	Hamming	-1.0 <= x <= +1.0	8-bit integer
hamming_whole_short	Hamming	-32767 <= x <= 32767	16-bit integer
hamming_whole_byte	Hamming	-127 <= x <= 127	8-bit integer
chebyshev_fraction_short	Chebyshev	-1.0 <= x <= +1.0	16-bit integer
chebyshev_fraction_byte	Chebyshev	-1.0 <= x <= +1.0	8-bit integer
chebyshev_whole_short	Chebyshev	-32767 <= x <= 32767	16-bit integer
chebyshev_whole_byte	Chebyshev	-127 <= x <= 127	8-bit integer
minkowskihalf_fraction_short	MinkowskiHalf	-1.0 <= x <= +1.0	16-bit integer
minkowskihalf_fraction_byte	MinkowskiHalf	-1.0 <= x <= +1.0	8-bit integer
minkowskihalf_whole_short	MinkowskiHalf	-32767 <= x <= 32767	16-bit integer
minkowskihalf_whole_byte	MinkowskiHalf	-127 <= x <= 127	8-bit integer
jeccard_fraction_short	Jeccard	-1.0 <= x <= +1.0	16-bit integer
jeccard_fraction_byte	Jeccard	-1.0 <= x <= +1.0	8-bit integer
jeccard_whole_short	Jeccard	-32767 <= x <= 32767	16-bit integer
jeccard_whole_byte	Jeccard	-127 <= x <= 127	8-bit integer
euclidean_fraction_float	Euclidean	float	32-bit float
euclidean_whole_float	Euclidean	float	32-bit float
cosine_fraction_float	Cosine	float	32-bit float
cosine_whole_float	Cosine	float	32-bit float
innerproduct_fraction_float	InnerProduct	float	32-bit float

innerproduct_whole_float	InnerProduct	float	32-bit float
manhatten_fraction_float	Manhatten	float	32-bit float
manhatten_whole_float	Manhatten	float	32-bit float
hamming_fraction_float	Hamming	float	32-bit float
hamming_whole_float	Hamming	float	32-bit float
chebyshev_fraction_float	Chebyshev	float	32-bit float
chebyshev_whole_float	Chebyshev	float	32-bit float
minkowskihalf_fraction_float	MinkowskiHalf	float	32-bit float
minkowskihalf_whole_float	MinkowskiHalf	float	32-bit float
jeccard_fraction_float	Jeccard	float	32-bit float
jeccard_whole_float	Jeccard	float	32-bit float

Adding Vectors

JaguarDB can integrate all application and vector data, facilitating streamlined data management for real-world scenarios. It enables the incorporation of vector data alongside other pertinent information related to business objects, allowing for comprehensive and cohesive data representation.

```
insert into STORE ( …, VECCOL, …) values (…, 'VECTOR_STRING', … )

insert into STORE values (…, 'VECTOR_STRING', … )
```

Where VECTOR_STRING is a list of comma-separated components of the vector. In the second statement, the values must be provided according to the correct order of the columns in the store. Once the vector is added, the value of the field for VECCOL will be replaced with an integer as the unique identifier for the vector. With a vector ID, the components of the vector can be retrieved from the vector database.

Similarity Search

Similarity search using JaguarDB vectors involves the process of finding vectors within the database that are most similar to a given query vector. This search is conducted based on predefined similarity metrics, such as cosine similarity or Euclidean distance similarity, which quantify the resemblance between vectors. The API for similarity search is as follows:

```
select
similarity(v, 'QUERY_VECTOR',
'topk=K,type=DISANCE_INPUT_QUANTIZATION')
from STORE;
```

where QUERY_VECTOR is a list of comma-separated component values of the vector. The number “K” specifies the number of most similar vectors to be found and returned for the query vector. The returned result is in the JSON format and the developer can call the jsonString() function to parse the JSON format and retrieve the ID and distance values.

As an example, the following statement returns the top 5 most similar vectors to the query vector:

```
select similarity(v, '0.1, 0.2, 0.3, 0.4, 0.5, 0.3, 0.1',
'topk=5,type=manhatten_fraction_byte') from vec1;
```

Combining Vector Search and Exact Search

JaguarDB empowers users with a unique synergy of similarity search and exact predicate search. In the context of this integration, consider the following Python illustration: it finds textual instances similar to a given input text while concurrently sifting through records that adhere to specific criteria. The outcome of this combined endeavor is the assignment of similarity values to the retrieved records, a direct consequence of the similarity search's operation. It is noted that the governing criterion, in this case, relates to the source of the text. However, in practical implementation, a number of predicates can be applied.

```
select

similarity(v, 'QUERY_VECTOR',

'topk=K,type=DISTANCE_INPUT_QUANTIZATION')

from STORE;
```

An example of integrating both similarity search and predicate based search is shown below:

```
def retrieveTopKWithCriteria( jag, model, queryText, src, K ):
    print("Query: " + queryText )

    sentences = [ queryText ]
    embeddings = model.encode(sentences, normalize_embeddings=False)
    comma_str = ",".join( [str(x) for x in embeddings[0] ])

    qstr = "select similarity(v, '" + comma_str
    qstr += "', 'topk=" + str(K) + ",type=cosine_fraction_short')"
    qstr += " from textvec"
    qstr += " where source='" + src + "'"

    jag.query( qstr )

    print("\n")
    print("Result: ")
    while jag.reply():
        print('zid={}'.format(jag.getValue("zid")) )
        print('v={}'.format(jag.getValue("v")) )
        print('vectorid={}'.format(jag.getValue("vectorid")) )
        print('rank={}'.format(jag.getValue("rank")) )
        print('distance={}'.format(jag.getValue("distance")) )
        print('source={}'.format(jag.getValue("source")) )
        print('text={}'.format(jag.getValue("text")) )
        print("\n")
```

Anomaly Detection

Jaguar vector database is revolutionizing the way businesses approach anomaly detection. It provides a structured and efficient means of storing and querying data, enabling organizations to analyze patterns and deviations with remarkable precision. This innovative technique not only streamlines the process of anomaly detection but also enhances the accuracy of identifying potential threats. As the business landscape continues to evolve in an increasingly digital world, leveraging vector databases for anomaly detection has become a strategic imperative for enterprises seeking to safeguard their operations and data from malicious activities.

The API for detecting anomaly is shown below:

```
select
anomalous(VECCOL,
          'type=DISTANCE_INPUT_QUANTIZATION,activation=[sigma:perc]')
from STORE
```

where the type specifies the distance type and quantization levels of vectors; the optional parameter sigma is the number of standard deviations, perc is percentage of vector components pass the sigma value.

```
select
anomalous(vc,
          'type=euclidean_whole_float')
from myvector;



select
anomalous(vc,
          'type=euclidean_whole_float, activation=[0.3:40;1.5:30]')
from myvector;
```

Result:

```
json  {"anomalous":"YES","prate":"0.388671875"}
```

Retrieving Vectors

In cases where users need to retrieve the component values of a vector, the following API can be used:

```
select
vector(VECCOL, 'type=DISTANCE_INPUT_QUANTIZATION')
from STORE
where KEY=…
```

For example,

```
select vector(v, 'type=manhatten_fraction_short')
from vec1
where fid=’ANjf848223@01’
```

The utilized KEY in the query must uniquely identify a record housing the vector, typically involving the exclusive use of the ZeroMove unique ID.

Updating Vectors

The vector components can be updated with two approaches:

```
update STORE
set VECCOL:vector='VECTOR_STRING'
where KEY=…

update STORE
set VECCOL:vector='VECTOR_ID:VECTOR_STRING'
where 1
```

where VECTOR_ID is the integer value of the vector ID, and VECTOR_STRING is a list of comma-separated component values.

Deleting Vectors

The vector components cannot be deleted separately without deleting the record containing the vector. A store record can be deleted with the following command:

```
Delete from STORE

where KEY=…
```

The KEY in the above statement must uniquely identify a record housing the vector, typically the ZeroMove unique ID. In addition, dropping or truncating a store will delete the associated vectors as well.

Conclusion

JaguarDB technology provides a powerful and eco-friendly solution for efficient and scalable data management for artificial intelligence. Leveraging ZeroMove hashing technology, its focus on performance, advanced features, and sustainability makes it a promising choice for organizations seeking reliable and environmentally conscious solutions for artificial intelligence.