Apache Cassandra : Vector Search

Overview of Vector Search

Overview of Vector Search

Vector Search is a new feature added to **Cassandra 5.0**. It is a powerful technique for finding relevant content within large document collections and is particularly useful for AI applications.

Vector Search concepts

Use Vector Search to create powerful relevant content search for any CQL table.

Vector Search quickstart

Follow the steps to get started quickly with Vector Search.

Working with Vector Search

Create, check, alter, drop, and query Vector Search.

Data modeling with Vector Search

The ins and outs of data modeling with Vector Search.

Reference: CREATE TABLE, CREATE INDEX, CREATE CUSTOM INDEX

Vector search concepts

Vector search concepts

Vector Search is a new feature added to **Cassandra 5.0**. It is a powerful technique for finding relevant content within large datasets and is particularly useful for AI applications. Vector Search also makes use of **Storage-Attached Indexes(SAI)**, leveraging the new modularity of the latter feature. Vector Search is the first instance of validating the extensibility of SAI.

Data stored in a database is useful, but the context of that data is critical to applications. Machine learning in applications allows users to get product recommendations, match similar images, and a host of other capabilities. A machine learning model is a program that can find patterns or make decisions from a previously unseen dataset. To power a machine learning model in an application, Vector Search does similarity comparison of stored database data to discover connections in data that may not be explicitly defined.

One key to doing similarity comparisons in a machine learning model is the ability to store embeddings vectors, arrays of floating-point numbers that represent the similarity of specific objects or entities. Vector Search brings that functionality to the high availability **Apache Cassandra** database.

Want to get started quickly? Here's how!

Vector Search Quickstart

The foundation of Vector Search lies within the embeddings, compact representations of text or images as high-dimensional vectors of floating-point numbers. For text processing, embeddings are generated by feeding the text to a machine learning model. These models generally use a neural network to transform the input into a fixed-length vector. When words are represented as high-dimensional vectors, the aim is to arrange the vectors so that similar words end up closer together in the vector space and dissimilar word end up further apart. Creating the vectors in this manner is referred to as preserving semantic or structural similarity. Embeddings capture the semantic meaning of the text, which in turn, allow queries to rely on a more nuanced understanding of the text as opposed to traditional term-based approaches.

Large Language Models (LLMs) generate contextual embeddings for the data, and optimize embeddings for queries. Trained embeddings like those produced by LLMs can be used in Natural Language Processing (NLP) tasks such as text classification, sentiment analysis, and machine translation. You can embed almost any kind of data and retrieve good results with vector search. As models continue to improve, the quality of results will also continue to improve.

Storage Attached Indexing (SAI)

SAI is a required feature providing unparalleled I/O throughput for databases to use Vector Search as well as other search indexing. SAI is a highly-scalable and globally-distributed index that adds column-level indexes to any vector data type column.

SAI provides the most indexing functionality available - indexing both queries and content (large inputs include such items as documents, words, and images) to capture semantics.

For more about SAI, see the **Storage Attached Index** documentation.

NOTE	You cannot change index settings without dropping and rebuilding the index.

It is better to create the index and then load the data. This method avoids the concurrent building of the index as data loads.

New Vector CQL data type

A new **vector data type** is added to CQL to support Vector Search. It is designed to save and retrieve embeddings vectors.

Vector Search Quickstart

Vector Search Quickstart

To enable your machine learning model, Vector Search uses data to be compared by similarity within a database, even if it is not explicitly defined by a connection. A vector is an array of floating point type that represents a specific object or entity.

The foundation of Vector Search lies within the embeddings, which are compact representations of text as vectors of floating-point numbers. These embeddings are generated by feeding the text through an API, which uses a neural network to transform the input into a fixed-length vector. Embeddings capture the semantic meaning of the text, providing a more nuanced understanding than traditional term-based approaches. The vector representation allows for input that is substantially similar to produce output vectors that are geometrically close; inputs that are not similar are geometrically further apart.

To enable Vector Search, a new `vector` data type is available in your **Cassandra** database with Vector Search.

Prerequisites

There are no prerequisite tasks.

In general, to use Vector Search with Apache Cassandra, you'll follow these instructions:

The embeddings were randomly generated in this quickstart. Generally, you would run both your source documents/contents through an embeddings generator, as well as the query you were asking to match. This example is simply to show the mechanics of how to use CQL to create vector search data objects.

Create vector keyspace

Create the keyspace you want to use for your Vector Search table. This example uses `cycling` as the `keyspace name`:

```
CREATE KEYSPACE IF NOT EXISTS cycling
   WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : '1' };
```

Use vector keyspace

Select the keyspace you want to use for your Vector Search table. This example uses `cycling` as the `keyspace name`:

```
USE cycling;
```

Create vector table

Create a new table in your keyspace, including the `comments_vector` column for vector. The code below creates a vector with five values:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
```

Optionally, you can alter an existing table to add a vector column:

```
ALTER TABLE cycling.comments_vs
   ADD comment_vector VECTOR <FLOAT, 5> ;
```

Create vector index

Create the custom index with Storage Attached Indexing (SAI):

```
CREATE INDEX IF NOT EXISTS ann_index
  ON cycling.comments_vs(comment_vector) USING 'sai';
```

For more about SAI, see the **Storage Attached Indexing** documentation.

IMPORTANT

The index can be created with options that define the similarity function:

```
CREATE INDEX IF NOT EXISTS ann_index
    ON vsearch.com(item_vector) USING 'sai'
WITH OPTIONS = { 'similarity_function': 'DOT_PRODUCT' };
```

> Valid values for the `similarity_function` are `DOT_PRODUCT`, `COSINE`, or `EUCLIDEAN`.

Load vector data into your database

Insert data into the table using the new type:

```
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-02-14 12:43:20-0800',
      'Raining too hard should have postponed',
      'Alex',
      [0.45, 0.09, 0.01, 0.2, 0.11]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-03-21 13:11:09.999-0800',
      'Second rest stop was out of water',
      'Alex',
      [0.99, 0.5, 0.99, 0.1, 0.34]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-04-01 06:33:02.16-0800',
      'LATE RIDERS SHOULD NOT DELAY THE START',
      'Alex',
      [0.9, 0.54, 0.12, 0.1, 0.95]
);

INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      totimestamp(now()),
      'The gift certificate for winning was the best',
      'Amy',
```

```
      [0.13, 0.8, 0.35, 0.17, 0.03]
);

INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-02-17 12:43:20.234+0400',
      'Glad you ran the race in the rain',
      'Amy',
      [0.3, 0.34, 0.2, 0.78, 0.25]
);

INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-03-22 5:16:59.001+0400',
      'Great snacks at all reststops',
      'Amy',
      [0.1, 0.4, 0.1, 0.52, 0.09]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-04-01 17:43:08.030+0400',
      'Last climb was a killer',
      'Amy',
      [0.3, 0.75, 0.2, 0.2, 0.5]
);
```

Query vector data with CQL

To query data using Vector Search, use a `SELECT` query:

```
SELECT * FROM cycling.comments_vs
    ORDER BY comment_vector ANN OF [0.15, 0.1, 0.1, 0.35, 0.55]
    LIMIT 3;
```

To obtain the similarity calculation of the best scoring node closest to the query data as part of the

results, use a `SELECT` query:

```
SELECT comment, similarity_cosine(comment_vector, [0.2, 0.15, 0.3, 0.2, 0.05])
    FROM cycling.comments_vs
    ORDER BY comment_vector ANN OF [0.1, 0.15, 0.3, 0.12, 0.05]
    LIMIT 1;
```

The supported functions for this type of query are:

- similarity_dot_product
- similarity_cosine
- similarity_euclidean

with the parameters of (<vector_column>, <embedding_value>). Both parameters represent vectors.

NOTE

- The limit must be 1,000 or fewer.
- Vector Search utilizes Approximate Nearest Neighbor (ANN) that in most cases yields results almost as good as the exact match. The scaling is superior to Exact Nearest Neighbor (KNN).
- Least-similar searches are not supported.
- Vector Search works optimally on tables with no overwrites or deletions of the `item_vector` column. For an `item_vector` column with changes, expect slower search results.

With the code examples, you have a working example of our Vector Search. Load your own data and use the search function.

Vector CQL data type

Vector CQL data type

A new CQL data type, a `VECTOR` is required for Vector Search. https://issues.apache.org/jira/browse/CASSANDRA-18504 added this data type to **Cassandra 5.0**. The new type `VECTOR` has the following properties:

- fixed length array
- elements may not be null
- flatten array (aka multi-cell = false)

Although the data type specified for a vector used in vector search is a floating point number, the vector data type can be used to create a vector of any data type. For example, a vector of `int` or `bigint` can be created, but cannot be used with vector search.

NOTE	Vectors are limited to a maximum dimension of 8K (2^13) items.

Create a keyspace:

```
CREATE KEYSPACE IF NOT EXISTS cycling
   WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : '1' };
```

Use the keyspace:

```
USE cycling;
```

This data type can be used in a `CREATE TABLE` statement:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
```

① Create a 5-dimensional embedding

Index the vector column:

```
CREATE INDEX IF NOT EXISTS ann_index
  ON cycling.comments_vs(comment_vector) USING 'sai';
```

Insert data:

```
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-02-14 12:43:20-0800',
      'Raining too hard should have postponed',
      'Alex',
      [0.45, 0.09, 0.01, 0.2, 0.11]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-03-21 13:11:09.999-0800',
      'Second rest stop was out of water',
      'Alex',
      [0.99, 0.5, 0.99, 0.1, 0.34]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-04-01 06:33:02.16-0800',
      'LATE RIDERS SHOULD NOT DELAY THE START',
      'Alex',
      [0.9, 0.54, 0.12, 0.1, 0.95]
);

INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      totimestamp(now()),
      'The gift certificate for winning was the best',
      'Amy',
      [0.13, 0.8, 0.35, 0.17, 0.03]
);
```

```
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-02-17 12:43:20.234+0400',
      'Glad you ran the race in the rain',
      'Amy',
      [0.3, 0.34, 0.2, 0.78, 0.25]
);

INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-03-22 5:16:59.001+0400',
      'Great snacks at all reststops',
      'Amy',
      [0.1, 0.4, 0.1, 0.52, 0.09]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-04-01 17:43:08.030+0400',
      'Last climb was a killer',
      'Amy',
      [0.3, 0.75, 0.2, 0.2, 0.5]
);
```

Select data:

```
SELECT * FROM cycling.comments_vs
    ORDER BY comment_vector ANN OF [0.15, 0.1, 0.1, 0.35, 0.55]
    LIMIT 3;
```

Data Modeling

Data Modeling

As you develop AI and Machine Learning (ML) applications using Vector Search, here are some data modeling considerations. These factors help effectively leverage vector search to produce accurate and efficient search responses within your application.

Data representation

Vector search relies on representing data points as high-dimensional vectors. The choice of vector representation depends on the nature of the data.

For data that consists of text documents, techniques like word embeddings (e.g., Word2Vec) or document embeddings (e.g., Doc2Vec) can be used to convert text into vectors. More complex models can also be used to generate embeddings using Large Language Models (LLMs) like OpenAI GPT-4 or Meta LLaMA 2. Word2Vec is a relatively simple model that uses a shallow neural network to learn embeddings for words based on their context. The key concept is that Word2Vec generates a single fixed vector for each word, regardless of the context in which the word is used. LLMs are much more complex models that use deep neural networks, specifically transformer architectures, to learn embeddings for words based on their context. Unlike Word2Vec, these models generate contextual embeddings, meaning the same word can have different embeddings depending on the context in which it is used.

Images can be represented using deep learning techniques like convolutional neural networks (CNNs) or pre-trained models such as Contrastive Language Image Pre-training (CLIP). Select a vector representation that captures the essential features of the data.

Dataset dimensions

A vector search only works when the vectors have the same dimensions, because vector operations like dot product and cosine similarity require vectors to have the same number of dimensions.

For vector search, it is crucial that all embeddings are created in the same vector space. This means that the embeddings should follow the same principles and rules to enable proper comparison and analysis. Using the same embedding library guarantees this compatibility because the library consistently transforms data into vectors in a specific, defined way. For example, comparing Word2Vec embeddings with BERT (an LLM) embeddings could be problematic because these models have different architectures and create embeddings in fundamentally different ways.

Thus, the vector data type is a fixed-length vector of floating-point numbers. The dimension value is defined by the embedding model you use. Some machine learning libraries will tell you the dimension value, but you must define it with the embedding model. Selecting an embedding model for your

dataset that creates good structure by ensuring related objects are nearby each other in the embedding space is important. You may need to test out different embedding models to determine which one works best for your dataset.

Preprocessing embeddings vectors

Normalizing is about scaling the data so it has a length of one. This is typically done by dividing each element in a vector by the vector's length.

Standardizing is about shifting (subtracting the mean) and scaling (dividing by the standard deviation) the data so it has a mean of zero and a standard deviation of one.

It is important to note that standardizing and normalizing in the context of embedding vectors are not the same. The correct preprocessing method (standardizing, normalizing, or even something else) depends on the specific characteristics of your data and what you are trying to achieve with your machine learning model. Preprocessing steps may involve cleaning and tokenizing text, resizing and normalizing images, or handling missing values.

Normalizing embedding vectors

Normalizing embedding vectors is a process that ensures every embedding vector in your vector space has a length (or norm) of one. This is done by dividing each element of the vector by the vector's length (also known as its `Euclidean` norm or `L2` norm).

For example, look at the embedding vectors from the **Vector Search Quickstart** and their normalized counterparts, where a consistent length has been used for all the vectors:

Original

```
[0.1, 0.15, 0.3, 0.12, 0.05]
[0.45, 0.09, 0.01, 0.2, 0.11]
[0.1, 0.05, 0.08, 0.3, 0.6]
```

Normalized

```
[0.27, 0.40, 0.80, 0.32, 0.13]
[0.88, 0.18, 0.02, 0.39, 0.21]
[0.15, 0.07, 0.12, 0.44, 0.88]
```

The primary reason you would normalize vectors when working with embeddings is that it makes comparisons between vectors more meaningful. By normalizing, you ensure that comparisons are not affected by the scale of the vectors, and are solely based on their direction. This is particularly useful to calculate the cosine similarity between vectors, where the focus is on the angle between vectors

(directional relationship), not their magnitude.

Normalizing embedding vectors is a way of standardizing your high-dimensional data so that comparisons between different vectors are more meaningful and less affected by the scale of the original vectors. Since `dot product` and cosine are equivalent for normalized vectors, but the `dot product` algorithm is 50% faster, it is recommended that developers use `dot product` for the similarity function.

However, if embeddings are NOT normalized, then `dot product` silently returns meaningless query results. Therefore, `dot product` is not set as the default similarity function in Vector Search.

When you use OpenAI, PaLM, or Simsce to generate your embeddings, they are normalized by default. If you use a different library, you want to normalize your vectors and set the similarity function to `dot product`. See how to set the similarity function in the **Vector Search quickstart**.

Normalization is not required for all vector search examples.

Standardizing embedding vectors

Standardizing embedding vectors typically refers to a process similar to that used in statistics where data is standardized to have a mean of zero and a standard deviation of one. The goal of standardizing is to transform the embedding vectors so they have properties of a standard normal Gaussian distribution.

If you are using a machine learning model that uses distances between points (like nearest neighbors or any model that uses Euclidean distance or cosine similarity), standardizing can ensure that all features contribute equally to the distance calculations. Without standardization, features on larger scales can dominate the distance calculations.

In the context of neural networks, for example, having input values that are on a similar scale can help the network learn more effectively, because it ensures that no particular feature dominates the learning process simply because of its scale.

Indexing and Storage

SAI indexing and storage mechanisms are tailored for large datasets like vector search. Currently, SAI uses JVector, an algorithm for Approximate Nearest Neighbor (ANN) search and close cousin to Hierarchical Navigable Small World (HNSW).

The goal of ANN search algorithms like JVector is to find the data points in a dataset that are closest (or most similar) to a given query point. However, finding the exact nearest neighbors can be computationally expensive, particularly when dealing with high-dimensional data. Therefore, ANN algorithms aim to find the nearest neighbors approximately, prioritizing speed and efficiency over exact accuracy.

JVector achieves this goal by creating a hierarchy of graphs, where each level of the hierarchy corresponds to a `small world` graph that is navigable. It is inspired by DiskANN, a disk-backed ANN library, to store the graphs on disk. For any given node (data point) in the graph, it is easy to find a path to any other node. The higher levels of the hierarchy have fewer nodes and are used for coarse navigation, while the lower levels have more nodes and are used for fine navigation. Such indexing structures enable fast retrieval by narrowing down the search space to potential matches.

JVector also uses the Panama SIMD API to accelerate index build and queries.

Similarity Metric

Vector search relies on computing the similarity or distance between vectors to identify relevant matches. Choosing an appropriate similarity metric is crucial, as different metrics may be more suitable for specific types of data. Common similarity metrics include cosine similarity, Euclidean distance, or Jaccard similarity. The choice of metric should align with the characteristics of the data and the desired search behavior.

Vector Search supports three similarity metrics: `cosine similarity`, `dot product`, and `Euclidean distance`. The default similarity algorithm for the Vector Search indexes is `cosine similarity`. The recommendation is to use the `dot product` on normalized embeddings for most applications, because `dot product` is 50% faster than `cosine similarity`.

Scalability and Performance

Scalability is a critical consideration as your dataset expands. Vector search algorithms should be designed to handle large-scale datasets efficiently. Your **Cassandra** database using Vector Search efficiently distributes data and accesses that data with parallel processing to enhance performance.

Evaluation and iteration

Continuously evaluate and iterate the data to refine search results against known truth and user feedback. This also helps identify areas for improvement. Iteratively refining the vector representations, similarity metrics, indexing techniques, or preprocessing steps can lead to better search performance and user satisfaction.

Use cases

Vector databases with well-optimized embeddings allow for new ways to search and associate data, generating results which previously would not have been possible with traditional databases.

Examples:

- Search for items that are similar to a given item, without needing to know the exact item name or

IDs

- Retrieve documents based on similarity of context and content rather than exact string or keyword matches
- Expand search results across dissimilar items, such as searching for a product and retrieving contextually similar products from a different category
- Execute word similarity searches, and suggest to users ways to rephrase queries or passages
- Encode text, images, audio, or video as queries and retrieve media that are conceptually, visually, audibly, or contextually similar to the input
- Reduce time spent on metadata and curation by automatically generating associations for data
- Improve data quality by automatically identifying and removing duplicates

Best practices

- Store relevant metadata about a vector in other columns in your table. For example, if your vector is an image, store the original image in the same table.
- Select a pre-trained model based on the queries you will need to make to your database.

Limitations

While the vector embeddings can replace or augment some functions of a traditional database, vector embeddings are not a replacement for other data types. Embeddings are best applied as a supplement to existing data because of the limitations:

- Vector embeddings are not human-readable. Embeddings are not recommended when seeking to directly retrieve data from a table.
- The model might not be able to capture all relevant information from the data, leading to incorrect or incomplete results.

Working with Vector Search

Working with Vector Search

Create vector keyspace

Create the keyspace you want to use for your Vector Search table. This example uses `cycling` as the `keyspace name`:

```
CREATE KEYSPACE IF NOT EXISTS cycling
   WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : '1' };
```

Use vector keyspace

Select the keyspace you want to use for your Vector Search table. This example uses `cycling` as the `keyspace name`:

```
USE cycling;
```

Create vector table

Create a new table in your keyspace, including the `comments_vector` column for vector. The code below creates a vector with five values:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
```

Optionally, you can alter an existing table to add a vector column:

```
ALTER TABLE cycling.comments_vs
   ADD comment_vector VECTOR <FLOAT, 5> ;
```

Create vector index

Create the custom index with Storage Attached Indexing (SAI):

```
CREATE INDEX IF NOT EXISTS ann_index
  ON cycling.comments_vs(comment_vector) USING 'sai';
```

For more about SAI, see the **Storage Attached Indexing** documentation.

IMPORTANT

The index can be created with options that define the similarity function:

```
CREATE INDEX IF NOT EXISTS ann_index
    ON vsearch.com(item_vector) USING 'sai'
WITH OPTIONS = { 'similarity_function': 'DOT_PRODUCT' };
```

Valid values for the `similarity_function` are `DOT_PRODUCT`, `COSINE`, or `EUCLIDEAN`.

Load vector data into your database

Insert data into the table using the new type:

```
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-02-14 12:43:20-0800',
      'Raining too hard should have postponed',
      'Alex',
      [0.45, 0.09, 0.01, 0.2, 0.11]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-03-21 13:11:09.999-0800',
      'Second rest stop was out of water',
      'Alex',
      [0.99, 0.5, 0.99, 0.1, 0.34]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
```

```
comment_vector)
   VALUES (
      now(),
      e7ae5cf3-d358-4d99-b900-85902fda9bb0,
      '2017-04-01 06:33:02.16-0800',
      'LATE RIDERS SHOULD NOT DELAY THE START',
      'Alex',
      [0.9, 0.54, 0.12, 0.1, 0.95]
);

INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      totimestamp(now()),
      'The gift certificate for winning was the best',
      'Amy',
      [0.13, 0.8, 0.35, 0.17, 0.03]
);

INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-02-17 12:43:20.234+0400',
      'Glad you ran the race in the rain',
      'Amy',
      [0.3, 0.34, 0.2, 0.78, 0.25]
);

INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-03-22 5:16:59.001+0400',
      'Great snacks at all reststops',
      'Amy',
      [0.1, 0.4, 0.1, 0.52, 0.09]
);
INSERT INTO cycling.comments_vs (record_id, id, created_at, comment, commenter,
comment_vector)
   VALUES (
      now(),
      c7fceba0-c141-4207-9494-a29f9809de6f,
      '2017-04-01 17:43:08.030+0400',
```

```
      'Last climb was a killer',
      'Amy',
      [0.3, 0.75, 0.2, 0.2, 0.5]
);
```

Query vector data with CQL

To query data using Vector Search, use a `SELECT` query:

```
SELECT * FROM cycling.comments_vs
    ORDER BY comment_vector ANN OF [0.15, 0.1, 0.1, 0.35, 0.55]
    LIMIT 3;
```

To obtain the similarity calculation of the best scoring node closest to the query data as part of the results, use a `SELECT` query:

```
SELECT comment, similarity_cosine(comment_vector, [0.2, 0.15, 0.3, 0.2, 0.05])
    FROM cycling.comments_vs
    ORDER BY comment_vector ANN OF [0.1, 0.15, 0.3, 0.12, 0.05]
    LIMIT 1;
```

The supported functions for this type of query are:

- similarity_dot_product
- similarity_cosine
- similarity_euclidean

with the parameters of (<vector_column>, <embedding_value>). Both parameters represent vectors.

NOTE

- The limit must be 1,000 or fewer.
- Vector Search utilizes Approximate Nearest Neighbor (ANN) that in most cases yields results almost as good as the exact match. The scaling is superior to Exact Nearest Neighbor (KNN).
- Least-similar searches are not supported.
- Vector Search works optimally on tables with no overwrites or deletions of the `item_vector` column. For an `item_vector` column with changes, expect slower search results.