Pre-General Availability: 2023-11-27

Oracle® Database

Oracle AI Vector Search User's Guide

Release 23.4
F87786-01
November 2023

Oracle Database Oracle AI Vector Search User's Guide, Release 23.4

F87786-01

Primary Author: Jean-Francois Verrier

Contributing Authors: Jessica True, Jody Glover

Contributors: Doug Hood, Yuan Zhou, Usha Krishnamurthy, Prakash Jashnani

Contents

1

Oracle AI Vector Search Overview

Oracle AI Vector Search stores and indexes vector embeddings for fast retrieval and similarity search.

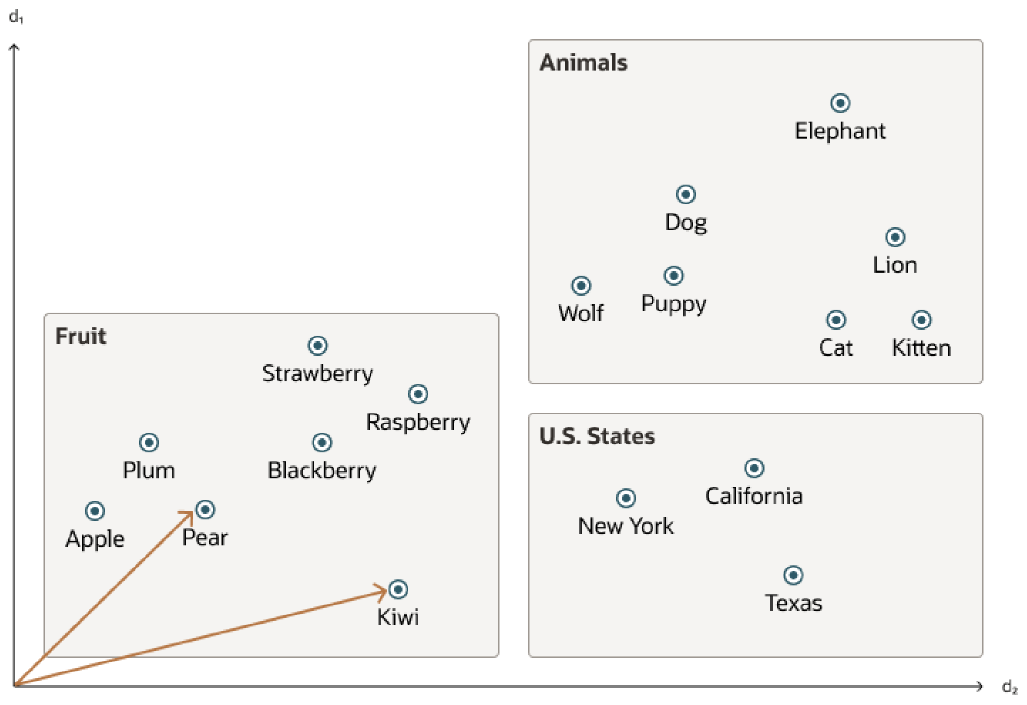
About Oracle AI Vector Search

Vector Databases are a new classification of specialized databases that are designed for Artificial Intelligence (AI) workloads that allow you to query data based on semantics, rather than keywords.

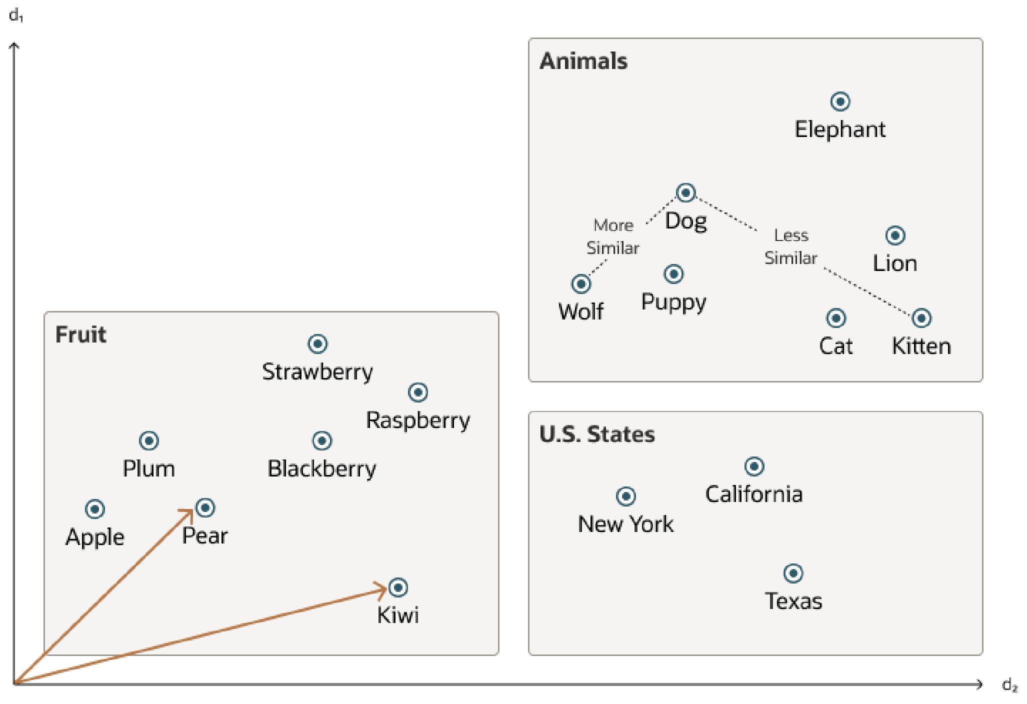
If you've ever used applications like voice assistants, chatbots, language translators, recommendation systems, anomaly detection, or video search and recognition, you've implicitly used features of a vector database.

Oracle AI Vector Search stores vector embeddings, which are mathematical vector representations of data points. These vector embeddings describe the semantic meaning behind content like words, documents, audio tracks, or images. This representation translates the semantic similarity of objects, as perceived by humans, into proximity in a mathematical vector space. This vector space has usually multihundreds, if not thousands, of dimensions with nonzero values in each coordinate. Said differently, vector embeddings are a way of representing almost any kind of data, like text, images, videos, users, music, and so on, as points in a multidimensional space where the locations of those points in space are semantically meaningful.

This simplified diagram illustrates a vector space where vectors are collapsed into two dimensions instead of hundreds:

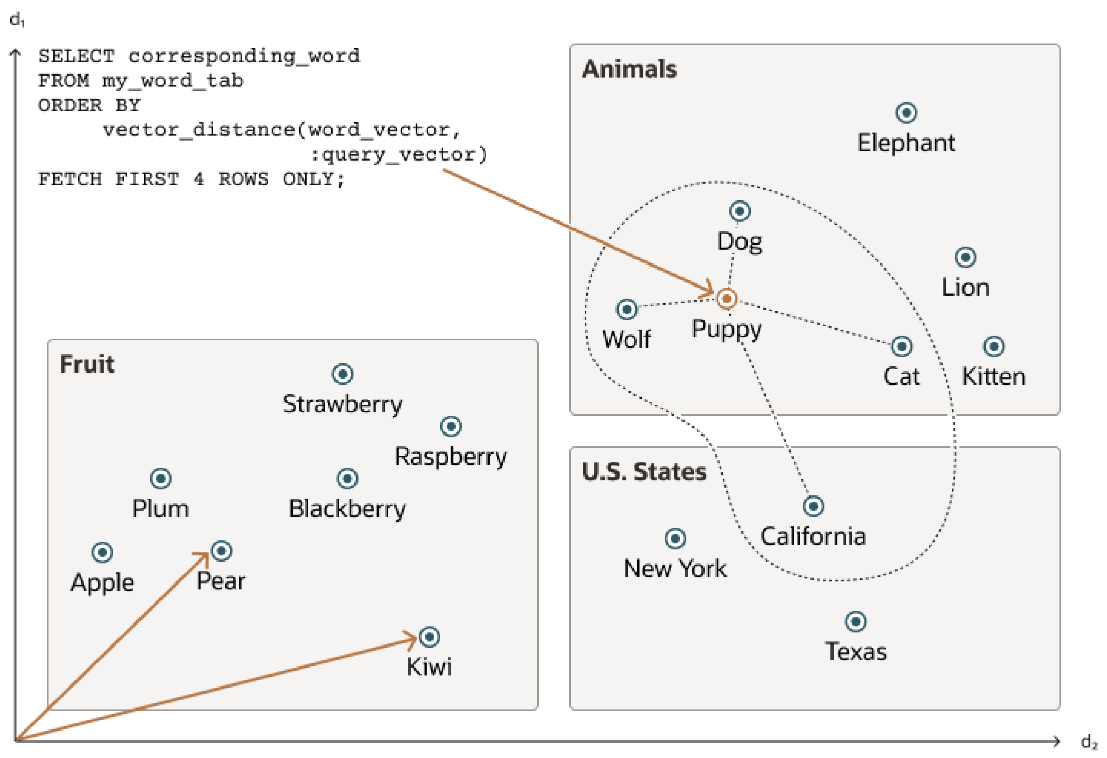


Searching semantic similarity in a data set is now equivalent to searching nearest neighbors in a vector space instead of using traditional keyword searches using query predicates. As illustrated in the following diagram, the distance between *dog* and *wolf* in this vector space is shorter than the distance between *dog* and *kitten.* In this space, a dog is more similar to a wolf than it is to a kitten.



Vector data tends to be unevenly distributed and clustered into groups that are semantically related. Doing a similarity search based on a given query vector is equivalent to retrieving the n-nearest vectors to your query vector in your vector space. Basically, you need to find an ordered list of vectors where the first row in the list is the closest or most similar vector to the query vector, the second row in the list is the second closest vector to the query vector, and so on. When doing a similarity search, the relative order of distances is what really matters rather than the actual distance.

Using the preceding vector space, here is an illustration of a semantic search where your query vector is the one corresponding to the word *Puppy* and you want to identify the four closest words :

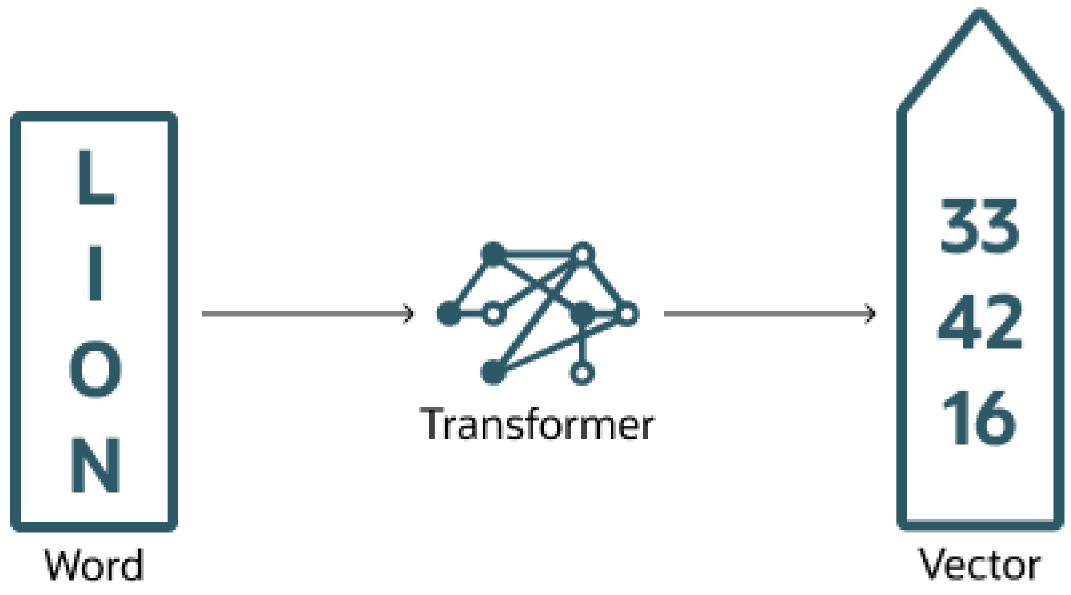


```
SELECT corresponding_word
FROM my_word_tab
ORDER BY
     vector_distance(word_vector,
                     :query_vector)
FETCH FIRST 4 ROWS ONLY;
```

Similarity searches tend to get data from one or more clusters depending on the value of the query vector and the fetch size.

One way of creating such vector embeddings could be to use someone's domain expertise to quantify a predefined set of features or dimensions like shape, texture, color, sentiment, and many others, depending on the object type with which you're dealing. However, this method is not cost effective and no longer used.

Instead, vector embeddings are created via neural networks. Most modern vector embeddings use a transformer model, as illustrated by the following diagram, but convolutional neural networks are still sometimes used.



Depending on the type of your data, you can use different pretrained, open-source models to create vector embeddings. For example:

- For textual data, Word2Vec or BERT transform words, sentences, or paragraphs into vector embeddings.
- For visual data, you can use Residual Network (ResNet) to generate vector embeddings.
- For audio data, you can use their visual spectrogram representation to fall back into the visual data case.

Each model also determines the number of dimensions for your vectors. For example:

- Cohere embed-english-v2.0 has 4096 dimensions.
- Cohere embed-english-light-v2.0 has 1024 dimensions.
- Cohere embed-multilingual-v2.0 has 768 dimensions.
- OpenAI text-embedding-ada-002 has 1536 dimensions.

Of course, you could always create your own model that is trained with your own data set. However, this is rarely done, given the large list of open-source models that exist, covering many different spaces. Another technique involves fine-tuning an existing model with your own specialized data set.

Why Use Oracle AI Vector Search?

The biggest benefit of Oracle AI Vector Search is that semantic search on unstructured data can be combined with relational search on business data in one single system.

This is not only powerful but also significantly more effective because you don't need to add a specialized vector database, eliminating the pain of data fragmentation between multiple systems.

For example, you use an application that allows you to find a house that is similar to a picture you took of one you like that is located in your preferred area for a certain budget. Finding a good match in this case requires combining a semantic picture search with searches on business data.

With Oracle AI Vector Search, you could create the following table:

```
CREATE TABLE house_for_sale (house_id     NUMBER,
                             price        NUMBER,
                             city         VARCHAR2(400),
                             house_photo  BLOB,
                             house_vector VECTOR);
```

With that table, you could run the following query to answer your basic question:

```
SELECT house_photo, city, price
FROM   house_for_sale
WHERE  price <= :input_price AND
       city  = :input_city
ORDER BY VECTOR_DISTANCE(house_vector, :input_vector);
```

In addition, because Oracle has been building database technologies for so long, your vectors can benefit from all of Oracle Database's most powerful features, like the following:

- Partitioning
- Real Application Clusters scalability
- Exadata smart scans
- Shard processing across geographically distributed databases
- Transactions
- Parallel SQL
- Disaster recovery
- Security

2

Using the Oracle Database VECTOR Data Type

You can declare a table's column as a `VECTOR` data type.

The following command shows a simple example:

```
CREATE TABLE my_vectors (id NUMBER, embedding VECTOR);
```

In this example, you don't have to specify the number of dimensions or their format, which are both optional. If you don't specify any of them, you can enter vectors of different dimensions with different formats. This is a simplification to help you get started with using vectors in Oracle Database.

Here's a more complex example that imposes more constraints on what you can store:

```
CREATE TABLE my_vectors (id NUMBER, embedding VECTOR(768, INT8)) ;
```

In this example, each vector that is stored must have 768 dimensions, and each of them will be formatted as an `INT8`. The number of dimensions must be strictly greater than 0 with no practical limit. The possible dimension formats are `INT8` (8-bit integers), `FLOAT32` (32-bit floating-point numbers), and `FLOAT64` (64-bit floating-point numbers). `FLOAT32` and `FLOAT64` are the IEEE standards. Oracle Database automatically casts the values as needed.

You can declare the following forms for a vector:

- `VECTOR` equivalent to `VECTOR(*, *)`: Vectors can have an arbitrary number of dimensions and formats.
- `VECTOR(`*`number_of_dimensions`*`, *)` equivalent to `VECTOR(`*`number_of_dimensions`*`)`: Vectors must all have the specified number of dimensions or an error is thrown. Every vector will have its dimensions stored without format modification.
- `VECTOR(*, dimension_element_format)`: Vectors can have an arbitrary number of dimensions, but their format will be up-converted or down-converted to the specified dimension element format `(INT8, FLOAT32, or FLOAT64)`.
- `VECTOR(number_of_dimensions, dimension_element_format)`: Vectors must all have the specified number of dimensions or an error is thrown, and each will be up-converted or down-converted to the specified dimension element format (`INT8`, `FLOAT32`, or `FLOAT64`).

A vector can be `NULL` but its dimensions cannot (for example, you cannot have `[1.1, NULL, 2.2]`).

The following SQL*Plus code example shows how the system interprets various vector definitions:

```
SQL> CREATE TABLE my_vect_tab (
2    v1 VECTOR(3, FLOAT32),
3    v2 VECTOR(2, FLOAT64),
4    v3 VECTOR(1, INT8),
5    v4 VECTOR(1, *),
6    v5 VECTOR(*, FLOAT32),
7    v6 VECTOR(*, *),
8    v7 VECTOR
9  );

Table created.

SQL> DESC my_vect_tab;
 Name                                      Null?    Type
 ----------------------------------------- --------
----------------------------
 V1                                                 VECTOR(3 , FLOAT32)
 V2                                                 VECTOR(2 , FLOAT64)
 V3                                                 VECTOR(1 , INT8)
 V4                                                 VECTOR(1 , *)
 V5                                                 VECTOR(* , FLOAT32)
 V6                                                 VECTOR(* , *)
 V7                                                 VECTOR(* , *)
```

3

Using Oracle AI Vector Search Operations

There are a number of SQL functions and operators that you use in Oracle AI Vector Search to create and manipulate vectors.

Base Operations

Base operations for Oracle AI Vector Search involve creating, converting, sizing, and describing vectors.

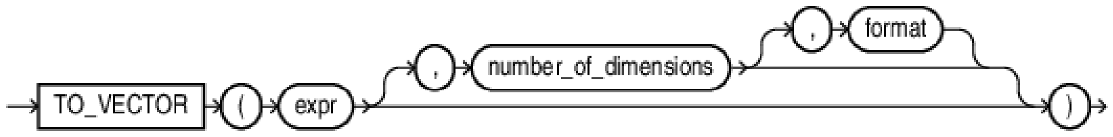
Vector Constructors

`TO_VECTOR()` and `VECTOR()` are synonymous constructors of vectors. The functions take a string of type `VARCHAR2` or `CLOB` as input and return a vector as output.

TO_VECTOR

Syntax

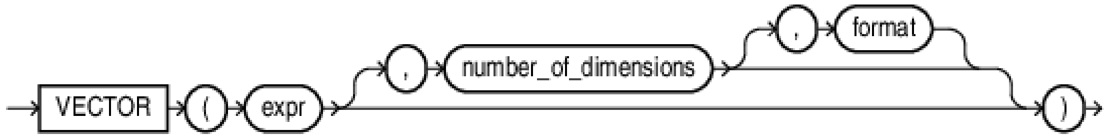
```
TO_VECTOR(expr [ , number_of_dimensions [ , format ] ] )
```



VECTOR

Syntax

```
VECTOR ( expr [ , number_of_dimensions [ , format ] ] )
```



Parameters

- *`expr`* must evaluate to either (a) a string that represents a vector or (b) a vector. The valid string representation of a vector is in the form of an array of non-null numbers enclosed with a bracket and separated by commas, such as `'[1, 3.4, -05.60, 3e+4]'`. If it evaluates to a vector, the function serves the purpose of converting it to the specified or default format. The expression can be `NULL`, in which case the result is `NULL` as well.
- *`number_of_dimensions`* must be a numeric value that describes the number of dimensions of the vector to construct. Specifying the number of dimensions as constants helps with identifying dimension mismatches during certain data processing actions, such as distance calculation at compile time. The number of dimensions may also be specified as an asterisk `*`, in which case the dimension is determined by `expr`.
- *`format`* must be one of the following tokens: `INT8`, `FLOAT32`, `FLOAT64`, or `*`. This is the target internal storage format of the vector. If `*` is used, the format will be `FLOAT32`. Note that this behavior is a bit different than declaring a vector column. If you declare a column of type `VECTOR(3, *)`, then all inserted vectors will NOT have their storage format modified (stored as is).

Examples

```
SELECT TO_VECTOR('[34.6, 77.8]');
SELECT TO_VECTOR('[34.6, 77.8]', 2, FLOAT32);

SQL> SELECT TO_VECTOR('[34.6, 77.8]', 2, FLOAT32) FROM dual;

TO_VECTOR('[34.6,77.8]',2,FLOAT32)
-----------------------------------------------------------------------
---------
[3.45999985E+001,7.78000031E+001]

1 row selected.

SQL> SELECT TO_VECTOR('[34.6, 77.8, -89.34]', 3, FLOAT32);

TO_VECTOR('[34.6,77.8,-89.34]',3,FLOAT32)
-----------------------------------------------------------------------
---------
[3.45999985E+001,7.78000031E+001,-8.93399963E+001]

1 row selected.
```

Note:

- Applications using Oracle Client 23c libraries or Thin mode drivers can insert vector data directly as a string or a `CLOB`. For example:

  ```
  INSERT INTO vecTab VALUES ('[1.1, 2.9, 3.14]');
  ```

- For applications using pre-Oracle Client 23c libraries connected to Oracle Database 23c, use the `TO_VECTOR()` SQL function to insert vector data. For example:

  ```
  INSERT INTO vecTab VALUES(TO_VECTOR('[1.1, 2.9, 3.14]'));
  ```

Vector Serializers

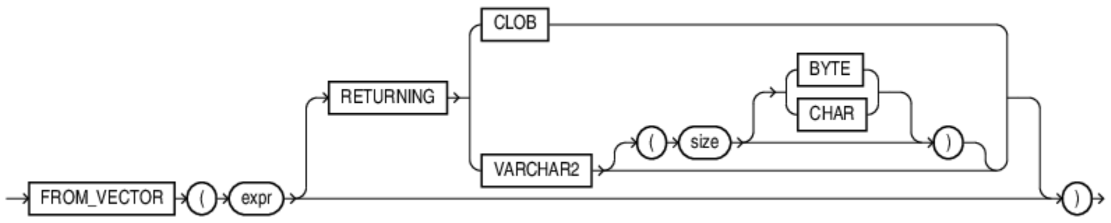
FROM_VECTOR() and VECTOR_SERIALIZE() are synonymous serializers of vectors.

Both functions take a vector as input and return a string of type `VARCHAR2` or `CLOB` as output. They optionally take a `RETURNING` clause to specify the returned data type. If `VARCHAR2` is specified without size, the returned value size is 32767. There is no support to convert to `CHAR`, `NCHAR`, and `NVARCHAR2`.

FROM_VECTOR

Syntax

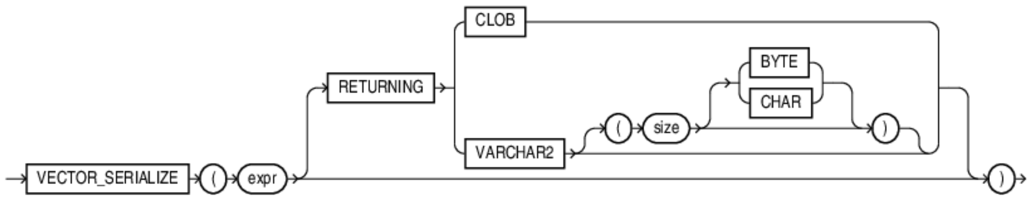
```
FROM_VECTOR ( expr [ RETURNING ( CLOB | VARCHAR2 [ ( size [BYTE |
CHAR] ) ] ) ] )
```



VECTOR_SERLIAZE

Syntax

```
VECTOR_SERIALIZE ( expr [ RETURNING ( CLOB | VARCHAR2 [ ( size [BYTE |
CHAR] ) ] ) ] )
```



Parameters

expr must be a vector type. The function returns `NULL` if *expr* is `NULL`.

Examples

```
SELECT FROM_VECTOR(TO_VECTOR('[1, 2, 3]') );

SQL> SELECT FROM_VECTOR(TO_VECTOR('[1.1, 2.2, 3.3]', 3, FLOAT32) );

VECTOR_SERIALIZE(VECTOR('[1.1,2.2,3.3]',3,FLOAT32))
-----------------------------------------------------------------------
---------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.

SQL> SELECT FROM_VECTOR( TO_VECTOR('[1.1, 2.2, 3.3]', 3, FLOAT32)
RETURNING VARCHAR2(1000));

VECTOR_SERIALIZE(VECTOR('[1.1,2.2,3.3]',3,FLOAT32)RETURNINGVARCHAR2(100
0))
-----------------------------------------------------------------------
---------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.

SQL> SELECT FROM_VECTOR(TO_VECTOR('[1.1, 2.2, 3.3]', 3, FLOAT32)
RETURNING CLOB );

VECTOR_SERIALIZE(VECTOR('[1.1,2.2,3.3]',3,FLOAT32)RETURNINGCLOB)
-----------------------------------------------------------------------
---------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.
```

Note:

Applications using Oracle Client 23c libraries or Thin mode drivers can fetch vector data directly, as shown in the following example:

```
SELECT dataVec FROM vecTab;
```

For applications using pre-Oracle Client 23c libraries connected to Oracle Database 23c, use the `FROM_VECTOR()` SQL function to fetch vector data, as shown by the following example:

```
SELECT FROM_VECTOR(dataVec) FROM vecTab;
```

VECTOR_NORM

The `VECTOR_NORM()` function returns the Euclidean norm of a vector in the format of `BINARY_DOUBLE`.

This value is also called *magnitude* or *size* and represents the Euclidean distance between the vector and the origin.

Syntax

```
VECTOR_NORM(expr)
```

VECTOR_NORM
(
expr
)

Parameters

expr must evaluate to a vector. The function returns `NULL` if *expr* is `NULL`.

Example

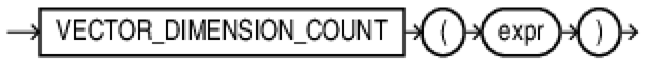
```
SQL> SELECT VECTOR_NORM( TO_VECTOR('[4, 3]', 2, FLOAT32) );

VECTOR_NORM(TO_VECTOR('[4,3]',2,FLOAT32))
_____________________________________________
5.0
```

VECTOR_DIMENSION_COUNT

The `VECTOR_DIMENSION_COUNT()` function returns the number of dimensions of a vector in the format of an Oracle number.

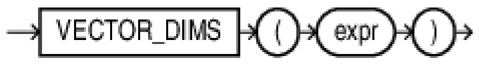
Syntax

```
VECTOR_DIMENSION_COUNT( expr )
```



`VECTOR_DIMS` is a synonym of `VECTOR_DIMENSION_COUNT()`.

```
VECTOR_DIMS( expr )
```



Parameters

`expr` must evaluate to a vector. These functions return `NULL` if expr is `NULL`.

Examples

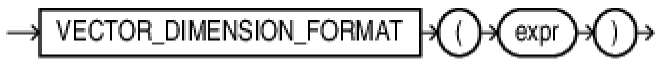
```
SQL> SELECT VECTOR_DIMENSION_COUNT( TO_VECTOR('[34.6, 77.8]', 2,
FLOAT64) );
VECTOR_DIMENSION_COUNT(TO_VECTOR('[34.6,77.8]',2,FLOAT64))
----------------------------------------------------------

                                                         2
SQL>
```

VECTOR_DIMENSION_FORMAT

The `VECTOR_DIMENSION_FORMAT()` returns the storage format of the vector. It returns a string of `'INT8'`, `'FLOAT32'`, or `'FLOAT64'`.

Syntax

```
VECTOR_DIMENSION_FORMAT( expr )
```



Parameters

`expr` must evaluate to a vector. This function returns `NULL` if *`expr`* is `NULL`.

Examples

```
SQL> SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8]', 2,
FLOAT64));
FLOAT64

SQL> SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8, 9]', 3,
FLOAT32));
FLOAT32
```

```
SQL> SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8, 9, 10]', 3,
INT8));
SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8, 9, 10]', 3,
INT8))
                                                                     *
ERROR at line 1:
ORA-51803: Vector dimension count must match the dimension count specified
inthe column definition (actual: 4, required: 3).

SQL> SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8, 9.10]', 3,
INT8));
VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6,77.8,9.10
--------------------------------------------------
INT8

SQL> SELECT TO_VECTOR('[34.6, 77.8, 9.10]', 3, INT8);
TO_VECTOR('[34.6,77.8,9.10]',3,INT8)
-----------------------------------------------------------------------------
---
[3.5E+001,7.8E+001,9.0E+000]
SQL>
```

Distance Metrics

Measuring distances in a vector space is at the heart of identifying the most relevant results for a given query vector. That process is very different from the well-known keyword filtering in the relational database world.

In the vector world, multiple mathematical methods called distances determine how similar, or how far, two vectors are in your vector space. The time it takes to calculate those distances grows with the number of vector dimensions, which can go into thousands of dimensions. Generally it's best to match the distance metric you use to the one that was used to train your vector embedding model.

Oracle AI Vector Search supports the following distance metrics:

- Euclidean
- Squared Euclidean
- Cosine distance
- Negated Dot Product
- Manhattan
- Hamming

VECTOR_DISTANCE

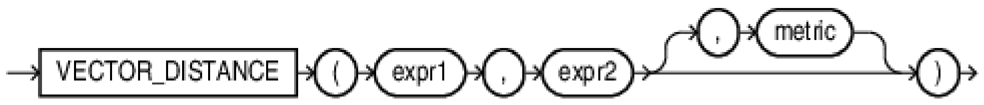
`VECTOR_DISTANCE()` is the main function that allows you to calculate distances between two vectors. This function takes two vectors as parameters and you can optionally specify a distance metric to calculate the distance in the way that you want. If you do not specify a distance metric, then the default distance metric is Squared Euclidean distance.

You can optionally use the following shorthand functions too : `L1_DISTANCE`, `L2_DISTANCE`, `COSINE_DISTANCE` and `INNER_PRODUCT`. These functions take two vectors as input and return the distance between them.

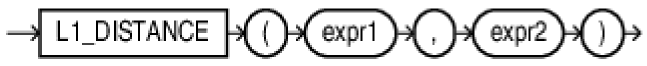
All of these functions return the vectors' distance as a `BINARY_DOUBLE`.

Syntax

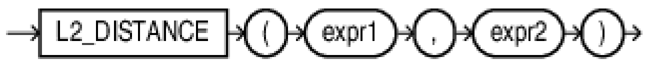
```
VECTOR_DISTANCE( expr1, expr2 [, metric] )
```



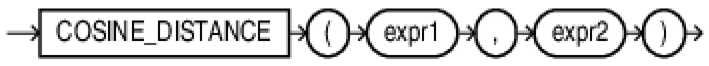
```
L1_DISTANCE ( expr1, expr2 )
```



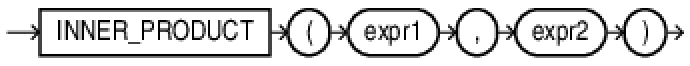
```
L2_DISTANCE ( expr1, expr2 )
```



```
COSINE_DISTANCE ( expr1, expr2 )
```



```
INNER_PRODUCT ( expr1, expr2 )
```



Parameters

- *`expr1`* and *`expr2`* must evaluate to vectors and have the same number of dimensions. This function returns `NULL` if either *`expr1`* or *`expr2`* is `NULL`.

- *metric* must be one of the following tokens: `MANHATTAN`, `EUCLIDEAN`, `DOT`, `COSINE`, `HAMMING`, and `EUCLIDEAN_SQUARED` or `L2_SQUARED`.
  - **`EUCLIDEAN_SQUARED`** (or `L2_SQUARED`) is the default metric. It is the Euclidean distance without taking the square root.
  - `MANHATTAN` metric calculates the Manhattan distance (also known as L1 distance or taxicab distance) between two vectors. Using `L1_DISTANCE()` is equivalent to this metric.
  - `EUCLIDEAN` metric calculates the Euclidean distance (also known as L2 distance) between two vectors. Using `L2_DISTANCE()` is equivalent to this metric.
  - `DOT` metric calculates the negated dot product (also known as inner product) of two vectors. Using `INNER_PRODUCT()` is equivalent to the negation of this metric.
  - `COSINE` metric calculates the cosine distance between two vectors. Using `COSINE_DISTANCE()` is equivalent to this metric.
  - `HAMMING` metric calculates the hamming distance between two vectors.

Examples

- `VECTOR_DISTANCE(expr1, expr2, EUCLIDEAN) is equivalent to L2_DISTANCE(expr1, expr2)`
- `VECTOR_DISTANCE(expr1, expr2, COSINE) is equivalent to COSINE_DISTANCE(expr1, expr2)`
- `VECTOR_DISTANCE(expr1, expr2, DOT) is equivalent to -1*INNER_PRODUCT(expr1, expr2)`
- `VECTOR_DISTANCE(expr1, expr2, MANHATTAN) is equivalent to L1_DISTANCE(expr1, expr2)`

Shorthand Operators for Distances

You can use the following shorthand distance operators in lieu of their corresponding distance functions:

- `<->` is the Euclidian distance operator: `expr1 <-> expr2` is equivalent to `L2_DISTANCE(expr1, expr2)` or `VECTOR_DISTANCE(expr1, expr2, EUCLIDEAN)`
- `<=>` is the cosine distance operator: `expr1 <=> expr2` is equivalent to `COSINE_DISTANCE(expr1, expr2)` or `VECTOR_DISTANCE(expr1, expr2, COSINE)`
- `<#>` is the negative dot product operator: `expr1 <#> expr2` is equivalent to `-1*INNER_PRODUCT(expr1, expr2)` or `VECTOR_DISTANCE(expr1, expr2, DOT)`

Syntax

```
<expr1> <-> <expr2>

<expr1> <#> <expr2>

<expr1> <=> <expr2>
```

Parameters

`expr1` and `expr2` must evaluate to vectors and have the same number of dimensions. These operations return `NULL` if either `expr1` or `expr2` is `NULL`.

Examples

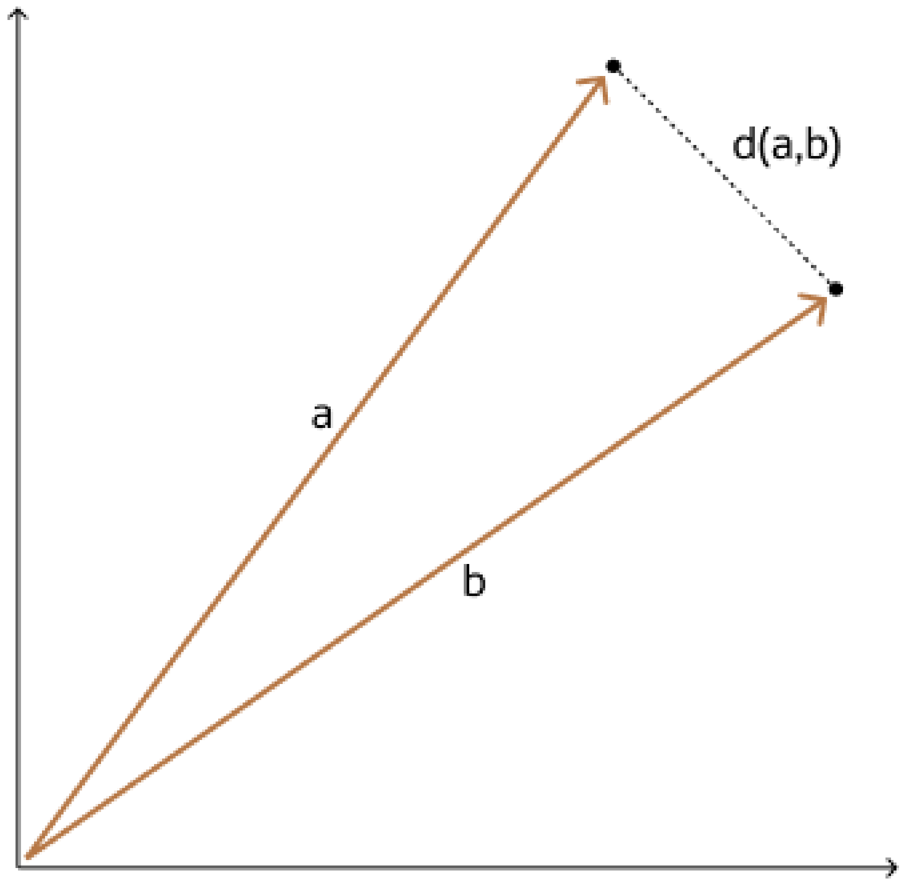
- `'[1, 2]' <-> '[0,1]'`
- `v1 <-> '[' || '1,2,3' || ']'` is equivalent to `v1 <-> '[1, 2, 3]'`
- `v1 <-> '[1,2]'` is equivalent to `L2_DISTANCE(v1, '[1,2]')`
- `v1 <=> v2` is equivalent to `COSINE_DISTANCE(v1, v2)`
- `v1 <#> v2` is equivalent to `-1*INNER_PRODUCT(v1, v2)`

Euclidean and Squared Euclidean Distances

Euclidean distance reflects the distance between each of the vectors' coordinates being compared—basically the straight-line distance between two vectors. This is calculated using the Pythagorean theorem applied to the vector's coordinates.

This metric is sensitive to both the vector's size and it's direction.

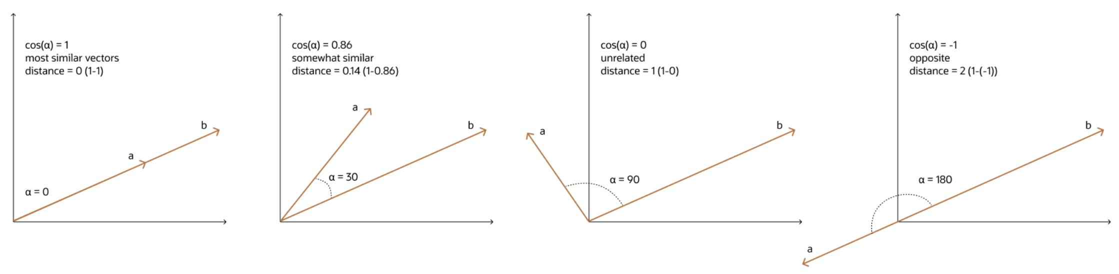
With Euclidean distances, comparing squared distances is equivalent to comparing distances. So, when ordering is more important than the distance values themselves, the Squared Euclidean distance is very useful as it is faster to calculate than the Euclidean distance (avoiding the square-root calculation).



Cosine Similarity

One of the most widely used similarity metric, especially in natural language processing (NLP), is cosine similarity, which measures the cosine of the angle between two vectors.

The smaller the angle, the more similar are the two vectors. Cosine similarity measures the similarity in the direction or angle of the vectors, ignoring differences in their size (also called *magnitude*). The smaller the angle, the bigger is its cosine. So the cosine distance and the cosine similarity have an inverse relationship. While cosine distance measures how different two vectors are, cosine similarity measures how similar two vectors are.



Dot Product Similarity

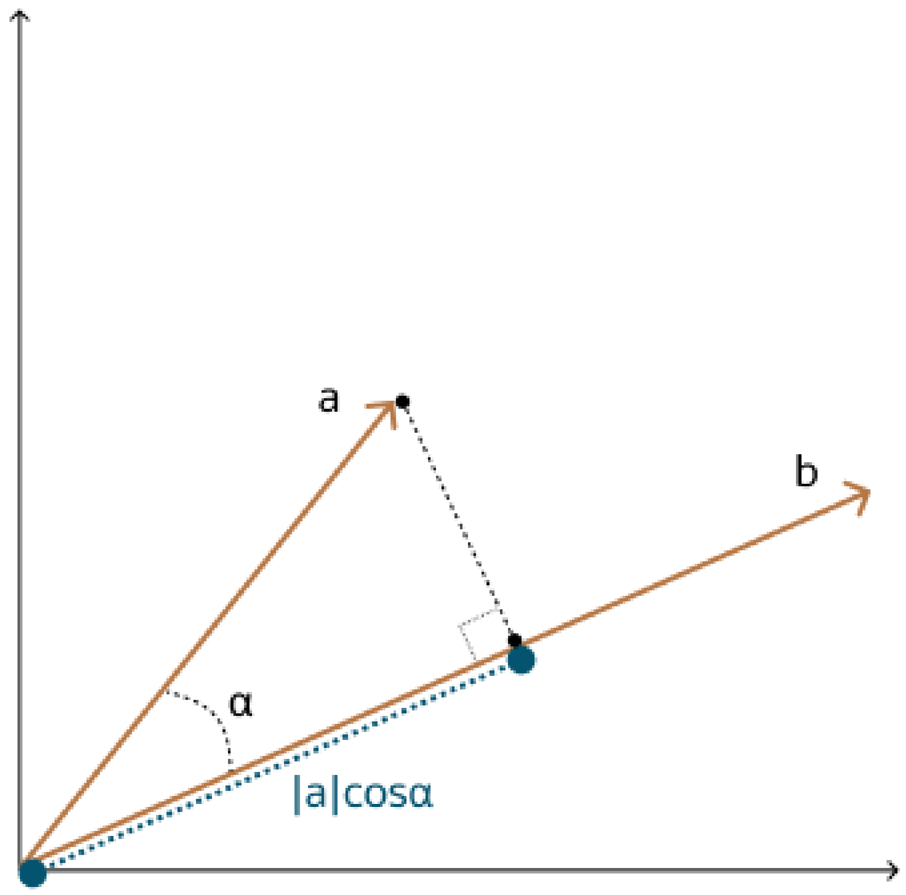
The dot product similarity of two vectors can be viewed as multiplying the size of each vector by the cosine of their angle. The corresponding geometrical interpretation of this definition is equivalent to multiplying the size of one of the vectors by the size of the projection of the second vector onto the first one, or vice versa.

As illustrated in the following diagram, you project one vector on the other and multiply the resulting vector sizes.

Incidentally, this is equivalent to the sum of the products of each vector's coordinate. Often, you do not have access to the cosine of the two vector's angle, hence this calculation is easier.

Larger dot product values imply that the vectors are more similar, while smaller values imply that they are less similar. Compared to using Euclidean distance, using the dot product similarity is specially useful for high-dimensional vectors.

Note that normalizing vectors and using the dot product similarity is equivalent to using cosine similarity. There are cases where the first one is faster to evaluate than the second one, and vice versa. A normalized vector is created by dividing each dimension by the norm (or length) of the vector, such that the norm of the normalized vector is equal to 1.

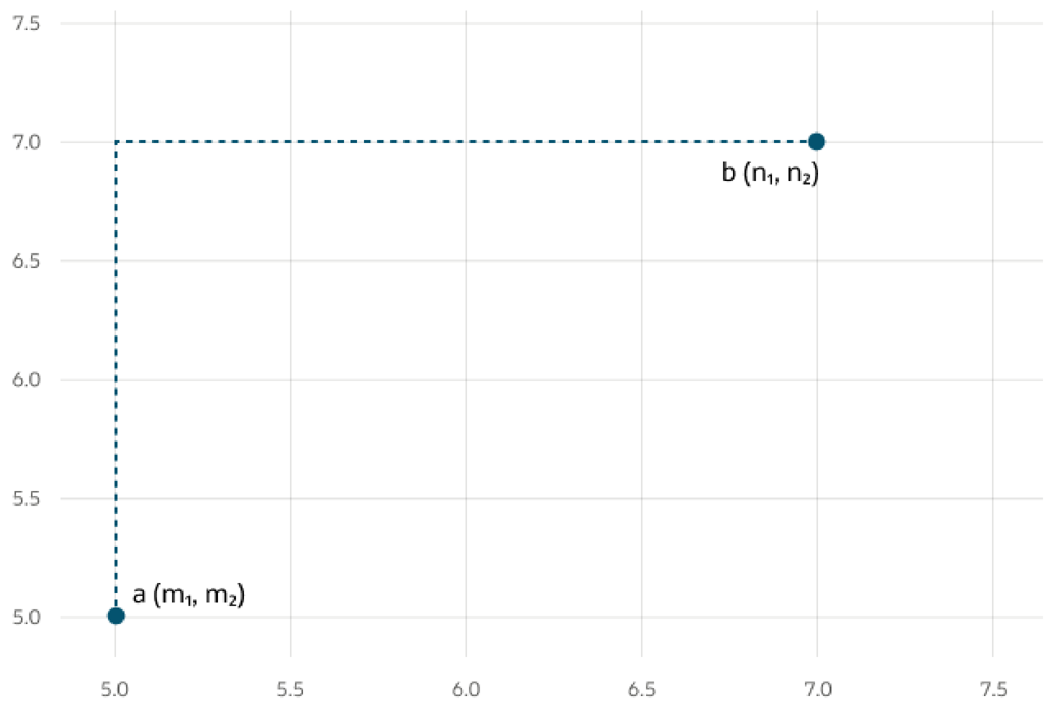


Manhattan Distance

This metric is calculated by summing the distance between the coordinates of the two vectors that you want to compare.

Imagine yourself in the streets of Manhattan trying to go from point A to point B. A straight line is not possible.

This metric is most useful for vectors describing objects on a uniform grid, like city blocks, power grids, or a chessboard. Compared to the Euclidean metric, the Manhattan metric is faster for calculations and you can use it advantageously for higher dimensional vector spaces.

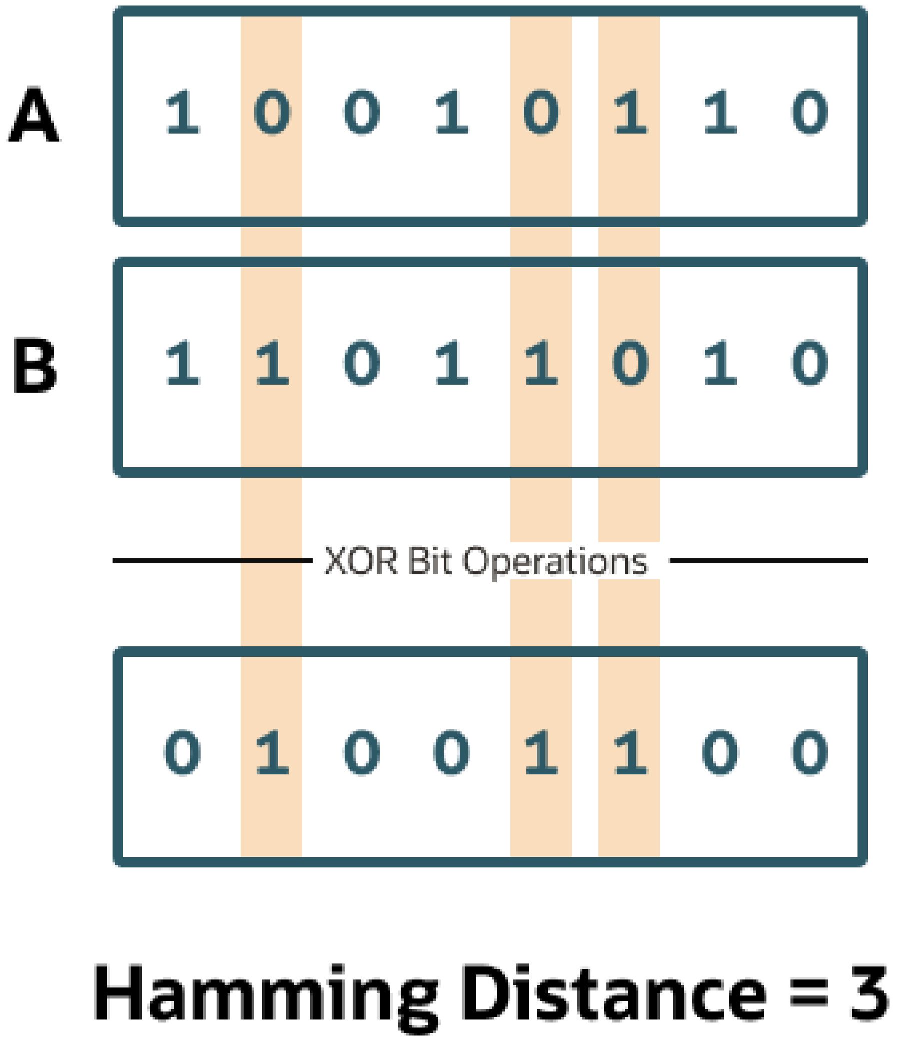


Hamming Similarity

The Hamming distance between two vectors represents the number of coordinates where they differ.

For example, when using binary vectors, the Hamming distance between two vectors is the number of bits you must change to change one vector into the other. To compute the Hamming distance between two vectors, you need to compare the position of each bit in the sequence. You can do this by using `exclusive or` (also called the XOR bit operation), which outputs 1 if the bits in the sequence do not match, and 0 otherwise. It's important to note that the bit strings need to be of equal length for the comparison to make sense.

The Hamming metric is mainly used with binary vectors for error detection over networks.


A
1 0 0 1 0 1 1 0
B
1 1 0 1 1 0 1 0
XOR Bit Operations
0 1 0 0 1 1 0 0
Hamming Distance = 3

4

Using Similarity Search

A similarity search looks for the relative order of vectors compared to a query vector. Naturally, the comparison is done using a particular distance metric but what is important is the result set of your top closest vectors, not the distance between them.

For example, the Euclidean similarity search involves retrieving the top-k nearest vectors in your space relative to the Euclidean distance metric and a query vector. Here's an example that retrieves the top 10 vectors from the `vector_tab` table that are the nearest to `query_vector`:

```
SELECT docID FROM vector_tab ORDER BY
VECTOR_DISTANCE( embedding, :query_vector, EUCLIDEAN ) FETCH FIRST 10 ROWS
ONLY;
```

In this example, `docID` and `embedding` are columns defined in the `vector_tab` table and `embedding` has the `VECTOR` data type.

In the case of Euclidean distances, comparing squared distances is equivalent to comparing distances. So, when ordering is more important than the distance values themselves, the Squared Euclidean distance is very useful as it is faster to calculate than the Euclidean distance (avoiding the square-root calculation). Consequently, the previous similarity search example could be advantageously rewritten like this:

```
SELECT docID FROM vector_tab ORDER BY
VECTOR_DISTANCE( embedding, :query_vector) FETCH FIRST 10 ROWS ONLY;
```

Squared Euclidean distance is the Oracle AI Vector Search default when using the `VECTOR_DISTANCE()` function.

Here is a complete scenario that shows the difference between Cosine and Euclidean similarity searches and how it is important to select the right one. Let's imagine we have the following text documents classifying galaxies by their types:

- **DOC1**: "Messier 31 is a barred spiral galaxy in the Andromeda constellation which has a lot of barred spiral galaxies."
- **DOC2**: "Messier 33 is a spiral galaxy in the Triangulum constellation."
- **DOC3**: "Messier 58 is an intermediate barred spiral galaxy in the Virgo constellation."
- **DOC4**: "Messier 63 is a spiral galaxy in the Canes Venatici constellation."
- **DOC5**: "Messier 77 is a barred spiral galaxy in the Cetus constellation."
- **DOC6**: "Messier 91 is a barred spiral galaxy in the Coma Berenices constellation."
- **DOC7**: "NGC 1073 is a barred spiral galaxy in Cetus constellation."
- **DOC8**: "Messier 49 is a giant elliptical galaxy in the Virgo constellation."
- **DOC9**: "Messier 60 is an elliptical galaxy in the Virgo constellation."

We could create vectors representing the preceding galaxy's classes using the following five dimensions vector space based on the count of important words appearing in each document:

Table 4-1 Five dimension vector space

Galaxy Classes	Intermediate	Barred	Spiral	Giant	Elliptical
M31	0	2	2	0	0
M33	0	0	1	0	0
M58	1	1	1	0	0
M63	0	0	1	0	0
M77	0	1	1	0	0
M91	0	1	1	0	0
M49	0	0	0	1	1
M60	0	0	0	0	1
NGC1073	0	1	1	0	0

This naturally gives you the following vectors:

- **M31**: `[0,2,2,0,0]`
- **M33**: `[0,0,1,0,0]`
- **M58**: `[1,1,1,0,0]`
- **M63**: `[0,0,1,0,0]`
- **M77**: `[0,1,1,0,0]`
- **M91**: `[0,1,1,0,0]`
- **M49**: `[0,0,0,1,1]`
- **M60**: `[0,0,0,0,1]`
- **NGC1073**: `[0,1,1,0,0]`

The idea is now to do a similarity search that finds the top-3 galaxies that have a similar type as NGC 1073. In other words, find the galaxies that are barred spirals or similar to a barred spiral type. We are showing you how to manually calculate the various distances using the Cosine and Euclidean metrics. We will then do the same using similarity search queries using Oracle AI Vector Search. Note that here, "distance" or "close" *do not* represent the distance in light years between the galaxies but the difference between their types as our vector space only represents galaxies' types and not the number of light years between them.

Let's first do a Cosine similarity search by calculating the cosine distances between NGC 1073 and all the other galaxies and then order the result to finally extract the first three. Here is one possible way to manually compute the Cosine distances between the vectors:

1. We normalize the vectors by dividing each vector's coordinates by the Euclidean norm of that vector:
   - **M31**: `[0,0.7071067811865475244008443621048 5,0.70710678118654752440084436210485,0,0]`

- **M33**: `[0,0,1,0,0]`
- **M58**: `[0.5773502691896257645091487805019 6,0.5773502691896257645091487805019 6,0.5773502691896257645091487805019 6,0,0]`
- **M63**: `[0,0,1,0,0]`
- **M77**: `[0,0.7071067811865475244008443621048 5,0.7071067811865475244008443621048 5,0,0]`
- **M91**: `[0,0.7071067811865475244008443621048 5,0.7071067811865475244008443621048 5,0,0]`
- **M49**: `[0,0,0,0.7071067811865475244008443621048 5,0.7071067811865475244008443621048 5]`
- **M60**: `[0,0,0,0,1]`
- **NGC 1073**: `[0,0.7071067811865475244008443621048 5,0.7071067811865475244008443621048 5,0,0]`

2. We calculate the Cosine similarity by calculating the Dot Product distances (`SUM(Xi.Yi)`) between NGC 1073 and all the other galaxies using the preceding normalized vectors. Because the vectors are normalized, Cosine similarity is the same as the Dot Product:
   - **NGC 1073 x M31** = 1
   - **NGC 1073 x M33** = 0.70710678118654752440084436210485
   - **NGC 1073 x M58** = 0.81649658092772603273242802490197
   - **NGC 1073 x M63** = 0.70710678118654752440084436210485
   - **NGC 1073 x M77** = 1
   - **NGC 1073 x M91** = 1
   - **NGC 1073 x M49** = 0
   - **NGC 1073 x M60** = 0
3. We deduce from the preceding calculation, the Cosine distances (1 - Cosine similarity) between NGC 1073 and:
   - **M31 is** 0
   - **M33 is** 0.29289321881345247559915563789515
   - **M58 is** 0.18350341907227396726757197509803
   - **M63 is** 0.29289321881345247559915563789515
   - **M77 is** 0
   - **M91 is** 0
   - **M49 is** 1
   - **M60 is** 1

   From the preceding calculation, the three closest vectors from NGC 1073 are M31, M77, and M91 as their distances to NGC 1073 are 0. They are all indeed barred spiral galaxies. Note that the next one on the list is M58 as its distance to NGC 1073 is the smallest compared to the remaining ones. It makes sense because M58 is classified as an intermediate barred spiral galaxy. Clearly, the Cosine similarity search is giving good results in this case.

Now, let's do the same type of manual calculation but this time using the Euclidean similarity search. The Euclidean distance (`SQRT(SUM((xi-yi)2))`) between NGC 1073 and:

- **M31 is** 1.4142135623730950488016887242097
- **M33 is** 1
- **M58 is** 1
- **M63 is** 1
- **M77 is** 0
- **M91 is** 0
- **M49 is** 2
- **M60 is** 1.7320508075688772935274463415059

In this result, you can see that the first three most similar galaxies are now M77, M91, and M33. However, M33 is not a barred spiral but just a spiral one. Because Euclidean distance is looking more at the vectors' magnitude rather than their orientations, it eliminated M31 although it is a barred spiral.

You can clearly see with this example that deciding which distance metric to use is key when doing similarity searches. Most of the time, the distance metric you need to use for the best results is driven by the model you are using to generate your vector embeddings.

Now, using Oracle Database, this is the code that automates all of these calculations:

```
create table galaxies (id number, name varchar2(50), doc
varchar2(500), embedding vector);

insert into galaxies values (1, 'M31', 'Messier 31 is a barred spiral
galaxy in the Andromeda constellation which has a lot of barred spiral
galaxies.', '[0,2,2,0,0]');
insert into galaxies values (2, 'M33', 'Messier 33 is a spiral galaxy
in the Triangulum constellation.', '[0,0,1,0,0]');
insert into galaxies values (3, 'M58', 'Messier 58 is an intermediate
barred spiral galaxy in the Virgo constellation.', '[1,1,1,0,0]');
insert into galaxies values (4, 'M63', 'Messier 63 is a spiral galaxy
in the Canes Venatici constellation.', '[0,0,1,0,0]');
insert into galaxies values (5, 'M77', 'Messier 77 is a barred spiral
galaxy in the Cetus constellation.', '[0,1,1,0,0]');
insert into galaxies values (6, 'M91', 'Messier 91 is a barred spiral
galaxy in the Coma Berenices constellation.', '[0,1,1,0,0]');
insert into galaxies values (7, 'M49', 'Messier 49 is a giant
elliptical galaxy in the Virgo constellation.', '[0,0,0,1,1]');
insert into galaxies values (8, 'M60', 'Messier 60 is an elliptical
galaxy in the Virgo constellation.', '[0,0,0,0,1]');
insert into galaxies values (9, 'NGC1073', 'NGC 1073 is a barred
spiral galaxy in Cetus constellation.', '[0,1,1,0,0]');
commit;
```

```
SELECT name, VECTOR_DISTANCE( embedding, (SELECT embedding FROM galaxies
WHERE name = 'NGC1073'), COSINE ) as vector_similarity
FROM galaxies
WHERE name <> 'NGC1073'
ORDER BY VECTOR_DISTANCE( embedding, (SELECT embedding FROM galaxies WHERE
name = 'NGC1073'), COSINE ) FETCH FIRST 3 ROWS ONLY;


NAME    VECTOR_SIMILARITY
_______ _______________________________________________________________
M31     0.0000000000000002220446049250313
M91     0.0000000000000002220446049250313
M77     0.0000000000000002220446049250313



SELECT name, VECTOR_DISTANCE( embedding, (SELECT embedding FROM galaxies
WHERE name = 'NGC1073'), EUCLIDEAN ) as vector_distance
FROM galaxies
WHERE name <> 'NGC1073'
ORDER BY VECTOR_DISTANCE( embedding, (SELECT embedding FROM galaxies WHERE
name = 'NGC1073'), EUCLIDEAN ) FETCH FIRST 3 ROWS ONLY;


NAME    VECTOR_DISTANCE
_______ _______________________________________________________________
M77     0.0
M91     0.0
M33     1.0
```

In the preceding `SELECT` statements, not only can you filter by vector distances but you can simultaneously use relational predicates on traditional data types. This avoids the need to store vectors and relational data in different specialized databases eliminating the need for dedicated synchronization applications. This is a huge gain in terms of performance and data maintenance.

Here is how this similarity search is run by the optimizer:

```
EXPLAIN PLAN FOR
 SELECT name, VECTOR_DISTANCE( embedding, (SELECT embedding FROM galaxies
WHERE name = 'NGC1073'), COSINE )
 FROM galaxies
 WHERE name <> 'NGC1073'
 ORDER BY VECTOR_DISTANCE( embedding, (SELECT embedding FROM galaxies WHERE
name = 'NGC1073'), COSINE ) FETCH FIRST 3 ROWS ONLY;



select plan_table_output from
table(dbms_xplan.display('plan_table',null,'all'));



PLAN_TABLE_OUTPUT
_____________________________________________________________________________
```

```
__________
Plan hash value: 3226012187


-----------------------------------------------------------------------
-------------
| Id  | Operation               | Name      | Rows  | Bytes | Cost
(%CPU)| Time     |
-----------------------------------------------------------------------
-------------
|   0 | SELECT STATEMENT        |           |     3 |    99 |     7
(15)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL      | GALAXIES |     1 |  4129 |     3
(0)| 00:00:01 |
|*  2 |  COUNT STOPKEY          |           |       |
|             |           |
|   3 |   VIEW                  |           |     8 |   264 |     7
(15)| 00:00:01 |
|*  4 |    TABLE ACCESS FULL    | GALAXIES |     1 |  4129 |     3
(0)| 00:00:01 |
|*  5 |    SORT ORDER BY STOPKEY|           |     8 | 33032 |     7
(15)| 00:00:01 |
|*  6 |     TABLE ACCESS FULL   | GALAXIES |     8 | 33032 |     3
(0)| 00:00:01 |
-----------------------------------------------------------------------
-------------



PLAN_TABLE_OUTPUT
________________________________________________________________
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------


   1 - SEL$2 / "GALAXIES"@"SEL$2"
   2 - SEL$4
   3 - SEL$1 / "from$_subquery$_004"@"SEL$4"
   4 - SEL$3 / "GALAXIES"@"SEL$3"
   5 - SEL$1
   6 - SEL$1 / "GALAXIES"@"SEL$1"


Predicate Information (identified by operation id):
---------------------------------------------------


   1 - filter("NAME"='NGC1073')


PLAN_TABLE_OUTPUT
_______________________________________________________________________
_______________
   2 - filter(ROWNUM<=3)
   4 - filter("NAME"='NGC1073')
   5 - filter(ROWNUM<=3)
   6 - filter("NAME"<>'NGC1073')


Column Projection Information (identified by operation id):
-----------------------------------------------------------



   1 - "EMBEDDING" /*+ LOB_BY_VALUE */ [VECTOR,8200]
   2 - "from$_subquery$_004"."NAME"[VARCHAR2,50],

"from$_subquery$_004"."EMBEDDING<=>(SELECTEMBEDDINGFROMGALAXIESWHERENAME='NG
       C1073')"[BINARY_DOUBLE,8]
   3 - "from$_subquery$_004"."NAME"[VARCHAR2,50],

"from$_subquery$_004"."EMBEDDING<=>(SELECTEMBEDDINGFROMGALAXIESWHERENAME='NG


PLAN_TABLE_OUTPUT
_____________________________________________________________________________
__
       C1073')"[BINARY_DOUBLE,8]
   4 - "EMBEDDING" /*+ LOB_BY_VALUE */ [VECTOR,8200]
   5 - (#keys=1) VECTOR_DISTANCE("EMBEDDING" /*+ LOB_BY_VALUE */ ,  (SELECT
       "EMBEDDING" /*+ LOB_BY_VALUE */  FROM "GALAXIES" "GALAXIES" WHERE
       "NAME"='NGC1073'), COSINE)[8], "NAME"[VARCHAR2,50], "EMBEDDING" /*+
       LOB_BY_VALUE */ [VECTOR,8200]
   6 - "NAME"[VARCHAR2,50], "EMBEDDING" /*+ LOB_BY_VALUE */ [VECTOR,8200]


Note
-----
   - dynamic statistics used: dynamic sampling (level=2)


53 rows selected.


SQL>
```

5

Supported Clients for Oracle AI Vector Search

The first Limited Availability release of Oracle AI Vector Search supports the following clients:

- The latest python-oracledb and node-oracledb SQL drivers support binding with native vector types.
- The existing JDBC, ODP.NET and OCI SQL drivers support binding with CLOB types.

The second Limited Availability release of Oracle AI Vector Search plans to support the following clients:

- The latest python-oracledb and node-oracledb SQL drivers support binding with native vector types.
- The latest JDBC, ODP.NET and OCI SQL drivers will support binding with native vector types.
- PL/SQL will support defining vector types.

Oracle Database 23c release 23.4 will be the general availability release of Oracle AI Vector Search.

- Oracle Database 23c release 23.4 will support binding with native vector types for all Oracle clients.

For applications that use pre-Oracle Client 23c libraries connected to Oracle Database 23c, release 23.4:

- use the `TO_VECTOR()` SQL function to insert vector data, as shown in this example:
  `INSERT INTO vecTab VALUES(TO_VECTOR('[1.1, 2.9, 3.14]'));`
- use the `FROM_VECTOR()` SQL function to fetch vector data, as shown in this example:
  `SELECT FROM_VECTOR(dataVec) FROM vecTab;`

6
Search Code Examples

Use these code examples to start exploring Oracle AI Vector Search.

SQL Code Examples

Note:

Code examples provided here are for learning purposes only. Do not use these code examples on production databases.

Create and Describe a Table Containing Vectors

Use the following SQL code examples to create a table with vector columns, insert vectors, and describe vectors.

Note:

A table can have multiple vector columns like in this example. The number of vector columns you have per table is a data modeling choice.

```
SQL> create table my_vect_tab (
  2    v1 vector(3, float32),
  3    v2 vector(2, float64),
  4    v3 vector(1, int8),
  5    v4 vector(1, *),
  6    v5 vector(*, float32),
  7    v6 vector(*, *),
  8    v7 vector
  9  );

Table created.

SQL>
SQL> desc my_vect_tab;
 Name                                      Null?    Type
 ----------------------------------------- --------
----------------------------
 V1                                                 VECTOR(3 , FLOAT32)
 V2                                                 VECTOR(2 , FLOAT64)
 V3                                                 VECTOR(1 , INT8)
 V4                                                 VECTOR(1 , *)
 V5                                                 VECTOR(* , FLOAT32)
 V6                                                 VECTOR(* , *)
 V7                                                 VECTOR(* , *)

SQL>

SQL> CREATE TABLE vect_tab (id number, e VECTOR(2));

Table VECT_TAB created.

SQL> SELECT securefile, value_based, cache, max_inline, column_name,
table_name
  2  FROM user_lobs
  3* WHERE table_name = 'VECT_TAB' ;

SECUREFILE    VALUE_BASED    CACHE       MAX_INLINE COLUMN_NAME
TABLE_NAME
_____________ ______________ ________ _____________ ______________
_____________
YES           YES            YES               8000 E
VECT_TAB

SQL> SELECT column_name, data_type, data_length, char_length,
data_precision, data_scale
  2  FROM user_tab_columns
  3* WHERE table_name = 'VECT_TAB' and column_name = 'E';

COLUMN_NAME    DATA_TYPE       DATA_LENGTH    CHAR_LENGTH
DATA_PRECISION    DATA_SCALE
______________ ____________ ______________ ______________
_________________ _____________
E              VECTOR                 8200              2

SQL>
```

Select Vectors and Use Vector Functions : Examples of Valid and Invalid Syntax

Use the following SQL code examples to select vectors and use vector functions.

```
SQL> SELECT * FROM vect_tab ORDER BY id;

   ID E
_____ ______________________
    1 [1.0E+000,2.0E+000]
    2 [2.0E+000,3.0E+000]


SQL> SELECT VECTOR_NORM(e) FROM vect_tab ORDER BY id;

VECTOR_NORM(E)
_________________
2.236068
3.6055512

SQL> SELECT distinct e FROM vect_tab;

Error starting at line : 1 in command -
```

```
select distinct e from vect_tab
Error at Command Line : 1 Column : 17
Error report -
SQL Error: ORA-22848: cannot use VECTOR type as comparison key

Note: This is expected behavior : vectors cannot be used as comparison keys.
Use supported vector functions only.

SQL> SELECT e FROM vect_tab UNION ALL SELECT e FROM vect_tab;

E
______________________
[1.0E+000,2.0E+000]
[2.0E+000,3.0E+000]
[1.0E+000,2.0E+000]
[2.0E+000,3.0E+000]

SQL> SELECT id FROM vect_tab WHERE VECTOR_DISTANCE(e, VECTOR('[34.6, -3.4]',
2, FLOAT32)) > 2 ORDER BY id;

   ID
_____
    1
    2

SQL> SELECT id FROM vect_tab ORDER BY VECTOR_DISTANCE(e, VECTOR('[34.6,
-3.4]', 2, FLOAT32));

   ID
_____
    2
    1
```

Python Code Examples

These Python code examples show several steps, from creating a table to generating Cohere and OpenAI embeddings and inserting them into an Oracle Database. For more information refer to the Using Oracle Database Vectors in python-oracledb - Limited Availability Release technical brief.

Note:

Code examples provided here are for learning purposes only. Do not use these code examples on production databases.

Create a Table and Insert and Select Vectors

Use this Python code example to create a table containing vectors, insert, and select vectors.

```
import oracledb

connection = oracledb.connect(
    user="username",
    password="password",
    dsn="cdb1_pdb1",
    config_dir="drectory_path"
    )

print("Successfully connected to Oracle Database")

cursor = connection.cursor()

cursor.execute("""
    begin
        execute immediate 'drop table t1';
        exception when others then if sqlcode <> -942 then raise; end
if;
    end;""")

cursor.execute("""
    create table t1 (
        id number,
        v vector(3, float32),
        primary key (id))""")

id_val = 4
vector_val = [15.625, 21.25, 5.9]

cursor.setinputsizes(None, oracledb.DB_TYPE_VECTOR)
cursor.execute("insert into t1 values (:1, :2)", [id_val, vector_val])

connection.commit()

cursor.execute('select * from t1')
for row in cursor:
     print(row)
```

Run the Python Script

This Python code shows you how to run your Python script.

```
$ python vec.py

Successfully connected to Oracle Database
(4, [15.625, 21.25, 5.9])
```

Generate OpenAI Embeddings and Insert Them into Oracle Database

Use this Python code to generate OpenAI embeddings and insert them into Oracle Database.

```
import oracledb
import os
import openai
import numpy
```

```
openai.api_key = os.getenv("MY_OPENAI_API_KEY")

response = openai.Embedding.create(
  model="text-embedding-ada-002",
  input="The food was delicious and the waiter..."
)

embeddings = response['data'][0]['embedding']
print(len(embeddings))

connection = oracledb.connect(
    user="username",
    password="password",
    dsn="cdb1_pdb1",
    config_dir="path"
    )

cursor = connection.cursor()
cursor.execute("""
    begin
        execute immediate 'drop table t1';
        exception when others then if sqlcode <> -942 then raise; end if;
    end;""")

cursor.execute("""
    create table t1 (
        id number,
        v vector(1536, float64),
        primary key (id))""")

id_val = 1

cursor.setinputsizes(None, oracledb.DB_TYPE_VECTOR)
cursor.execute("insert into t1 values (:1, :2)", [id_val, embeddings])
connection.commit()

cursor.execute('select * from t1')
for row in cursor:
     print(row)

print("Bye bye ")
```

Generate Cohere Embeddings and Insert Them into Oracle Database

Use this Python code to generate Cohere embeddings and insert them into Oracle Database.

```
import oracledb
import cohere
import os

co = cohere.Client(os.getenv("CO_API_KEY"))

response = co.embed(
  texts=['hello'],
  model='small',
)

vec = response.embeddings[0]
print(len(vec))

connection = oracledb.connect(
    user="username",
    password="password",
    dsn="cdb1_pdb1",
    config_dir="path"
    )

cursor = connection.cursor()
cursor.execute("""
    begin
        execute immediate 'drop table t1';
        exception when others then if sqlcode <> -942 then raise; end
if;
    end;""")

cursor.execute("""
    create table t1 (
        id number,
        v vector(1024, float64),
        primary key (id))""")

id_val = 1
cursor.setinputsizes(None, oracledb.DB_TYPE_VECTOR)
cursor.execute("insert into t1 values (:1, :2)", [id_val, vec])
connection.commit()

cursor.execute('select * from t1')
for row in cursor:
     print(row)

print("Bye bye ")
```

Generate Hugging Face Embeddings and Insert Them into Oracle Database

Use this Python code to generate Hugging Face embeddings and insert them into Oracle Database.

```
import oracledb
from sentence_transformers import SentenceTransformer

sentences = ["This is an example sentence"]

print("Hugging Face Sentence Transformers")
print("Using all-MiniLM-L6-v2 with 384 dimensions")

model = SentenceTransformer('all-MiniLM-L6-v2')
embeddings = list(model.encode(sentences)[0])
print(len(embeddings))

connection = oracledb.connect(
    user="username",
    password="password",
    dsn="cdb1_pdb1",
    config_dir="path"
    )

cursor = connection.cursor()
cursor.execute("""
    begin
        execute immediate 'drop table t1';
        exception when others then if sqlcode <> -942 then raise; end if;
    end;""")

cursor.execute("""
    create table t1 (
        id number,
        v vector(384, float64),
        primary key (id))""")

id_val = 1
cursor.setinputsizes(None, oracledb.DB_TYPE_VECTOR)
cursor.execute("insert into t1 values (:1, :2)", [id_val, embeddings])
connection.commit()

cursor.execute('select * from t1')
for row in cursor:
     print(row)

print("Bye bye")
```

Node.js Code Examples

These Node.js code examples show several steps, from creating a table containing vectors, inserting vectors, to selecting them from the Oracle Database. For more information, refer to the *Using Oracle Database Vectors in node-oracledb Limited Availability Release* technical brief.

Note:

Code examples provided here are for learning purposes only. Do not use these code examples on production databases.

Create Table, Insert Vectors

Use these Node.js code examples to create and insert vectors in the Oracle Database.

```
'use strict';

Error.stackTraceLimit = 50;

const oracledb = require('oracledb');
const config = {
user: "username",
password: "password",
connectString: "(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=ip_address)
(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=cdb1_pdb1)))"
};

const tableName = "t1";
const dropSql = `drop table ${tableName}`;
const createSql = `create table ${tableName} (id number, v vector(5,
float32))`;
const insertSql = `insert into ${tableName} values (:1, :2)`;

const vectorData = [3.25, 4.78, 2.625, 1.5, 9.14];

const binds = [
  1,
  new oracledb.Vector(vectorData)
];

async function run() {
  const conn = await oracledb.getConnection(config);
  await conn.execute(dropSql);
  await conn.execute(createSql);
  await conn.execute(insertSql, binds);
  console.log("all good!");
  await conn.commit();
  await conn.close();
}

run();
```

Select Vectors

Use this Node.js code example to select vectors from the Oracle Database.

```
'use strict';

Error.stackTraceLimit = 50;

const oracledb = require('oracledb');

const config = {
  user: "username",
  password: "password",
  connectString: "(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=ip_address)
(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=cdb1_pdb1)))"
};

const selectSql = `select * from t1`;

async function run() {
  const conn = await oracledb.getConnection(config);
  const results = await conn.execute(selectSql);
  console.log("results:", results);
  console.log("array:", results.rows[0][1]);
  await conn.close();
}

run();
```

7

Frequently Asked Questions

This section is dedicated to answering general questions that you may have about Oracle AI Vector Search.

Question 1

What is the maximum supported number of dimensions for the `VECTOR` datatype?

Answer: Oracle AI Vector Search `VECTOR` datatype supports a maximum of 65533 dimensions.

Question 2

Is there a difference between the following two declarations : `VECTOR` and `VECTOR(*,*)`?

Answer: No, there is no difference between the two declarations. Both allow you to store vectors of different dimensions with different formats in the same column.

Question 3

Which vector embedding model does Oracle AI Vector Search support?

Answer: Oracle AI Vector Search supports all Vector Embedding models that support `FLOAT32`, `FLOAT64`, or `INT8` formats. However, these models are not stored within the database. You can access these models from various service providers to generate your vector embeddings and store them in the Oracle Database.

Question 4

Is it possible to have some vector dimensions set to `NULL`?

Answer: No, Oracle AI Vector Search does not allow you to set a vector dimension to `NULL`; for example, you can't have `[1.1, NULL, 2.2]`. However, a vector can be set to `NULL`; for example, `SELECT to_vector(NULL);` is supported.

Question 5

How are vectors stored inside the Oracle Database?

Answer: Vectors are internally stored as Securefile BLOBs.

Question 6

What is Oracle AI Vector Search?

Answer: Oracle AI Vector Search stores and indexes vector embeddings for fast retrieval and similarity search.

Question 7

In which release Oracle AI vector Search was introduced?

Answer: Oracle AI Vector Search is introduced in Oracle Database 23c (23.4.0.10.23).

Question 8

What are the Oracle Database client tools that support Oracle AI vector Search?

Answer: SQL*Plus, SQLcl, SQL Developer, SQL Developer Web and Oracle Developer Tools for VS Code enable you to insert, update, delete and select vector values in Oracle 23.4.

Question 9

What are the Oracle Database drivers that support Oracle AI vector Search?

Answer: Python-oracledb and Node-oracledb enable biding with native vector types. JDBC, ODP.NET and OCI enable binding with CLOBs.

Question 10

What is a typical Oracle AI Vector Search workflow?

Answer: A typical Oracle AI Vector Search workflow contains the following three main parts:

1. You need to generate vector embeddings from your unstructured data. This step is done outside the Oracle Database for Limited Availability 1 release.
2. You store the resulting vector embeddings and associated data with your relational business data in the Oracle Database.
3. You can use Oracle AI Vector Search native SQL operations to combine similarity with relational searches.

Question 11

What are the main advantages of using Oracle AI Vector Search compared to other Vector Databases?

Answer: The biggest benefit of Oracle AI Vector Search is that semantic search on unstructured data can be combined with relational search on business data in one single system. Additionally, because Oracle has been building database technologies for so long, your vectors can benefit from all of Oracle Database's most powerful features, such as partitioning, RAC, Exadata smart scans, transactions, and security.

Question 12

How do you impose `CHECK` constraints when defining a `VECTOR` column?

Answer: When defined, the number of dimensions and the number formats effectively become `CHECK` constraints. If you do not define the number of dimensions or the number format, the number format is considered to be `FLOAT32` and is thus a `CHECK` constraint. The number of dimensions is a strict check constraint and throws an error if not respected. However, the number format is used to up-convert or down-convert the entered data.