ORACLE

Oracle AI Database 26ai vectors in node-oracledb

Driving Generative AI through vector support in Node.js applications running Oracle AI Database 26ai and beyond

November 2025, Version 4.0

Public

Disclaimer

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle software license and service agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without the prior written consent of Oracle. This document is not part of your license agreement, nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This document is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

Table of Contents

Introduction

Oracle AI Database 26ai provides a VECTOR data type for advanced AI/ML[1] search operations as part of its Oracle AI Vector Search feature set. Vectors are commonly used in AI to represent the semantics of unstructured data such as images, documents, video, and audio. They are generated using vector embedding models. This data type is a homogeneous array of 8-bit unsigned integers, 8-bit signed integers, 32-bit floating-point numbers, or 64-bit floating-point numbers. The storage format of a vector can be specified as *Dense* (default) or *Sparse*. See the introductory blog from Oracle's AI Vector Search team on the comprehensive list of benefits and use cases for Oracle AI Database 26ai vector support.

The node-oracledb add-on for Node.js is a database driver module for high-performance Oracle Database applications written in JavaScript or TypeScript. You can quickly write complex applications or build sophisticated web services that expose REST or GraphQL endpoints. Check the node-oracledb documentation for complete details on the driver.

Vector data in Oracle Database

VECTOR columns in Oracle AI Database 26ai can be created as type:

```
VECTOR(<vectorDimensions>, <vectorFormat>, [SPARSE])
```

where the attributes are:

- *vectorDimensions*: defines the number of dimensions for the vector data. For example, a point in 3D space is defined by vector data of 3 dimensions, i.e., the (*x,y,z*) coordinates. For the BINARY vector format, the number of dimensions should be a multiple of 8.
- *vectorFormat*: one of the keywords BINARY, INT8, FLOAT32, or FLOAT64 to define the storage format[2] of each dimension value in the vector. The INT8, FLOAT32, or FLOAT64 formats are supported from Oracle Database 23.4 onwards. The BINARY format is supported starting from Oracle Database 23.5 onwards.
- *SPARSE*: optional keyword to identify if the vector is a SPARSE VECTOR column. It is available from Oracle Database 23.7 onwards.

For example:

To create and insert data into a table with a single default (dense) FLOAT32 vector column:

```
CREATE TABLE vecTab(dataVec VECTOR(3, FLOAT32));
```

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

For more information about using vectors, refer to the Oracle Documentation:

Oracle Database AI Vector Search User Guide

Sparse vector support in Oracle AI Database 26ai

Sparse vectors provide an efficient way to represent VECTOR data where most dimensions have zero values. They are characterized by the total number of dimensions (including zero and non-zero values), indices of non-zero values, and the non-zero values at the specified indices. This compact representation ensures that only essential data is stored. The indices are zero-based, meaning they start at zero.

[1] Artificial Intelligence / Machine Learning

[2] BINARY – 8-bit unsigned integer, INT8 – 8 bit signed integer, FLOAT32 – 32-bit floating point number, FLOAT64 – 64-bit floating point number

For example:

```
CREATE TABLE vecSparse (float32sparseCol VECTOR(25, FLOAT64, SPARSE));
```

An example of a sparse vector represented in string form, with 25 dimensions, non-zero values at indices 5, 8, and 11, and non-zero values 25.25, 6.125, and 8.25, corresponding to their respective indices is:

```
[25, [5, 8, 11], [25.25, 6.125, 8.25]]
```

Installing the node-oracledb driver

Please make sure that Node.js (version 14.17 or later) and npm are installed on your machine and that you have the connection details to an Oracle AI Database 26ai (or later) release that supports the VECTOR data type.

The node-oracledb driver with vector support is available on npm or GitHub.

To install the driver, use the npm module. Run the following in a command line terminal:

```
npm install oracledb
```

This command installs the '*oracledb*' Node.js package.

For more details on installing the driver, refer to the node-oracledb installation manual.

Vector support in node-oracledb

The node-oracledb driver provides direct access to Oracle Database through its default Thin mode, which is implemented purely in JavaScript. An optional Thick mode can also be enabled at runtime in node-oracledb, which uses Oracle Client libraries to connect to the Oracle Database.

The node-oracledb 6.5 release introduced support for binding and fetching the VECTOR data type. Further enhancements, such as Binary Vector Support, were made in the 6.6 release. In version 6.8, Sparse Vector support was introduced. Vectors are represented as Node.js *TypedArray* objects or JavaScript Arrays in both the Thin and Thick modes of node-oracledb.

Vectors can be fetched and inserted using standard node-oracledb APIs. Vector data will be returned or fetched as *TypedArrays* of unsigned integer (8-bit), signed integer (8-bit), float (32-bit), or double (64-bit) values depending on whether the VECTOR column in Oracle Database has a BINARY, INT8, FLOAT32, or FLOAT64 format.

The code snippets in this section use the *vecTab* table created in an earlier section.

The code below returns the data type and value of the vector array in the *dataVec* column of the *vecTab* table.

```
const result = await connection.execute('select dataVec from vecTab');

const vec = result.rows[0].dataVec;

console.log('Returned Array Type:', vec.constructor);

console.log('Returned Array:', vec);
```

This code snippet will give the output:

```
Returned Array type: [Function: Float32Array]
Returned Array: Float32Array(3) [
  1.100000023841858,
  2.190000057220459,
  3.140000104904175
]
```

This output indicates that a *TypedArray* of 32-bit floating point numbers is being returned since the *dataVec* column is a `FLOAT32` VECTOR column.

A new node-oracledb constant, *oracledb.DB_TYPE_VECTOR* has been created for vectors. This type will be returned as an attribute in the metadata returned for queries and can be used as a type in bind information supplied by the developer.

A fetchTypeHandler function can be used to convert the vector data to a JavaScript object if required. For example, the following code snippet converts a *TypedArray* object to a JavaScript array:

```
oracledb.fetchTypeHandler = function(metadata) {
  if (metadata.dbType === oracledb.DB_TYPE_VECTOR) {
    const myConverter = (v) => {
      if (v !== null) {
        return Array.from(v);
      }
      return v;
    };
    return {converter: myConverter};
  }
};

const result = await connection.execute('select dataVec from vecTab');
const vec = result.rows[0].dataVec;
console.log('Returned Array Type:', vec.constructor);
console.log('Returned Array:', vec);
```

Running this code gives the output:

```
Returned Array type: [Function: Array]
Returned Array: [ 1.100000023841858, 2.190000057220459, 3.140000104904175 ]
```

The attributes *vectorDimensions* and *vectorFormat* have also been added to the metadata returned for queries.

- The *vectorDimensions* attribute returns the number of dimensions of the VECTOR column. This attribute will contain the value 'undefined' for non-VECTOR columns. It will also be *undefined* for VECTOR columns where the number of dimensions is flexible.
- The *vectorFormat* attribute defines the storage format of each dimension value in the VECTOR column. The storage format will be one of the following node-oracledb global constants – `VECTOR_FORMAT_BINARY`, `VECTOR_FORMAT_INT8`, `VECTOR_FORMAT_FLOAT32`, and `VECTOR_FORMAT_FLOAT64`. This attribute will contain the value *undefined* for non-VECTOR columns and also for VECTOR columns whose storage format is flexible.

In node-oracledb 6.8, a new attribute, *isSparseVector*, is part of the metadata returned from queries to identify the type of data stored in a column. It returns `True` if the column contains sparse vectors, `False` if the column contains dense vectors, and `undefined` if the column does not contain vector data.

Continuing with the *vecTab* table example, to fetch the *vectorDimensions* and *vectorFormat* attributes:

```
const vecDimensions = result.metadata[0].vectorDimensions;

const vecStorageFormat = result.metadata[0].vectorFormat;

let vecStorageFormatString;

if (vecStorageFormat == oracledb.VECTOR_FORMAT_FLOAT32)
  vecStorageFormatString = 'FLOAT32';
else if (vecStorageFormat == oracledb.VECTOR_FORMAT_FLOAT64)
  vecStorageFormatString = 'FLOAT64';
else if (vecStorageFormat == oracledb.VECTOR_FORMAT_INT8)
  vecStorageFormatString = 'INT8';
else if (vecStorageFormat == oracledb.VECTOR_FORMAT_BINARY)
  vecStorageFormatString = 'BINARY';
else
  vecStorageFormatString = 'UNKNOWN TYPE';

console.log('Vector dimensions for the dataVec column:', vecDimensions);

console.log('Vector storage format for the dataVec column:',
vecStorageFormatString);
```

This will give the output:

```
Vector dimensions for the dataVec column: 3
Vector storage format for the dataVec column: FLOAT32
```

The above output indicates that the *dataVec* column in the *vecTab* table is a 3-dimensional FLOAT32 vector.

All *TypedArray* formats (*UInt8Array*, *Int8Array*, *Float32Array* and *Float64Array*) and JavaScript arrays of numbers will be accepted as input for vector data. To pass these arrays as inputs to flexible[3] VECTOR columns as bind values, pass in `oracledb.DB_TYPE_VECTOR` as a bind type attribute. For VECTOR columns with a defined vector storage format, pass the array directly as the bind value.

These semantics are shown in the examples in the section 'Sample Node.js applications using vectors'.

Sample Node.js programs using vectors

Dense vectors

The following Node.js app, *vectorDense.js*, works with VECTOR column data types in the default dense storage format of Oracle Database using node-oracledb.

```
// vectorDense.js sample code

const oracledb = require('oracledb');


const tableName = 'sampleVectorTab';
```

[3] Flexible VECTOR columns do not have their vector storage formats defined at the time of table creation

```
// To run the script in Thick mode, uncomment the following line:
// oracledb.initOracleClient()

// Add the DB user credentials and connect string
const dbConfig = {
  user: "myuser",
  password: "mypw",
  connectString: "db_connectstring"
};

oracledb.outFormat = oracledb.OUT_FORMAT_OBJECT;

// By default, TypedArrays are returned. A Fetch Type Handler like
// below is used to convert TypedArray to JavaScript Array objects.
// This is optional.
oracledb.fetchTypeHandler = function(metadata) {
  if (metadata.dbType === oracledb.DB_TYPE_VECTOR) {
    const myConverter = (v) => {
      if (v !== null) {
        return Array.from(v);
      }
      return v;
    };
    return {converter: myConverter};
  }
};

// Main function
async function run() {
  const connection = await oracledb.getConnection(dbConfig);
  try {
    let result;
    const serverVersion = connection.oracleServerVersion;
    if (serverVersion < 2305000000) {
      console.log('This DB version does not support all the VECTOR data types');
```

```
    return;
  }

  await connection.execute(`DROP TABLE IF EXISTS ${tableName}`);
  await connection.execute(`CREATE TABLE ${tableName} (
    ID NUMBER,
    VCOL VECTOR(3),
    VCOL32 VECTOR(3, FLOAT32),
    VCOL64 VECTOR(3, FLOAT64),
    VCOL8 VECTOR(3, INT8),
    VCOLB VECTOR(16, BINARY)
    )`
  );
  console.log('Table created');

  // JavaScript Array
  const arr = [1.1, 2.2, 3.3];
  // 32-bit floating point TypedArray
  const float32arr = new Float32Array([4.4, 5.51, 6.6]);
  // 64-bit floating point TypedArray
  const float64arr = new Float64Array([7.7, 8.8, 9.9]);
  // 8-bit signed integer TypedArray
  const int8arr = new Int8Array([126, 125, -23]);
  // 8-bit unsigned integer TypedArray for binary vector
  const binarr = new Uint8Array([240, 200]);


  result = await connection.execute(
    `INSERT INTO ${tableName}
     (ID, VCOL, VCOL32, VCOL64, VCOL8, VCOLB)
     VALUES (:id, :vec, :vec32, :vec64, :vec8, :vecbinary)`,
    { id: 1,
      vec: {type: oracledb.DB_TYPE_VECTOR, val: arr},
      vec32: float32arr,
      vec64: float64arr,
      vec8: int8arr,
```

```
        vecbinary: binarr
      });
    console.log('Rows inserted: ' + result.rowsAffected);

    result = await connection.execute(
      `SELECT ID, VCOL, VCOL32, VCOL64, VCOL8, VCOLB FROM ${tableName}`
    );

    console.log("Query output:");
    console.log(result.rows[0]);
  } catch (err) {
    console.error(err);
  } finally {
    if (connection) {
      try {
        await connection.close();
      } catch (err) {
        console.error(err);
      }
    }
  }
}
run();
```

The script creates a table '*sampleVectorTab*' with four VECTOR columns:

- *VCOL32* is a FLOAT32 format VECTOR column
- *VCOL64* is a FLOAT64 format VECTOR column
- *VCOL8* is an 8-bit signed integer (INT8) format VECTOR column
- *VCOLB* is an 8-bit unsigned integer (BINARY) format VECTOR column
- *VCOL* is a flexible VECTOR column

Then, data is inserted into the table. *TypedArrays* are used as bind values for inserting data into VECTOR columns with a specific *VectorFormat* attribute. To insert data into the VECTOR column with an unspecified *VectorFormat* attribute (*VCOL*), a JavaScript array is used as a bind value with the *type* property set to *DB_TYPE_VECTOR* in this example.

VECTOR columns are fetched as node-oracledb Array objects using the FetchTypeHandler global function.

The output is similar to:

```
$ node vectorDense.js
Table created
Rows inserted: 1
Query output:
{
  ID: 1,
  VCOL: [ 1.1, 2.2, 3.3 ]
  VCOL32: [ 4.400000095367432, 5.510000228881836, 6.599999904632568 ],
  VCOL64: [ 7.7, 8.8, 9.9 ],
  VCOL8: [ 126, 125, -23 ],
  VCOLB: [240, 200]
}
```

The minor discrepancies between the input (see the *arr* variable) and output values of the *Float32 TypedArray* are due to the side effects of floating-point operations in JavaScript.

Sparse vectors

The following Node.js app, *vectorSparse.js*, works with the sparse VECTOR column data type in Oracle Database using node-oracledb.

```
// vectorSparse.js sample code
const oracledb = require('oracledb');


const tableName = 'VectorSparseTab';


// To run the script in Thick mode, uncomment the following line:
// oracledb.initOracleClient()


// Add the DB user credentials and connect string
const dbConfig = {
    user: "myuser",
    password: "mypw",
    connectString: "db_connectstring"
  };


oracledb.outFormat = oracledb.OUT_FORMAT_OBJECT;
```

```
// Main function
async function run() {
  const connection = await oracledb.getConnection(dbConfig);
  try {
      let result;
      const serverVersion = connection.oracleServerVersion;
      if (serverVersion < 2307000000) {
        console.log('This DB version does not support SPARSE VECTOR data type');
      return;
      }

      await connection.execute(`CREATE TABLE ${tableName} (
        ID NUMBER GENERATED ALWAYS AS IDENTITY,
        FLOAT32SPARSECOL VECTOR(25, float32, sparse),
        FLOAT64SPARSECOL VECTOR(30, float64, sparse),
        INT8SPARSECOL VECTOR(35, int8, sparse)
        )`
      );
      console.log('Table created');

      // Sparse Vector Object with 32-bit floating point non-zero values
      const float32arr = new Float32Array([4.4, 5.51, 6.6]);
      const sparseVecObjFloat32 = new oracledb.SparseVector(
        { values: float32arr, indices: [1, 3, 7], numDimensions: 25 });
      // Sparse Vector Object with 64-bit floating point non-zero values
      const float64arr = new Float64Array([7.7, 8.8, 9.9]);
      const sparseVecObjFloat64 = new oracledb.SparseVector(
        { values: float64arr, indices: [1, 3, 6], numDimensions: 30 });
      // Sparse Vector Object with 8-bit signed integer non-zero values
      const int8arr = new Int8Array([126, -25]);
      const sparseVecObjInt8 = new oracledb.SparseVector(
        { values: int8arr, indices: [4, 7], numDimensions: 35 });

      result = await connection.execute(
      `INSERT INTO ${tableName}
        (FLOAT32SPARSECOL, FLOAT64SPARSECOL, INT8SPARSECOL)
```

```
      VALUES (:sparseVec32, :sparseVec64, :sparseVec8)`,
      { sparseVec32: sparseVecObjFloat32,
        sparseVec64: sparseVecObjFloat64,
        sparseVec8: sparseVecObjInt8,
      }
    );
    console.log('Rows inserted: ' + result.rowsAffected);

    result = await connection.execute(
    `SELECT ID, FLOAT32SPARSECOL, FLOAT64SPARSECOL, INT8SPARSECOL FROM
      ${tableName}`
    );

    console.log("Query output:");
    console.log("ID:", result.rows[0].ID);
    console.log("FLOAT32 SPARSE COL:",
      JSON.stringify(result.rows[0].FLOAT32SPARSECOL));
    console.log("FLOAT64 SPARSE COL:",
      JSON.stringify(result.rows[0].FLOAT64SPARSECOL));
    console.log("INT8 SPARSE COL:",
      JSON.stringify(result.rows[0].INT8SPARSECOL));

    for (data of result.metaData) {
      console.log(data.name, "is sparse:", data.isSparseVector);
    }
  } catch (err) {
    console.error(err);
  } finally {
    if (connection) {
      try {
        await connection.execute(`DROP TABLE IF EXISTS ${tableName}`);
        console.log("Table dropped");
        await connection.close();
      } catch (err) {
        console.error(err);
      }
```

```
    }
  }
}


run();
```

The script creates a table '*VectorSparseTab*' with three SPARSE VECTOR columns and an auto-generated ID NUMBER column:

- *FLOAT32SPARSECOL* is a FLOAT32 format SPARSE VECTOR column
- *FLOAT64SPARSECOL* is a FLOAT64 format SPARSE VECTOR column
- *INT8SPARSECOL* is an 8-bit signed integer (INT8) format SPARSE VECTOR column

Then, data is inserted into the table. Node-oracledb's *oracledb.SparseVector* objects of the required data type are used as bind values for inserting data into VECTOR columns with a specific *VectorFormat* (*Float64*, *Float32,* and *Int8* data types) attribute.

The SPARSE VECTOR columns are fetched as node-oracledb's *oracledb.SparseVector* objects.

In this example, the *isSparseVector* attribute of the returned result's metadata property on each column is used to check whether the column is a SPARSE VECTOR column.

Note that if the column is not a VECTOR column, the *isSparseVector* metadata attribute is *undefined*.

The output is similar to:

```
$ node vectorSparse.js

Table created

Rows inserted: 1

Query output:

ID: 1

FLOAT32 SPARSE COL:
{"numDimensions":25,"indices":{"0":1,"1":3,"2":7},"values":{"0":4.400000095367432
,"1":5.510000228881836,"2":6.599999904632568}}

FLOAT64 SPARSE COL:
{"numDimensions":30,"indices":{"0":1,"1":3,"2":6},"values":{"0":7.7,"1":8.8,"2":9
.9}}

INT8 SPARSE COL:
{"numDimensions":35,"indices":{"0":4,"1":7},"values":{"0":126,"1":-25}}

ID is sparse: undefined

FLOAT32SPARSECOL is sparse: true

FLOAT64SPARSECOL is sparse: true

INT8SPARSECOL is sparse: true

Table dropped
```

Using embedding models with vectors for intuitive applications

The vector support in node-oracledb enables Node.js developers to use a variety of embedding models from AI frameworks such as Cohere, OpenAI, and HuggingFace. These embedding models can be used to generate vector data that can be stored in the Oracle Database. The vectors can empower Node.js applications with similarity search and natural language processing capabilities.

Using a random seed dataset, the sample application below implements a similarity search for any text-based input from the user. This application uses embedding models from the Cohere AI framework and has three component JavaScript files that:

- Create the source dataset (*createSchema.js*)
- Generate and embed vectors into the database based on the source dataset (*vectorizeTableCohere.js*)
- Implement similarity search with reranking based on the embedded vectors (*similaritySearchCohere.js*)

Once the source data is created and vectors are embedded, the application user can run a similarity search on the source dataset with any phrase or sentence and get the top N closely matching or similar sentences from the source dataset based on the vector comparison. We also use a reranking model on top of the embedding model to improve accuracy.

Create source dataset

The following is a sample file (*createSchema.js*), which can be used to create a source dataset:

https://gist.github.com/sharadraju/108275cc79f111ad94b6830948d1fa10

The file in the link above will create the source dataset (*my_data* table) with a VECTOR column initialized to null values. This VECTOR column will be updated when we embed vector data using the Cohere embedding models. You can modify the sample file to add more rows and improve the source dataset.

When the *createSchema.js* file is run with the node command, a successful output is similar to the following:

```
$ node createSchema.js

Connected to Oracle Database

Created table and inserted data

Thin mode selected

Run at: Mon Sep 09 2024 21:40:25 GMT+0530 (India Standard Time)

Oracle Database version: 23.5.0.24.5
```

Generate and Embed vectors into Oracle Database

Now that the source dataset is ready, the next step is to generate and embed vectors into the source dataset using Cohere in this case. First, the user must create an account at http://www.cohere.com and generate the Cohere API key.

The environment variable CO_API_KEY must be set to the Cohere API key.

The *cohere-ai* npm module must be installed:

```
$ npm install cohere-ai
```

Note: If the application user is running behind a firewall or a corporate HTTP/HTTPS proxy, a relevant npm module to connect to Cohere through the proxy can be downloaded and used in the *vectorizeCohere.js* file for running the embedding models and similarity searches.

The following program embeds vectors into the source dataset (*my_data table* in this case):

```
// vectorizeCohere.js file
const oracledb = require('oracledb');
const cohere = require('cohere-ai');

async function vectorize() {
  let connection;

  // Add the DB user credentials
  // and connect string
  const dbConfig = {
    user: "myuser",
    password: "mypw",
    connectString: "db_connectstring"
  };
  // To run the script in Thick mode, uncomment the following line:
  // oracledb.initOracleClient();

  // Get your Cohere API Key from the environment
  const apiKey = process.env.CO_API_KEY;

  // Select/Set your Embedding model below
  // const embeddingModel = 'embed-english-light-v3.0';
  // const embeddingModel = 'embed-english-v3.0';
  // const embeddingModel = 'embed-multilingual-light-v3.0';
  const embeddingModel = 'embed-multilingual-v3.0';

  console.log('Using embedding model ' + embeddingModel);

  const co = new cohere.CohereClient({ token: apiKey });

  try {
```

```
// Get a standalone Oracle Database connection
connection = await oracledb.getConnection(dbConfig);

//Connect only to an Oracle Database version that supports vectors
if (connection.oracleServerVersion < 2304000000) {
  console.log('This example requires Oracle Database 23.4 or later');
  process.exit();
}
console.log('Connected to Oracle Database');

console.log('Vectorizing the following data:');

// Loop over the rows and vectorize the VARCHAR2 data
const sql = 'SELECT id, info FROM my_data ORDER BY 1';
const result = await connection.execute(sql);

const binds = [];

for (const row of result.rows) {
  // Convert to a format that Cohere wants
  const data = [row[1]];
  console.log(row);

  // Create the vector embedding [a JSON object]
  const response = await co.embed({
    texts: data,
    model: embeddingModel,
    inputType: 'search_query',
  });

  // Extract the vector from the JSON object & convert it to TypedArray
  const float32VecArray = new Float32Array(response.embeddings[0]);

  // Record the array and key
  binds.push([float32VecArray, row[0]]);
}
```

```
    // Do an update to add or replace the vector values
    await connection.executeMany('UPDATE my_data SET v = :1 WHERE id = :2',
      binds,
      { autoCommit: true });

    console.log(`Added ${binds.length} vectors to the table`);
  } catch (err) {
    console.error(err);
  } finally {
    if (connection)
      await connection.close();
  }
}

vectorize();
```

This code snippet uses the *'embed-multilingual-v3.0'* embedding model of Cohere here. Developers can also use other Cohere embedding models depending on their preference.

Running this file will embed the vectors in the VECTOR column of the *my_data* table.

When the *vectorizeCohere.js* file is run with the node command, a successful output is similar to the following:

```
$ node vectorizeTableCohere.js
Using embedding model embed-multilingual-v3.0
Connected to Oracle Database
Vectorizing the following data:
[ 1, 'San Francisco is in California.' ]
[ 2, 'San Jose is in California.' ]
[ 3, 'Los Angeles is in California.' ]
[ 4, 'Buffalo is in New York.' ]
[ 5, 'Brooklyn is in New York.' ]
...
[ 100, 'Ferraris are often red.' ]
[ 101, 'Teslas are electric.' ]
[ 102, 'Mini coopers are small.' ]
```

```
[ 103, 'Fiat 500s are small.' ]
[ 104, 'Dodge Vipers are wide.' ]
...
[ 1100, 'Mumbai is in India.' ]
[
  1101,
  'Mumbai is the capital city of the Indian state of Maharashtra.'
]
[ 1102, 'Mumbai is the Indian state of Maharashtra.' ]
[ 1103, 'Mumbai is on the west coast of India.' ]
[ 1104, 'Mumbai is the de facto financial centre of India.' ]
[ 1105, 'Mumbai has a population of about 12.5 million people.' ]
[
  1106,
  'Mumbai is hot with an average minimum temperature of 24 degrees Celsius.'
]
[
  1107,
 'Common languages in Mumbai are Marathi, Hindi, Gujarati, Urdu, Bambaiya and
English.'
]
Added 134 vectors to the table
```

Note: A compressed version of the output is shown above, as the original output can span multiple lines depending on the amount of data in the *my_data* table.

This file will have updated the VECTOR columns in the source dataset (*my_data* table).

Run Similarity Search with user inputs

Finally, we run the *similaritySearchCohere.js* file to enable the application users to search for similar information to their questions or inputs in the source dataset (*my_data* table). We also use a reranking model to improve the accuracy of the similarity search results.

```
// similaritySearchCohere.js file
const oracledb = require('oracledb');
const cohere = require('cohere-ai');
const readline = require('readline');

const readLineAsync = () => {
  const rl = readline.createInterface({
```

```
    input: process.stdin
  });

  return new Promise((resolve) => {
    rl.prompt();
    rl.on('line', (line) => {
      rl.close();
      resolve(line);
    });
  });
};

async function runSimilaritySearch() {
  let connection;

  // Add the DB user credentials
  // and connect string
  const dbConfig = {
    user: "myuser",
    password: "mypw",
    connectString: "db_connectstring"
  };

  const topK = 5; // Return the top 5 similar results
  let reRank = true;

  // Get your Cohere API Key from the environment
  const apiKey = process.env.CO_API_KEY;

  // Select/Set your Embedding model here
  // const embeddingModel = 'embed-english-light-v3.0';
  // const embeddingModel = 'embed-english-v3.0';
  // const embeddingModel = 'embed-multilingual-light-v3.0';
  const embeddingModel = 'embed-multilingual-v3.0';

  // Cohere re-ranking models
```

```
// const rerankModel = 'rerank-english-v2.0';
const rerankModel = 'rerank-multilingual-v2.0';

console.log('Using embedding model ' + embeddingModel);

if (reRank)
  console.log('Using reranker ' + rerankModel);
else
  console.log('Not using reranking');

console.log('TopK = ' + topK);

const co = new cohere.CohereClient({ token: apiKey });

try {
  // To run the script in Thick mode, uncomment the following line:
  // oracledb.initOracleClient();

  // Get a standalone Oracle Database connection
  connection = await oracledb.getConnection(dbConfig);

  // Connect only to an Oracle Database version that supports vectors
  if (connection.oracleServerVersion < 2304000000) {
    console.log('This example requires Oracle Database 23.4 or later');
    process.exit();
  }
  console.log('Connected to Oracle Database');


  // Using the EUCLIDEAN Vector Distance function
  const sql = `SELECT info FROM my_data
               ORDER BY VECTOR_DISTANCE(v, :1, EUCLIDEAN)
               FETCH FIRST :2 ROWS ONLY`;

  while (true) {
    // Get the text input to vectorize
```

```
    console.log("\nEnter a phrase. Type 'quit' or 'exit' to exit: ");
    const text = await readLineAsync();

    if (text === 'quit' || text === 'exit')
      break;
    if (text === '')
      continue;

    // Create the vector embedding [a JSON object]
    const sentence = [text];
    const response = await co.embed({
      texts: sentence,
      model: embeddingModel,
      inputType: 'search_query',
    });

    // Extract the vector from the JSON object
    const float64VecArray = new Float64Array(response.embeddings[0]);

    const docs = [];

    // Do the Similarity Search
    const rows = (await connection.execute(sql, [float64VecArray, topK])).rows;
    for (const row of rows) {
      docs.push(row[0]);
    }
```

```
    if (!reRank) {
      // Rely on the vector distance for the resultset order
      console.log('\nWithout ReRanking');
      console.log('================');

      for (const hit of docs) {
        console.log(hit);
```

```
        }
      } else {

        // Rerank for better results
        const { results } = await co.rerank({ query: text, documents: docs, topN:
topK, model: rerankModel });

        console.log('\nReranked results');
        console.log('================');

        for (const hit of results) {
          console.log(docs[hit.index]);
        }
      }
    } // End of while loop
  } catch (err) {
    console.error(err);
  } finally {
    if (connection)
      await connection.close();
  }
}

runSimilaritySearch();
```

Note: If the application user is running behind a firewall or a corporate HTTP/HTTPS proxy, a relevant npm module to connect to Cohere through the proxy can be downloaded and used in the *vectorizeCohere.js* file for running the embedding models and similarity searches.

Based on the user input, the similarity search function will give the top five most closely related sentences from the source dataset based on the semantics and context obtained from the embedding models.

When the *similaritySearchCohere.js* file is run with the node command, the output is similar to the following:

```
$ node similaritySearchCohere.js
Using embedding model embed-multilingual-v3.0
Using reranker rerank-multilingual-v2.0
TopK = 5
Connected to Oracle Database
```

```
Enter a phrase. Type 'quit' or 'exit' to exit:
Talk about Cars


Reranked results
================
Porsches are fast and reliable.
Nissan GTRs are great.
Toyotas are reliable.
Ford 150s are popular.
Alfa Romeos are fun.

Enter a phrase. Type 'quit' or 'exit' to exit:
Tell me something about the Middle East

Reranked results
================
The United Arab Emirates consists of seven Emirates.
Emirates is the largest airline in the Middle East.
Dubai is in the Persian Gulf.
Dubai is in the United Arab Emirates.
Sheikh Mohamed bin Zayed Al Nahyan is the president of the United Arab Emirates.

Enter a phrase. Type 'quit' or 'exit' to exit:
quit
```

Based on the user input (e.g.,'**`Talk about Cars`**' or '**`Tell me something about the Middle East`**'), the top five semantically and contextually similar statements from the source dataset are displayed.

Conclusion

Oracle AI Vector Search, with Oracle AI Database 26ai, enables a new class of applications powered by semantic searches using LLMs augmented with existing business data. Node-oracledb brings that capability to JavaScript and TypeScript developers.

References

- Oracle AI Vector Search User's Guide
- node-oracledb documentation

Connect with us

Call +**1.800.ORACLE1** or visit **oracle.com**. Outside North America, find your local office at: **oracle.com/contact**.

blogs.oracle.com facebook.com/oracle twitter.com/oracle