14
Working with Vectors

Starting with Oracle Database Release 23ai, you can declare a table's column as a Vector. A Vector is an array of one or more numeric values that may be integers or fractional numbers.

You can use the Vector data for machine learning. Oracle AI (Artificial Intelligence) Vector Search enables you to query data based on semantics, rather than keywords. For more information about Oracle AI Vector Search, refer to the *Oracle AI Vector Search User's Guide*.

- JDBC APIs and Types for Vectors
- SQL to Java Conversions with CallableStatement
- SQL to Java Conversions with CallableStatment and ResultSet
- Java to SQL Conversions with PreparedStatement and CallableStatement
- The VECTOR Datum Class
- Backward Compatibility with Earlier JDBC Drivers

14.1 JDBC APIs and Types for Vectors

JDBC drivers represent SQL data types as instances of the `java.sql.SQLType` interface. For each data type of Oracle Database, the Oracle JDBC Driver declares an instance of `SQLType` as a member of `oracle.jdbc.OracleType`.

14.1.1 JDBC Types for Vectors

This section describes the new instances of SQLType that are added to the `oracle.jdbc.OracleType` enum for Vector support. These instances represent the VECTOR data type.

> **Note:**
>
> Java object types that can be converted to and from the VECTOR data type are specified in the Java API Reference documentation of each type.

VECTOR
`OracleType.VECTOR` represents a Vector of any type, that is, a type with an asterisk (`*`) wild card.

VECTOR_INT8
`OracleType.VECTOR_INT8` represents a Vector of `INT8` values.

VECTOR_FLOAT32
`OracleType.VECTOR_FLOAT32` represents a Vector of `FLOAT32` values.

VECTOR_FLOAT64
`OracleType.VECTOR_FLOAT64` represents a Vector of `FLOAT64` values.

You *must* use these type codes when a `PreparedStatement` has a Vector type parameter. You can provide this type as an argument to the `setObject(int, Object, SQLType)` method or the `setObject(int, Object, int)` method. A Vector parameter *cannot* be set by calling the `setObject(int, Object)` method.

14.1.2 JDBC Interfaces for Vectors

This section describes the JDBC interfaces that have been added or updated for Vector support.

14.1.2.1 The VectorMetaData Interface

The new interface, `oracle.jdbc.VectorMetaData` stores the metadata for a Vector column or parameter.

14.1.2.2 The DatabaseMetaData Interface

JDBC drivers represent metadata of tables and stored procedures as instances of the `java.sql.DatabaseMetaData` interface.

> **See Also:**
>
> The Java SE Documentation

14.1.2.3 The OracleResultSetMetaData and OracleParameterMetaData Interfaces

JDBC drivers represent metadata of columns and parameters as instances of the `java.sql.ResultSetMetaData` and `java.sql.ParameterMetaData` interfaces respectively. The Oracle JDBC Driver extends the `ResultSetMetaData` and `ParameterMetaData` interfaces with `OracleResultSetMetaData` and `OracleParameterMetaData` interfaces, respectively.

14.1.2.4 The SparseArray Interface

Sparse vectors are vectors that typically have a large number of dimensions but with very few non-zero dimension values. For JDBC applications, conversion must happen between the Java objects and the sparse vectors. As Java has no built-in object to represent sparse data, the `SparseArray` interface is used for this purpose.

> **See Also:**
>
> - *Oracle AI Vector Search User's Guide*
> - Oracle Database JDBC Java API Reference

The `SparseArray` interface contains the following sub-interfaces that use a certain Java numeric type to store non-zero values like `double`, `float`, `byte`, and `boolean`:

- `SparseDoubleArray`
- `SparseFloatArray`

- `SparseArray`
- `SparseDoubleArray`

You can use these sub-interfaces as bind values for the `PreparedStatement.setObject(int, Object)` method and as return values for the `ResultSet.getObject(int, Class)` method.

14.1.3 JDBC Methods for Vectors

This section describes the JDBC methods that have been added or updated for Vector support. It also describes the behavior of standard JDBC methods with respect to Vector data.

The getVectorMetaData Method

A method named `getVectorMetaData` is added to the `OracleResultSetMetaData` and `OracleParameterMetaData` interfaces. The method returns an instance of the `oracle.jdbc.VectorMetaData` interface for a Vector column or parameter, which enables the applications to identify the size and the type of the Vector data at run time. The method returns `null` for a column or parameter that is not a Vector.

The getLength Method

A method named `getLength` returns the number of values in a Vector column or parameter. For example:

- The method returns 3 for a column declared as VECTOR(3, `INT8`).
- The method returns -1 for a Vector column or parameter with a variable length, that is, where asterisk (`*`) is specified as the length. For example, VECTOR(`*`, `INT8`).

The getArrayClass Method

A method named `getArrayClass` returns the class of an array object, which you can use in conversions of a Vector column or parameter. For example,

- `double[].class` is returned for a column or parameter that is a Vector of any type. For example, VECTOR(`*`, `*`)
- `byte[].class` is returned for a column or parameter that is a Vector of `INT8` values.
- `float[].class` is returned for a column or parameter that is a Vector of `FLOAT32` values.
- `double[].class` is returned for a column or parameter that is a Vector of `FLOAT64` values.

The getType Method

The `getType` method returns one of the following `OracleTypes` for a Vector column or parameter:

- `OracleTypes.VECTOR` is returned for a column or parameter that is a Vector of any type, that is, a VECTOR(<*length*>,*) type
- `OracleTypes.VECTOR_INT8` is returned for a column or parameter that is a Vector of `INT8` values.
- `OracleTypes.VECTOR_FLOAT32` is returned for a column or parameter that is a Vector of `FLOAT32` values.
- `OracleTypes.VECTOR_FLOAT64` is returned for a column or parameter that is a Vector of `FLOAT64` values.

The getColumns Method

The `getColumns` method retrieves Vector columns in the following ways:

- The `getColumns` method returns the `int` value of `oracle.jdbc.OracleTypes.VECTOR (-105)` as the `DATA_TYPE` for a Vector column.
- The `getColumns` method returns the String value of `"VECTOR"` as the `TYPE_NAME` for a Vector column.

The getObject(int) or getObject(String) Methods

When you call the `getObject(int)` or `getObject(String)` methods of the `java.sql.ResultSet` interface, there is no default mapping for the VECTOR data type. But, you can choose a default mapping with the `oracle.jdbc.vectorDefaultGetObjectType` connection property.

The getPrecision and getScale Methods

The `getPrecision` and `getScale` methods return 0 for a Vector column or parameter. The JDBC 4.3 Specification does not define the correct behavior of these methods for the VECTOR data type. The return values of 0 indicate that precision and scale are not applicable to the data type. All other methods behave as specified by the JDBC 4.3 Specification.

14.2 SQL to Java Conversions with CallableStatement

JDBC drivers represent procedural SQL calls as instances of the `java.sql.CallableStatement` interface. The interface defines the `registerOutParameter` methods that specify the SQL type of an `out` parameter. A corresponding `getObject` method is defined, which converts the registered SQL type to a Java object.

> **See Also:**
>
> The CallableStatement Interface

This section specifies the conversions of the `getObject` method, when converting a Vector to a Java object.

> **Note:**
>
> This section does *not* specify the behavior of the `getObject` methods that accept a `Class` argument. The behavior of these methods is specified in the SQL to Java Conversions with CallableStatment and ResultSet section. These methods are not influenced by a SQL type registered with the registerOutParameter method.

This section discusses the behavior of the *widening* and *narrowing* conversions that are possible with Vectors:

> **See Also:**
>
> - Widening Primitive Conversion
> - Narrowing Primitive Conversion

OracleType.VECTOR Registrations

The `registerOutParameter` methods recognize `OracleType.VECTOR` and `OracleTypes.VECTOR`. With this registration, the following conversions are performed by the `getObject` methods, if no `Class` argument is provided:

- A Vector of `INT8` values is converted to a `byte[]`
- A Vector of `FLOAT32` values is converted to a `float[]`
- A Vector of `FLOAT64` values is converted to a `double[]`

OracleType.VECTOR_INT8 Registrations

The `registerOutParameter` methods recognize `OracleType.VECTOR_INT8` and `OracleTypes.VECTOR_INT8`. With this registration, the following conversions are performed by the `getObject` methods, if no `Class` argument is provided:

- A Vector of `INT8` values is converted to a `byte[]`. No additional conversion occurs in this case.
- A Vector of `FLOAT32` values is converted to a `byte[]`, where, a widening conversion of `byte` to `float` occurs.

OracleType.VECTOR_FLOAT32 Registrations

The `registerOutParameter` methods recognize `OracleType.VECTOR_FLOAT32` and `OracleTypes.VECTOR_FLOAT31`. With this registration, the following conversions are performed by the `getObject` methods, if no Class argument is provided:

- A Vector of `INT8` values is converted to a `float[]`, where, a widening conversion of `byte` to `float` occurs.
- A Vector of `FLOAT32` values is converted to a `float[]`. No additional conversion occurs in this case.
- A Vector of `FLOAT64` values is converted to a `float[]`, where, a narrowing conversion of `double` to `float` occurs.

OracleType.VECTOR_FLOAT64 Registrations

The `registerOutParameter` methods recognize `OracleType.VECTOR_FLOAT64` and `OracleTypes.VECTOR_FLOAT64`. With this registration, the following conversions are performed by the `getObject` methods, if no `Class` argument is provided:

- A Vector of `INT8` values is converted to a `double[]`, where, a widening conversion of `byte` to `double` occurs.
- A Vector of `FLOAT32` values is converted to a `double[]`, where, a widening conversion of `float` to `double` occurs.

- A Vector of `FLOAT64` values is converted to a `double[]`. No additional conversion occurs in this case.

14.3 SQL to Java Conversions with CallableStatment and ResultSet

JDBC drivers represent the values of out parameters and columns as instances of the `java.sql.CallableStatement` and `java.sql.ResultSet` interfaces respectively. Both the interfaces define `getObject` methods that convert a SQL type to a Java object.

This section describes the behavior of the `getObject` methods, when converting Vector parameters and columns to Java objects.

This section discusses the behavior of the *widening* and *narrowing* conversions that are possible with Vectors:

> **See Also:**
>
> - Widening Primitive Conversion
> - Narrowing Primitive Conversion

Default Conversions

The `getObject(int)` and `getObject(String)` methods of the `ResultSet` interface do *not* support conversions of Vector columns. The JDBC 4.3 Specification does not specify any class of Java object as the default conversion of Vector.

The `getObject(int)` and `getObject(String)` methods of the `CallableStatement` interface support conversions of Vector columns. Refer to the SQL to Java Conversions with CallableStatement section for more details.

boolean[] Conversions

The `getObject` methods recognize `boolean[].class` as a target Java type. The following conversions are performed:

- A Vector of `INT8` values is converted to `boolean[]`. A value of 0 is converted to `false`, and a value that is not 0, is converted to `true`.
- A Vector of `FLOAT32` values is converted to `boolean[]`. A value of 0.0 is converted to `false`, and a value that is not 0.0, is converted to `true`.
- A Vector of `FLOAT64` values is converted to `boolean[]`. A value of 0.0 is converted to `false`, and a value that is not 0.0, is converted to `true`.

byte[] Conversions

The `getObject` methods recognize `byte[].class` as a target Java type. The following conversions are performed:

- A Vector of `INT8` values is converted to `byte[]`. No additional conversion is performed in this case.

- A Vector of `FLOAT32` values is converted to `byte[]`, where, a narrowing conversion of `float` to `byte` occurs.
- A Vector of `FLOAT64` values is converted to `byte[]`, where, a narrowing conversion of `double` to `byte` occurs.

short[] Conversions

The `getObject` methods recognize `short[].class` as a target Java type. The following conversions are performed:

- A Vector of `INT8` values is converted to `short[]`, where, a widening conversion of `byte` to `short` occurs.
- A Vector of `FLOAT32` values is converted to `short[]`, where, a narrowing conversion of `float` to `short` occurs.
- A Vector of `FLOAT64` values is converted to `short[]`, where, a narrowing conversion of `double` to `short` occurs.

int[] Conversions

The `getObject` methods recognize `int[].class` as a target Java type. The following conversions are performed:

- A Vector of `INT8` values is converted to `int[]`, where, a widening conversion of `byte` to `int` occurs.
- A Vector of `FLOAT32` values is converted to `int[]`, where, a narrowing conversion of `float` to `int` occurs.
- A Vector of `FLOAT64` values is converted to `byte[]`, where, a narrowing conversion of `double` to `int` occurs.

long[] Conversions

The `getObject` methods recognize `long[].class` as a target Java type. The following conversions are performed:

- A Vector of `INT8` values is converted to `long[]`, where, a widening conversion of `byte` to `long` occurs.
- A Vector of `FLOAT32` values is converted to `long[]`, where, a narrowing conversion of `float` to `long` occurs.
- A Vector of `FLOAT64` values is converted to `long[]`, where, a narrowing conversion of `double` to `long` occurs.

float[] Conversions

The `getObject` methods recognize `float[].class` as a target Java type. The following conversions are performed:

- A Vector of `INT8` values is converted to `float[]`, where, a widening conversion of `byte` to `int` occurs.
- A Vector of `FLOAT32` values is converted to `float[]`. No additional conversion is performed in this case.
- A Vector of `FLOAT64` values is converted to `float[]`, where, a narrowing conversion of `double` to `float` occurs.

double[] Conversions

The `getObject` methods recognize `double[].class` as a target Java type. The following conversions are performed:

- A Vector of `INT8` values is converted to `double[]`, where, a widening conversion of`byte` to `double` occurs.
- A Vector of `FLOAT32` values is converted to `double[]`, where, a widening conversion of`float` to `double` occurs.
- A Vector of `FLOAT64` values is converted to `double[]`, where, a narrowing conversion of `double` to `float` occurs.

14.4 Java to SQL Conversions with PreparedStatement and CallableStatement

JDBC drivers represent SQL commands as instances of the `java.sql.PreparedStatement` and `java.sql.CallableStatement` interfaces. These interfaces define the `setObject` methods that convert a Java object to a SQL type. This section describes the behavior of the `setObject` method, when converting Java objects to a Vector.

> **See Also:**
>
> - The PreparedStatement Interface
> - The CallableStatement Interface

This section discusses the behavior of the *widening* and *narrowing* conversions that are possible with Vectors:

> **See Also:**
>
> - Widening Primitive Conversion
> - Narrowing Primitive Conversion

Default Conversion

The `setObject(int, Object)` method does not support conversions to Vectors. The JDBC 4.3 Specification does not specify Vector as the default conversion for any class of Java object.

OracleType.VECTOR Conversions

The `setObject` methods recognize `OracleType.VECTOR` and `OracleTypes.VECTOR` as a target SQL type. The following conversions are supported for this type:

- A `byte[]` is converted to a Vector of `INT8` values
- A `float[]` is converted to a Vector of `FLOAT32` values

- A `double[]` is converted to a Vector of `FLOAT64` values.

OracleType.VECTOR_INT8 Conversions

The `setObject` methods recognize `OracleType.VECTOR_INT8` and `OracleTypes.VECTOR_INT8` as a target SQL type. The following conversions are supported for this target SQL type:

- A `boolean[]` is converted to a Vector of `INT8` values. A boolean value of `false` is converted to 0, and a value of `true` is converted to 1.
- A `byte[]` is converted to a Vector of `INT8` values. No additional conversion occurs in this case.
- A `short[]` is converted to a Vector of `INT8` values, where, a narrowing conversion of `short` to `byte` occurs.
- An `int[]` is converted to a Vector of `INT8` values, where, a narrowing conversion of `int` to `byte` occurs.
- A `long[]` is converted to a Vector of `INT8` values, where, a narrowing conversion of `long` to `byte` occurs.
- A `float[]` is converted to a Vector of `INT8` values, where, a narrowing conversion of `float` to `byte` occurs.
- A `double[]` is converted to a Vector of `INT8` values, where, a narrowing conversion of `double` to `byte` occurs.

OracleType.VECTOR_FLOAT32 Conversions

The `setObject` methods recognize `OracleType.VECTOR_FLOAT32` and `OracleTypes.VECTOR_FLOAT32` as a target SQL type. The following conversions are supported for this target SQL type:

- A `boolean[]` is converted to a Vector of `FLOAT32` values. A boolean value of `false` is converted to 0.0, and a value of `true` is converted to 1.0.
- A `byte[]` is converted to a Vector of `FLOAT32` values, where, a widening conversion of `byte` to `float` occurs.
- A `short[]` is converted to a Vector of `FLOAT32` values, where, a widening conversion of `short` to `float` occurs.
- An `int[]` is converted to a Vector of `FLOAT32` values, where, a widening conversion of `int` to `float` occurs.
- A `long[]` is converted to a Vector of `FLOAT32` values, where, a widening conversion of `long` to `float` occurs.
- A `float[]` is converted to a Vector of `FLOAT32` values. No additional conversion occurs in this case.
- A `double[]` is converted to a Vector of `FLOAT32` values, where, a narrowing conversion of `double` to `float` occurs.

OracleType.VECTOR_FLOAT64 Conversions

The `setObject` methods recognize `OracleType.VECTOR_FLOAT64` and `OracleTypes.VECTOR_FLOAT64` as a target SQL type. The following conversions are supported for this target SQL type:

- A `boolean[]` is converted to a Vector of `FLOAT64` values, where, a boolean value of `false` is converted to 0.0 and `true` is converted to 1.0.
- A `byte[]` is converted to a Vector of `FLOAT64` values, where, a widening conversion of `byte` to `double` occurs.
- A `short[]` is converted to a Vector of `FLOAT64` values, where, a widening conversion of `short` to `double` occurs.
- An `int[]` is converted to a Vector of `FLOAT64` values, where, a widening conversion of `int` to `double` occurs.
- A `long[]` is converted to a Vector of `FLOAT64` values, where, a widening conversion of `long` to `double` occurs.
- A `float[]` is converted to a Vector of `FLOAT64` values, where, a widening conversion of `float` to `double` occurs.
- A `double[]` is converted to a Vector of `FLOAT64` values. No additional conversion occurs in this case.

14.5 The VECTOR Datum Class

The `oracle.sql` package defines a `Datum` class with subclasses that represent each Oracle SQL data type. For example, the `oracle.sql.NUMBER` subclass represents a value of the `NUMBER` data type, and `oracle.sql.TIMESTAMP` represents values of the `TIMESTAMP` data type.

A new subclass `oracle.sql.VECTOR` is added to the `Datum` class to represent values of Vector columns. The `VECTOR` class supports conversions between Java objects and Oracle's binary encoding of a Vector.

Conversions from Java Objects

The `VECTOR` class defines factory methods that create an instance of a Vector. These factory methods convert a Java object into the binary encoding of a Vector in the following ways:

- An `ofFloat64Values(Object)` method converts a Java object into a Vector of `FLOAT64` values.
- An `ofFloat32Values(Object)` method converts a Java object into a Vector of `FLOAT32` values.
- An `ofInt8Values(Object)` method converts a Java object into a Vector of `INT8` values.

> **See Also:**
>
> Java to SQL Conversions with PreparedStatement and CallableStatement

Conversions to Java Objects

The `VECTOR` class defines instance methods that return a Java object representation of a Vector. These instance methods convert the binary encoding of a Vector into a Java object in the following ways:

- The `toBooleanArray()` method converts a Vector into an array of `boolean` values
- The `toByteArray()` method converts a Vector into an array of `byte` values

- The `toShortArray()` method converts a Vector into an array of `short` values
- The `toIntArray()` method converts a Vector into an array of `int` values
- The `toLongArray()` method converts a Vector into an array of `long` values
- The `toFloatArray()` method converts a Vector into an array of `float` values
- The `toDoubleArray()` method converts a Vector into an array of `double` values

> **See Also:**
>
> SQL to Java Conversions with CallableStatment and ResultSet

14.6 Backward Compatibility with Earlier JDBC Drivers

Earlier releases of Oracle JDBC may connect to an Oracle Database 23ai. These JDBC builds do not have built-in support for the VECTOR data type, but they do support the `VARCHAR` and `CLOB` data types, and applications may use these types for DML and query operations on Vector columns.

JDBC supports conversions of `String` with `VARCHAR` and `java.sql.Clob` with `CLOB`. These conversions have consistent behavior in Oracle Database 19c, 21c, and 23ai releases.

The following code example demonstrates these conversions, where a `String` and `java.sql.Clob` are passed to the `PreparedStatement.setObject` method. Conversions of `CLOB` to `String` are demonstrated by calling the `ResultSet.getObject(int, Class)` method with the `String.class` method.

```
import oracle.jdbc.OracleStatement;

import java.sql.Clob;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
import java.util.Arrays;
import java.util.Random;

/**
 * This example can be run with 19.x releases of Oracle JDBC. It uses String
to
 * represent VECTOR data, which may be suitable for database tools.
 */
public class VectorStringTest {

  public static void main(String[] args) throws SQLException {

    try (
      Connection connection =
        DriverManager.getConnection("jdbc:oracle:thin:@test");
```

```
      Table table = new Table(connection);
      PreparedStatement insert = connection.prepareStatement(
        "INSERT INTO vector_test(id, value) VALUES (?, ?)");
      PreparedStatement query = connection.prepareStatement(
        "SELECT id, value FROM vector_test ORDER BY id")) {

      // Toy example to show the VARCHAR literal syntax of a VECTOR: A comma
      // separated numbers enclosed in square brackets.
      String vectorLiteral = "[0.1, 0.2, 0.3]";
      insert.setString(1, "0");
      insert.setString(2, vectorLiteral);
      System.out.println("Inserting VECTOR (VARCHAR):\n\t" + vectorLiteral);
      insert.executeUpdate();

      // Generate a Vector of 256 dimensions, each having many decimal point
      // digits. Arrays.toString(double[]) conveniently generates the Vector
      // literal syntax, so it may be used to convert the Vector to String.
      double[] vector = getVector(256);
      String vectorString = Arrays.toString(vector);
      insert.setObject(1, "1");
      insert.setObject(2, vectorString);
      System.out.println("Inserting VECTOR (VARCHAR):\n\t" + vectorString);
      insert.executeUpdate();

      // If the String is longer than 32k characters, then it must be
converted
      // to a CLOB (32k is the maximum length of a VARCHAR).
      // This example results in:
      //   ORA-42552: VECTOR() library processing error
'LVECTOR_ERR_INPUT_NAN_OR_INF' in 'qvcCons:lvector_from_oratext'.
      // The 2048 length is commented out for this reason. A 256 length is
used
      // just to demonstrate the conversion to CLOB.
      // double[] largeVector = getVector(2048);
      double[] largeVector = getVector(256);
      Clob vectorClob = connection.createClob();
      try {
        String largeVectorString = Arrays.toString(largeVector);
        vectorClob.setString(1L, largeVectorString);
        insert.setString(1, "2");
        insert.setObject(2, vectorClob);
        System.out.println("Inserting VECTOR (CLOB):\n\t" +
largeVectorString);
        insert.executeUpdate();
      }
      finally {
        vectorClob.free();
      }

      // Query the VECTOR column. For a 19c JDBC client, the database sends
the
      // VECTOR as a CLOB. For a 23ai JDBC client, it sends it as the VECTOR
binary
      // encoding. When the getString method has JDBC convert the VECTOR, both
      // client versions should return the same text value.
      try (ResultSet resultSet = query.executeQuery()) {
```

```
      ResultSetMetaData metaData = resultSet.getMetaData();
      while (resultSet.next()) {
        System.out.println("Queried VECTOR:");
        System.out.printf(
          "\t%s (%s) : %s%n",
          metaData.getColumnName(1),
          metaData.getColumnTypeName(1),
          resultSet.getString(1));
        System.out.printf(
          "\t%s (%s) : %s%n",
          metaData.getColumnName(2),
          metaData.getColumnTypeName(2),
          resultSet.getString(2));
      }
    }

    // Applications can request that the database always sends the VECTOR
    // as a CLOB. The defineColumnType method is used to specify the CLOB
    // type.
    System.out.println("\nQuerying VECTOR as CLOB");
    query.unwrap(OracleStatement.class)
      .defineColumnType(2, Types.CLOB);
    try (ResultSet resultSet = query.executeQuery()) {
      ResultSetMetaData metaData = resultSet.getMetaData();
      while (resultSet.next()) {
        System.out.println("Queried VECTOR:");
        System.out.printf(
          "\t%s (%s) : %s%n",
          metaData.getColumnName(1),
          metaData.getColumnTypeName(1),
          resultSet.getString(1));
        System.out.printf(
          "\t%s (%s) : %s%n",
          metaData.getColumnName(2),
          metaData.getColumnTypeName(2),
          resultSet.getString(2));
      }
    }

  }
}

static double[] getVector(int length) {
  return new Random(0).doubles()
    .limit(length)
    .toArray();
}

static class Table implements AutoCloseable {

  private final Connection connection;

  Table(Connection connection) throws SQLException {
    try (Statement statement = connection.createStatement()) {
      statement.addBatch("DROP TABLE IF EXISTS vector_test");
      statement.addBatch(
```

```
            "CREATE TABLE vector_test" +
              " (id NUMBER PRIMARY KEY, value VECTOR(*,*))");
          statement.executeBatch();
        }
        this.connection = connection;
      }

      @Override
      public void close() throws SQLException {
        try (Statement statement = connection.createStatement()) {
          statement.execute("DROP TABLE IF EXISTS vector_test");
        }
      }
    }
}
```