**Bring AI-Powered Search to Your Db2 Data with Vectors**

**By Shaikh Quader**

A person holding a phone and a device

AI-generated content may be incorrect.

This walkthrough demonstrates how to build a **semantic product recommendation system** using the **AI vector features in IBM Db2**, scheduled for general availability in **June 2025** with Db2 version 12, Mod Pack 2.

💡 *New to Db2 vectors?*  
Check out the [first installment](https://community.ibm.com/community/user/blogs/shaikh-quader/2025/03/19/vectors-intro) — **AI Vectors and Similarity Search: A Gentle Introduction** — for a beginner-friendly overview of how vector search works and why it matters.

**The Use Case**

Let’s ground this with a familiar scenario:

You walk into a shoe store holding a picture of a sneaker you liked online. It’s a black knit running shoe with waterproof material and a flat arch. You ask the associate:

“Do you have anything like this in stock?”

Now imagine the store’s system runs on traditional SQL. It uses filters like:

**brand = 'Zentrax' AND color = 'Black' AND material = 'Knit'**

That means if the store has a nearly identical shoe—but in **gray** or **mesh**, or from another brand—it won’t show up at all.

You end up missing great alternatives just because one field doesn’t match exactly. This is the **limitation of keyword search** and traditional SQL filters. They’re rigid and binary. A match either qualifies or it doesn’t.

But what if the system could understand similarity the way people do? What if it could say:

“You liked this black waterproof running shoe with a flat arch. Here’s a gray version with similar build and style.”

That’s where **semantic search** powered by **vector embeddings** comes in. In this post, I’ll show how to implement this in Db2—so your queries don’t just ask *what’s the same*, but *what feels similar*. Even if the exact words don’t match.

A person using a phone and a computer

AI-generated content may be incorrect.

Watch the following quick demo video showing how Db2’s vector search powers intelligent shoe recommendations based on semantic similarity — from a reference selection to visually similar matches on the storefront.

This short clip provides a high-level overview of the capability in action, followed by a technical walkthrough in this post.

[A screenshot of a video

AI-generated content may be incorrect.](https://video.ibm.com/recorded/134363955)

**The Demo Setup**

To run this demo locally, start by reviewing the [README.md](https://github.com/shaikhq/db2-samples/blob/master/ai-vectors/product-recommendation/README.md) file in the [Db2 samples GitHub repository](https://github.com/shaikhq/db2-samples/tree/master/ai-vectors/product-recommendation). It contains complete instructions on how to set up your Db2 environment, prepare the tables, and execute the demo workflow.

This demo uses a **synthetic dataset of shoes**, where each record includes:

* Descriptive attributes: **BRAND**, **COLOR**, **MATERIAL**
* Metadata: **SKU**, **SIZE**, **CITY**, **GENDER**

The dataset was generated using the notebook [shoes-data-generation.ipynb](https://github.com/shaikhq/db2-samples/blob/master/ai-vectors/product-recommendation/shoes-data-generation.ipynb). You can explore or modify this notebook if you're interested in generating your own variants.

To run the actual semantic search demo, open and execute:  
[shoes-search.ipynb](https://github.com/shaikhq/db2-samples/blob/master/ai-vectors/product-recommendation/shoes-search.ipynb)

All vector embeddings have been **precomputed** and saved in [shoes-vectors.csv](https://github.com/shaikhq/db2-samples/blob/master/ai-vectors/product-recommendation/shoes-vectors.csv).

So, you can run the entire workflow without needing live access to watsonx.ai.  
If you do want to regenerate the embeddings yourself, you’ll need:

* A watsonx.ai API key and a project
* Python SDK installed and configured

The remainder of the workflow—vector storage, filtering, and search—is handled **natively inside Db2** using SQL and built-in vector functions.

**When SQL Alone Isn't Enough — Enter Vector Search**

**SQL is powerful. But it’s exact.** You write queries to retrieve rows that match the **WHERE** clause *precisely*. That’s great when you want to:

* List all shoes under $100
* Find all shoes in stock in Toronto
* Filter size = 12 AND gender = 'Men'

But some queries need more nuanced answers:

“What if you want **similar**, not **exact**?”

**When “Close Enough” is Exactly What You Want**

Let’s say you run the above shoe store with an email campaign that goes out every Friday. You want to recommend products to customers based on their past purchases. But here’s the challenge: they’re not all buying the same brand or sticking to the same material or color. What they’re really looking for are **feel-alike products**—items that are similar in purpose, look, or experience, even if the details differ slightly.

In traditional SQL, expressing this kind of “semantic similarity” is **clunky**. You’re stuck with:

* Handcrafting combinations of filters
* Losing flexibility
* Missing out on *almost-right* options

**Vector search** changes that.

**What Is a Vector, Really?**

Before we dive deeper, it helps to understand what vectors actually are and how Db2 uses them.

A **vector embedding** is a numerical representation of data—typically high-dimensional—that captures the meaning or semantics of the original input. In simpler terms, it’s a way to translate human-readable descriptions (like text) into a list of numbers that a machine can compare.

Think of a vector embedding as **compressing the essence of a product into a point in space**. If two products are close in that space, they’re semantically similar—even if they don’t share exact text values.

For example, the description, a text,

“*Running shoe, knit, black, waterproof, flat arch*”

...might be represented as a vector like this:

[0.132, -0.087, 0.456, 0.901, -0.334, ... , 0.274] # (1024 total dimensions)

This list of numbers is what the model uses to compare with other products based on semantic similarity.

In our use case, we pass a text into a **pretrained language model**, which returns a vector of 1024 floating-point numbers. This vector acts like a **semantic fingerprint** of the shoe.

To support this, **Db2 now includes native vector support**—meaning you can store and compare vectors directly in a table column. The column is declared like this:

VECTOR(**1024**, **FLOAT32**)

* **1024** refers to the number of dimensions in the vector, determined by the embedding model in use.
* **FLOAT32** indicates that each dimension is represented as a 32-bit float.

All vectors stored in a column must use the same dimensionality and coordinate type, ensuring consistency for distance calculations.

With this support, you can now **store**, **filter**, and **rank by similarity**—entirely from SQL—enabling intelligent, flexible, AI-powered queries inside your relational database.

**Step-by-Step Implementation of The Shoe Recommendation Use Case**

**Step 1: Create the Shoes Table**

**CREATE TABLE** SQ\_SHOES **(**

SKU **VARCHAR(20)**,

BRAND **VARCHAR(50)**,

MODEL **VARCHAR(50)**,

COLOR **VARCHAR(30),**

MATERIAL **VARCHAR(30)**,

SIZE **INT**,

GENDER **VARCHAR(10)**,

CITY **VARCHAR(50)**

**)**;

*This creates a standard Db2 table to store structured attributes of shoes, such as size, gender, city.*

**IMPORT FROM** **'shoes.csv'** **OF** DEL **INSERT INTO** SQ\_SHOES;

*Loads the shoe records from a CSV file into the newly created table.*

**Step 2: Add a Vector Column**

**ALTER TABLE** SQ\_SHOES **ADD COLUMN** EMBEDDING VECTOR(**1024**, **FLOAT32**);

*Adds a new column that will store the semantic embeddings of each shoe as a 1024-dimensional vector of float values.*

* **FLOAT32** is the most common coordinate type used for vector embeddings generated by modern language models.
* **1024** is the number of dimensions produced by the selected embedding model selected.

All vectors stored in this column must have the **same dimensionality and coordinate type**. This ensures consistency and compatibility during similarity comparisons using vector distance functions.

**Step 3: Choosing the Right Features for Embedding**

Not all columns are useful for semantic similarity.

* ✅ Keep: Descriptive **text-based features** like **brand**, **color**, **material**
* ❌ Skip: Arbitrary IDs like **SKU** (not repeatable or semantically meaningful)

We build a descriptive sentence:

**Type: Running; Material: Knit; Color: Black; Weather Resistance: Waterproof; Arch Support: Flat**

*This structured sentence is passed to a text embedding model to generate a meaningful vector representation.*

**Step 4: Generate Embeddings**

# Initialize the Embeddings object

embeddings = Embeddings(

model\_id=**"intfloat/multilingual-e5-large",**

credentials=credentials,

project\_id**="YOUR\_PROJECT\_ID",**

params=embed\_params

)

# Text to be embedded

text = **"Type: Running; Material: Knit; Color: Black; Weather Resistance: Waterproof; Arch Support: Flat"**

# Generate the embedding vector

vector = embeddings.embed\_query(text)

**Code breakdown:**

* **embeddings = Embeddings(...)**: Creates an instance of the Embeddings class from IBM's **watsonx.ai** SDK, which is used to generate vector representations (embeddings) of text inputs.
* **model\_id="intfloat/multilingual-e5-large"**: Specifies the embedding model to use. The **intfloat/multilingual-e5-large** model is a multilingual embedding model developed by Microsoft, capable of handling text in multiple languages.
* **credentials=credentials**: Provides the necessary authentication credentials to access IBM's watsonx.ai services.
* **project\_id="YOUR\_PROJECT\_ID"**: Indicates the specific IBM Cloud project under which the embedding operations will be performed. Replace **"YOUR\_PROJECT\_ID"** with your actual project ID.
* **params=embed\_params**: Optional parameters that can customize the embedding process, such as truncating input tokens or specifying return options.
* **text = ...**: Defines the text input that you want to convert into an embedding vector. The text describes characteristics of a product, which can be useful for tasks like product recommendation or similarity searches.
* **vector = embeddings.embed\_query(text)**: Calls the **embed\_query** method on the embeddings object, passing in the defined text. The method processes the text using the specified embedding model and returns a numerical vector representation. This vector captures the semantic meaning of the text and can be used in various downstream tasks like clustering, classification, or semantic search.

**UPDATE** SQ\_SHOES

**SET** EMBEDDING **=** VECTOR**('{vector}'**, **1024**, **FLOAT32)**

**WHERE** SKU **=** **'ZEN-2061';**

*Stores the generated vector into the table, using the VECTOR constructor function to convert the string of numbers into a native Db2 vector type.*

**Step 5: Combine SQL Filters + Vector Similarity**

*This query narrows the candidate shoes to those that match basic structured filters like city, gender, and size.*

**SELECT** **\***, VECTOR\_DISTANCE(

(**SELECT** EMBEDDING **FROM** SQ\_SHOES **WHERE** SKU **=** **'ZEN-2061'**),

EMBEDDING,

EUCLIDEAN

) **AS** similarity

**FROM** SQ\_SHOES

**WHERE** CITY = **'Toronto'** **AND** GENDER **=** **'Men'** **AND** SIZE **=** **12**

**ORDER** **BY** similarity **ASC**

**FETCH** **FIRST** **5** **ROWS** ONLY;

*This query performs a semantic similarity search using the* ***VECTOR\_DISTANCE*** *function. It works in several parts:*

* **Reference Vector**: The subquery **(SELECT EMBEDDING FROM SKU\_SHOES WHERE SKU = 'ZEN-2061')** retrieves the embedding of the reference shoe that the user is interested in.
* **Candidate Vectors**: The outer query filters the dataset using structured attributes (location, gender, size) to limit the number of comparisons.
* **Distance Metric**: **'EUCLIDEAN'** is specified as the distance function. Smaller euclidean distances indicate greater similarity.
* **Ordering and Limiting**: The results are ordered by similarity score in ascending order and only the top 5 most similar items are returned.

*In essence, this query identifies the five most semantically similar shoes—based on descriptive features—among those that meet the basic filters. It blends traditional SQL filtering with vector-based ranking in a single query.*

**Interpreting Results**

If you compare the **embedding vector** of the reference shoe with those of the returned shoes:

* **Top matches** appear closer in the vector space, meaning they share more semantic features with the reference shoe (e.g., similar material, purpose, or color).
* As you **move down the ranked list**, the shoes become less similar. Differences may emerge in key attributes—perhaps the color changes, or the material is different—but they still retain enough similarity to be considered relevant.

This behavior illustrates the power of vector search: instead of returning only exact matches, the system prioritizes *semantic closeness*. It opens up opportunities to recommend alternatives that a rigid SQL filter would miss entirely.

I used a dimension reduction technique to reduce the vectors to 2D. The reference shoe appears at the center. Similar shoes cluster nearby. Less relevant shoes appear farther away.

This makes the intuition behind **vector similarity** visible.

A diagram of a running graph

AI-generated content may be incorrect.

**Final Thoughts**

**Vector search adds a powerful new search tool to relational data.**

With Db2’s native vector support, you can now:

* Store dense vector embeddings alongside your structured data.
* Use SQL to run flexible similarity queries.
* Blend metadata filters with semantic recommendations.

It opens the door to smarter, more human-like use cases—from product recommendations to document discovery.

If you’re already using Db2, you don’t need to move your data elsewhere. This demo shows how to bring semantic capabilities **into your existing SQL world**.

GitHub: [Db2 samples GitHub repository](https://github.com/shaikhq/db2-samples/tree/master/ai-vectors/product-recommendation)

**Acknowledgments**

This post wouldn’t be complete without acknowledging the colleagues who helped bring the video demo to life. I want to thank Varun M, Lakshmi Narayanan PV, Jasper Zhu, and [@Aniruddha Joshi](https://community.ibm.com/community/user/people/aniruddha-joshi) for their contributions in building and refining the product recommendation demo featured here.

I'm also grateful to [@Christian Garcia-Arellano](https://community.ibm.com/community/user/people/christian-garcia-arellano) and [@JANA WONG](https://community.ibm.com/community/user/people/jana-wong) for reviewing this post and helping shape it into something clearer and more polished.