```
@NikhilSharma ➜ C:\..\dev-genAI (base 3.12.7) 298ms
[100%] python .\06_sqlite_vec.py
sqlite_version=3.45.3, vec_version=v0.1.6


Inserted embeddings:
(1, [0.1, 0.1, 0.1, 0.1])
(2, [0.2, 0.2, 0.2, 0.2])
(3, [0.3, 0.3, 0.3, 0.3])
(4, [0.4, 0.4, 0.4, 0.4])
(5, [0.5, 0.5, 0.5, 0.5])

Query Vector:
[0.3, 0.3, 0.3, 0.3]

Similar vectors found:
----------------------
ID: 3, Distance: 0.0
ID: 4, Distance: 0.19999998807907104
ID: 2, Distance: 0.20000001788139343


==================Distance Metrics====================

L2 (Euclidean Distance) and Cosine Distance:
(1, 2, 0.20000000298023224, 0.0)
(1, 3, 0.40000003576278687, 1.552204231813903e-08)
(1, 4, 0.6000000238418579, 0.0)
(1, 5, 0.800000011920929, 2.0489094865183688e-08)
(2, 3, 0.20000001788139343, 1.552204231813903e-08)
(2, 4, 0.4000000059604645, 0.0)
```

sqlite-vec

What is sqlite-vec?

`sqlite-vec` extends SQLite with native vector support. It introduces a new vector data type and adds a suite of functions for working with vectors. Think of it as embedding a minimal vector database engine directly into your local `.db` file.

You get:

- Native vector types: `float32`, `int8`, and `bit` (binary vectors)
- Distance metrics: L2 (Euclidean), L1 (Manhattan), cosine similarity, and Hamming
- SQL functions for manipulating vectors
- KNN search via virtual tables
- SIMD acceleration with AVX and NEON

This makes `sqlite-vec` ideal for use cases like:

- Embedding-based semantic search
- Local AI-powered search engines
- Lightweight ML applications
- Offline recommender systems

What is sqlite-vec and what capabilities does it add to SQLite?

`sqlite-vec` is an SQLite extension that introduces native support for vector data, allowing users to store, manipulate, and query vector information directly within SQLite. It offers features such as K-Nearest Neighbor (KNN) search, various distance metrics (like Euclidean and cosine similarity), and

SIMD-accelerated performance, making it suitable for applications like semantic search and recommendation engines.

How does `sqlite-vec` ensure type safety when storing different types of vectors?

`sqlite-vec` uses SQLite's subtype feature to track the type of each vector column, enforcing type safety. This means that if a user attempts to compare vectors of incompatible types (like `float32` and `int8`), the system will return a clear error message instead of allowing accidental operations that could lead to undefined behavior.

What are some of the distance metrics supported by `sqlite-vec` and how are they used?

`sqlite-vec` supports several distance metrics for vector comparisons, including L2 (Euclidean) distance, L1 (Manhattan) distance, cosine similarity, and Hamming distance. These metrics are used with SQL functions to compute the similarity between vectors, helping in tasks like semantic search and nearest neighbor queries.

In what scenarios is using `sqlite-vec` particularly advantageous?

`sqlite-vec` is ideal when lightweight, embedded vector search is needed, especially for applications utilizing SQLite, such as AI-powered apps on edge devices or mobile platforms. It provides a reproducible and portable solution without the need for external dependencies, making it suitable for local and medium-scale applications.

What advanced features does `sqlite-vec` offer for managing and querying metadata alongside vectors?

`sqlite-vec` allows users to attach custom metadata columns (like labels or timestamps) to vector data, enabling efficient filtering during queries. This capability is combined with an internal indexing system that can skip irrelevant rows before performing distance calculations, thereby improving performance in KNN searches.

Vector search has become a foundational tool for modern applications — from powering recommendation engines to enabling semantic search in LLM pipelines. Traditionally, developers reached for dedicated vector databases like FAISS, Annoy, or Pinecone.

When to Use This

`sqlite-vec` is perfect when:

- You want lightweight, embedded vector search
- You already use SQLite and want to avoid external dependencies
- You're building AI-powered apps on the edge or mobile
- You want reproducible, portable vector pipelines

It may *not* be ideal for large-scale distributed search across millions of high-dimensional vectors — but for local, embedded, or small-medium applications, it's shockingly capable.

Implementation

```
import sqlite3
from typing import Tuple, Optional, List, Any, Union, Iterable
import sqlite_vec
from sqlite_vec import serialize_float32

class VectorDatabase:
    def __init__(self, db_path: str):
        self.db_path = db_path
        self.connection: Optional[sqlite3.Connection] = None
        self._connect()

    def _connect(self) -> None:
        self.connection = sqlite3.connect(self.db_path)
        self.connection.enable_load_extension(True)
        sqlite_vec.load(self.connection)
        self.connection.enable_load_extension(False)

    def get_versions(self) -> Tuple[str, str]:
        if not self.connection:
            self._connect()
        return self.connection.execute(
            "SELECT sqlite_version(), vec_version()"
        ).fetchone()

    def display_version_info(self) -> None:
        sqlite_version, vec_version = self.get_versions()
        print(f"sqlite_version={sqlite_version}, vec_version={vec_version}")

    def execute(self, query: str, params: Union[tuple, dict, list, None] = None) ->
sqlite3.Cursor:
        if not self.connection:
            self._connect()

        try:
            if params:
                return self.connection.execute(query, params)
            else:
                return self.connection.execute(query)
        except sqlite3.Error as e:
            print(f"Query execution error: {e}")
            raise

    def executemany(self, query: str, params_seq: Iterable[Union[tuple, dict, list]]) ->
sqlite3.Cursor:
        if not self.connection:
            self._connect()

        try:
            return self.connection.executemany(query, params_seq)
        except sqlite3.Error as e:
            print(f"Query execution error: {e}")
            raise

    def execute_fetchall(self, query: str, params: Union[tuple, dict, list, None] = None) -
> List[tuple]:
```

```
        cursor = self.execute(query, params)
        return cursor.fetchall()

    def execute_fetchone(self, query: str, params: Union[tuple, dict, list, None] = None) -
> Optional[tuple]:
        cursor = self.execute(query, params)
        return cursor.fetchone()

    def create_vector_table(self, table_name: str, dimensions: int) -> None:
        self.execute(f"CREATE VIRTUAL TABLE IF NOT EXISTS {table_name} USING vec0(embedding
float[{dimensions}])")

    def commit(self) -> None:
        if self.connection:
            self.connection.commit()

    def close(self) -> None:
        if self.connection:
            self.connection.close()
            self.connection = None

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.connection and exc_type is None:
            self.connection.commit()
        self.close()


# def serialize_float32(vector: List[float]) -> bytes:
#     return struct.pack(f'{len(vector)}f', *vector)


def find_similar_vectors(db: VectorDatabase, table_name: str, query_vector: List[float],
limit: int = 3) -> List[tuple]:
    return db.execute_fetchall(
        f"""
        SELECT
          rowid,
          distance
        FROM {table_name}
        WHERE embedding MATCH ?
        ORDER BY distance
        LIMIT ?
        """,
        [serialize_float32(query_vector), limit]
    )


def load_vectors(db: VectorDatabase, table_name: str, items: List[Tuple[int, List[float]]])
-> None:
    serialized_items = [(item[0], serialize_float32(item[1])) for item in items]

    with db:
        for item_id, serialized_vector in serialized_items:
            db.execute(
                f"INSERT INTO {table_name}(rowid, embedding) VALUES (?, ?)",
```

```
                [item_id, serialized_vector]
            )


def main():
    items = [
        (1, [0.1, 0.1, 0.1, 0.1]),
        (2, [0.2, 0.2, 0.2, 0.2]),
        (3, [0.3, 0.3, 0.3, 0.3]),
        (4, [0.4, 0.4, 0.4, 0.4]),
        (5, [0.5, 0.5, 0.5, 0.5]),
    ]
    query_vector = [0.3, 0.3, 0.3, 0.3]
    table_name = "vec_data"
    vector_dimensions = 4

    with VectorDatabase("local.sqlite") as db:
        db.display_version_info()

        # uncomment below lines when creating the table

        # db.create_vector_table(table_name, vector_dimensions)
        # load_vectors(db, table_name, items)
        similar_vectors = find_similar_vectors(db, table_name, query_vector)

        print("\n\nInserted embeddings:")
        for row in items:
            print(row)
        print("\nQuery Vector:")
        print(query_vector)



        print("\nSimilar vectors found:")
        print("----------------------")
        for row_id, distance in similar_vectors:
            print(f"ID: {row_id}, Distance: {distance}")


        print('\n\n'+"="*20 +"Distance Metrics" + "="*20)
        print("\nL2 (Euclidean Distance) and Cosine Distance:")
        custom_results = db.execute_fetchall(
            f"""
            SELECT
                a.rowid AS vector1_id,
                b.rowid AS vector2_id,
                vec_distance_l2(a.embedding, b.embedding) AS l2_distance,
                vec_distance_cosine(a.embedding, b.embedding) AS cosine_distance
            FROM {table_name} a
            CROSS JOIN {table_name} b
            WHERE a.rowid < b.rowid
            ORDER BY a.rowid, b.rowid
            LIMIT 6
            """
        )
        for row in custom_results:
            print(row)
```

```python
if __name__ == "__main__":
    main()
```

```
@NikhilSharma ➜ C:\..\dev-genAI (base 3.12.7) 298ms
[100%] python .\06_sqlite_vec.py
sqlite_version=3.45.3, vec_version=v0.1.6


Inserted embeddings:
(1, [0.1, 0.1, 0.1, 0.1])
(2, [0.2, 0.2, 0.2, 0.2])
(3, [0.3, 0.3, 0.3, 0.3])
(4, [0.4, 0.4, 0.4, 0.4])
(5, [0.5, 0.5, 0.5, 0.5])

Query Vector:
[0.3, 0.3, 0.3, 0.3]

Similar vectors found:
----------------------
ID: 3, Distance: 0.0
ID: 4, Distance: 0.19999998807907104
ID: 2, Distance: 0.20000001788139343


==================Distance Metrics====================

L2 (Euclidean Distance) and Cosine Distance:
(1, 2, 0.20000000298023224, 0.0)
(1, 3, 0.40000003576278687, 1.552204231813903e-08)
(1, 4, 0.6000000238418579, 0.0)
(1, 5, 0.800000011920929, 2.0489094865183688e-08)
(2, 3, 0.20000001788139343, 1.552204231813903e-08)
(2, 4, 0.4000000059604645, 0.0)
```