SQLiteVec Hybrid Search Demo

This notebook demonstrates how to implement hybrid search using SQLiteVec with Microsoft Semantic Kernel, combining full-text search (FTS5) and vector search for enhanced retrieval capabilities.

Overview

- **Vector Storage**: Store and retrieve embeddings using SQLiteVec
- **Full-Text Search**: Traditional keyword-based search with FTS5
- **Hybrid Search**: Combine both approaches for optimal results
- **Dependency Injection**: Clean architecture with DI container
- **Data Models**: Use VectorData attributes for schema definition

1. Package Installation

Installing the required packages for SQLiteVec, vector data operations, and Semantic Kernel integration.

```
#r "nuget: Microsoft.SemanticKernel"
#r "nuget: Microsoft.SemanticKernel.Connectors.InMemory, *-*"
#r "nuget: Microsoft.SemanticKernel.Connectors.SqliteVec, *-*"
#r "nuget: Microsoft.SemanticKernel.Connectors.OpenAI"
#r "nuget: Microsoft.Extensions.VectorData.Abstractions,*-*"
#r "nuget: Microsoft.Data.Sqlite"

#r "nuget: Microsoft.Extensions.DependencyInjection"
#r "nuget: Microsoft.Extensions.Hosting"
#r "nuget: Microsoft.Extensions.Logging"
```

2. Imports and Configuration

Importing necessary namespaces for SQLiteVec operations, vector data handling, and dependency injection.

```
#pragma warning disable SKEXP0001 // SQLiteVec is experimental
#pragma warning disable SKEXP0020 // Vector store is experimental

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
```

```
using Microsoft.Data.Sqlite;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.VectorData;

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.InMemory;
using Microsoft.SemanticKernel.Connectors.SqliteVec;
using Microsoft.SemanticKernel.Embeddings;
```

3. Data Model Definition

Defining the Hotel data model with VectorData attributes for SQLiteVec integration. This model supports both traditional data storage and vector embeddings.

Key Features:

- **VectorStoreKey**: Unique identifier for each record
- **VectorStoreData**: Regular data fields with optional storage mapping
- **VectorStoreVector**: Vector embedding with dimension and distance function configuration

```
public class Hotel
{
    [VectorStoreKey]
    public int HotelId { get; set; }

    [VectorStoreData]
    public string? HotelName { get; set; }

    [VectorStoreData]
    public string? Description { get; set; }

    [VectorStoreData]
    public string? Location { get; set; }

    [VectorStoreData]
    public double Rating { get; set; }

    [VectorStoreVector(Dimensions: 1536, DistanceFunction =
DistanceFunction.CosineDistance)]
    public ReadOnlyMemory<float>? NameEmbedding { get; set; }

    [VectorStoreVector(Dimensions: 1536, DistanceFunction =
DistanceFunction.CosineDistance)]
    public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
}
```

4. Hotel Search Service

Creating a service that encapsulates hybrid search functionality, combining FTS5 and vector search capabilities with Reciprocal Rank Fusion (RRF) for optimal result ranking.

Service Features:

- **Dependency Injection**: Clean separation of concerns
- **Hybrid Search**: Combines keyword and semantic search using RRF algorithm
- **Reciprocal Rank Fusion**: Scientifically proven method for merging ranked lists
- **Weighted Scoring**: Configurable weights for keyword (60%) vs vector (40%) search
- **Detailed Analytics**: Comprehensive ranking information and search type classification
- **Async Operations**: Non-blocking database operations
- **Performance Optimized**: Intelligent search limits for better fusion quality

RRF Algorithm:

- **Formula**: `score = weight / (k + rank)` where k=60 (standard RRF constant)
- **Rank-based Fusion**: Higher-ranked results from either method receive more weight
- **Missing Result Handling**: Works effectively even when queries don't match in one search method
- **Search Type Classification**: Results tagged as Hybrid, Keyword-only, or Vector-only

```
public interface IHotelSearchService
{
    Task<SearchResult<Hotel>> SearchHotelsAsync(string query,
SearchOptions options = default, CancellationToken cancellationToken =
default);

    Task<IReadOnlyList<Hotel>> GetAllHotelsAsync(CancellationToken
cancellationToken = default);
    Task InitializeDataAsync(CancellationToken cancellationToken);
}

public record SearchConfiguration
{
    public double KeywordWeight { get; init; } = 0.6;
    public double VectorWeight { get; init; } = 0.4;
    public int RrfConstant { get; init; } = 60;
    public int SearchMultiplier { get; init; } = 2;
    public int MinSearchLimit { get; init; } = 20;
}

public record SearchMetrics
{
    public TimeSpan SearchDuration { get; init; }
    public int KeywordResults { get; init; }
    public int VectorResults { get; init; }
    public int HybridResults { get; init; }
    public string QueryType { get; init; } = string.Empty;
```

```
}

public record SearchResult<T>
{
    public IReadOnlyList<T> Items { get; init; } = Array.Empty<T>();
    public SearchMetrics? Metrics { get; init; }
    public int TotalCount => Items.Count;
}

public record SearchOptions
{
    public int MaxResults { get; init; } = 10;
    public double? MinRating { get; init; }
    public string? Location { get; init; }
    public bool IncludeMetrics { get; init; } = false;
}

public class RRFScore
{
    public Hotel Hotel { get; set; }
    public int? KeywordRank { get; set; }
    public int? VectorRank { get; set; }
    public double KeywordScore { get; set; }
    public double VectorScore { get; set; }
    public double TotalScore { get; set; }
    public double VectorSimilarity { get; set; }
}

public class RankedResult
{
    public Hotel Hotel { get; set; }
    public double RRFScore { get; set; }
    public int? KeywordRank { get; set; }
    public int? VectorRank { get; set; }
    public double KeywordScore { get; set; }
    public double VectorScore { get; set; }
    public double VectorSimilarity { get; set; }
    public string SearchType { get; set; }
}

public sealed class HotelSearchService : IHotelSearchService
{
    private readonly SqliteCollection<int, Hotel> _vectorCollection;
    private readonly IEmbeddingGenerator<string, Embedding<float>>
_embeddingService;
    private readonly string _connectionString;
    private readonly ILogger<HotelSearchService> _logger;
    private readonly SearchConfiguration _config;

    public HotelSearchService(
```

```
        SqliteCollection<int, Hotel> vectorCollection,
        IEmbeddingGenerator<string, Embedding<float>>
embeddingService,
        string connectionString,
        ILogger<HotelSearchService> logger,
        SearchConfiguration? config = null)
    {
        _vectorCollection = vectorCollection ?? throw new
ArgumentNullException(nameof(vectorCollection));
        _embeddingService = embeddingService ?? throw new
ArgumentNullException(nameof(embeddingService));
        _connectionString = connectionString ?? throw new
ArgumentNullException(nameof(connectionString));
        _logger = logger ?? throw new
ArgumentNullException(nameof(logger));
        _config = config ?? new SearchConfiguration();
    }

    public async Task<SearchResult<Hotel>> SearchHotelsAsync(string
query,  SearchOptions options = default,  CancellationToken
cancellationToken = default)
    {
        ArgumentException.ThrowIfNullOrWhiteSpace(query);

        options ??= new SearchOptions();
        var stopwatch = Stopwatch.StartNew();

        try
        {
            var (keywordResults, vectorResults) = await
ExecuteParallelSearchAsync(query, options, cancellationToken);

            var rankedResults =
ApplyReciprocalRankFusion(keywordResults, vectorResults, options);

            var metrics = options.IncludeMetrics ? new SearchMetrics
            {
                SearchDuration = stopwatch.Elapsed,
                KeywordResults = keywordResults.Count,
                VectorResults = vectorResults.Count,
                HybridResults = rankedResults.Count,
                QueryType = DetermineQueryType(query)
            } : null;

            return new SearchResult<Hotel>
            {
                Items = rankedResults.Select(r => r.Hotel).ToList(),
                Metrics = metrics
            };
        }
```

```
        catch (Exception ex)
        {
            _logger.LogError(ex, "Search failed for query: {Query}",
query);
            throw;
        }
        finally
        {
            stopwatch.Stop();
            Debug.WriteLine($"Search duration: {stopwatch.Elapsed}");
        }
    }

    public async Task<IReadOnlyList<Hotel>> KeywordSearchAsync(string
query, int maxResults = 10, CancellationToken cancellationToken =
default)
    {
        var results = await PerformKeywordSearchAsync(query,
maxResults, cancellationToken);
        return results;
    }

    public async Task<IReadOnlyList<Hotel>>
GetAllHotelsAsync(CancellationToken cancellationToken = default)
    {
        const string sql = "SELECT HotelId, HotelName, Description,
Location, Rating FROM hotels_fts";

        Debug.WriteLine($"Getting all hotels from
{_connectionString}");
        Stopwatch stopwatch = Stopwatch.StartNew();

        using var connection = new
SqliteConnection(_connectionString);
        await connection.OpenAsync(cancellationToken);

        using var command = connection.CreateCommand();
        command.CommandText = sql;

        var results = new List<Hotel>();
        using var reader = await
command.ExecuteReaderAsync(cancellationToken);

        while (await reader.ReadAsync(cancellationToken))
        {
            results.Add(new Hotel
            {
                HotelId = reader.GetInt32(0),
                HotelName = reader.GetString(1),
                Description = reader.GetString(2),
```

```
                Location = reader.GetString(3),
                Rating = reader.GetDouble(4)
            });
        }

        stopwatch.Stop();
        Debug.WriteLine($"Found {results.Count} hotels in
{stopwatch.Elapsed}");

        return results;
    }

    public async Task InitializeDataAsync(CancellationToken
cancellationToken = default)
    {
        await CreateFtsTableAsync(cancellationToken);
        await _vectorCollection.EnsureCollectionExistsAsync();
        await SeedDataAsync(cancellationToken);
    }

    private async Task<(List<Hotel> keyword,
List<VectorSearchResult<Hotel>> vector)> ExecuteParallelSearchAsync(
        string query,
        SearchOptions options,
        CancellationToken cancellationToken)
    {
        var searchLimit = Math.Max(options.MaxResults *
_config.SearchMultiplier, _config.MinSearchLimit);

        var keywordTask = PerformKeywordSearchAsync(query,
searchLimit, cancellationToken);
        var vectorTask = GenerateEmbeddingAndSearchAsync(query,
searchLimit, cancellationToken);

        await Task.WhenAll(keywordTask, vectorTask);

        return (keywordTask.Result, vectorTask.Result);
    }

    private async Task<List<VectorSearchResult<Hotel>>>
GenerateEmbeddingAndSearchAsync(
        string query,
        int limit,
        CancellationToken cancellationToken)
    {
        var embedding = await _embeddingService.GenerateAsync(query);

        return await PerformVectorSearchAsync(embedding, limit,
cancellationToken);
    }
```

```
    private List<RankedResult> ApplyReciprocalRankFusion(
        List<Hotel> keywordResults,
        List<VectorSearchResult<Hotel>> vectorResults,
        SearchOptions options)
    {
        var rrfScores = new Dictionary<int, RRFScore>();

        ProcessKeywordResults(keywordResults, rrfScores);
        ProcessVectorResults(vectorResults, rrfScores);

        var results = rrfScores.Values
            .Where(x => PassesFilters(x.Hotel, options))
            .OrderByDescending(x => x.TotalScore)
            .Take(options.MaxResults)
            .Select(CreateRankedResult)
            .ToList();

        LogSearchResults(results);
        return results;
    }

    private void ProcessKeywordResults(List<Hotel> results,
Dictionary<int, RRFScore> scores)
    {
        Debug.WriteLine($"Processing {results.Count} keyword
results");
        for (int i = 0; i < results.Count; i++)
        {
            var hotel = results[i];
            var rank = i + 1;
            var score = _config.KeywordWeight / (_config.RrfConstant +
rank);

            UpdateOrCreateScore(scores, hotel, score: score,
keywordRank: rank);
        }
        Debug.WriteLine($"Processed {results.Count} keyword results");
    }

    private void ProcessVectorResults(List<VectorSearchResult<Hotel>>
results, Dictionary<int, RRFScore> scores)
    {
        Debug.WriteLine($"Processing {results.Count} vector results");
        for (int i = 0; i < results.Count; i++)
        {
            var result = results[i];
            var rank = i + 1;
            var score = _config.VectorWeight / (_config.RrfConstant +
rank);
```

```
            UpdateOrCreateScore(scores, result.Record, score: score,
vectorRank: rank, similarity: result.Score ?? 0.0);
        }
        Debug.WriteLine($"Processed {results.Count} vector results");
    }

    private static void UpdateOrCreateScore(
        Dictionary<int, RRFScore> scores,
        Hotel hotel,
        double score,
        int? keywordRank = null,
        int? vectorRank = null,
        double similarity = 0.0)
    {
        if (scores.TryGetValue(hotel.HotelId, out var existing))
        {
            existing.TotalScore += score;
            existing.KeywordRank ??= keywordRank;
            existing.VectorRank ??= vectorRank;
            existing.KeywordScore += keywordRank.HasValue ? score : 0;
            existing.VectorScore += vectorRank.HasValue ? score : 0;
            existing.VectorSimilarity =
Math.Max(existing.VectorSimilarity, similarity);
        }
        else
        {
            scores[hotel.HotelId] = new RRFScore
            {
                Hotel = hotel,
                KeywordRank = keywordRank,
                VectorRank = vectorRank,
                KeywordScore = keywordRank.HasValue ? score : 0,
                VectorScore = vectorRank.HasValue ? score : 0,
                TotalScore = score,
                VectorSimilarity = similarity
            };
        }
    }

    private static bool PassesFilters(Hotel hotel, SearchOptions
options) =>
        (options.MinRating is null || hotel.Rating >=
options.MinRating) &&
        (string.IsNullOrEmpty(options.Location) ||
hotel.Location?.Contains(options.Location,
StringComparison.OrdinalIgnoreCase) == true);

    private static RankedResult CreateRankedResult(RRFScore score) =>
new()
```

```
    {
        Hotel = score.Hotel,
        RRFScore = score.TotalScore,
        KeywordRank = score.KeywordRank,
        VectorRank = score.VectorRank,
        KeywordScore = score.KeywordScore,
        VectorScore = score.VectorScore,
        VectorSimilarity = score.VectorSimilarity,
        SearchType = GetSearchType(score.KeywordRank,
score.VectorRank)
    };

    private static string GetSearchType(int? keywordRank, int?
vectorRank) =>
        (keywordRank.HasValue, vectorRank.HasValue) switch
        {
            (true, true) => "Hybrid",
            (true, false) => "Keyword",
            (false, true) => "Vector",
            _ => "Unknown"
        };

    private string DetermineQueryType(string query)
    {
        // Simple heuristic - could be enhanced with NLP
        var words = query.Split(' ',
StringSplitOptions.RemoveEmptyEntries);
        return words.Length switch
        {
            1 => "Single-term",
            2 => "Two-term",
            _ => "Multi-term"
        };
    }

    private async Task<List<Hotel>> PerformKeywordSearchAsync(string
query, int limit, CancellationToken cancellationToken)
    {
        const string sql = """
            SELECT HotelId, HotelName, Description, Location, Rating
            FROM hotels_fts
            WHERE hotels_fts MATCH $query
            LIMIT $limit
            """;

        Stopwatch stopwatch = Stopwatch.StartNew();
        Debug.WriteLine($"Performing keyword search for {query} with
limit {limit}");

        using var connection = new
```

```
SqliteConnection(_connectionString);
        await connection.OpenAsync(cancellationToken);

        using var command = connection.CreateCommand();
        command.CommandText = sql;
        command.Parameters.AddWithValue("$query", query);
        command.Parameters.AddWithValue("$limit", limit);

        var results = new List<Hotel>();
        using var reader = await
command.ExecuteReaderAsync(cancellationToken);

        while (await reader.ReadAsync(cancellationToken))
        {
            results.Add(new Hotel
            {
                HotelId = reader.GetInt32(0),
                HotelName = reader.GetString(1),
                Description = reader.GetString(2),
                Location = reader.GetString(3),
                Rating = reader.GetDouble(4)
            });
        }

        stopwatch.Stop();
        Debug.WriteLine($"Keyword search completed in
{stopwatch.Elapsed}");

        return results;
    }

    private async Task<List<VectorSearchResult<Hotel>>>
PerformVectorSearchAsync(
        Embedding<float> queryEmbedding,
        int limit,
        CancellationToken cancellationToken)
    {
        var searchOptions = new VectorSearchOptions<Hotel>
        {
            VectorProperty = static v => v.DescriptionEmbedding
        };

        var results = new List<VectorSearchResult<Hotel>>();

        Stopwatch stopwatch = Stopwatch.StartNew();
        Debug.WriteLine($"Performing vector search for
{queryEmbedding} with limit {limit}");
        await foreach (var result in
_vectorCollection.SearchAsync(queryEmbedding, limit,
searchOptions).WithCancellation(cancellationToken))
```

```
        {
            results.Add(result);
        }
        stopwatch.Stop();
        Debug.WriteLine($"Vector search completed in
{stopwatch.Elapsed}");

        return results;
    }

    public async Task<IReadOnlyList<Hotel>> VectorSearchAsync(string
query, int maxResults = 10, CancellationToken cancellationToken =
default)
    {
        var embedding = await _embeddingService.GenerateAsync(query);

        var results = await PerformVectorSearchAsync(embedding,
maxResults, cancellationToken);

        return results.Select(r => r.Record).ToList();
    }

    private async Task CreateFtsTableAsync(CancellationToken
cancellationToken)
    {
        const string sql = """
            CREATE VIRTUAL TABLE IF NOT EXISTS hotels_fts USING fts5(
                HotelId UNINDEXED,
                HotelName,
                Description,
                Location,
                Rating UNINDEXED
            )
            """;

        Stopwatch stopwatch = Stopwatch.StartNew();
        Debug.WriteLine($"Creating FTS table");

        using var connection = new
SqliteConnection(_connectionString);
        await connection.OpenAsync(cancellationToken);

        using var command = connection.CreateCommand();
        command.CommandText = sql;
        await command.ExecuteNonQueryAsync(cancellationToken);

        stopwatch.Stop();
        Debug.WriteLine($"FTS table created in {stopwatch.Elapsed}");
    }
```

```
    private async Task SeedDataAsync(CancellationToken
cancellationToken)
    {
        var sampleHotels = new[]
        {
            new Hotel { HotelId = 1, HotelName = "Grand Luxury
Resort", Description = "Luxury resort with spa, multiple pools, and
fine dining", Location = "Miami Beach", Rating = 4.8 },
            new Hotel { HotelId = 2, HotelName = "Mountain View
Lodge", Description = "Cozy mountain lodge with hiking trails and
scenic views", Location = "Colorado", Rating = 4.5 },
            new Hotel { HotelId = 3, HotelName = "Urban Business
Hotel", Description = "Modern business hotel in downtown with
conference facilities", Location = "New York", Rating = 4.2 },
            new Hotel { HotelId = 4, HotelName = "Seaside Wellness
Retreat", Description = "Wellness retreat with spa treatments and
ocean views", Location = "California", Rating = 4.7 },
            new Hotel { HotelId = 5, HotelName = "Historic Boutique
Inn", Description = "Charming historic inn with antique furnishings
and local charm", Location = "Vermont", Rating = 4.3 }
        };

        // Generate embeddings and store in vector collection
        var embeddingTasks = sampleHotels.Select(async hotel =>
        {
            var embedding = await
_embeddingService.GenerateAsync(hotel.Description);
            hotel.DescriptionEmbedding = embedding.Vector;
            await _vectorCollection.UpsertAsync(hotel)
                .ConfigureAwait(false);
            return hotel;
        });

        await Task.WhenAll(embeddingTasks)
            .ConfigureAwait(false);

        // Store in FTS table
        using var connection = new
SqliteConnection(_connectionString);
        await connection.OpenAsync(cancellationToken)
            .ConfigureAwait(false);

        const string insertSql = """
            INSERT OR REPLACE INTO hotels_fts (HotelId, HotelName,
Description, Location, Rating)
            VALUES ($id, $name, $description, $location, $rating)
            """;

        foreach (var hotel in sampleHotels)
        {
```

```
            using var command = connection.CreateCommand();
            command.CommandText = insertSql;
            command.Parameters.AddWithValue("$id", hotel.HotelId);
            command.Parameters.AddWithValue("$name", hotel.HotelName);
            command.Parameters.AddWithValue("$description",
hotel.Description);
            command.Parameters.AddWithValue("$location",
hotel.Location);
            command.Parameters.AddWithValue("$rating", hotel.Rating);

            await command.ExecuteNonQueryAsync(cancellationToken)
                .ConfigureAwait(false);
        }
    }

    private void LogSearchResults(List<RankedResult> results)
    {
        if (!_logger.IsEnabled(LogLevel.Information)) return;

        _logger.LogInformation("RRF Fusion Results:");
        foreach (var result in results.Take(3))
        {
            _logger.LogInformation(
                "  {HotelName}: RRF={RRFScore:F4},
KW_Rank={KeywordRank}, Vec_Rank={VectorRank}, Type={SearchType}",
                result.Hotel.HotelName,
                result.RRFScore,
                result.KeywordRank?.ToString() ?? "N/A",
                result.VectorRank?.ToString() ?? "N/A",
                result.SearchType);
        }
    }
}
```

5. Dependency Injection Setup

Configuring the DI container with all required services for SQLiteVec operations, following the same pattern as the Semantic Kernel agents example.

Services Registered:

- **SQLiteVec Vector Store**: For vector storage and retrieval
- **OpenAI Embedding Service**: For generating text embeddings
- **Hotel Search Service**: Custom service for hybrid search
- **Logging**: For monitoring and debugging

```
#pragma warning disable SKEXP0010 // SQLiteVec is experimental

var connectionString = "Data Source=hotels.db";
var collectionName = "hotels";
```

```
var builder = Host.CreateDefaultBuilder([])
    .ConfigureServices((context, services) =>
    {
        services.AddLogging(builder =>
builder.AddConsole().SetMinimumLevel(LogLevel.Information));

        services.AddOpenAIChatCompletion(
            "gpt-4o-mini",
            Environment.GetEnvironmentVariable("OPENAI_API_KEY")
        );

        services.AddOpenAIEmbeddingGenerator(
            "text-embedding-3-small",
            Environment.GetEnvironmentVariable("OPENAI_API_KEY")
        );

        services.AddSingleton(provider =>
        {
            var vectorStore = new SqliteCollection<int,
Hotel>(connectionString, collectionName);
            return vectorStore;
        });

        services.AddSingleton<IHotelSearchService>(provider =>
        {
            return new HotelSearchService(
                provider.GetRequiredService<SqliteCollection<int,
Hotel>>(),

provider.GetRequiredService<IEmbeddingGenerator<string,
Embedding<float>>>(),
                connectionString,

provider.GetRequiredService<ILogger<HotelSearchService>>()
            );
        });
    });

IHost host = builder.Build();
IServiceProvider serviceProvider = host.Services;

var hotelSearchService =
serviceProvider.GetRequiredService<IHotelSearchService>();
var logger =
serviceProvider.GetRequiredService<ILogger<IHotelSearchService>>();

Console.WriteLine("□ Dependency injection configured successfully");
Console.WriteLine($"□ Database: {connectionString}");
Console.WriteLine($"□ Collection: {collectionName}");
```

```
 Dependency injection configured successfully
 Database: Data Source=hotels.db
 Collection: hotels
```

6. Data Initialization

Setting up the database schema, creating FTS5 tables, and seeding sample data with embeddings.

Initialization Steps:

1. **Vector Collection**: Create SQLiteVec collection for embeddings
2. **FTS5 Table**: Create full-text search table for keyword matching
3. **Sample Data**: Insert hotels with generated embeddings
4. **Dual Storage**: Data stored in both vector and FTS tables

```
Console.WriteLine(" Initializing database and sample data...");

try
{
    await
hotelSearchService.InitializeDataAsync(CancellationToken.None);
    Console.WriteLine(" Database initialized successfully");

    var allHotels = await hotelSearchService.GetAllHotelsAsync();
    Console.WriteLine($" Total hotels in database:
{allHotels.Count()}");

    Console.WriteLine("\n Sample Hotels:");
    foreach (var hotel in allHotels.Take(3))
    {
        Console.WriteLine($"  • {hotel.HotelName} ({hotel.Rating}★) -
{hotel.Location}");
        Console.WriteLine($"    {hotel.Description}");
    }
}
catch (Exception ex)
{
    Console.WriteLine($" Error during initialization: {ex.Message}");
    throw;
}
```

```
 Initializing database and sample data...
 Database initialized successfully
 Total hotels in database: 30

 Sample Hotels:
  • Grand Luxury Resort (4.8★) - Miami Beach
    Luxury resort with spa, multiple pools, and fine dining
  • Mountain View Lodge (4.5★) - Colorado
```

```
    Cozy mountain lodge with hiking trails and scenic views
  • Urban Business Hotel (4.2★) - New York
    Modern business hotel in downtown with conference facilities
```

7. Hybrid Search Demonstration

Demonstrating the hybrid search capabilities with various query types to show how keyword and semantic search complement each other.

Search Features:

- **Keyword Priority**: Exact matches ranked higher
- **Semantic Understanding**: Finds conceptually similar results
- **Result Fusion**: Combines both approaches intelligently
- **Relevance Scoring**: Orders results by relevance

```
var searchQueries = new[]
{
    "luxury spa resort",
    "mountain hiking",
    "business conference",
    "relaxation wellness",
    "historic charm"
};

foreach (var query in searchQueries)
{
    Console.WriteLine($"\n Searching for: '{query}'");
    Console.WriteLine(new string('=', 50));

    try
    {
        var results = await
hotelSearchService.SearchHotelsAsync(query, new SearchOptions
{ MaxResults = 5 });

        if (results.Items.Any())
        {
            var resultsList = results.Items;
            Console.WriteLine($" Found {resultsList.Count()}
result(s):");

            for (int i = 0; i < resultsList.Count; i++)
            {
                var hotel = resultsList[i];
                Console.WriteLine($"\n  {i + 1}. {hotel.HotelName}
({hotel.Rating}★)");
                Console.WriteLine($"     {hotel.Location}");
                Console.WriteLine($"     {hotel.Description}");
```

```
            }
        }
        else
        {
            Console.WriteLine("□ No results found");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"□ Search error: {ex.Message}");
    }
}


□ Searching for: 'luxury spa resort'
==================================================
□ Found 5 result(s):

  1. Grand Luxury Resort (4.8★)
     □ Miami Beach
     □ Luxury resort with spa, multiple pools, and fine dining

  2. Seaside Wellness Retreat (4.7★)
     □ California
     □ Wellness retreat with spa treatments and ocean views

  3. Mountain View Lodge (4.5★)
     □ Colorado
     □ Cozy mountain lodge with hiking trails and scenic views

  4. Urban Business Hotel (4.2★)
     □ New York
     □ Modern business hotel in downtown with conference facilities

  5. Historic Boutique Inn (4.3★)
     □ Vermont
     □ Charming historic inn with antique furnishings and local charm

□ Searching for: 'mountain hiking'
==================================================
□ Found 5 result(s):

  1. Mountain View Lodge (4.5★)
     □ Colorado
     □ Cozy mountain lodge with hiking trails and scenic views

  2. Seaside Wellness Retreat (4.7★)
     □ California
     □ Wellness retreat with spa treatments and ocean views
```

```
  3. Grand Luxury Resort (4.8★)
     Miami Beach
     Luxury resort with spa, multiple pools, and fine dining

  4. Historic Boutique Inn (4.3★)
     Vermont
     Charming historic inn with antique furnishings and local charm

  5. Urban Business Hotel (4.2★)
     New York
     Modern business hotel in downtown with conference facilities

 Searching for: 'business conference'
==================================================
 Found 5 result(s):

  1. Urban Business Hotel (4.2★)
     New York
     Modern business hotel in downtown with conference facilities

  2. Seaside Wellness Retreat (4.7★)
     California
     Wellness retreat with spa treatments and ocean views

  3. Grand Luxury Resort (4.8★)
     Miami Beach
     Luxury resort with spa, multiple pools, and fine dining

  4. Mountain View Lodge (4.5★)
     Colorado
     Cozy mountain lodge with hiking trails and scenic views

  5. Historic Boutique Inn (4.3★)
     Vermont
     Charming historic inn with antique furnishings and local charm

 Searching for: 'relaxation wellness'
==================================================
 Found 5 result(s):

  1. Seaside Wellness Retreat (4.7★)
     California
     Wellness retreat with spa treatments and ocean views

  2. Grand Luxury Resort (4.8★)
     Miami Beach
     Luxury resort with spa, multiple pools, and fine dining

  3. Mountain View Lodge (4.5★)
     Colorado
```

```
     Cozy mountain lodge with hiking trails and scenic views

  4. Historic Boutique Inn (4.3★)
     Vermont
     Charming historic inn with antique furnishings and local charm

  5. Urban Business Hotel (4.2★)
     New York
     Modern business hotel in downtown with conference facilities

 Searching for: 'historic charm'
==================================================
 Found 5 result(s):

  1. Historic Boutique Inn (4.3★)
     Vermont
     Charming historic inn with antique furnishings and local charm

  2. Mountain View Lodge (4.5★)
     Colorado
     Cozy mountain lodge with hiking trails and scenic views

  3. Urban Business Hotel (4.2★)
     New York
     Modern business hotel in downtown with conference facilities

  4. Grand Luxury Resort (4.8★)
     Miami Beach
     Luxury resort with spa, multiple pools, and fine dining

  5. Seaside Wellness Retreat (4.7★)
     California
     Wellness retreat with spa treatments and ocean views
```

8. Performance Analysis

Analyzing the performance characteristics of different search approaches to understand when to use each method.

Performance Metrics:

- **Search Time**: Measure execution time for different approaches
- **Result Quality**: Compare relevance of results
- **Resource Usage**: Monitor memory and CPU usage
- **Scalability**: Understand performance with larger datasets

```
var testQuery = "luxury spa resort";
var iterations = 5;

Console.WriteLine($"\n⚡ Performance Analysis - Query: '{testQuery}'");
```

```
Console.WriteLine(new string('=', 60));

var hybridTimes = new List<long>();
for (int i = 0; i < iterations; i++)
{
    var stopwatch = Stopwatch.StartNew();
    var results = await
hotelSearchService.SearchHotelsAsync(testQuery, maxResults: 10);
    stopwatch.Stop();
    hybridTimes.Add(stopwatch.ElapsedMilliseconds);
}

var avgHybridTime = hybridTimes.Average();
var minHybridTime = hybridTimes.Min();
var maxHybridTime = hybridTimes.Max();

Console.WriteLine($"□ Hybrid Search Performance ({iterations}
iterations):");
Console.WriteLine($"   Average: {avgHybridTime:F2} ms");
Console.WriteLine($"   Min: {minHybridTime} ms");
Console.WriteLine($"   Max: {maxHybridTime} ms");

GC.Collect();
GC.WaitForPendingFinalizers();
var memoryBefore = GC.GetTotalMemory(false);

var memoryResults = await
hotelSearchService.SearchHotelsAsync(testQuery, maxResults: 10);

var memoryAfter = GC.GetTotalMemory(false);
var memoryUsed = memoryAfter - memoryBefore;

Console.WriteLine($"□ Memory Usage:");
Console.WriteLine($"   Before: {memoryBefore / 1024:N0} KB");
Console.WriteLine($"   After: {memoryAfter / 1024:N0} KB");
Console.WriteLine($"   Used: {memoryUsed / 1024:N0} KB");

Console.WriteLine($"\n□ Search Quality Analysis:");
var qualityResults = await
hotelSearchService.SearchHotelsAsync(testQuery, maxResults: 5);
var qualityList = await qualityResults.ToListAsync();

Console.WriteLine($"   Results returned: {qualityList.Count()}");
Console.WriteLine($"   Top result:
{qualityList.FirstOrDefault()?.HotelName ?? "None"}");
Console.WriteLine($"   Average rating: {qualityList.Average(h =>
h.Rating):F1}★");


⚡ Performance Analysis - Query: 'luxury spa resort'
```

```
============================================================
 Hybrid Search Performance (5 iterations):
   Average: 264.00 ms
   Min: 223 ms
   Max: 325 ms
 Memory Usage:
   Before: 52,154 KB
   After: 52,325 KB
   Used: 171 KB

 Search Quality Analysis:
   Results returned: 5
   Top result: Grand Luxury Resort
   Average rating: 4.5★
```

9. Advanced Features Demo

Demonstrating advanced SQLiteVec features including custom distance functions, filtering, and batch operations.

Advanced Capabilities:

- **Custom Filtering**: Filter results by rating, location, etc.
- **Batch Operations**: Process multiple embeddings efficiently
- **Distance Functions**: Compare different similarity metrics
- **Metadata Filtering**: Combine vector search with traditional filters

```
Console.WriteLine("\n Advanced Features Demo");
Console.WriteLine(new string('=', 50));

// 1. Filtered search - high-rated hotels only
Console.WriteLine("\n High-rated hotels (4.5+ stars) with
'luxury':");
try
{
    Console.WriteLine("Searching for luxury hotels...");
    var luxuryQuery = "luxury amenities";
    var allResults = await
hotelSearchService.SearchHotelsAsync(luxuryQuery, maxResults: 10);
    Console.WriteLine($"Found {allResults.Count()} results");
    var highRatedResults = allResults.Where(h => h.Rating >= 4.5);

    foreach (var hotel in highRatedResults)
    {
        Console.WriteLine($"  • {hotel.HotelName} ({hotel.Rating}★) -
{hotel.Location}");
    }

    if (!highRatedResults.Any())
```

```
    {
        Console.WriteLine("  No high-rated luxury hotels found");
    }
}
catch (Exception ex)
{
    Console.WriteLine($" Error in filtered search: {ex.Message}");
}

// 2. Location-based filtering
Console.WriteLine("\n Hotels in specific locations:");
var locations = new[] { "California", "Colorado", "New York" };

foreach (var location in locations)
{
    var locationResults = await
hotelSearchService.GetAllHotelsAsync();
    var locationHotels = locationResults.Where(h =>
h.Location?.Contains(location) == true);

    Console.WriteLine($"\n  {location}:");
    foreach (var hotel in locationHotels)
    {
        Console.WriteLine($"    - {hotel.HotelName}
({hotel.Rating}★)");
    }
}

// 3. Batch embedding demonstration
Console.WriteLine("\n Batch processing demo:");
var embeddingService =
serviceProvider.GetRequiredService<IEmbeddingGenerator<string,
Embedding<float>>>();

var batchQueries = new[]
{
    "ocean view resort",
    "mountain cabin retreat",
    "city business hotel"
};

var batchStopwatch = Stopwatch.StartNew();
var batchEmbeddings = new List<ReadOnlyMemory<float>>();

foreach (var query in batchQueries)
{
    var embedding = await embeddingService.GenerateVectorAsync(query);
    batchEmbeddings.Add(embedding);
}
```

```
batchStopwatch.Stop();
Console.WriteLine($"  Generated {batchEmbeddings.Count} embeddings in
{batchStopwatch.ElapsedMilliseconds} ms");
Console.WriteLine($"  Average: {batchStopwatch.ElapsedMilliseconds /
(double)batchEmbeddings.Count:F1} ms per embedding");


 Advanced Features Demo
==================================================

 High-rated hotels (4.5+ stars) with 'luxury':
Searching for luxury hotels...
Found 5 results
  • Grand Luxury Resort (4.8★) - Miami Beach
  • Seaside Wellness Retreat (4.7★) - California
  • Mountain View Lodge (4.5★) - Colorado

 Hotels in specific locations:

  California:
    - Seaside Wellness Retreat (4.7★)
    - Seaside Wellness Retreat (4.7★)
    - Seaside Wellness Retreat (4.7★)
    - Seaside Wellness Retreat (4.7★)
    - Seaside Wellness Retreat (4.7★)

  Colorado:
    - Mountain View Lodge (4.5★)
    - Mountain View Lodge (4.5★)
    - Mountain View Lodge (4.5★)
    - Mountain View Lodge (4.5★)
    - Mountain View Lodge (4.5★)

  New York:
    - Urban Business Hotel (4.2★)
    - Urban Business Hotel (4.2★)
    - Urban Business Hotel (4.2★)
    - Urban Business Hotel (4.2★)
    - Urban Business Hotel (4.2★)

 Batch processing demo:
  Generated 3 embeddings in 2664 ms
  Average: 888.0 ms per embedding
```

10. Cleanup and Best Practices

Demonstrating proper resource cleanup and sharing best practices for production deployments.

Best Practices:

- **Connection Management**: Proper disposal of database connections

- **Error Handling**: Robust error handling patterns
- **Resource Cleanup**: Clean shutdown of services
- **Performance Monitoring**: Track key metrics in production

```
Console.WriteLine("\n Cleanup and Best Practices");
Console.WriteLine(new string('=', 50));

// Best practice: Always dispose of resources properly
Console.WriteLine(" Resource Management:");
Console.WriteLine("  • Database connections are auto-disposed with
'using' statements");
Console.WriteLine("  • Vector collections are managed by DI
container");
Console.WriteLine("  • Embedding service is properly registered as
singleton");

// Performance recommendations
Console.WriteLine(" Performance Recommendations:");
Console.WriteLine("  • Use connection pooling for high-throughput
scenarios");
Console.WriteLine("  • Cache frequently used embeddings");
Console.WriteLine("  • Consider batch operations for bulk inserts");
Console.WriteLine("  • Monitor vector index size vs. search
performance");

// Error handling patterns
Console.WriteLine(" Error Handling:");
Console.WriteLine("  • Wrap database operations in try-catch blocks");
Console.WriteLine("  • Implement retry logic for transient failures");
Console.WriteLine("  • Log errors with sufficient context");
Console.WriteLine("  • Gracefully degrade when embeddings are
unavailable");

// Cleanup demonstration
try
{
    // In a real application, you might want to close connections,
flush caches, etc.
    await host.StopAsync();
    Console.WriteLine(" Services stopped gracefully");
}
catch (Exception ex)
{
    Console.WriteLine($"⚠ Cleanup warning: {ex.Message}");
}
finally
{
    host?.Dispose();
    Console.WriteLine(" Host disposed");
}
```

```
 Cleanup and Best Practices
================================================
 Resource Management:
  • Database connections are auto-disposed with 'using' statements
  • Vector collections are managed by DI container
  • Embedding service is properly registered as singleton
 Performance Recommendations:
  • Use connection pooling for high-throughput scenarios
  • Cache frequently used embeddings
  • Consider batch operations for bulk inserts
  • Monitor vector index size vs. search performance
 Error Handling:
  • Wrap database operations in try-catch blocks
  • Implement retry logic for transient failures
  • Log errors with sufficient context
  • Gracefully degrade when embeddings are unavailable
 Services stopped gracefully
 Host disposed
```

Summary

This notebook demonstrated comprehensive SQLiteVec integration with Microsoft Semantic Kernel:

Key Concepts Covered:

1. **Vector Data Models**: Using VectorData attributes for schema definition
2. **Hybrid Search**: Combining FTS5 and vector search for optimal results
3. **Dependency Injection**: Clean architecture with proper service registration
4. **Performance Analysis**: Measuring and optimizing search performance
5. **Advanced Features**: Filtering, batch operations, and custom configurations
6. **Best Practices**: Resource management and error handling patterns

Architecture Benefits:

- **Scalable**: DI pattern supports easy testing and extensibility
- **Flexible**: Multiple search strategies can be combined
- **Performant**: SQLiteVec provides efficient vector operations
- **Maintainable**: Clean separation of concerns with service interfaces

Next Steps:

- Implement custom distance functions for domain-specific similarity
- Add real-time indexing for dynamic content updates
- Explore multi-modal embeddings (text + images)
- Build distributed search across multiple SQLite instances
- Integrate with semantic caching for improved performance