

Design and implement intelligent search with SQL

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/01-introduction>

Introduction

Completed

Finding relevant information in a database often requires more than matching keywords. Users search for concepts, not just words. A customer looking for "lightweight hiking backpack" might find nothing if the product is listed as "ultralight trail bag." SQL Server and Azure SQL Database provide multiple search approaches to handle these challenges: full-text search for keyword matching, vector search for semantic similarity, and hybrid search that combines both.

Full-text search excels when users know the specific terms they want. Vector search uses embeddings to find content based on meaning even when the exact words differ. Hybrid search runs both approaches and merges the results, giving users the best of both worlds. Choosing the right approach depends on how your users search and what trade-offs you can accept between precision, recall, and performance.

Imagine a retail team building a product search feature. They have embeddings stored for their product descriptions and support articles. Now they need search capabilities that help customers find products and answers quickly. Some customers type exact product names. Others describe what they need in natural language. The team decides to implement hybrid search so their application handles both patterns effectively, using full-text indexes for keyword matching and vector search for semantic similarity.

After completing this module, you'll be able to:

- To select the right approach for different query types, evaluate full-text search, vector search, and hybrid search.
- Implement full-text search using full-text indexes and query predicates.
- Prepare SQL databases for vector search by storing embeddings and choosing distance metrics.

- Write vector search queries using VECTOR_DISTANCE and VECTOR_SEARCH functions.
- Combine full-text and vector search results using Reciprocal Rank Fusion (RRF).

2. Choose an intelligent search approach

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/02-choose-intelligent-search-approach>

Choose an intelligent search approach

Completed

SQL Server and Azure SQL Database provide you with three ways to search text data, full-text search, vector search, and hybrid search. Each works differently, and choosing the right one depends on what your users need to find.

To see why the right search approach matters, imagine a customer searching your product catalog. They type "lightweight hiking backpack" but your best-selling item is listed as "ultralight trail bag." A traditional keyword search finds nothing. A smarter search option understands that both descriptions mean the same thing. The search approach you choose determines whether your users find what they need or not.

Understand the three search approaches

Let's look at how each approach works and when you'd use it.

Full-text search looks for words and phrases. When a customer searches for "mountain bike helmet," full-text search finds products containing those exact terms. It understands language rules, so searching for "ride" can also find "riding" or "rode." Use full-text search when users know the specific words they want.

Semantic vector search looks for meaning. Instead of matching words, it compares the mathematical representation of concepts. When a customer searches for "something to keep me visible on evening rides," vector search can find products described as "reflective cycling vest" or "LED bike lights" even though those words don't appear in the search. This approach works well when users describe what they need rather than name it.

Hybrid search uses both. It runs a full-text search and a vector search, then combines the results. A customer searching for "comfortable seat for long tours" gets products matching those keywords and products like "touring saddle" or "gel bike seat" that vector search identifies as semantically related.

Match the approach to the query intent

Your users express different intentions when they search. Some know exactly what they want. Others are exploring.

Consider these two searches against a product database:

- "Model XR-500" - The user wants that specific product. Full-text search handles this search perfectly.
- "something to keep drinks cold on a hike" - The user describes a need. Vector search understands this means "insulated water bottle" or "portable cooler."

When you can't predict how users search, hybrid search covers both cases. The user who types "XR-500 portable cooler" gets results matching the model number and results matching the concept.

Evaluate the trade-offs

Choosing a search approach involves trade-offs. Consider these factors as you design your solution.

Precision versus recall - Full-text search returns fewer results, but they closely match the query terms. Vector search returns more results, including items that are conceptually related but use different words. If your application needs exact matches, lean toward full-text. If it needs discovery, lean toward vector search.

Data requirements - Full-text search needs a full-text index on your text columns. Vector search needs embeddings stored in vector columns. You might need to prepare your data differently depending on which approach you choose.

Performance characteristics - Full-text indexes are optimized for fast keyword lookup. Vector search performance depends on how many vectors you're comparing and whether you use approximate or exact methods. Hybrid search runs both, so it takes longer than either one alone.

These trade-offs aren't about which approach is better. The right choice depends on which approach fits your scenario.

Combine approaches with hybrid search

Full-text or vector search alone doesn't handle every situation. Full-text search misses documents that express the same idea with different words. Vector search might return conceptually related items that don't contain important terms the user specified.

Hybrid search solves this issue by running both approaches and merging the results. A user searching for "Azure SQL backup strategies" gets documents containing those keywords and documents about "database recovery planning" that vector search identifies as related.

The challenge is combining two ranked lists into one. How do you decide which result should appear first when full-text search ranks it #3 and vector search ranks it #7?

Merge results with Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF) is an algorithm for combining ranked results from different searches. Instead of comparing raw scores, which use different scales, RRF calculates a score based on each document's position (the **rank**) using the formula $1/(\text{rank} + k)$. The constant **k** is 60, a value established by the original RRF research and used by Azure AI Search. Documents appearing in multiple result sets have their scores summed, which is where RRF really shines.

Here's how it works. Suppose a customer searches for "comfortable bike seat" and you run both full-text and vector search:

Rank	Full-text results	Vector search results
1	Bike Seat Comfort Pro	Ergonomic Touring Saddle
2	Comfortable Handlebar Grips	Gel Cushion Saddle
3	Racing Bike Seat	Bike Seat Comfort Pro

Full-text search finds products containing the words *comfortable*, *bike*, and *seat*. Vector search finds semantically similar products, including saddles that don't use the word *seat* at all.

Notice that "Bike Seat Comfort Pro" appears in both lists: rank 1 in full-text and rank 3 in vector search. RRF calculates its combined score by summing:

- Full-text rank 1: $1/(60+1) = 0.0164$
- Vector rank 3: $1/(60+3) = 0.0159$
- **Combined: 0.0323**

Compare that to "Ergonomic Touring Saddle," which only appears in vector search at rank 1:

- Vector rank 1: $1/(60+1) = 0.0164$
- **Combined: 0.0164**

Even though "Ergonomic Touring Saddle" ranked #1 in vector search, "Bike Seat Comfort Pro" wins the combined ranking because appearing in both lists doubled its score. The merged results:

Rank	Combined results	RRF Score
1	Bike Seat Comfort Pro	0.0323
2	Ergonomic Touring Saddle	0.0164
3	Comfortable Handlebar Grips	0.0161
4	Gel Cushion Saddle	0.0161
5	Racing Bike Seat	0.0159

Notice that ranks 3 and 4 have identical scores. When RRF produces ties, the ordering between those results depends on the implementation and might be arbitrary or alphabetical. In practice, tied

results are equally relevant, so the exact order between them matters less than ensuring they all appear above results with lower scores.

RRF rewards documents that perform well across multiple search methods, surfacing results that match both keywords and meaning.

Key takeaways

Full-text search, vector search, and hybrid search each serve different purposes. Full-text search excels at finding exact keywords and phrases, vector search captures semantic meaning across different phrasings, and hybrid search combines both to maximize relevance. Your choice depends on how your users search and what trade-offs you can accept between precision, recall, and performance. When you implement hybrid search, Reciprocal Rank Fusion (RRF) is a powerful technique for merging ranked results from full-text and vector searches, ensuring that items relevant to both methods rise to the top of the results list.

3. Implement full-text search

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/03-implement-full-text-search>

Implement full-text search

Completed

Full-text search lets you run linguistic searches against text data in SQL Server and Azure SQL Database. Unlike the `LIKE` operator, which only matches character patterns, full-text search understands words, phrases, and language rules.

When a customer searches your product catalog for "ride," they expect to find results that say "riding," "rides," or "rode." Full-text search handles these scenarios by working with language-aware indexes that understand word forms. It doesn't automatically match synonyms or abbreviations (like "MTB" for "mountain bike") unless you configure a thesaurus, but it handles inflections out of the box.

When to use full-text search

Full-text search works best when your users search using specific words or phrases. Consider these scenarios where full-text search is the right choice:

- A customer types a product name or part number

- A support agent searches for tickets containing specific error messages
- A user looks for documents with exact technical terms

If users instead describe what they need without using specific terms, vector search might be more appropriate. But when users know the words they want, full-text search delivers fast, precise results.

Understand full-text indexes and predicates

Full-text search requires two things: a full-text index on your text columns, and predicates to query that index.

A full-text index stores information about words and their locations within your text columns. SQL Server breaks down the text into individual tokens, removes common words like "the" and "is" (called stopwords), and builds an inverted index that maps words to the rows containing them. This structure allows fast lookups without scanning every row.

To query a full-text index, you use predicates in your `WHERE` clause. SQL Server provides two main predicates:

- **CONTAINS** searches for exact matches of words and phrases. Use it when you need precise control over what matches.
- **FREETEXT** searches for meaning rather than exact words. It automatically looks for inflectional forms of your search terms.

SQL Server also provides table-valued functions **CONTAINSTABLE** and **FREETEXTTABLE** that return ranked results with relevance scores, which is useful when you want to sort results by how well they match.

Query a full-text index

The following example shows how to search a product catalog using both `CONTAINS` and `FREETEXT`. Assume the `Production.Product` table has a full-text index on the `Name` column.

Use `CONTAINS` when you need exact word matching:

```
SELECT ProductID, Name, ListPrice
FROM Production.Product
WHERE CONTAINS(Name, 'mountain')
      AND ListPrice < 500;
```

This query returns products with "mountain" in the name under \$500. The `CONTAINS` predicate checks the full-text index while the `ListPrice` filter uses the regular column.

Use `FREETEXT` when you want SQL Server to find related word forms automatically:

```
SELECT ProductID, Name
FROM Production.Product
```

```
WHERE FREETEXT(Name, 'riding bikes');
```

This query finds products related to "riding bikes," including matches for "ride," "rides," and "biking." `FREETEXT` handles inflections automatically without requiring you to specify them.

When you need ranked results, use `CONTAINSTABLE` to get full-text relevance scores. Full-text search assigns a rank to each result based on how well it matches the query. This rank is useful for sorting results by relevance:

```
SELECT p.ProductID, p.Name, ft.RANK
FROM Production.Product AS p
INNER JOIN CONTAINSTABLE(Production.Product, Name,
    'NEAR((mountain, bike))') AS ft
    ON p.ProductID = ft.[KEY]
ORDER BY ft.RANK DESC;
```

This query finds products where "mountain" and "bike" appear near each other, sorted by full-text relevance. The `RANK` column indicates how well each row matches the search criteria. Later, when you combine full-text search with vector search, you use these full-text ranks alongside vector similarity scores to produce merged results.

Common query patterns

Full-text search supports several query patterns. Each addresses a different search need.

Term search finds rows containing a specific word. A search for `CONTAINS(Description, 'aluminum')` returns all products with "aluminum" in the description.

Phrase search finds rows containing words in a specific order. Wrapping the search in quotes like `CONTAINS(Description, '"mountain bike"')` matches only that exact phrase, not rows where "mountain" and "bike" appear separately.

Prefix search finds words that start with specific characters. Searching for `CONTAINS(Description, '"light*")` matches "light," "lights," "lighter," "lightweight," and any other word beginning with "light."

Inflectional search finds different forms of a word. Using `CONTAINS(Description, 'FORMSOF(INFLECTIONAL, "ride")')` matches "ride," "rides," "riding," and "rode."

Proximity search finds words that appear near each other. Searching for `CONTAINS(Description, 'NEAR((light, aluminum))')` finds products where "light" and "aluminum" appear close together, which often indicates a lightweight aluminum product.

Evaluate full-text search results

After implementing full-text search, evaluate whether it meets your users' needs. Watch for these signals:

Precision problems occur when results include irrelevant items. If searching for "brake" returns results about "brake-resistant" coatings when users want bicycle brakes, your search is too broad. Consider using phrase searches or more specific terms.

Noise happens when common words dilute results. Full-text search uses stoplists to filter out words like "the" and "and," but domain-specific noise might require custom stoplists. If searching for "the mountain" returns too many results because "mountain" appears everywhere, you might need to adjust your approach.

Query intent mismatch appears when users search for concepts rather than words. If customers search for "something for rainy commutes" and expect to find waterproof gear, full-text search doesn't help because it matches words, not meaning. This signal suggests you might need vector search or hybrid search instead.

Key takeaways

Full-text search excels when users search with specific words and phrases. It uses full-text indexes to enable fast linguistic searches and provides predicates like `CONTAINS` and `FREETEXT` to query those indexes. Different query patterns (term, phrase, prefix, inflectional, and proximity) address different search needs. When full-text search returns too many irrelevant results, or when users search for concepts instead of specific words, the next search method to consider is vector search.

4. Prepare SQL for vector search

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/04-prepare-sql-vector-search>

Prepare SQL for vector search

Completed

Before you can run vector searches, you need to store vectors in your database and decide how those searches execute. This unit covers the design decisions you make before writing your first vector query.

Vector search finds rows based on mathematical similarity rather than exact matches. To make this work, you store embeddings as vectors, choose how to measure similarity, and decide whether to search all vectors exactly or use an index for faster approximate results.

Store vectors with the vector data type

SQL Server and Azure SQL Database provide a native **vector** data type designed for storing embeddings. Each vector is an array of floating-point numbers, stored in an optimized binary format but exposed as JSON arrays for convenience.

When you define a vector column, you specify the number of dimensions:

```
CREATE TABLE dbo.Products
(
    ProductID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Description NVARCHAR(MAX),
    DescriptionVector VECTOR(1536) NOT NULL
);
```

The number in parentheses matches the dimensions your embedding model produces. For example, OpenAI's text-embedding-3-small model produces 1,536 dimensions, so your column would be `VECTOR(1536)`. The maximum supported is 1,998 dimensions.

Each element is stored as a single-precision (4-byte) float. A 1,536-dimension vector uses about 6 KB per row. When designing your table, consider how this column affects storage and memory as your data grows.

Understand distance metrics

Vector search works by calculating the distance between vectors. Vectors that are closer together represent more similar concepts. SQL Server supports three distance metrics:

Cosine distance measures the angle between vectors, ignoring their magnitude. Two vectors pointing in the same direction have a cosine distance of 0, regardless of their length. This metric works well when you care about the direction of meaning, not the intensity. Cosine distance ranges from 0 (identical) to 2 (opposite).

Euclidean distance measures the straight-line distance between two points in vector space. It considers both direction and magnitude. Euclidean distance ranges from 0 (identical) to infinity.

Dot product calculates the sum of element-wise products. SQL Server returns the negative dot product so that smaller values indicate more similar vectors, consistent with the other metrics.

Most embedding models are optimized for cosine similarity, making cosine distance the common choice. However, the best metric depends on how your embeddings were trained.

Choose between exact and approximate search

When you search for similar vectors, you have two options: exact nearest neighbor (ENN) search and approximate nearest neighbor (ANN) search.

Exact nearest neighbor search compares your query vector against every vector in the table. It guarantees the most accurate results but requires calculating the distance to every row. Use the `VECTOR_DISTANCE` function for exact search:

```
DECLARE @query VECTOR(1536) = '[0.1, 0.2, ...]';

SELECT TOP 10 ProductID, Name,
    VECTOR_DISTANCE('cosine', @query, DescriptionVector) AS Distance
FROM dbo.Products
ORDER BY Distance;
```

This query calculates the cosine distance between `@query` and every row's `DescriptionVector`, then returns the 10 closest matches. For small tables (under 50,000 vectors as a general guideline), exact search performs well.

Approximate nearest neighbor search uses a vector index to find similar vectors without scanning every row. It trades perfect accuracy for speed, returning results that are very close to exact but not guaranteed to be the absolute nearest neighbors.

Create a vector index using `CREATE VECTOR INDEX`:

```
CREATE VECTOR INDEX idx_Products_DescriptionVector
ON dbo.Products(DescriptionVector)
WITH (METRIC = 'cosine', TYPE = 'DiskANN');
```

SQL Server uses the DiskANN algorithm, which builds a graph structure that allows fast navigation to nearby vectors. Once the index exists, use the `VECTOR_SEARCH` function for approximate search:

```
DECLARE @query VECTOR(1536) = '[0.1, 0.2, ...]';

SELECT t.ProductID, t.Name, s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.Products AS t,
    COLUMN = DescriptionVector,
    SIMILAR_TO = @query,
    METRIC = 'cosine',
    TOP_N = 10
) AS s
ORDER BY s.distance;
```

Decide when to use each approach

The choice between exact and approximate search depends on your dataset size and accuracy requirements.

Use exact search (`VECTOR_DISTANCE`) when:

- Your table has fewer than 50,000 vectors

- Your query filters reduce the candidate set to a few rows
- You need guaranteed accurate results and can accept longer query times

Use approximate search (`VECTOR_SEARCH` with an index) when:

- Your table has hundreds of thousands or millions of vectors
- Query speed matters more than perfect accuracy
- A recall close to 1 (meaning most true nearest neighbors are found) is acceptable

Recall measures how many of the true nearest neighbors the approximate search returns compared to an exact search. DiskANN typically achieves high recall, meaning the results are very close to what exact search would return.

Consider index limitations

Vector indexes in SQL Server have specific requirements to be aware of:

- The table must have a single-column integer primary key with a clustered index
- Tables with vector indexes become read-only while the index exists—you must drop the index to modify data, then recreate it
- Vector indexes can't be partitioned

Note

Vector indexes are currently in preview. In Azure SQL Database and SQL database in Microsoft Fabric, you can set the `ALLOW_STALE_VECTOR_INDEX` database scoped configuration to `ON` to allow writes, but the index doesn't reflect new data until you rebuild it. This option isn't currently available in SQL Server 2025. Check the current documentation for the latest on these limitations.

These limitations affect how you design your tables and maintenance workflows. For tables that change frequently, you might use exact search until the data stabilizes, then add a vector index.

Key takeaways

Preparing SQL for vector search means making three decisions: what data type and dimensions to use, which distance metric matches your embedding model, and whether exact or approximate search fits your dataset size. Exact search with `VECTOR_DISTANCE` works well for smaller datasets or filtered queries, while approximate search with `VECTOR_SEARCH` and a DiskANN index handles larger datasets efficiently. In the next unit, you'll learn the specific functions and query patterns for running vector searches.

5. Implement vector search query patterns

Implement vector search query patterns

Completed

With vectors stored and a search strategy chosen, you're ready to write queries. This unit covers the T-SQL functions that power vector search and shows you how to combine them effectively.

Vector search queries follow a common pattern: generate or retrieve a query vector, calculate distances to stored vectors, and return the closest matches. SQL Server provides four functions that work together to support this pattern: `VECTOR_DISTANCE`, `VECTOR_SEARCH`, `VECTOR_NORMIMIZE`, and `VECTORPROPERTY`.

Calculate distances with `VECTOR_DISTANCE`

The `VECTOR_DISTANCE` function calculates the distance between two vectors using the metric you specify. This function performs exact nearest neighbor search, computing the distance to every qualifying row. In practical terms, it compares your search embedding against every stored embedding to find which ones are most similar, like checking every product in a catalog to find the best matches.

```
VECTOR_DISTANCE(metric, vector1, vector2)
```

The metric must be `'cosine'`, `'euclidean'`, or `'dot'`. Both vectors must have the same number of dimensions.

Here's a typical pattern for finding similar products based on a description embedding:

```
-- Generate an embedding for the user's search phrase
DECLARE @searchVector VECTOR(1536);
SELECT @searchVector = AI_GENERATE_EMBEDDINGS('lightweight hiking boots' USE MODEL

SELECT TOP 10
    ProductID,
    Name,
    VECTOR_DISTANCE('cosine', @searchVector, DescriptionVector) AS Distance
FROM dbo.Products
ORDER BY Distance;
```

This query works in two steps. First, it converts the search phrase "lightweight hiking boots" into an embedding using the same model that created the stored `DescriptionVector` values. Second, it calculates the cosine distance between `@searchVector` and every row's `DescriptionVector`. The

`ORDER BY Distance` clause sorts the results from smallest distance (most similar) to largest, and `TOP 10` returns only the first 10 rows. Since smaller distances mean more similar vectors, this combination gives you the 10 products most semantically related to "lightweight hiking boots."

You can also use `VECTOR_DISTANCE` in a `WHERE` clause to find all vectors within a specific distance threshold:

```
SELECT ProductID, Name
FROM dbo.Products
WHERE VECTOR_DISTANCE('cosine', @searchVector, DescriptionVector) < 0.3
ORDER BY VECTOR_DISTANCE('cosine', @searchVector, DescriptionVector);
```

This approach finds all products within a similarity threshold rather than returning a fixed count. The 0.3 value here's just an example. Cosine distance ranges from 0 (identical vectors) to 2 (completely opposite), but what counts as "similar enough" depends entirely on your data and embedding model. To find the right threshold, run test queries, examine the distance values in your results, and identify where relevant results stop and the noise begins. Use this pattern when you want everything meeting a quality bar rather than a fixed number of results.

Use `VECTOR_SEARCH` for approximate retrieval

When your table grows beyond tens of thousands of rows, calculating distances to every row becomes slow. The `VECTOR_SEARCH` function uses a vector index to find approximate nearest neighbors quickly.

```
VECTOR_SEARCH(
    TABLE = object AS alias,
    COLUMN = vector_column,
    SIMILAR_TO = query_vector,
    METRIC = 'cosine' | 'euclidean' | 'dot',
    TOP_N = k
)
```

This function returns a result set containing all columns from the source table plus a `distance` column. Here's how you'd use it:

```
DECLARE @searchVector VECTOR(1536);
SELECT @searchVector = AI_GENERATE_EMBEDDINGS('lightweight hiking boots' USE MODEL

SELECT t.ProductID, t.Name, s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.Products AS t,
    COLUMN = DescriptionVector,
    SIMILAR_TO = @searchVector,
    METRIC = 'cosine',
    TOP_N = 10
) AS s
ORDER BY s.distance;
```

The function returns the approximate 10 nearest neighbors. If a matching vector index exists on `DescriptionVector` with the same metric, `VECTOR_SEARCH` uses it for fast retrieval. Without an index, it falls back to exact search and raises a warning.

Handle post-filtering carefully

`VECTOR_SEARCH` applies any WHERE clause conditions *after* finding the nearest neighbors, not before. Consider this query:

```
SELECT t.ProductID, t.Name, s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.Products AS t,
    COLUMN = DescriptionVector,
    SIMILAR_TO = @searchVector,
    METRIC = 'cosine',
    TOP_N = 10
) AS s
WHERE t.CategoryID = 5
ORDER BY s.distance;
```

This query first finds the 10 nearest vectors across all categories, then filters to only category 5. If none of the 10 nearest products are in category 5, you get no results. To work around this issue, request more candidates than you need:

```
SELECT TOP 10 t.ProductID, t.Name, s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.Products AS t,
    COLUMN = DescriptionVector,
    SIMILAR_TO = @searchVector,
    METRIC = 'cosine',
    TOP_N = 50
) AS s
WHERE t.CategoryID = 5
ORDER BY s.distance;
```

By requesting 50 candidates from `VECTOR_SEARCH` and then taking the top 10 after filtering, you're more likely to have enough results in the target category.

Normalize vectors for consistent comparisons

Different embedding models produce vectors with different lengths (magnitudes). When comparing vectors from multiple sources or when your model doesn't normalize outputs, you can use `VECTOR_NORMIMIZE` to scale vectors to unit length.

```
VECTOR_NORMIMIZE(vector, norm_type)
```

The `norm_type` can be:

- `'norm2'` : Euclidean norm (most common)
- `'norm1'` : Sum of absolute values
- `'norminf'` : Maximum absolute value

```
DECLARE @v VECTOR(3) = '[3, 4, 0]';
SELECT VECTOR_NORMALIZE(@v, 'norm2') AS NormalizedVector;
-- Returns: [0.6, 0.8, 0.0]
```

Normalizing ensures that cosine distance and dot product behave consistently. Most modern embedding models (like those from OpenAI) produce normalized vectors, so you often don't need this step. But if you're working with embeddings from models that don't normalize, applying `VECTOR_NORMALIZE` before storing or comparing ensures consistent similarity scores.

You can calculate a vector's magnitude using the related `VECTOR_NORM` function:

```
DECLARE @v VECTOR(3) = '[3, 4, 0]';
SELECT VECTOR_NORM(@v, 'norm2') AS Magnitude;
-- Returns: 5.0
```

Inspect vectors with VECTORPROPERTY

The `VECTORPROPERTY` function returns metadata about a vector. This metadata is useful for validating data or debugging dimension mismatches.

```
VECTORPROPERTY(vector, property)
```

Two properties are supported:

- `'Dimensions'` : Returns the number of dimensions as an integer
- `'BaseType'` : Returns the data type name (currently always `float`)

```
DECLARE @v VECTOR(1536) = '[0.1, 0.2, ...]';
SELECT VECTORPROPERTY(@v, 'Dimensions') AS Dims;
-- Returns: 1536
```

This function helps when troubleshooting queries where vectors might have mismatched dimensions—a common issue when different embedding models are used.

Choose the right function for your scenario

Each function serves a distinct purpose in vector search:

Scenario	Function	When to use
Small dataset or filtered queries	<code>VECTOR_DISTANCE</code>	Under 50,000 vectors, or when a WHERE clause significantly reduces candidates

Scenario	Function	When to use
Large dataset with vector index	<code>VECTOR_SEARCH</code>	Hundreds of thousands of vectors or more, when speed matters
Comparing vectors from different models	<code>VECTOR_NORMALIZE</code>	When embedding sources vary or model doesn't produce normalized vectors
Validating vector data	<code>VECTORPROPERTY</code>	Checking dimensions during troubleshooting or data validation

Key takeaways

Vector search queries in SQL follow a predictable pattern: prepare a query vector, find neighbors using either `VECTOR_DISTANCE` (exact) or `VECTOR_SEARCH` (approximate), and order by distance. `VECTOR_NORMALIZE` and `VECTORPROPERTY` support data quality and consistency. Remember that `VECTOR_SEARCH` applies filters after finding nearest neighbors, so request more candidates than you need when combining with WHERE clauses. In the next unit, you'll learn how to combine vector search with full-text search in hybrid queries.

6. Implement hybrid search and ranking

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/06-implement-hybrid-search-ranking>

Implement hybrid search and ranking

Completed

Full-text search and vector search each have strengths, but neither alone handles every query. Full-text search excels at finding exact keywords but misses documents that express the same idea differently. Vector search captures semantic meaning but might return conceptually related items that don't contain important terms the user specified. Hybrid search combines both approaches to maximize relevance.

In this unit, you learn how to implement hybrid search in SQL by running full-text and vector searches in parallel, then merging the results using Reciprocal Rank Fusion (RRF).

Understand the hybrid search pattern

A hybrid search query runs two searches against the same data:

1. **Full-text search** uses `FREETEXTTABLE` or `CONTAINSTABLE` to find documents matching keywords, returning results ranked by BM25 relevance.
2. **Vector search** uses `VECTOR_DISTANCE` or `VECTOR_SEARCH` to find documents with similar embeddings, returning results ranked by distance.

Each search produces a ranked list. The challenge is combining these two lists into a single result set that benefits from both ranking signals.

Consider a user searching for "Azure SQL backup strategies." Full-text search finds documents containing those exact words. Vector search finds documents about "database recovery planning" or "disaster recovery for SQL databases" that are semantically related but use different terms. A hybrid approach returns both, giving users the best of both worlds.

Merge results with Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF) is a technique for combining ranked lists from different sources. Instead of comparing raw scores, which might use different scales, RRF focuses on rank positions.

The formula for RRF is:

$$\text{RRF_score} = 1/(k + \text{rank_from_source_1}) + 1/(k + \text{rank_from_source_2})$$

The constant `k` (typically 60) prevents high-ranked items from dominating. A document ranked #1 in both lists receives a higher combined score than one ranked #1 in only one list.

This approach has two key advantages:

- **Score normalization isn't needed.** Full-text BM25 scores and cosine distances use completely different scales. RRF sidesteps these differences by using ranks instead.
- **Neither source dominates.** If vector search consistently returns higher scores than full-text search, raw score fusion would favor vector results. RRF treats both sources equally.

Implement hybrid search in SQL

Here's a complete hybrid search implementation using Common Table Expressions (CTEs) to structure the query:

```
DECLARE @searchText NVARCHAR(1000) = 'lightweight hiking boots';
DECLARE @searchVector VECTOR(1536);
DECLARE @topN INT = 50;
DECLARE @rrfK INT = 60;

-- Generate embedding for the search phrase
SELECT @searchVector = AI_GENERATE_EMBEDDINGS(@searchText USE MODEL MyEmbeddingModel);

-- Run hybrid search with RRF
WITH keyword_search AS (
```

```

SELECT TOP(@topN)
    p.ProductID,
    RANK() OVER (ORDER BY ftt.[RANK] DESC) AS keyword_rank
FROM dbo.Products p
INNER JOIN FREETEXTTABLE(dbo.Products, Description, @searchText) AS ftt
    ON p.ProductID = ftt.[KEY]
),
vector_search AS (
    SELECT TOP(@topN)
        ProductID,
        RANK() OVER (ORDER BY distance) AS vector_rank
    FROM (
        SELECT
            ProductID,
            VECTOR_DISTANCE('cosine', @searchVector, DescriptionVector) AS distance
        FROM dbo.Products
    ) AS similar_products
),
combined AS (
    SELECT TOP(@topN)
        COALESCE(ks.ProductID, vs.ProductID) AS ProductID,
        ks.keyword_rank,
        vs.vector_rank,
        COALESCE(1.0 / (@rrfK + ks.keyword_rank), 0.0) +
        COALESCE(1.0 / (@rrfK + vs.vector_rank), 0.0) AS rrf_score
    FROM keyword_search ks
    FULL OUTER JOIN vector_search vs ON ks.ProductID = vs.ProductID
)
SELECT
    p.ProductID,
    p.Name,
    p.Description,
    c.keyword_rank,
    c.vector_rank,
    c.rrf_score
FROM combined c
INNER JOIN dbo.Products p ON c.ProductID = p.ProductID
ORDER BY c.rrf_score DESC;

```

Let's break down what each CTE does:

1. **keyword_search**: Runs a full-text search using `FREETEXTTABLE`, which returns BM25-ranked results. The `RANK()` function assigns position numbers.
2. **vector_search**: Calculates cosine distance for all products and assigns rank positions based on distance (smallest first).
3. **combined**: Joins the two result sets using `FULL OUTER JOIN` so products appearing in either list are included. The RRF formula calculates the combined score. `COALESCE` handles products that appear in only one list by treating the missing rank as contributing zero.
4. **Final SELECT**: Joins back to the Products table to retrieve the full product details, ordered by the combined RRF score.

Use VECTOR_SEARCH for large datasets

When your table has hundreds of thousands of rows, replace the exact `VECTOR_DISTANCE` calculation with `VECTOR_SEARCH` for better performance:

```
vector_search AS (  
    SELECT TOP(@topN)  
        t.ProductID,  
        RANK() OVER (ORDER BY s.distance) AS vector_rank  
    FROM VECTOR_SEARCH(  
        TABLE = dbo.Products AS t,  
        COLUMN = DescriptionVector,  
        SIMILAR_TO = @searchVector,  
        METRIC = 'cosine',  
        TOP_N = @topN  
    ) AS s  
)
```

This version uses the DiskANN index for approximate nearest neighbor search, which is faster for large datasets while still providing high-quality results.

Tune hybrid search parameters

Several factors affect hybrid search quality:

Result set size (@topN): The number of candidates from each search. Larger values give RRF more candidates to work with but increase query time. Start with 50 and adjust based on your relevance testing.

The RRF constant (@rrfK): The formula uses $1/(k + \text{rank})$. A larger k (like 60) smooths out differences between ranks. A smaller k amplifies the advantage of top-ranked items. The value 60 is standard and comes from the original RRF research.

Weighting: You can weight one source more than the other by multiplying its contribution:

```
-- Weight vector search 2x more than keyword search  
COALESCE(1.0 / (@rrfK + ks.keyword_rank), 0.0) +  
COALESCE(2.0 / (@rrfK + vs.vector_rank), 0.0) AS rrf_score
```

Filtering: Apply filters before the hybrid search when possible. Filtering after RRF might eliminate highly ranked results.

Evaluate search quality

Measuring search quality helps you tune parameters and choose between approaches. Common metrics include:

Precision: Of the results returned, what percentage are relevant? High precision means few irrelevant results.

Recall: Of all relevant documents in your database, what percentage did the search find? High recall means you're not missing good results.

Mean Reciprocal Rank (MRR): How high does the first relevant result appear? MRR rewards searches that put the best result at the top.

To evaluate, you need a test set of queries with known relevant documents. Run each approach (full-text only, vector only, hybrid) and compare metrics. Hybrid search typically improves recall compared to either approach alone, often with minimal precision loss.

Key takeaways

Hybrid search combines full-text and vector search to handle both keyword-focused and concept-focused queries. Reciprocal Rank Fusion merges the ranked results without requiring score normalization, treating both sources fairly. You can tune the result set size, RRF constant, and source weights to optimize for your data. Evaluating precision, recall, and MRR helps you measure whether hybrid search improves relevance for your specific use case.

7. Exercise - Implement intelligent search with full-text, vector, and hybrid queries

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/07-exercise-implement-intelligent-search>

Exercise - Implement intelligent search with full-text, vector, and hybrid queries

Completed

In this exercise, you implement different search approaches in an Azure SQL Database. You create full-text indexes, run vector searches using stored embeddings, and combine both techniques with hybrid search using Reciprocal Rank Fusion (RRF). You then compare how each search approach handles the same query and observe the differences in results and performance.

Note

To complete this exercise, you need an [Azure subscription](#) and be approved for Azure OpenAI access. If you need Azure OpenAI access, apply at the [Azure OpenAI limited access](#) page.

Launch the exercise and follow the instructions.

Launch Exercise

8. Knowledge check

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/08-knowledge-check>

Knowledge check

Completed

9. Summary

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/09-summary>

Summary

Completed

This module covered designing and implementing intelligent search capabilities for SQL Server and Azure SQL Database. You learned how to evaluate full-text search, vector search, and hybrid search approaches for different query types. You also explored the T-SQL functions for vector operations and implemented hybrid search using Reciprocal Rank Fusion to combine ranked results from multiple sources.

After completing this module, you can:

- To select the right approach for different query types, evaluate full-text search, vector search, and hybrid search.
- Implement full-text search using full-text indexes and query predicates.

- Prepare SQL databases for vector search by storing embeddings and choosing distance metrics.
- Write vector search queries using VECTOR_DISTANCE and VECTOR_SEARCH functions.
- Combine full-text and vector search results using Reciprocal Rank Fusion (RRF).

Additional reading

- [Full-text search overview](#)
- [Vector data type](#)
- [VECTOR_DISTANCE function](#)
- [VECTOR_SEARCH function](#)
- [Reciprocal Rank Fusion in hybrid search](#)