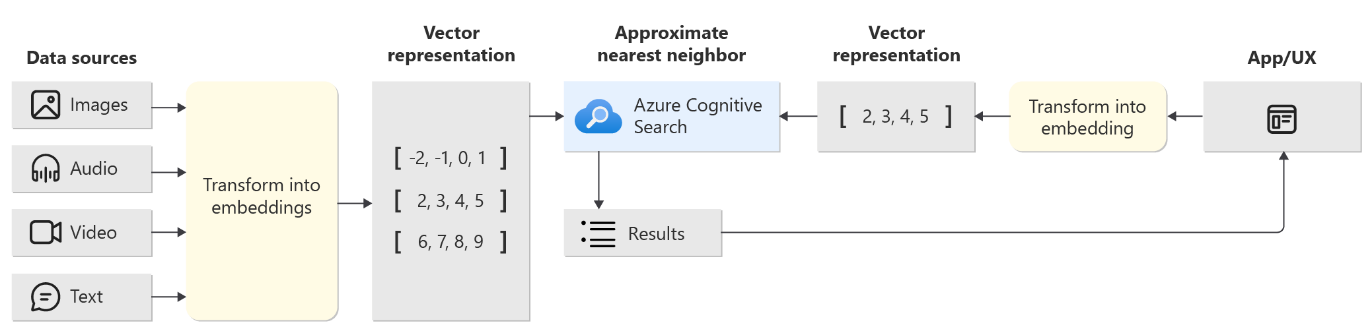
**Vector Search**

Vector search is an approach in information retrieval that stores numeric representations of content for search scenarios. Because the content is numeric rather than plain text, the search engine matches on vectors that are the most similar to the query, with no requirement for matching on exact terms.

Vector support includes indexing, storing, and querying of vector embeddings from a search index.

The following diagram shows the indexing and query workflows for vector search.



**Architecture of vector search workflow.**

On the indexing side, Azure AI Search takes vector embeddings and uses a **nearest neighbors algorithm** to place similar vectors close together in an index. Internally, it creates vector indexes for each vector field.

* **How you get embeddings from your source content into Azure AI Search ?**

It depends on your approach and whether you can use preview features. You can **vectorize or generate embeddings as a preliminary step** using models from OpenAI, Azure OpenAI, and any number of providers, over a wide range of source content including text, images, and other content types supported by the models. You can then push prevectorized content to vector fields in a vector store. That's the generally available approach. If you can use preview features, **Azure AI Search** offers **integrated data chunking and vectorization** in an indexer pipeline. You still provide the resources (endpoints and connection information to Azure OpenAI), but Azure AI Search makes all of the calls and handles the transitions.

* **Query Side**

On the query side, in your client application, you collect the query input from a user, usually through a prompt workflow. You can then add an encoding step that converts the input into a vector, and then send the vector query to your index on Azure AI Search for a similarity search. As with indexing, you can deploy the integrated vectorization (preview) to convert the question into a vector. For either approach, Azure AI Search returns documents with the requested **k nearest neighbors (kNN)** in the results.

Azure AI Search supports **hybrid scenarios** that run **vector and keyword search in parallel**, returning a unified result set that often provides better results than just vector or keyword search alone. For hybrid, vector and nonvector content is ingested into the same index, for queries that run side by side.

**What scenarios can vector search support?**

Scenarios for vector search include:

* **Vector database**. Azure AI Search stores the data that you query over. Use it as a pure vector store any time you need long-term memory or a knowledge base, or grounding data for Retrieval Augmented Generation (RAG) architecture, or any app that uses vectors.
* **Similarity search**. Encode text using embedding models such as OpenAI embeddings or open source models such as SBERT, and retrieve documents with queries that are also encoded as vectors.
* **Search across different content types (multimodal)**. Encode images and text using multimodal embeddings (for example, with OpenAI CLIP or GPT-4 Turbo with Vision in Azure OpenAI) and query an embedding space composed of vectors from both content types.
* **Hybrid search**. In Azure AI Search, hybrid search refers to **vector and keyword query execution** from the same request. Vector support is implemented at the field level, with an index containing both vector fields and searchable text fields. The queries execute in parallel and the **results are merged** into a single response. Optionally, add **semantic ranking** for more accuracy with **L2 reranking** using the same language models that power Bing.
* **Multilingual search**. Providing a search experience in the users own language is possible through embedding models and chat models trained in multiple languages. If you need more control over translation, you can supplement with the multi-language capabilities that Azure AI Search supports for nonvector content, in hybrid search scenarios.
* **Filtered vector search**. A query request can include a vector query and a filter expression. Filters apply to text and numeric fields, and are useful for metadata filters, and including or excluding search results based on filter criteria. Although a vector field isn't filterable itself, you can set up a filterable text or numeric field. The search engine can process the filter before or after the vector query executes.

**Azure integration and related services**

Azure AI Search is deeply integrated across the Azure AI platform. The following table lists several that are useful in vector workloads.

| **Product** | **Integration** |
| --- | --- |
| Azure OpenAI Studio | In the chat with your data playground, **Add your own data** uses Azure AI Search for grounding data and conversational search. This is the easiest and fastest approach for chatting with your data. |
| Azure OpenAI | Azure OpenAI provides embedding models and chat models. Demos and samples target the [text-embedding-ada-002](https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models#embeddings-models). We recommend Azure OpenAI for generating embeddings for text. |
| Azure AI Services | [Image Retrieval Vectorize Image API(Preview)](https://learn.microsoft.com/en-us/azure/ai-services/computer-vision/how-to/image-retrieval#call-the-vectorize-image-api) supports vectorization of image content. We recommend this API for generating embeddings for images. |
| Azure data platforms: Azure Blob Storage, Azure Cosmos DB | You can use [indexers](https://learn.microsoft.com/en-us/azure/search/search-indexer-overview) to automate data ingestion, and then use [integrated vectorization (preview)](https://learn.microsoft.com/en-us/azure/search/vector-search-integrated-vectorization) to generate embeddings. Azure AI Search can automatically index vector data from two data sources: [Azure blob indexers](https://learn.microsoft.com/en-us/azure/search/search-howto-indexing-azure-blob-storage) and [Azure Cosmos DB for NoSQL indexers](https://learn.microsoft.com/en-us/azure/search/search-howto-index-cosmosdb). For more information, see [Add vector fields to a search index.](https://learn.microsoft.com/en-us/azure/search/vector-search-how-to-create-index). |

It's also commonly used in open-source frameworks like [LangChain](https://js.langchain.com/docs/integrations/vectorstores/azure_aisearch).

**Vector search concepts**

If you're new to vectors, this section explains some core concepts.

**About vector search**

Vector search is a method of information retrieval where documents and queries are represented as vectors instead of plain text. In vector search, machine learning models generate the vector representations of source inputs, which can be text, images, or other content. Having a mathematic representation of content provides a common basis for search scenarios. If everything is a vector, a query can find a match in vector space, even if the associated original content is in different media or language than the query.

**Why use vector search**

When searchable content is represented as vectors, a query can find close matches in similar content. The embedding model used for vector generation knows which words and concepts are similar, and it places the resulting vectors close together in the embedding space. For example, vectorized source documents about "clouds" and "fog" are more likely to show up in a query about "mist" because they're semantically similar, even if they aren't a lexical match.

**Embeddings and vectorization**

Embeddings are a specific type of vector representation of content or a query, created by machine learning models that capture the semantic meaning of text or representations of other content such as images. Natural language machine learning models are trained on large amounts of data to identify patterns and relationships between words. During training, they learn to represent any input as a vector of real numbers in an intermediary step called the encoder. After training is complete, these language models can be modified so the intermediary vector representation becomes the model's output. The resulting embeddings are high-dimensional vectors, where words with similar meanings are closer together in the vector space, as explained in [Understand embeddings (Azure OpenAI)](https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/understand-embeddings).

The effectiveness of vector search in retrieving relevant information depends on the effectiveness of the embedding model in distilling the meaning of documents and queries into the resulting vector. The best models are well-trained on the types of data they're representing. You can evaluate existing models such as Azure OpenAI text-embedding-ada-002, bring your own model that's trained directly on the problem space, or fine-tune a general-purpose model. Azure AI Search doesn't impose constraints on which model you choose, so pick the best one for your data.

In order to create effective embeddings for vector search, it's important to take input size limitations into account. We recommend following the [guidelines for chunking data](https://learn.microsoft.com/en-us/azure/search/vector-search-how-to-chunk-documents) before generating embeddings. This best practice ensures that the embeddings accurately capture the relevant information and enable more efficient vector search.

**What is the embedding space?**

Embedding space is the corpus for vector queries. Within a search index, an embedding space is all of the vector fields populated with embeddings from the same embedding model. Machine learning models create the embedding space by mapping individual words, phrases, or documents (for natural language processing), images, or other forms of data into a representation comprised of a vector of real numbers representing a coordinate in a high-dimensional space. In this embedding space, similar items are located close together, and dissimilar items are located farther apart.

For example, documents that talk about different species of dogs would be clustered close together in the embedding space. Documents about cats would be close together, but farther from the dogs cluster while still being in the neighborhood for animals. Dissimilar concepts such as cloud computing would be much farther away. In practice, these embedding spaces are abstract and don't have well-defined, human-interpretable meanings, but the core idea stays the same.

**Nearest neighbors search**

In vector search, the search engine scans vectors within the embedding space to identify vectors that are closest to the query vector. This technique is called [nearest neighbor search](https://en.wikipedia.org/wiki/Nearest_neighbor_search). Nearest neighbors help quantify the similarity between items. A high degree of vector similarity indicates that the original data was similar too. To facilitate fast nearest neighbor search, the search engine performs optimizations, or employs data structures and data partitioning, to reduce the search space. Each vector search algorithm solves the nearest neighbor problems in different ways as they optimize for minimum latency, maximum throughput, recall, and memory. To compute similarity, similarity metrics provide the mechanism for computing distance.

Azure AI Search currently supports the following algorithms:

* **Hierarchical Navigable Small World (HNSW)**: HNSW is a leading ANN algorithm optimized for high-recall, low-latency applications where data distribution is unknown or can change frequently. It organizes high-dimensional data points into a hierarchical graph structure that enables fast and scalable similarity search while allowing a tunable a trade-off between search accuracy and computational cost. Because the algorithm requires all data points to reside in memory for fast random access, this algorithm consumes [vector index size](https://learn.microsoft.com/en-us/azure/search/vector-search-index-size) quota.
* **Exhaustive K-nearest neighbors (eKNN):** Calculates the distances between the query vector and all data points. It's computationally intensive, so it works best for smaller datasets. Because the algorithm doesn't require fast random access of data points, this algorithm doesn't consume vector index size quota. However, this algorithm provides the global set of nearest neighbors.

Within an index definition, you can specify one or more algorithms, and then for each vector field specify which algorithm use:

* [Create a vector store](https://learn.microsoft.com/en-us/azure/search/vector-search-how-to-create-index) to specify an algorithm in the index and on fields.
* For exhaustive KNN, use [2023-11-01](https://learn.microsoft.com/en-us/rest/api/searchservice/indexes/create-or-update), [2023-10-01-Preview](https://learn.microsoft.com/en-us/rest/api/searchservice/indexes/create-or-update?view=rest-searchservice-2023-10-01-preview&preserve-view=true), or Azure SDK beta libraries that target either REST API version.

Algorithm parameters that are used to initialize the index during index creation are immutable and can't be changed after the index is built. However, parameters that affect the query-time characteristics (efSearch) can be modified.

In addition, fields that specify HNSW algorithm also support exhaustive KNN search using the [query request](https://learn.microsoft.com/en-us/azure/search/vector-search-how-to-query) parameter "exhaustive": true. The opposite isn't true however. If a field is indexed for exhaustiveKnn, you can't use HNSW in the query because the extra data structures that enable efficient search don’t exist.

* **Approximate Nearest Neighbors**

Approximate Nearest Neighbor search (ANN) is a class of algorithms for finding matches in vector space. This class of algorithms employs different data structures or data partitioning methods to significantly reduce the search space to accelerate query processing.

ANN algorithms sacrifice some accuracy, but offer scalable and faster retrieval of approximate nearest neighbors, which makes them ideal for balancing accuracy against efficiency in modern information retrieval applications. You can adjust the parameters of your algorithm to fine-tune the recall, latency, memory, and disk footprint requirements of your search application.

**Azure AI Search uses HNSW for its ANN algorithm.**

## Algorithms used in vector search

Vector search algorithms include exhaustive k-nearest neighbors (KNN) and Hierarchical Navigable Small World (HNSW).

* Exhaustive KNN performs a brute-force search that scans the entire vector space.
* HNSW performs an [approximate nearest neighbor (ANN)](https://learn.microsoft.com/en-us/azure/search/vector-search-overview#approximate-nearest-neighbors) search.

Only vector fields marked as searchable in the index, or as searchFields in the query, are used for searching and scoring.

**When to use exhaustive KNN (eKNN)**

Exhaustive KNN calculates the distances between all pairs of data points and finds the exact k nearest neighbors for a query point. It's intended for scenarios where high recall is of utmost importance, and users are willing to accept the trade-offs in search performance. Because it's computationally intensive, use exhaustive KNN for small to medium datasets, or when precision requirements outweigh query performance considerations.

Another use case is to build a dataset to evaluate approximate nearest neighbor algorithm recall. Exhaustive KNN can be used to build the ground truth set of nearest neighbors.

Exhaustive KNN support is available through [2023-11-01 REST API](https://learn.microsoft.com/en-us/rest/api/searchservice/search-service-api-versions#2023-11-01), [2023-10-01-Preview REST API](https://learn.microsoft.com/en-us/rest/api/searchservice/search-service-api-versions#2023-10-01-Preview), and in Azure SDK client libraries that target either REST API version.

**When to use HNSW**

During indexing, HNSW creates extra data structures for faster search, organizing data points into a hierarchical graph structure. HHNSW has several configuration parameters that can be tuned to achieve the throughput, latency, and recall objectives for your search application. For example, at query time, you can specify options for exhaustive search, even if the vector field is indexed for HNSW.

During query execution, HNSW enables fast neighbor queries by navigating through the graph. This approach strikes a balance between search accuracy and computational efficiency. HNSW is recommended for most scenarios due to its efficiency when searching over larger data sets.

**How nearest neighbor search works**

Vector queries execute against an embedding space consisting of vectors generated from the same embedding model. Generally, the input value within a query request is fed into the same machine learning model that generated embeddings in the vector store. The output is a vector in the same embedding space. Since similar vectors are clustered close together, finding matches is equivalent to finding the vectors that are closest to the query vector, and returning the associated documents as the search result.

For example, if a query request is about hotels, the model maps the query into a vector that exists somewhere in the cluster of vectors representing documents about hotels. Identifying which vectors are the most similar to the query, based on a similarity metric, determines which documents are the most relevant.

When vector fields are indexed for exhaustive KNN, the query executes against "all neighbors". For fields indexed for HNSW, the search engine uses an HNSW graph to search over a subset of nodes within the vector store.

**Creating the HNSW graph**

During indexing, the search service constructs the HNSW graph. The goal of indexing a new vector into an HNSW graph is to add it to the graph structure in a manner that allows for efficient nearest neighbor search. The following steps summarize the process:

1. **Initialization**: Start with an empty HNSW graph, or the existing HNSW graph if it's not a new index.
2. **Entry point:** This is the top-level of the hierarchical graph and serves as the starting point for indexing.
3. **Adding to the graph**: Different hierarchical levels represent different granularities of the graph, with higher levels being more global, and lower levels being more granular. Each node in the graph represents a vector point.
   * Each node is connected to up to m neighbors that are nearby. This is the m parameter.
   * The number of data points considered as candidate connections is governed by the efConstruction parameter. This dynamic list forms the set of closest points in the existing graph for the algorithm to consider. Higher efConstruction values result in more nodes being considered, which often leads to denser local neighborhoods for each vector.
   * These connections use the configured similarity metric to determine distance. Some connections are "long-distance" connections that connect across different hierarchical levels, creating shortcuts in the graph that enhance search efficiency.
4. **Graph pruning and optimization**: This can happen after indexing all vectors, and it improves navigability and efficiency of the HNSW graph.

**Navigating the HNSW graph at query time**

A vector query navigates the hierarchical graph structure to scan for matches. The following summarize the steps in the process:

1. Initialization: The algorithm initiates the search at the top-level of the hierarchical graph. This entry point contains the set of vectors that serve as starting points for search.
2. Traversal: Next, it traverses the graph level by level, navigating from the top-level to lower levels, selecting candidate nodes that are closer to the query vector based on the configured distance metric, such as cosine similarity.
3. Pruning: To improve efficiency, the algorithm prunes the search space by only considering nodes that are likely to contain nearest neighbors. This is achieved by maintaining a priority queue of potential candidates and updating it as the search progresses. The length of this queue is configured by the parameter efSearch.
4. Refinement: As the algorithm moves to lower, more granular levels, HNSW considers more neighbors near the query, which allows the candidate set of vectors to be refined, improving accuracy.
5. Completion: The search completes when the desired number of nearest neighbors have been identified, or when other stopping criteria are met. This desired number of nearest neighbors is governed by the query-time parameter k.

**Similarity metrics used to measure nearness**

The algorithm finds candidate vectors to evaluate similarity. To perform this task, a similarity metric calculation compares the candidate vector to the query vector and measures the similarity. The algorithm keeps track of the ordered set of most similar vectors that its found, which forms the ranked result set when the algorithm has reached completion.

| **Metric** | **Description** |
| --- | --- |
| cosine | This metric measures the angle between two vectors, and isn't affected by differing vector lengths. Mathematically, it calculates the angle between two vectors. Cosine is the similarity metric used by [Azure OpenAI embedding models](https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/understand-embeddings#cosine-similarity), so if you're using Azure OpenAI, specify cosine in the vector configuration. |
| dotProduct | This metric measures both the length of each pair of two vectors, and the angle between them. Mathematically, it calculates the products of vectors' magnitudes and the angle between them. For normalized vectors, this is identical to cosine similarity, but slightly more performant. |
| euclidean | (also known as l2 norm) This metric measures the length of the vector difference between two vectors. Mathematically, it calculates the Euclidean distance between two vectors, which is the l2-norm of the difference of the two vectors. |

**Scores in a vector search results**

Scores are calculated and assigned to each match, with the highest matches returned as k results. The **@search.score** property contains the score. The following table shows the range within which a score will fall.

| **Search method** | **Parameter** | **Scoring metric** | **Range** |
| --- | --- | --- | --- |
| vector search | @search.score | Cosine | 0.333 - 1.00 |

For cosine metric, it's important to note that the calculated @search.score isn't the cosine value between the query vector and the document vectors. Instead, Azure AI Search applies transformations such that the score function is monotonically decreasing, meaning score values will always decrease in value as the similarity becomes worse. This transformation ensures that search scores are usable for ranking purposes.

There are some nuances with similarity scores:

* Cosine similarity is defined as the cosine of the angle between two vectors.
* Cosine distance is defined as 1 - cosine\_similarity.

To create a monotonically decreasing function, the @search.score is defined as 1 / (1 + cosine\_distance).

Having the original cosine value can be useful in custom solutions that set up thresholds to trim results of low quality results.

**Tips for relevance tuning**

If you aren't getting relevant results, experiment with changes to [query configuration](https://learn.microsoft.com/en-us/azure/search/vector-search-how-to-query). There are no specific tuning features, such as a scoring profile or field or term boosting, for vector queries:

* Experiment with [chunk size and overlap](https://learn.microsoft.com/en-us/azure/search/vector-search-how-to-chunk-documents). Try increasing the chunk size and ensuring there's sufficient overlap to preserve context or continuity between chunks.
* For HNSW, try different levels of efConstruction to change the internal composition of the proximity graph. The default is 400. The range is 100 to 1,000.
* Increase k results to feed more search results into a chat model, if you're using one.
* Try [hybrid queries](https://learn.microsoft.com/en-us/azure/search/hybrid-search-how-to-query) with semantic ranking. In benchmark testing, this combination consistently produced the most relevant results.

# **Full text (keyword) search**

Full text search is an approach in information retrieval that matches on plain text stored in an index. For example, given a query string "hotels in San Diego on the beach", the search engine looks for tokenized strings based on those terms. To make scans more efficient, query strings undergo **lexical analysis**: lower-casing all terms, removing stop words like "the", and reducing terms to primitive root forms. When matching terms are found, the search engine retrieves documents, ranks them in order of relevance, and returns the top results.

Query execution can be complex. This article is for developers who need a deeper understanding of how full text search works in Azure AI Search. For text queries, Azure AI Search seamlessly delivers expected results in most scenarios, but occasionally you might get a result that seems "off" somehow. In these situations, having a background in the four stages of **Lucene query execution** **(query parsing, lexical analysis, document matching, scoring)** can help you identify specific changes to query parameters or index configuration that produce the desired outcome.

**Note :** Azure AI Search uses [**Apache Lucene**](https://lucene.apache.org/) for full text search, but Lucene integration is not exhaustive. We selectively expose and extend Lucene functionality to enable the scenarios important to Azure AI Search.

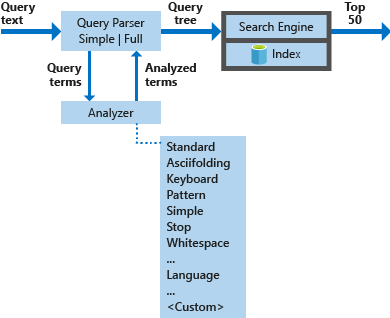
## Architecture overview and diagram

Query execution has four stages:

1. Query parsing
2. Lexical analysis
3. Document retrieval
4. Scoring

A full text search query starts with parsing the query text to extract search terms and operators. There are two parsers so that you can choose between speed and complexity. An analysis phase is next, where individual query terms are sometimes broken down and reconstituted into new forms. This step helps to cast a broader net over what could be considered as a potential match. The search engine then scans the index to find documents with matching terms and scores each match. A result set is then sorted by a relevance score assigned to each individual matching document. Those at the top of the ranked list are returned to the calling application.

The diagram below illustrates the components used to process a search request.



| **Key components** | **Functional description** |
| --- | --- |
| **Query parsers** | Separate query terms from query operators and create the query structure (a query tree) to be sent to the search engine. |
| **Analyzers** | Perform lexical analysis on query terms. This process can involve transforming, removing, or expanding of query terms. |
| **Index** | An efficient data structure used to store and organize searchable terms extracted from indexed documents. |
| **Search engine** | Retrieves and scores matching documents based on the contents of the inverted index. |

## Stage 1: Query parsing

As noted, the query string is the first line of the request:

"search": "Spacious, air-condition\* +\"Ocean view\"",

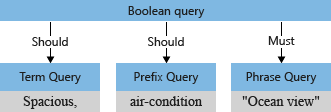
The query parser separates operators (such as \* and + in the example) from search terms, and deconstructs the search query into subqueries of a supported type:

* **term query** for standalone terms (like spacious)
* **phrase query** for quoted terms (like ocean view)
* **prefix query** for terms followed by a prefix operator \* (like air-condition)

For a full list of supported query types, see [Lucene query syntax](https://learn.microsoft.com/en-us/rest/api/searchservice/lucene-query-syntax-in-azure-search)

Operators associated with a subquery determine whether the query "must be" or "should be" satisfied in order for a document to be considered a match. For example, +"Ocean view" is "must" due to the + operator.

The query parser restructures the subqueries into a query tree (an internal structure representing the query) it passes on to the search engine. In the first stage of query parsing, the query tree looks like this.



### Supported parsers: Simple and Full Lucene

Azure AI Search exposes two different query languages, **simple (default) and full.** By setting the **queryType** parameter with your search request, you tell the query parser which query language you choose so that it knows how to interpret the operators and syntax.

* The [Simple query language](https://learn.microsoft.com/en-us/rest/api/searchservice/simple-query-syntax-in-azure-search) is intuitive and robust, often suitable to interpret user input as-is without client-side processing. It supports query operators familiar from web search engines.
* The [Full Lucene query language](https://learn.microsoft.com/en-us/rest/api/searchservice/lucene-query-syntax-in-azure-search), which you get by setting **queryType=full**, extends the default Simple query language by adding support for more operators and query types like wildcard, fuzzy, regex, and field-scoped queries. For example, a regular expression sent in Simple query syntax would be interpreted as a query string and not an expression. The example request in this article uses the Full Lucene query language.

### Impact of searchMode on the parser

Another search request parameter that affects parsing is the "**searchMode**" parameter. It controls the default operator for Boolean queries: any (default) or all.

When "**searchMode=any**", which is the default, the space delimiter between spacious and air-condition is OR (||), making the sample query text equivalent to:

Spacious,||air-condition\*+"Ocean view"

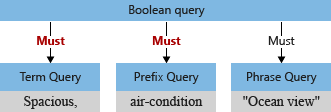
Explicit operators, such as + in +"Ocean view", are unambiguous in boolean query construction (the term must match). Less obvious is how to interpret the remaining terms: spacious and air-condition. Should the search engine find matches on ocean view and spacious and air-condition? Or should it find ocean view plus either one of the remaining terms?

By default ("searchMode=any"), the search engine assumes the broader interpretation. Either field should be matched, reflecting "or" semantics. The initial query tree illustrated previously, with the two "should" operations, shows the default.

Suppose that we now set **"searchMode=all"**. In this case, the space is interpreted as an "and" operation. Each of the remaining terms must both be present in the document to qualify as a match. The resulting sample query would be interpreted as follows:

+Spacious,+air-condition\*+"Ocean view"

A modified query tree for this query would be as follows, where a matching document is the intersection of all three subqueries:



**Note :** Choosing "searchMode=any" over "searchMode=all" is a decision best arrived at by running representative queries. Users who are likely to include operators (common when searching document stores) might find results more intuitive if "searchMode=all" informs boolean query constructs. For more about the interplay between "searchMode" and operators, see [**Simple query syntax**](https://learn.microsoft.com/en-us/rest/api/searchservice/simple-query-syntax-in-azure-search).

## Stage 2: Lexical analysis

Lexical analyzers process **term queries and phrase queries** after the query tree is structured. An analyzer accepts the text inputs given to it by the parser, processes the text, and then **sends back tokenized terms** to be incorporated into the query tree.

The most common form of lexical analysis is \***linguistic analysis** that transforms query terms based on rules specific to a given language:

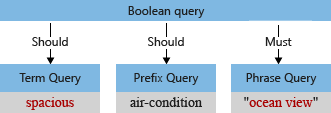
* Reducing a query term to the root form of a word
* Removing non-essential words ([stopwords](https://learn.microsoft.com/en-us/azure/search/reference-stopwords), such as "the" or "and" in English)
* Breaking a composite word into component parts
* Lower casing an upper case word

All of these operations tend to erase differences between the text input provided by the user and the terms stored in the index. Such operations go beyond text processing and require in-depth knowledge of the language itself. To add this layer of linguistic awareness, Azure AI Search supports a long list of [language analyzers](https://learn.microsoft.com/en-us/rest/api/searchservice/language-support) from both Lucene and Microsoft.

**Note :** Analysis requirements can range from minimal to elaborate depending on your scenario. You can control complexity of lexical analysis by the selecting one of the predefined analyzers or by creating your own [**custom analyzer**](https://learn.microsoft.com/en-us/rest/api/searchservice/Custom-analyzers-in-Azure-Search). Analyzers are scoped to searchable fields and are specified as part of a field definition. This allows you to vary lexical analysis on a per-field basis. Unspecified, the standard Lucene analyzer is used.

In our example, prior to analysis, the initial query tree has the term "Spacious," with an uppercase "S" and a comma that the query parser interprets as a part of the query term (a comma isn't considered a query language operator).

When the default analyzer processes the term, it will lowercase "ocean view" and "spacious", and remove the comma character. The modified query tree looks like:



### Testing analyzer behaviors

The behavior of an analyzer can be tested using the [Analyze API](https://learn.microsoft.com/en-us/rest/api/searchservice/test-analyzer). Provide the text you want to analyze to see what terms given analyzer generates. For example, to see how the standard analyzer would process the text "air-condition", you can issue the following request:

JSON

{

"text": "air-condition",

"analyzer": "standard"

}

The standard analyzer breaks the input text into the following two tokens, annotating them with attributes like **start and end offsets** (used for hit highlighting) as well as their **position** (used for phrase matching):

JSON

{

"tokens": [

{

"token": "air",

"startOffset": 0,

"endOffset": 3,

"position": 0

},

{

"token": "condition",

"startOffset": 4,

"endOffset": 13,

"position": 1

}

]

}

### Exceptions to lexical analysis

Lexical analysis applies only to query types that require complete terms – either a term query or a phrase query. It **doesn’t apply** to query types with **incomplete terms** – prefix query, wildcard query, regex query – or to a fuzzy query. Those query types, including the prefix query with term air-condition\* in our example, are added directly to the query tree, bypassing the analysis stage. The only transformation performed on query terms of those types is lowercasing.

## Stage 3: Document retrieval

Document retrieval refers to finding documents with matching terms in the index. This stage is understood best through an example. Let's start with a hotels index having the following simple schema:

JSON

{

"name": "hotels",

"fields": [

{ "name": "id", "type": "Edm.String", "key": true, "searchable": false },

{ "name": "title", "type": "Edm.String", "searchable": true },

{ "name": "description", "type": "Edm.String", "searchable": true }

]

}

Further assume that this index contains the following four documents:

JSON

{

"value": [

{

"id": "1",

"title": "Hotel Atman",

"description": "Spacious rooms, ocean view, walking distance to the beach."

},

{

"id": "2",

"title": "Beach Resort",

"description": "Located on the north shore of the island of Kauaʻi. Ocean view."

},

{

"id": "3",

"title": "Playa Hotel",

"description": "Comfortable, air-conditioned rooms with ocean view."

},

{

"id": "4",

"title": "Ocean Retreat",

"description": "Quiet and secluded"

}

]

}

**How terms are indexed**

To understand retrieval, it helps to know a few basics about indexing. The unit of storage is an inverted index, one for each searchable field. Within an inverted index is a sorted list of all terms from all documents. Each term maps to the list of documents in which it occurs, as evident in the example below.

To produce the terms in an inverted index, the search engine performs lexical analysis over the content of documents, similar to what happens during query processing:

1. **Text inputs** are passed to an analyzer, lower-cased, stripped of punctuation, and so forth, depending on the analyzer configuration.
2. **Tokens** are the output of lexical analysis.
3. **Terms**are added to the index.

It's common, but not required, to use the same analyzers for search and indexing operations so that query terms look more like terms inside the index.

**Note :** Azure AI Search lets you specify different analyzers for indexing and search via additional indexAnalyzer and searchAnalyzer field parameters. If unspecified, the analyzer set with the analyzer property is used for both indexing and searching.

**Inverted index for example documents**

Returning to our example, for the **title** field, the inverted index looks like this:

| **Term** | **Document list** |
| --- | --- |
| atman | 1 |
| beach | 2 |
| hotel | 1, 3 |
| ocean | 4 |
| playa | 3 |
| resort | 3 |
| retreat | 4 |

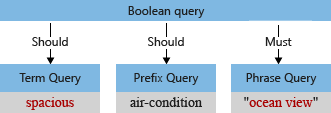
In the title field, only hotel shows up in two documents: 1, 3.

For the **description** field, the index is as follows:

| **Term** | **Document list** |
| --- | --- |
| air | 3 |
| and | 4 |
| beach | 1 |
| conditioned | 3 |
| comfortable | 3 |
| distance | 1 |
| island | 2 |
| kauaʻi | 2 |
| located | 2 |
| north | 2 |
| ocean | 1, 2, 3 |
| of | 2 |
| on | 2 |
| quiet | 4 |
| rooms | 1, 3 |
| secluded | 4 |
| shore | 2 |
| spacious | 1 |
| the | 1, 2 |
| to | 1 |
| view | 1, 2, 3 |
| walking | 1 |
| with | 3 |

**Matching query terms against indexed terms**

Given the inverted indexes above, let’s return to the sample query and see how matching documents are found for our example query. Recall that the final query tree looks like this:



During query execution, individual queries are executed against the searchable fields independently.

* The TermQuery, "spacious", matches document 1 (Hotel Atman).
* The PrefixQuery, "air-condition\*", doesn't match any documents.

This is a behavior that sometimes confuses developers. Although the term air-conditioned exists in the document, it's split into two terms by the default analyzer. Recall that prefix queries, which contain partial terms, aren't analyzed. Therefore terms with prefix "air-condition" are looked up in the inverted index and not found.

* The PhraseQuery, "ocean view", looks up the terms "ocean" and "view" and checks the proximity of terms in the original document. Documents 1, 2 and 3 match this query in the description field. Notice document 4 has the term ocean in the title but isn’t considered a match, as we're looking for the "ocean view" phrase rather than individual words.

**Note :** A search query is executed independently against all searchable fields in the Azure AI Search index unless you limit the fields set with the searchFields parameter, as illustrated in the example search request. Documents that match in any of the selected fields are returned.

On the whole, for the query in question, the documents that match are 1, 2, 3.

## Stage 4: Scoring

Every document in a search result set is assigned a relevance score. The function of the relevance score is to rank higher those documents that best answer a user question as expressed by the search query. The score is computed based on statistical properties of terms that matched. At the core of the scoring formula is [TF/IDF (term frequency-inverse document frequency)](https://en.wikipedia.org/wiki/Tf%E2%80%93idf). In queries containing rare and common terms, TF/IDF promotes results containing the rare term. For example, in a hypothetical index with all Wikipedia articles, from documents that matched the query the president, documents matching on president are considered more relevant than documents matching on the.

### Scoring example

Recall the three documents that matched our example query:

Search = Spacious, air-condition\* +"Ocean view"

JSON

{

"value": [

{

"@search.score": 0.25610128,

"id": "1",

"title": "Hotel Atman",

"description": "Spacious rooms, ocean view, walking distance to the beach."

},

{

"@search.score": 0.08951007,

"id": "3",

"title": "Playa Hotel",

"description": "Comfortable, air-conditioned rooms with ocean view."

},

{

"@search.score": 0.05967338,

"id": "2",

"title": "Ocean Resort",

"description": "Located on a cliff on the north shore of the island of Kauai. Ocean view."

}

]

}

Document 1 matched the query best because both the term spacious and the required phrase ocean view occur in the description field. The next two documents match only the phrase ocean view. It might be surprising that the relevance score for document 2 and 3 is different even though they matched the query in the same way. It's because the scoring formula has more components than just TF/IDF. In this case, document 3 was assigned a slightly higher score because its description is shorter. Learn about [Lucene's Practical Scoring Formula](https://lucene.apache.org/core/6_6_1/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html) to understand how field length and other factors can influence the relevance score.

Some query types (wildcard, prefix, regex) always contribute a constant score to the overall document score. This allows matches found through query expansion to be included in the results, but without affecting the ranking.

An example illustrates why this matters. Wildcard searches, including prefix searches, are ambiguous by definition because the input is a partial string with potential matches on a very large number of disparate terms (consider an input of "tour\*", with matches found on “tours”, “tourettes”, and “tourmaline”). Given the nature of these results, there's no way to reasonably infer which terms are more valuable than others. For this reason, we ignore term frequencies when scoring results in queries of types wildcard, prefix and regex. In a multi-part search request that includes partial and complete terms, results from the partial input are incorporated with a constant score to avoid bias towards potentially unexpected matches.

### Relevance tuning

There are two ways to tune relevance scores in Azure AI Search:

1. **Scoring profiles** promote documents in the ranked list of results based on a set of rules. In our example, we could consider documents that matched in the title field more relevant than documents that matched in the description field. Additionally, if our index had a price field for each hotel, we could promote documents with lower price. Learn more about [adding scoring profiles to a search index](https://learn.microsoft.com/en-us/rest/api/searchservice/add-scoring-profiles-to-a-search-index).
2. **Term boosting** (available only in the Full Lucene query syntax) provides a boosting operator ^ that can be applied to any part of the query tree. In our example, instead of searching on the prefix air-condition\*, one could search for either the exact term air-condition or the prefix, but documents that match on the exact term are ranked higher by applying boost to the term query: air-condition^2||air-condition\*. Learn more about [term boosting in a query](https://learn.microsoft.com/en-us/rest/api/searchservice/lucene-query-syntax-in-azure-search#bkmk_termboost).

### Scoring in a distributed index

All indexes in Azure AI Search are automatically split into multiple shards, allowing us to quickly distribute the index among multiple nodes during service scale up or scale down. When a search request is issued, it’s issued against each shard independently. The results from each shard are then merged and ordered by score (if no other ordering is defined). It's important to know that the scoring function weights query term frequency against its inverse document frequency in all documents within the shard, not across all shards!

This means a relevance score could be different for identical documents if they reside on different shards. Fortunately, such differences tend to disappear as the number of documents in the index grows due to more even term distribution. It’s not possible to assume on which shard any given document will be placed. However, assuming a document key doesn't change, it will always be assigned to the same shard.

In general, document score isn't the best attribute for ordering documents if order stability is important. For example, given two documents with an identical score, there's no guarantee that one appears first in subsequent runs of the same query. Document score should only give a general sense of document relevance relative to other documents in the results set.

This article explains the BM25 relevance scoring algorithm used to compute search scores for [full text search](https://learn.microsoft.com/en-us/azure/search/search-lucene-query-architecture). BM25 relevance is exclusive to full text search. Filter queries, autocomplete and suggested queries, wildcard search or fuzzy search queries aren't scored or ranked for relevance.

## Algorithms used in full text (keyword) search

Azure AI Search provides the following scoring algorithms for full text search:

| **Algorithm** | **Usage** | **Range** |
| --- | --- | --- |
| BM25Similarity | Fixed algorithm on all search services created after July 2020. You can configure this algorithm, but you can't switch to an older one (classic). | Unbounded. |
| ClassicSimilarity | Present on older search services. You can [opt-in for BM25](https://learn.microsoft.com/en-us/azure/search/index-ranking-similarity) and choose an algorithm on a per-index basis. | 0 < 1.00 |

Both BM25 and Classic **are TF-IDF-like retrieval functions** that use the term frequency (TF) and the inverse document frequency (IDF) as variables to calculate relevance scores for each document-query pair, which is then used for ranking results. While conceptually similar to classic, BM25 is rooted in probabilistic information retrieval that produces more intuitive matches, as measured by user research.

BM25 offers advanced customization options, such as allowing the user to decide how the relevance score scales with the term frequency of matched terms. For more information, see [Configure the scoring algorithm](https://learn.microsoft.com/en-us/azure/search/index-ranking-similarity).

**Note :** If you're using a search service that was created before July 2020, the scoring algorithm is most likely the previous default, ClassicSimilarity, which you can upgrade on a per-index basis. See [**Enable BM25 scoring on older services**](https://learn.microsoft.com/en-us/azure/search/index-ranking-similarity#enable-bm25-scoring-on-older-services) for details.

The following video segment fast-forwards to an explanation of the generally available ranking algorithms used in Azure AI Search. You can watch the full video for more background.

<https://youtu.be/Y_X6USgvB1g>

## How BM25 ranking works

Relevance scoring refers to the computation of a search score (**@search.score**) that serves as an indicator of an item's relevance in the context of the current query. The range is unbounded. However, the higher the score, the more relevant the item.

The search score is computed based on statistical properties of the string input and the query itself. Azure AI Search finds documents that match on search terms (some or all, depending on [searchMode](https://learn.microsoft.com/en-us/rest/api/searchservice/search-documents#query-parameters)), favoring documents that contain many instances of the search term. The search score goes up even higher if the term is rare across the data index, but common within the document. The basis for this approach to computing relevance is known as TF-IDF or term frequency-inverse document frequency.

Search scores can be repeated throughout a result set. When multiple hits have the same search score, the ordering of the same scored items is undefined and not stable. Run the query again, and you might see items shift position, especially if you're using the free service or a billable service with multiple replicas. Given two items with an identical score, there's no guarantee that one appears first.

To break the tie among repeating scores, you can add an **$orderby** clause to first order by score, then order by another sortable field (for example, $orderby=search.score() desc,Rating desc). For more information, see [$orderby](https://learn.microsoft.com/en-us/azure/search/search-query-odata-orderby).

Only fields marked as searchable in the index, or searchFields in the query, are used for scoring. Only fields marked as retrievable, or fields specified in select in the query, are returned in search results, along with their search score.

**Note:** A @search.score = 1 indicates an un-scored or un-ranked result set. The score is uniform across all results. Un-scored results occur when the query form is fuzzy search, wildcard or regex queries, or an empty search (search=\*, sometimes paired with filters, where the filter is the primary means for returning a match).

## Scores in a text results

Whenever results are ranked, **@search.score** property contains the value used to order the results.

The following table identifies the scoring property returned on each match, algorithm, and range.

| **Search method** | **Parameter** | **Scoring algorithm** | **Range** |
| --- | --- | --- | --- |
| full text search | @search.score | BM25 algorithm, using the [parameters specified in the index](https://learn.microsoft.com/en-us/azure/search/index-ranking-similarity#set-bm25-parameters). | Unbounded. |

### Score variation

Search scores convey general sense of relevance, reflecting the strength of match relative to other documents in the same result set. But scores aren't always consistent from one query to the next, so as you work with queries, you might notice small discrepancies in how search documents are ordered. There are several explanations for why this might occur.

| **Cause** | **Description** |
| --- | --- |
| Data volatility | Index content varies as you add, modify, or delete documents. Term frequencies will change as index updates are processed over time, affecting the search scores of matching documents. |
| Multiple replicas | For services using multiple replicas, queries are issued against each replica in parallel. The index statistics used to calculate a search score are calculated on a per-replica basis, with results merged and ordered in the query response. Replicas are mostly mirrors of each other, but statistics can differ due to small differences in state. For example, one replica might have deleted documents contributing to their statistics, which were merged out of other replicas. Typically, differences in per-replica statistics are more noticeable in smaller indexes. For more information about this condition, see [Concepts: search units, replicas, partitions, shards](https://learn.microsoft.com/en-us/azure/search/search-capacity-planning#concepts-search-units-replicas-partitions-shards) in the capacity planning documentation. |
| Identical scores | If multiple documents have the same score, any one of them might appear first. |

### Scoring statistics and sticky sessions

For scalability, Azure AI Search distributes each index horizontally through a sharding process, which means that [portions of an index are physically separate](https://learn.microsoft.com/en-us/azure/search/search-capacity-planning#concepts-search-units-replicas-partitions-shards).

By default, the score of a document is calculated based on statistical properties of the data within a shard. This approach is generally not a problem for a large corpus of data, and it provides better performance than having to calculate the score based on information across all shards. That said, using this performance optimization could cause two very similar documents (or even identical documents) to end up with different relevance scores if they end up in different shards.

If you prefer to compute the score based on the statistical properties across all shards, you can do so by adding scoringStatistics=global as a [query parameter](https://learn.microsoft.com/en-us/rest/api/searchservice/search-documents) (or add "scoringStatistics": "global" as a body parameter of the [query request](https://learn.microsoft.com/en-us/rest/api/searchservice/search-documents)).

HTTP

POST https://[service name].search.windows.net/indexes/hotels/docs/search?api-version=2020-06-30

{

"search": "<query string>",

"scoringStatistics": "global"

}

Using scoringStatistics will ensure that all shards in the same replica provide the same results. That said, different replicas may be slightly different from one another as they're always getting updated with the latest changes to your index. In some scenarios, you may want your users to get more consistent results during a "query session". In such scenarios, you can provide a sessionId as part of your queries. The sessionId is a unique string that you create to refer to a unique user session.

HTTP

POST https://[service name].search.windows.net/indexes/hotels/docs/search?api-version=2020-06-30

{

"search": "<query string>",

"sessionId": "<string>"

}

As long as the same sessionId is used, a best-effort attempt is made to target the same replica, increasing the consistency of results your users will see.

**Note:** Reusing the same sessionId values repeatedly can interfere with the load balancing of the requests across replicas and adversely affect the performance of the search service. The value used as sessionId cannot start with a '\_' character.

## Relevance tuning

In Azure AI Search, you can configure BM25 algorithm parameters, and tune search relevance and boost search scores through these mechanisms:

| **Approach** | **Implementation** | **Description** |
| --- | --- | --- |
| [Scoring algorithm configuration](https://learn.microsoft.com/en-us/azure/search/index-ranking-similarity) | Search index |  |
| [Scoring profiles](https://learn.microsoft.com/en-us/azure/search/index-add-scoring-profiles) | Search index | Provide criteria for boosting the search score of a match based on content characteristics. For example, you might want to boost matches based on their revenue potential, promote newer items, or perhaps boost items that have been in inventory too long. A scoring profile is part of the index definition, composed of weighted fields, functions, and parameters. You can update an existing index with scoring profile changes, without incurring an index rebuild. |
| [Semantic ranking](https://learn.microsoft.com/en-us/azure/search/semantic-search-overview) | Query request | Applies machine reading comprehension to search results, promoting more semantically relevant results to the top. |
| [featuresMode parameter](https://learn.microsoft.com/en-us/azure/search/index-similarity-and-scoring#featuresmode-parameter-preview) | Query request | This parameter is mostly used for unpacking a score, but it can be used for in code that provides a [custom scoring solution](https://github.com/Azure-Samples/search-ranking-tutorial). |

## featuresMode parameter (preview)

[Search Documents](https://learn.microsoft.com/en-us/rest/api/searchservice/preview-api/search-documents) requests have a new [featuresMode](https://learn.microsoft.com/en-us/rest/api/searchservice/preview-api/search-documents#featuresmode) parameter that can provide more detail about relevance at the field level. Whereas the @searchScore is calculated for the document all-up (how relevant is this document in the context of this query), through featuresMode you can get information about individual fields, as expressed in a @search.features structure. The structure contains all fields used in the query (either specific fields through **searchFields** in a query, or all fields attributed as **searchable** in an index). For each field, you get the following values:

* Number of unique tokens found in the field
* Similarity score, or a measure of how similar the content of the field is, relative to the query term
* Term frequency, or the number of times the query term was found in the field

For a query that targets the "description" and "title" fields, a response that includes @search.features might look like this:

JSON

"value": [

{

"@search.score": 5.1958685,

"@search.features": {

"description": {

"uniqueTokenMatches": 1.0,

"similarityScore": 0.29541412,

"termFrequency" : 2

},

"title": {

"uniqueTokenMatches": 3.0,

"similarityScore": 1.75451557,

"termFrequency" : 6

}

}

}

]

You can consume these data points in [custom scoring solutions](https://github.com/Azure-Samples/search-ranking-tutorial) or use the information to debug search relevance problems.

## Number of ranked results in a full text query response

By default, if you aren't using pagination, the search engine returns the top 50 highest ranking matches for full text search. You can use the top parameter to return a smaller or larger number of items (up to 1000 in a single response). Full text search is subject to a maximum limit of 1,000 matches (see [API response limits](https://learn.microsoft.com/en-us/azure/search/search-limits-quotas-capacity#api-response-limits)). Once 1,000 matches are found, the search engine no longer looks for more.

To return more or less results, use the paging parameters top, skip, and next. Paging is how you determine the number of results on each logical page and navigate through the full payload. For more information, see [How to work with search results](https://learn.microsoft.com/en-us/azure/search/search-pagination-page-layout).

# **Semantic ranking in Azure AI Search**

In Azure AI Search, semantic ranking measurably improves search relevance by using language understanding to rerank search results. This article is a high-level introduction.

## What is semantic ranking?

Semantic ranker is a collection of query-related capabilities that improve the quality of an initial [BM25-ranked](https://learn.microsoft.com/en-us/azure/search/index-similarity-and-scoring) or [RRF-ranked](https://learn.microsoft.com/en-us/azure/search/hybrid-search-ranking) search result for text-based queries. When you enable it on your search service, semantic ranking extends the query execution pipeline in two ways:

* First, it **adds secondary ranking** over an initial result set that was scored using BM25 or RRF. This secondary ranking uses **multi-lingual, deep learning models** adapted from Microsoft Bing to promote the most semantically relevant results.
* Second, it **extracts and returns captions and answers** in the response, which you can render on a search page to improve the user's search experience.

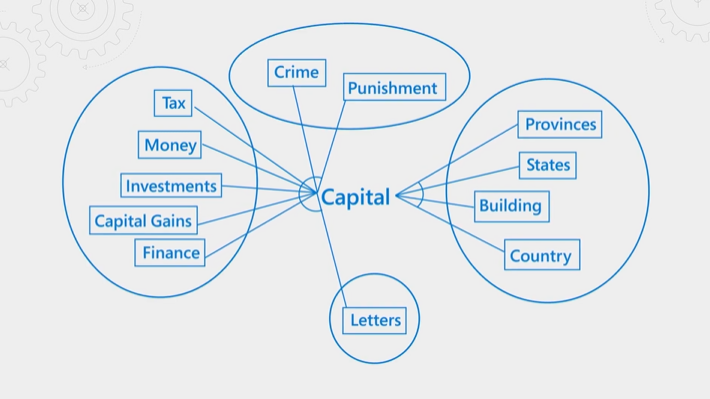
Here are the capabilities of the semantic reranker.

| **Feature** | **Description** |
| --- | --- |
| Semantic ranking | Uses the context or semantic meaning of a query to compute a new relevance score over preranked results. |
| [Semantic captions and highlights](https://learn.microsoft.com/en-us/azure/search/semantic-how-to-query-request) | Extracts verbatim sentences and phrases from a document that best summarize the content, with highlights over key passages for easy scanning. Captions that summarize a result are useful when individual content fields are too dense for the search results page. Highlighted text elevates the most relevant terms and phrases so that users can quickly determine why a match was considered relevant. |
| [Semantic answers](https://learn.microsoft.com/en-us/azure/search/semantic-answers) | An optional and extra substructure returned from a semantic query. It provides a direct answer to a query that looks like a question. It requires that a document has text with the characteristics of an answer. |

## How semantic ranker works

Semantic ranking feeds a query and results to language understanding models hosted by Microsoft and scans for better matches.

The following illustration explains the concept. Consider the term "capital". It has different meanings depending on whether the context is finance, law, geography, or grammar. Through language understanding, the semantic ranker can detect context and promote results that fit query intent.



Semantic ranking is both resource and time intensive. In order to complete processing within the expected latency of a query operation, inputs to the semantic ranker are consolidated and reduced so that the reranking step can be completed as quickly as possible.

There are two steps to semantic ranking: **summarization and scoring**. Outputs consist of rescored **results, captions, and answers**.

### How inputs are collected and summarized

In semantic ranking, the query subsystem passes search results as an input to summarization and ranking models. Because the ranking models have input size constraints and are processing intensive, search results must be sized and structured (summarized) for efficient handling.

1. Semantic ranking starts with a [BM25-ranked result](https://learn.microsoft.com/en-us/azure/search/index-ranking-similarity) from a text query or an [RRF-ranked result](https://learn.microsoft.com/en-us/azure/search/hybrid-search-ranking) from a hybrid query. Only text fields are used in the reranking exercise, and only the **top 50 results** progress to semantic ranking, even if results include more than 50. Typically, fields used in semantic ranking are informational and descriptive.
2. For each document in the search result, the summarization model accepts up to 2,000 tokens, where a token is approximately 10 characters. Inputs are assembled from the "title", "keyword", and "content" fields listed in the [semantic configuration](https://learn.microsoft.com/en-us/azure/search/semantic-how-to-configure).
3. Excessively long strings are trimmed to ensure the overall length meets the input requirements of the summarization step. This trimming exercise is why it's important to add fields to your semantic configuration in priority order. If you have very large documents with text-heavy fields, anything after the maximum limit is ignored.

| **Semantic field** | **Token limit** |
| --- | --- |
| "title" | 128 tokens |
| "keywords | 128 tokens |
| "content" | remaining tokens |

1. Summarization output is a summary string for each document, composed of the most relevant information from each field. Summary strings are sent to the ranker for scoring, and to machine reading comprehension models for captions and answers.

The maximum length of each generated summary string passed to the semantic ranker is 256 tokens.

### Outputs of semantic ranker

From each summary string, the machine reading comprehension models find passages that are the most representative.

Outputs are:

* A [semantic caption](https://learn.microsoft.com/en-us/azure/search/semantic-how-to-query-request) for the document. Each caption is available in a plain text version and a highlight version, and is frequently fewer than 200 words per document.
* An optional [semantic answer](https://learn.microsoft.com/en-us/azure/search/semantic-answers), assuming you specified the answers parameter, the query was posed as a question, and a passage is found in the long string that provides a likely answer to the question.

Captions and answers are always verbatim text from your index. There's no generative AI model in this workflow that creates or composes new content.

### How summaries are scored

Scoring is done over the caption, and any other content from the summary string that fills out the 256 token length.

1. Captions are evaluated for conceptual and semantic relevance, relative to the query provided.
2. A **@search.rerankerScore** is assigned to each document based on the semantic relevance of the document for the given query. Scores range from 4 to 0 (high to low), where a higher score indicates higher relevance.
3. Matches are listed in descending order by score and included in the query response payload. The payload includes answers, plain text and highlighted captions, and any fields that you marked as retrievable or specified in a select clause.

**Note**

Beginning on July 14, 2023, the **@search.rerankerScore** distribution is changing. The effect on scores can't be determined except through testing. If you have a hard threshold dependency on this response property, rerun your tests to understand what the new values should be for your threshold.

## Semantic capabilities and limitations

Semantic ranker is a newer technology so it's important to set expectations about what it can and can't do. What it can do:

* Promote matches that are semantically closer to the intent of original query.
* Find strings to use as captions and answers. Captions and answers are returned in the response and can be rendered on a search results page.

What semantic ranking can't do is rerun the query over the entire corpus to find semantically relevant results. Semantic ranking reranks the existing result set, consisting of the top 50 results as scored by the default ranking algorithm. Furthermore, semantic ranking can't create new information or strings. Captions and answers are extracted verbatim from your content so if the results don't include answer-like text, the language models won't produce one.

Although semantic ranking isn't beneficial in every scenario, certain content can benefit significantly from its capabilities. The language models in semantic ranking work best on searchable content that is information-rich and structured as prose. A knowledge base, online documentation, or documents that contain descriptive content see the most gains from semantic ranking capabilities.

The underlying technology is from Bing and Microsoft Research, and integrated into the Azure AI Search infrastructure as an add-on feature. For more information about the research and AI investments backing semantic ranking, see [How AI from Bing is powering Azure AI Search (Microsoft Research Blog)](https://www.microsoft.com/research/blog/the-science-behind-semantic-search-how-ai-from-bing-is-powering-azure-cognitive-search/).

The following video provides an overview of the capabilities.

<https://youtu.be/yOf0WfVd_V0>

When you enable semantic ranker, choose a pricing plan for the feature:

* At lower query volumes (under 1000 monthly), semantic ranking is free.
* At higher query volumes, choose the standard pricing plan.

Charges for semantic ranking are levied when query requests include queryType=semantic and the search string isn't empty (for example, search=pet friendly hotels in New York). If your search string is empty (search=\*), you aren't charged, even if the queryType is set to semantic.

# **Hybrid search (vectors + full text keyword)**

Hybrid search is a combination of full text and vector queries that execute against a search index that contains both searchable plain text content and generated embeddings. For query purposes, hybrid search is:

* A single query request that includes both **search** and **vectors** query parameters
* Executing in parallel
* With merged results in the query response, scored using [Reciprocal Rank Fusion (RRF)](https://learn.microsoft.com/en-us/azure/search/hybrid-search-ranking)

This article explains the concepts, benefits, and limitations of hybrid search. Watch this [embedded video](https://learn.microsoft.com/en-us/azure/search/hybrid-search-overview#why-choose-hybrid-search) for an explanation and short demos of how hybrid retrieval contributes to high quality chat-style and copilot apps.

## How does hybrid search work?

In Azure AI Search, vector fields containing embeddings can live alongside textual and numerical fields, allowing you to formulate hybrid queries that execute in parallel. Hybrid queries can take advantage of existing functionality like filtering, faceting, sorting, scoring profiles, and [semantic ranking](https://learn.microsoft.com/en-us/azure/search/semantic-search-overview) in a single search request.

Hybrid search combines results from both full text and vector queries, which use different ranking functions such as **BM25 and HNSW**. A [**Reciprocal Rank Fusion (RRF)**](https://learn.microsoft.com/en-us/azure/search/hybrid-search-ranking) algorithm merges the results. The query response provides just one result set, using RRF to pick the most relevant matches from each query.

## Structure of a hybrid query

Hybrid search is predicated on having a search index that contains fields of various [data types](https://learn.microsoft.com/en-us/rest/api/searchservice/supported-data-types), including plain text and numbers, geo coordinates for geospatial search, and vectors for a mathematical representation of a chunk of text. You can use almost all query capabilities in Azure AI Search with a vector query, **except** for client-side interactions such as **autocomplete and suggestions.**

A representative hybrid query might be as follows (notice the vector is trimmed for brevity):

HTTP

POST https://{{searchServiceName}}.search.windows.net/indexes/hotels-vector-quickstart/docs/search?api-version=2023-11-01

content-type: application/JSON

{

"count": true,

"search": "historic hotel walk to restaurants and shopping",

"select": "HotelId, HotelName, Category, Description, Address/City, Address/StateProvince",

"filter": "geo.distance(Location, geography'POINT(-77.03241 38.90166)') le 300",

"facets": [ "Address/StateProvince"],

"vectors": [

{

"value": [ <array of embeddings> ]

"k": 7,

"fields": "DescriptionVector"

},

{

"value": [ <array of embeddings> ]

"k": 7,

"fields": "Description\_frVector"

}

],

"queryType": "semantic",

"queryLanguage": "en-us",

"semanticConfiguration": "my-semantic-config"

}

Key points include:

* **search** specifies a full text search query.
* **vectors** for vector queries, which can be multiple, targeting multiple vector fields. If the embedding space includes multi-lingual content, vector queries can find the match with no language analyzers or translation required.
* **select** specifies which fields to return in results, which can be text fields that are human readable.
* **filters** can specify geospatial search or other include and exclude criteria, such as whether parking is included. The geospatial query in this example finds hotels within a 300-kilometer radius of Washington D.C.
* **facets** can be used to compute facet buckets over results that are returned from hybrid queries.
* **queryType=semantic** invokes semantic ranking, applying machine reading comprehension to surface more relevant search results.

Filters and facets target data structures within the index that are distinct from the inverted indexes used for full text search and the vector indexes used for vector search. As such, when filters and faceted operations execute, the search engine can apply the operational result to the hybrid search results in the response.

**Notice**: how there's no **orderby** in the query. Explicit sort orders override relevanced-ranked results, so if you want similarity and BM25 relevance, omit sorting in your query.

A response from the above query might look like this:

HTTPCopy

{

"@odata.count": 3,

"@search.facets": {

"Address/StateProvince": [

{

"count": 1,

"value": "NY"

},

{

"count": 1,

"value": "VA"

}

]

},

"value": [

{

"@search.score": 0.03333333507180214,

"@search.rerankerScore": 2.5229012966156006,

"HotelId": "49",

"HotelName": "Old Carrabelle Hotel",

"Description": "Spacious rooms, glamorous suites and residences, rooftop pool, walking access to shopping, dining, entertainment and the city center.",

"Category": "Luxury",

"Address": {

"City": "Arlington",

"StateProvince": "VA"

}

},

{

"@search.score": 0.032522473484277725,

"@search.rerankerScore": 2.111117362976074,

"HotelId": "48",

"HotelName": "Nordick's Motel",

"Description": "Only 90 miles (about 2 hours) from the nation's capital and nearby most everything the historic valley has to offer. Hiking? Wine Tasting? Exploring the caverns? It's all nearby and we have specially priced packages to help make our B&B your home base for fun while visiting the valley.",

"Category": "Boutique",

"Address": {

"City": "Washington D.C.",

"StateProvince": null

}

}

]

}

## Why choose hybrid search?

Hybrid search combines the strengths of vector search and keyword search. The advantage of vector search is finding information that's conceptually similar to your search query, even if there are no keyword matches in the inverted index. The advantage of keyword or full text search is **precision**, with the ability to apply semantic ranking that improves the quality of the initial results. Some scenarios - such as querying over product codes, highly specialized jargon, dates, and people's names - can perform better with keyword search because it can identify exact matches.

Benchmark testing on real-world and benchmark datasets indicates that hybrid retrieval with semantic ranking offers significant benefits in search relevance.

The following video explains how hybrid retrieval gives you optimal grounding data for generating useful AI responses.

[Outperform vector search with hybrid retrieval and ranking (Tech blog)](https://techcommunity.microsoft.com/t5/azure-ai-services-blog/azure-cognitive-search-outperforming-vector-search-with-hybrid/ba-p/3929167)

<https://youtu.be/Xwx1DJ0OqCk>

# **Algorithms used in hybrid search (RRF)**

Reciprocal Rank Fusion (RRF) is an algorithm that evaluates the search scores from multiple, previously ranked results to produce a unified result set. In Azure AI Search, **RRF** is **used whenever there are two or more queries** that **execute** in **parallel**. Each query produces a ranked result set, and RRF is used to **merge and homogenize** the **rankings** into a single result set, returned in the query response. Examples of scenarios where RRF is always used include [hybrid search](https://learn.microsoft.com/en-us/azure/search/hybrid-search-overview) and multiple vector queries executing concurrently.

RRF is based on the concept of **reciprocal rank**, which is the inverse of the rank of the first relevant document in a list of search results. The goal of the technique is to take into account the position of the items in the original rankings, and give higher importance to items that are ranked higher in multiple lists. This can help improve the overall quality and reliability of the final ranking, making it more useful for the task of fusing multiple ordered search results.

## How RRF ranking works

RRF works by taking the search results from multiple methods, assigning a reciprocal rank score to each document in the results, and then combining the scores to create a new ranking. The concept is that documents appearing in the top positions across multiple search methods are likely to be more relevant and should be ranked higher in the combined result.

Here's a simple explanation of the RRF process:

1. Obtain ranked search results from multiple queries executing in parallel.
2. Assign reciprocal rank scores for result in each of the ranked lists. RRF generates a new **@search.score** for each match in each result set. For each document in the search results, the engine assigns a reciprocal rank score based on its position in the list. The score is calculated as **1/(rank + k),** where **rank** is the position of the document in the list, and **k** is a constant, which was experimentally observed to perform best if it's set to a small value like 60. **Note that this k value is a constant in the RRF algorithm and entirely separate from the k that controls the number of nearest neighbors.**
3. Combine scores. For each document, the engine sums the reciprocal rank scores obtained from each search system, producing a combined score for each document.
4. The engine ranks documents based on combined scores and sorts them. The resulting list is the fused ranking.

Only fields marked as searchable in the index, or searchFields in the query, are used for scoring. Only fields marked as retrievable, or fields specified in select in the query, are returned in search results, along with their search score.

### Parallel query execution

RRF is used anytime there's more than one query execution. The following examples illustrate query patterns where parallel query execution occurs:

* A full text query, plus one vector query (simple hybrid scenario), equals two query executions.
* A full text query, plus one vector query targeting two vector fields, equals three query executions.
* A full text query, plus two vector queries targeting five vector fields, equals 11 query executions

## Scores in a hybrid search results

Whenever results are ranked, **@search.score** property contains the value used to order the results. Scores are generated by ranking algorithms that vary for each method. Each algorithm has its own range and magnitude.

The following chart identifies the scoring property returned on each match, algorithm, and range of scores for each relevance ranking algorithm.

| **Search method** | **Parameter** | **Scoring algorithm** | **Range** |
| --- | --- | --- | --- |
| full-text search | @search.score | BM25 algorithm | No upper limit. |
| vector search | @search.score | HNSW algorithm, using the similarity metric specified in the HNSW configuration. | 0.333 - 1.00 (Cosine), 0 to 1 for Euclidean and DotProduct. |
| hybrid search | @search.score | RRF algorithm | Upper limit is bounded by the number of queries being fused, with each query contributing a maximum of approximately 1 to the RRF score. For example, merging three queries would produce higher RRF scores than if only two search results are merged. |
| semantic ranking | @search.rerankerScore | Semantic ranking | 0.00 - 4.00 |

**Semantic ranking doesn't participate in RRF**. Its score (@search.rerankerScore) is always reported separately in the query response. Semantic ranking can rerank full text and hybrid search results, assuming those results include fields having semantically rich content.

## Number of ranked results in a hybrid query response

**By default**, if you aren't using pagination, the search engine returns the **top 50 highest** ranking matches for full text search, and the most similar k matches for vector search. In a hybrid query, top determines the number of results in the response. Based on defaults, the top 50 highest ranked matches of the unified result set are returned.

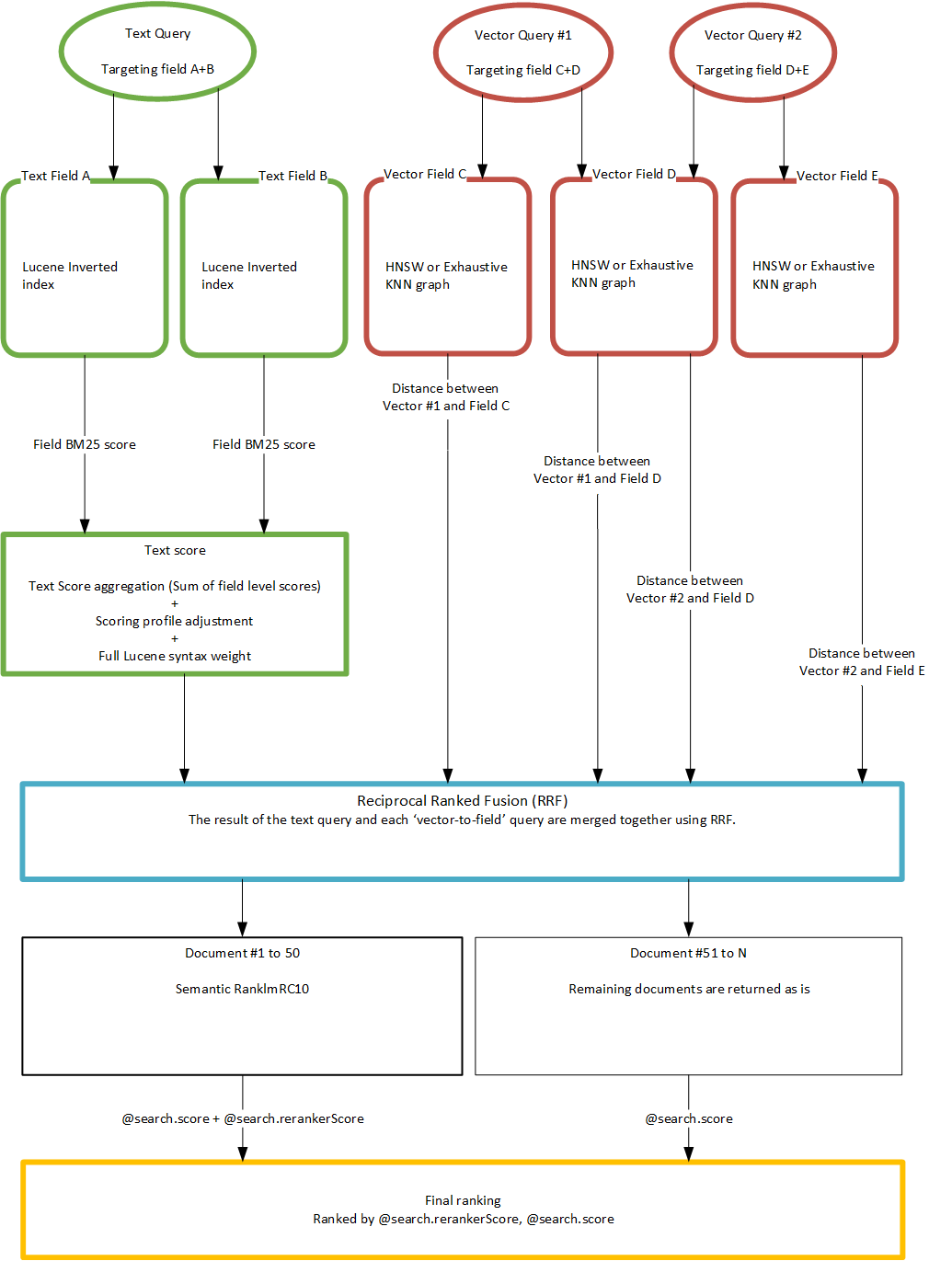
Often, the search engine finds more results than top and k. To return more results, use the paging parameters top, skip, and next. Paging is how you determine the number of results on each logical page and navigate through the full payload.

Full text search is subject to a maximum limit of 1,000 matches (see [API response limits](https://learn.microsoft.com/en-us/azure/search/search-limits-quotas-capacity#api-response-limits)). Once 1,000 matches are found, the search engine no longer looks for more.

For more information, see [How to work with search results](https://learn.microsoft.com/en-us/azure/search/search-pagination-page-layout).

## Diagram of a search scoring workflow

The following diagram illustrates a hybrid query that invokes keyword and vector search, with boosting through scoring profiles, and semantic ranking.

[](https://learn.microsoft.com/en-us/azure/search/media/hybrid-search/search-scoring-flow.png#lightbox)

A query that generates the previous workflow might look like this:

HTTPCopy

POST https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}/docs/search?api-version=2023-11-01

Content-Type: application/json

api-key: {{admin-api-key}}

{

"queryType":"semantic",

"search":"hello world",

"searchFields":"field\_a, field\_b",

"vectorQueries": [

{

"kind":"vector",

"vector": [1.0, 2.0, 3.0],

"fields": "field\_c, field\_d"

},

{

"kind":"vector",

"vector": [4.0, 5.0, 6.0],

"fields": "field\_d, field\_e"

}

],

"scoringProfile":"my\_scoring\_profile"

}