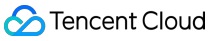
Elasticsearch Service

Vector Search Guide

Product Documentation

Copyright Notice

Trademark Notice



Service Statement

This document is intended to provide users with general information about Tencent Cloud's products and services only and does not form part of Tencent Cloud's terms and conditions. Tencent Cloud's products or services are subject to change. Specific products and services and the standards applicable to them are exclusively provided for in Tencent Cloud's applicable terms and conditions.

Contents

Vector Search Guide
Introduction to ES Vector Search

Last updated：2026-08-12 18:13:51

Introduction to Vector Search

1. What Is Vector Search

Vector search represents a paradigm shift in search technology. It converts unstructured data, such as text and images, into high-dimensional vectors using deep learning models. This process enables semantically similar content to be positioned close to each other in the vector space. When a user searches for "smartphone", the system matches not only literal keywords but also semantically relevant content such as "iPhone" and "Android phones".

2. Vector Search Scenarios

Vector search is not just a technological upgrade but also a cornerstone for building intelligent applications. Its primary scenarios include:

- **Semantic search:** It overcomes the limitations of traditional keyword-based search and searches for semantically similar content.
- **Accurate recommendation:** It recommends similar products or articles based on user preferences or the semantics of the currently viewed content.
- **Cross-modal search:** It enables cross-modal search experiences, such as text-to-image search, image-to-image search, and audio/video search.
- **Intelligent Q&A:** As the core of the Retrieval-Augmented Generation (RAG) architecture, it provides accurate and relevant knowledge sources for large language models.

With these capabilities, vector search in Tencent Cloud Elasticsearch Service (ES) delivers a set of core tools that transform data into deep semantic insights, helping you build more intelligent, user-intent-aware modern AI-powered search applications.

Introduction to ES Vector Search

As a leader in the search domain, ES has evolved from full-text search into an **AI-powered search engine that supports hybrid text-vector search**, and natively supports vector search technology **at the Lucene and ES kernel levels**.
ES's core advantages lie in its technology stack that simultaneously supports text search, vector search, aggregation analysis, and AI integration. This significantly distinguishes it from traditional vector databases. For details, see Comparison of Capabilities Between ES and Common Vector Databases.

1. Vector Search Algorithms

ES uses k-nearest neighbor (kNN) search to implement vector search. It searches for results based on semantics rather than exact keyword matching. The fundamental principle is to compare the query vector from the search request with vectors in the target vector collection. It evaluates the distance between them using similarity measurement methods such as cosine similarity or L2 norm, where a shorter distance indicates higher similarity. This process identifies the k vectors closest to the query vector. ES supports two kNN search methods:

Search Method	Scenario	Detailed Description
Exact brute-force kNN	Small datasets or scenarios requiring precise scoring, typically with a vector scale under 100,000.	This is a brute-force scanning method that compares the query vector against every vector in the target vector collection and identifies all eligible nearest neighbor vectors. Although it provides high accuracy, it is inefficient and consumes significant resources and time. It uses script_score to execute queries with vector functions.
Approximate nearest neighbor (ANN)	Most production workloads, supporting billions to tens of billions of vectors.	Commonly referred to as ANN. To address the limitations of brute-force kNN, the ANN algorithm requires the target vector collection to be pre-organized through indexing. The algorithm focuses on finding the top k similarity results with higher efficiency. The throughput, memory usage, and search accuracy of ANN search may vary depending on the selected index type. You need to strike a balance between search performance and accuracy.

2. Vector Indexing Algorithms

- ES primarily supports the Hierarchical Navigable Small World (HNSW) algorithm. This is an efficient graph indexing algorithm for high-dimensional vector similarity search, renowned for its excellent search speed and recall. HNSW combines the hierarchical concept of skip lists with the "shortcut" characteristic of small-world networks. It constructs a multi-layer graph structure. The top layer contains a small number of nodes with long-distance connections for fast navigation, while the bottom layer contains all nodes with short-distance connections for fine-grained search. This design achieves search complexity at an approximately logarithmic level. In the HNSW search process, a search starts from the top layer, greedily moves to the nearest neighbor node, and then descends to the next layer to continue the search after finding a local minimum value. This process repeats until the bottom layer is reached. This method effectively mitigates the local optimum trap common in traditional graph search.
- ES added support for the DiskBBQ algorithm in version 9.2. We will continue rolling out support for additional indexing algorithms in upcoming releases. Stay tuned.

3. Vector Similarity Algorithms

ES supports the following similarity algorithms, which you can specify during field configuration:

Algorithm	Description
l2_norm (Euclidean distance (L2 distance))	Calculates the absolute spatial distance, where a smaller value indicates greater similarity.
cosine (cosine similarity)	Focuses on vector direction and is suitable for semantic search.
dot_product (vector dot product)	Considers both direction and magnitude. Vectors should first be normalized to unit vectors (with a length of 1).
max_inner_product (maximum inner product)	Applies to unnormalized vectors where length carries meaningful information.

4. Vector Search Methods

ES supports both text and vector search and seamlessly supports flexible combinations of the two methods:

- **Vector search:** It uses a dedicated kNN query API to directly perform efficient approximate nearest neighbor search.
- **Hybrid search:** It enables you to easily apply business filter conditions (such as category and price range) before/after vector search, seamlessly integrating vector search with text search.
- **Intelligent fusion:** It enables you to intelligently rerank the results from vector search and full-text search (BM25) using algorithms such as Reciprocal Rank Fusion (RRF), balancing semantic relevance and keyword precision.

5. Seamless Integration with AI Applications

Tencent Cloud ES provides comprehensive capabilities to integrate with AI applications, enabling enterprises to efficiently implement semantic search, multimodal search, and RAG applications:

- **AI application end-to-end model inference capabilities**
  - **Out-of-the-box atomic services:** It provides online services such as document parsing, chunking, embedding, reranking, and large language model (LLM).
  - **Self-deployment using machine learning nodes:** You can add machine learning nodes to your ES cluster, which are dedicated to model inference. You can deploy built-in models on these nodes, or upload custom models to them.
  - **GPU-accelerated inference performance:** Through in-house development, Tencent Cloud ES is the first in the world to support GPU-based inference, including NVIDIA GPUs and domestic GPUs such as Zixiao.
- **One-stop RAG construction experience:** The Tencent Cloud ES console provides a one-stop RAG construction playground, which includes:
  - **Document upload:** It enables you to upload documents online, which are automatically parsed, chunked, embedded, and stored.

- **Online Q&A:** It offers visualized online Q&A experience. You can flexibly optimize Q&A effectiveness by adjusting configurations. It supports testing and comparison of multiple RAG configurations.
- **Effectiveness assessment:** It can generate a comprehensive assessment report for RAG, facilitating problem identification and targeted optimization.
- **Application deployment:** It supports code export and the online deployment of RAG services (coming soon).

6. Advantages of Tencent Cloud ES Based on Self-Developed Optimizations

Tencent Cloud has implemented **in-depth self-developed optimizations** on the open-source ES, covering aspects such as text search, vector search, hybrid search, fusion ranking, inference performance, and cost optimization. This makes Tencent Cloud ES a reliable, enterprise-grade technology foundation for the AI-powered search era.

- **Text search:** Traditional vector databases only support simple scalar filtering, while ES supports capabilities such as full-text search, keyword matching, phrase query, fuzzy match, and aggregation analysis. It can also effectively improve recall through nested fields, function_score, and other methods. In the field of text search, Tencent Cloud ES has accumulated long-term self-developed technologies, such as query pruning, adaptive routing, and core operator optimization. These technologies improve write performance by 100% and query performance by 200% to 1000%.
- **Vector search:** It supports HNSW graph indexing and features self-developed Multi-Path query parallelization. Compared to the traditional HNSW algorithm, it achieves up to four times the acceleration while maintaining the same recall.
- **Hybrid search:** It supports intelligent path analysis based on the cost-based optimizer (CBO), improving hybrid search query performance by 30% to 80%. The optimized built-in RRF reciprocal ranking algorithm brings a 10%—20% increase in recall.
- **Inference service:** Building upon productized machine learning nodes, it is compatible with GPUs such as NVIDIA and Zixiao, and supports the deployment of models such as Embedding, Rerank, NER, and Text Classification.
- **Model capabilities:** It provides the WeChat-developed image-text search model WeClip as well as text embedding models such as KaLM, Conan, and BGE series. Users can also leverage their own enterprise models through inference APIs.

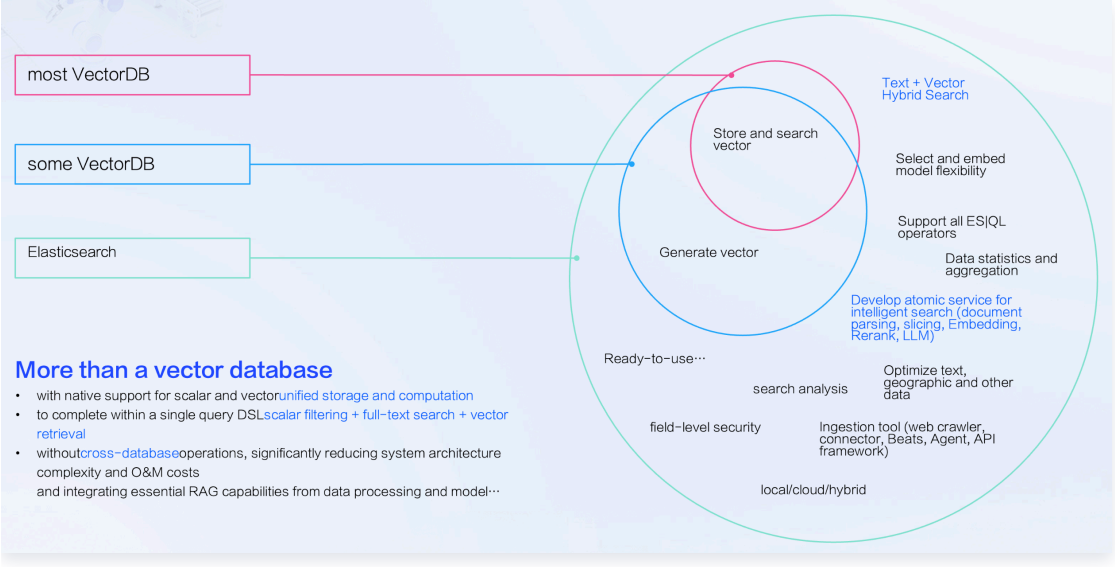
Comparison of ES and Common Vector Database Capabilities

Last updated：2026-08-12 18:13:52

As a leader in the search domain, ES has evolved from full-text search into an **AI-powered search engine that supports hybrid text-vector search**, and natively supports vector search technology **at the Lucene and ES kernel levels**.

ES's core advantages lie in its technology stack that simultaneously supports text search, vector search, aggregation analysis, and AI integration. This avoids the issues introduced by hybrid technology stacks, such as system complexity, high costs, difficult debugging and tracing, and high Ops investment, and also reduces the reliability assurance and security challenges associated with multi-system collaboration. Tencent Cloud has performed **in-depth self-developed optimization** on the open-source ES, covering aspects like hybrid search, fusion ranking, inference performance, cost optimization, and multimodal capabilities, making Tencent Cloud Elasticsearch Service (ES) a reliable, enterprise-grade technology foundation for the AI-powered search era.

In contrast, **the advantages of traditional vector databases mostly focus on storage and search in pure vector scenarios. They lack full-text search, aggregation analysis, and out-of-the-box AI integration capabilities**. Although some vector databases support text search, they mostly implement it through sparse vector methods, lacking text position information and unable to achieve basic search capabilities such as phrase queries. For example, when "climate change" is queried, the text search implemented via sparse vectors assigns the same score to both "Climate change is a global issue" and "The climate in this place changes every year".



Comparison of ES and Traditional Vector Database Capabilities

1. Basic Capabilities

Comparison Dimension	Tencent Cloud ES	Milvus	Other Vector Databases
Positioning	Preferred choice for text and vector hybrid search	More suitable for pure vector search scenarios	More suitable for pure vector search scenarios
Vector engine implementation	Implemented at kernel level	Implemented at kernel level	Implemented at kernel level or via plugins
Capability scope	Search: • Vector search • Text search (full-featured) • Geographic search Analysis: aggregation analysis AI integration: • Built-in model inference • One-stop Retrieval-Augmented Generation (RAG) building	Search: • Vector search • Text search (limited) • Geographic search Analysis: x (not supported) AI integration: x (dependent on third-party services)	Search: • Vector search • None or simple filtering • Geographic search: x (not supported) Analysis: x (not supported) AI integration: x (dependent on third-party services)
Query language and API	Single DSL syntax for all queries	APIs/SDKs, with less flexibility for complex queries than ES	-
Permission control	Level: index/document/field	Level: table level only	Level: generally table level only
Learning and Ops	Relatively gentle learning curve	Requires understanding of its unique architecture, with a steeper learning curve	-
Open-source ecosystem	Active and mature, offering end-to-end solutions covering data ingestion, management,	Focuses on the core vector engine, requiring users to integrate tools	-

	security, and observability	themselves to implement a solution	

2. Vector Search Capabilities

Comparison Dimension	Tencent Cloud ES	Milvus	Other Vector Databases
Performance	Billions to hundreds of billions of vectors, millisecond-level average response	Billions to hundreds of billions of vectors, millisecond-level average response	Billions to hundreds of billions of vectors, millisecond-level average response
Vector – index type	FLAT, HNSW, DiskBBQ (9.2), and so on	FLAT, HNSW, IVF, DiskANN, and so on	Generally supports FLAT, HNSW, IVF, and so on
Vector – quantization (memory saving)	Supports scalar quantization and binary quantization	Supports scalar quantization and binary quantization	Partially supports quantization
Vector – distance algorithm	L2_NORM, COSINE, DOT_PRODUCT, and MAX_INNER_PRODUCT	L2, IP, COSINE, JACCARD, and HAMMING	Supports mainstream distance algorithms
Vector – multiple vectors per table	Yes	Yes	Mostly yes
Vector – multiple vectors per field	Yes	Yes	Generally not supported

3. Text Search Capabilities

Comparison Dimension	Tencent Cloud ES	Milvus	Other Vector Databases
Chinese word segmentation	Yes	Yes	Supported by some
English word segmentation	Yes	Yes	Supported by some

Other multilingual word segmentation	Yes (extensive)	Limited	Mostly not supported
Multiple types per field	Yes	No	Mostly not supported
Synonym	Yes	No	Mostly not supported
Stopword	Yes	Yes	Generally supported
Custom plugin	Yes	No	Mostly not supported
Custom dictionary	Yes	No	Mostly not supported
Spelling error support	Yes	No	Mostly not supported
Highlight support	Yes	Yes	Mostly not supported
Custom relevance scoring	Yes, function_score	No	Mostly not supported
Full-text search	Yes (full-featured inverted index with BM25 scoring)	Yes (implemented through sparse vectors without positional information)	Mostly not supported
Phrase search	Yes	Yes (filtering only/no scoring)	Mostly not supported
NGram search	Yes	Yes	Mostly not supported
Custom script sorting	Yes (script_score combined with user profiles, click behaviors, and so on)	No	Mostly not supported

4. Hybrid Search Capabilities

Comparison Dimension	Tencent Cloud ES	Milvus	Other Vector Databases
Pre-filtering	Yes	Yes	Supported by some
Post-filtering	Yes	No	Supported by some
Multi-way merging	Yes	Yes (complex code)	Supported by some
Custom weighting	Yes	Yes	Mostly not supported
RRF fusion ranking	Yes	Yes	Mostly not supported
Rerank semantic ranking	Yes	Yes	Mostly not supported
Pre-filtering algorithm optimization	Yes	Yes	Supported by some
Geographic search	Yes	Yes	Mostly not supported

5. Aggregation Analysis

ES supports multi-level analysis capabilities, which are one of its core advantages. Basic metric aggregations can quickly calculate statistics such as sums, averages, and distinct counts, providing a macro business overview. The more powerful bucket aggregation mechanism can intelligently group data by time intervals, numerical ranges, geographic locations, or specific terms, acting as a multi-dimensional classification framework for your data. Pipeline aggregations further process aggregation results and support advanced analyses like moving averages and derivative calculations, enabling higher-level capabilities such as trend prediction.

Comparison Dimension	Tencent Cloud ES	Milvus	Other Vector Databases
Aggregation analysis	Yes	No	Mostly not supported

6. AI Integration Capabilities

Comparison Dimension	Tencent Cloud ES	Milvus	Other Vector Databases
Atomic service	Yes (self-developed) (parsing, chunking, vectorization, reranking, LLM, and so on)	Dependent on third-party inference services	Mostly not supported
Custom model inference	Yes Machine learning nodes (vectorization, reranking, and so on)	No	Mostly not supported
GPU-based inference	Yes (self-developed) NVIDIA GPUs or domestic GPUs such as Zixiao (cost-effective)	Dependent on third-party inference services (higher costs)	Mostly not supported
One-stop RAG building experience	Yes (self-developed)	No	Mostly not supported

ES Vector Cluster Configuration Assessment

Last updated：2026-08-12 18:13:52

You can refer to Cluster Specifications and Capacity Configuration Assessment > Vector Search Scenarios to learn how to properly plan vector cluster configurations based on your business characteristics.

Index Creation

Last updated：2026-08-12 18:13:52

This document describes the key points of vector index creation. For more details, see the elastic documentation.
Vector indexes serve as the foundation for building semantic search, recommendation systems, and Retrieval-Augmented Generation (RAG) applications. Proper configuration is crucial for search performance and quality. Based on the following example, this document introduces in detail how to create a vector index in ES and provides an in-depth explanation of the meaning and purpose of each configuration parameter.

Index Mapping Details

The following is an example of creating a vector index:

```
PUT /book-index
{
  "settings": {
    "index": {
      "number_of_shards": 3,
      "number_of_replicas": 1,
      "refresh_interval": "1s"
    }
  },
  "mappings": {
    "properties": {
     "id": {
        "type": "keyword"
      },
      "category": {
        "type": "keyword"
      },
     "price": {
        "type": "integer"
      },
      "title_text": {
        "type": "text"
      },
      "title_vector": {
        "type": "dense_vector", // Dense vector
```

```
        "dims": 768, // Vector dimensions
        "similarity": "cosine", // Vector similarity algorithm
        "index_options":{
            "type": "hnsw",
            "m": 16, // Maximum number of connections per node in the
HNSW graph
            "ef_construction": 100  // Number of candidate neighbors
examined for each new node during HNSW construction, which affects the
quality and speed of index creation
        }
      },
      "content_text": {
        "type": "text"
      },
      "content_vector": {
        "type": "dense_vector",
        "dims": 768,
        "similarity": "cosine",
        "index_options":{
            "type": "hnsw",
            "m": 16, // Maximum number of connections per node in the
HNSW graph
            "ef_construction": 100  // Number of candidate neighbors
examined for each new node during HNSW construction, which affects the
quality and speed of index creation
        }
      }
    }
  }
}
```

The following describes the main parameters. For more detailed information, see dense_vector.
Parameter description for vector fields:

Parameter	Required or Not	Description
type	Yes	dense_vector indicates a dense vector type.
dims	Yes	Number of vector dimensions. The maximum value is 4096. It must strictly match the dimensions of the vectors being

		written. Higher vector dimensions deliver richer information and higher search precision, while incurring greater storage and computational costs. You can start by testing recall/accuracy with 384 or 768 dimensions. If the results do not meet requirements, increase the dimensions; if they meet requirements, try reducing dimensions.
similarity	Yes	Similarity calculation method, which determines the vector distance measurement standard.

index_options (optional):

Parameter	Required or Not	Description
type	No	Specifies the index algorithm. The default value varies across ES versions: hnsw for 8.13, int8_hnsw for 8.16, and bbq_hnsw for 9.1.3. You can set the algorithm based on the vector scale by referring to the following information: • . hnsw: It is recommended for scenarios where the number of vectors is less than 100 million. • . int8_hnsw: It is recommended for scenarios where the number of vectors is between 100 million and 2 billion. • . bbq_hnsw: It is recommended for scenarios where the number of vectors is between 1 billion and 10 billion. • . bbq_disk: It is recommended for scenarios where the number of vectors is between 1 billion and 100 billion. (diskbbq is supported in ES 9.2 and later versions, and the cloud edition is expected to be released in Q1.)
m	No	Maximum number of connections per node in the HNSW graph, with a default value of 16. A larger m value improves recall and query speed but increases indexing time and memory usage. For most scenarios, start with 16. If extremely high recall is required and longer indexing time and larger index size are acceptable, you can try increasing it to 32 or higher.
ef_construction	No	Number of candidate neighbors examined when connections for each new node are found during HNSW index creation, with a default value of 100. Increasing this value improves index quality and recall but prolongs index creation time. You can start with 100. If the data distribution is complex or precision requirements are high, you can set it to 200 or higher.

The following four algorithms are supported for similarity calculation:

Algorithm	Description
l2_norm (Euclidean distance (L2 distance))	Calculates the absolute spatial distance, where a smaller value indicates greater similarity.
cosine (cosine similarity)	Focuses on vector direction and is suitable for semantic search.
dot_product (vector dot product)	Considers both direction and magnitude. Vectors should first be normalized to unit vectors (with a length of 1).
max_inner_product (maximum inner product)	Applies to unnormalized vectors where length carries meaningful information.

Note:
For image search, cosine or l2_norm is recommended. For text semantic search, cosine is recommended. For details, see kNN search in Elasticsearch.

Inference Service Creation
Inference Service Overview

Last updated：2026-08-12 18:13:52

The ES Inference Service is a core AI capability introduced in Elasticsearch 8 and later versions. It allows you to directly deploy models for inference within an ES cluster or call pre-deployed online inference services through a unified inference API, providing native support for semantic search, vector search, and Retrieval-Augmented Generation (RAG) applications.

Key Strengths

- Flexible options: It enables you to deploy your own models through machine learning nodes, directly call out-of-the-box atomic services, or directly call third-party inference services.
- Native integration: It seamlessly integrates with ES ingest pipelines, vector fields, and search APIs.

Preparations

Environment Requirements

- Elasticsearch version: 8.16 or later. It is recommended to use the highest stable version whenever possible.
- Machine learning node: If you deploy models using machine learning nodes, ensure that at least one machine learning node (dedicated to model inference) is configured in the cluster.

Model Preparation

Select an appropriate model type based on your business requirements. For example:

- Text embedding model: used to generate text vectors (such as bge-m3 and .multilingual-e5-small)
- Reranking model: used for fine ranking of search results.
- Large language model: used for text generation and Q&A.

> **Note:**
> When selecting a model, you should focus on its parameters. For example, different embedding models determine the vector dimensions, supported languages, and other aspects.

Inferencing with Machine Learning Nodes

Last updated：2026-08-12 18:13:52

By running inference services on machine learning nodes, you can gain high-performance, low-latency model inference capabilities and benefit from full integration with the Elasticsearch ecosystem.

Key Strengths

- GPU acceleration: Tencent Cloud Elasticsearch Service (ES) has independently developed support for NVIDIA and domestic GPUs (for example, support for GPU acceleration), improving inference efficiency by 30 times.
- Ecosystem integration: It seamlessly integrates with the Elasticsearch ecosystem for data collection and vector search.
- Model support: ES machine learning nodes currently support various models, including text embedding, reranking, named entity recognition, and classification. For details, refer to the related document.

Steps

1. **Creating a Machine Learning Node**

   If you have not created an ES cluster, you can enable Machine Learning Node when creating a cluster. If you have already created an ES cluster, you can enable Machine Learning Node through configuration adjustment.

   Note that you need to reserve sufficient memory space for different models. For example:

   - The .multilingual-e5-small model requires at least 8 GB of memory.
   - The bge-base-zh-v1.5 model requires at least 16 GB of memory.
   - The bge-m3 model requires at least 32 GB of memory.

   Selecting a GPU delivers higher inference efficiency and lower inference costs.
2. **Deploying a Model Inference Service**

   Go to the cluster details page and switch to Model Management:

   Click **Add Model** to select one from the preset models:

   You can also upload a custom model:

   The status after installation is as follows:

   After the model is installed, click **Deploy** to deploy the model:

   Switch to Deployment List to view the model deployment status:
3. **Testing the Inference Service**

   You can test the service in Kibana Dev Tools as follows:

```
POST _ml/trained_models/bge-base-zh/_infer
{
```

```
  "docs": [{ "text_field": "Hello" }]
}
```

4. **Creating an Inference Endpoint (Optional)**

   If you call the model service through an ingest pipeline within ES, you do not need to create an inference endpoint. If you want to call the inference service (for example, semantic search) through the inference API, you can refer to the following method to create an inference endpoint.

```
PUT _inference/text_embedding/bge-base-zh-service
{
  "service": "elasticsearch",
  "service_settings": {
    "model_id": "bge-base-zh",
    "num_allocations": 1,
    "num_threads": 1
  }
}
```

   After creating an inference endpoint, you can call _inference to test it:

```
POST _inference/text_embedding/bge-base-zh-service/_infer
{
  "input": "Hello"
}
```

Monitoring and Ops of Machine Learning Nodes

You can view the metrics of machine learning nodes through node monitoring and perform machine learning node scaling promptly based on the load.

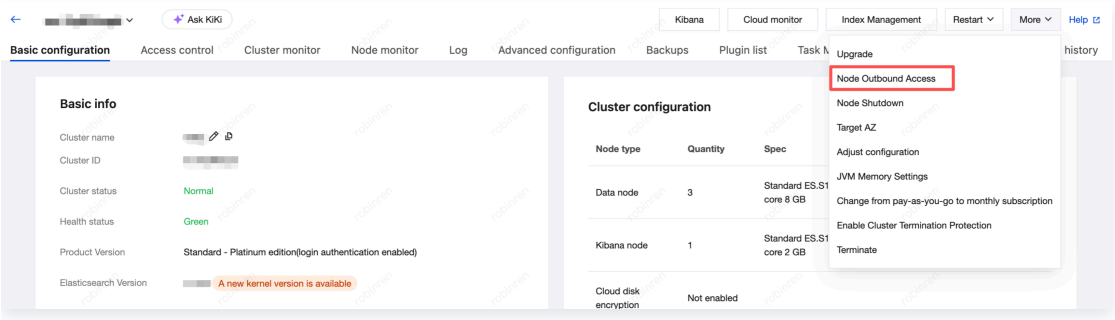
Calling Third-Party Inference Services

Last updated：2026-08-12 18:13:52

ES can integrate third-party inference services through its inference capabilities, enabling flexible and efficient calls of various required capabilities. For more details, refer to the related document.

Step 1: Enabling Node Outbound Access

To enable ES to access third-party services on the public network, you need to enable Node Outbound Access first.



Step 2: Creating an Inference Endpoint

The following is a reference template for creating a custom inference:

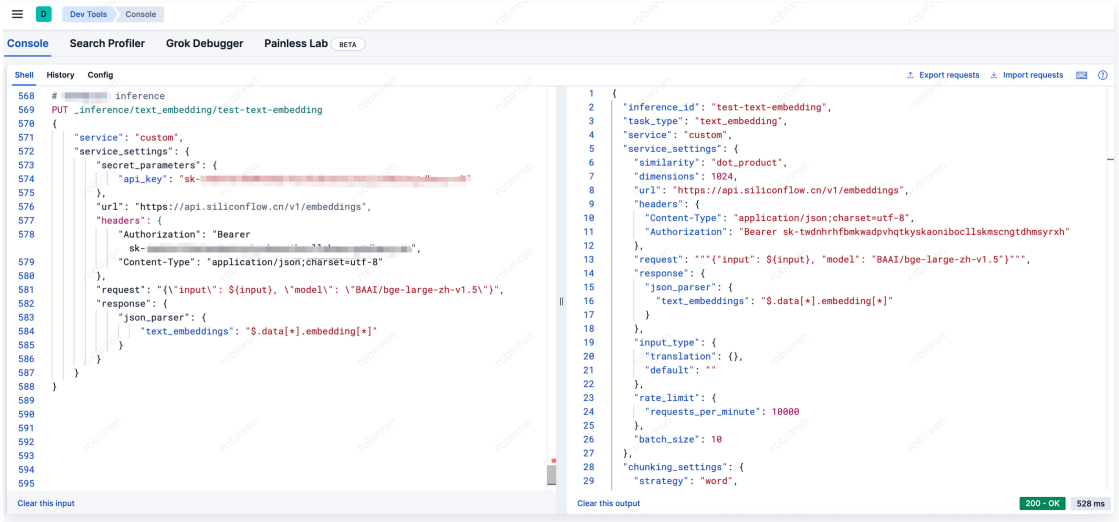
```
PUT _inference/text_embedding/test-text-embedding
{
  "service": "custom",
  "service_settings": {
     "secret_parameters": {
          "api_key": "<some api key>"
     },
     "url": "...endpoints.huggingface.cloud/v1/embeddings",
     "headers": {
         "Authorization": "Bearer ${api_key}",
         "Content-Type": "application/json"
     },
     "request": "{\"input\": ${input}}",
     "response": {
         "json_parser": {
             "text_embeddings":"$.data[*].embedding[*]"
         }
     }
```

```
    }
}
```

For example, the following is an example of using the SiliconFlow embedding service. You can refer to the template and fill in your API key and model name:

```
PUT _inference/text_embedding/test-text-embedding
{
    "service": "custom",
    "service_settings": {
        "secret_parameters": {
            "api_key": "sk-your_actual_API_key"
        },
        "url": "https://api.siliconflow.cn/v1/embeddings",
        "headers": {
            "Authorization": "Bearer sk-your_actual_API_key",
            "Content-Type": "application/json;charset=utf-8"
        },
        "request": "{\"input\": ${input}, \"model\": \"BAAI/bge-large-
zh-v1.5\"}",
        "response": {
            "json_parser": {
                "text_embeddings": "$.data[*].embedding[*]"
            }
        }
    }
}
```

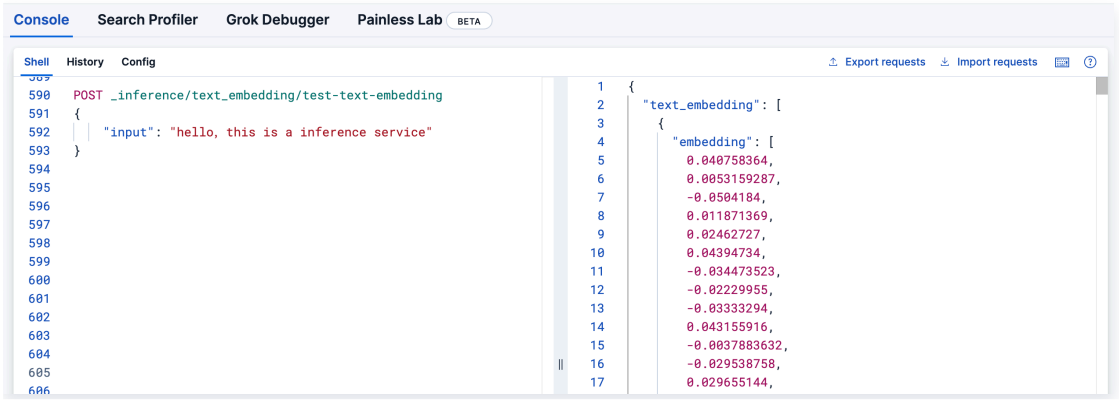
After execution, it indicates that the creation is successful if the following information is returned:



Step 3: Verifying the Inference Service

You can verify the inference service using the following statement. If successful, the embedding result is returned.

```
POST _inference/text_embedding/test-text-embedding
{
    "input": "hello, this is an inference service"
}
```



Data Writing

Last updated：2026-08-12 18:13:52

This document describes the key points of vector data writing. For more details, refer to the related document.

ES provides multiple flexible methods for writing vector data into the system, whether you have precomputed vectors or need to generate vectors in real time during the write process. This document describes the different writing methods in detail.

Method 1: Directly Writing Precomputed Vectors

If you already have precomputed vectors (for example, generated by an external Python script or a model service), you can directly write these vectors.

```
//Write a single document.
PUT /book-index/_doc/1
{
  "id": "1001",
  "category": "Fiction",
  "price":10,
  "title_text": "One Hundred Years of Solitude",
  "title_vector": [0.1, 0.2, 0.3 ...],
  "content_text": "The rise and fall of the town of Macondo and the
legendary stories of the Buendía family",
  "content_vector": [0.5, 0.6, 0.7, ... ]
}

//Perform bulk write.
POST /_bulk
{ "index": { "_index": "book-index", "_id": "2" } }
{ "id": "1002", "category": "History", "price": 20, "title_text":
"Sapiens: A Brief History of Humankind", "title_vector": [0.1, 0.2, 0.3
...], "content_text": "The developmental history of humanity from the
Cognitive Revolution to the Scientific Revolution", "content_vector":
[0.1, 0.2, 0.3, ...]}
{ "index": { "_index": "book-index", "_id": "3" } }
{ "id": "1003", "category": "Science Fiction", "price": 30,
"title_text": "The Three-Body Problem", "title_vector": [0.8, 0.7, 0.6
```

```
...], "content_text": "Human civilization and the laws of cosmic
sociology", "content_vector": [0.1, 0.2, 0.3, ...]}
```

Method 2: Automatically Generating Vectors Through Ingest Pipeline

1. **Creating an Ingest Pipeline**

   Based on the text embedding model inference service created earlier, you can create an ingest pipeline to perform embedding processing during the write process:

```
PUT /_ingest/pipeline/text-embedding
{
  "description": "Text embedding pipeline",
  "processors": [
    {
      "inference": {
        "model_id": "bge-base-zh",
        "ignore_missing": true,    // Ignore missing input fields.
        "input_output": [
          {
            "input_field": "title_text",
            "output_field": "title_vector"
          },
          {
            "input_field": "content_text",
            "output_field": "content_vector"
          }
        ]
      }
    }
  ]
}
```

> **Note:**
> For the model_id value in the ingest pipeline, if the model is deployed through a machine learning node, directly use the value of the model_id parameter from that node; if the model is deployed through an _inference endpoint, use the name of the _inference endpoint. If both are set up, either can be used. They both correspond to inference from the same underlying model. The

> difference is that the former accesses the model for inference directly, while the latter performs inference through the _inference endpoint.

2. **Writing Data**

When writing data, you only need to write scalar data. The model inference service in the ingest pipeline will automatically generate vector data.

```
//Write a single document.
PUT /book-index/_doc/1?pipeline=text-embedding
{
  "id": "1001",
  "category": "Fiction",
  "price":10,
  "title_text": "One Hundred Years of Solitude",
  "content_text": "The rise and fall of the town of Macondo and the
legendary stories of the Buendía family"
}

//Perform bulk write.
POST /_bulk?pipeline=text-embedding
{ "index": { "_index": "book-index", "_id": "2" } }
{ "id": "1002", "category": "History", "price": 20, "title_text":
"Sapiens: A Brief History of Humankind", "content_text": "The
developmental history of humanity from the Cognitive Revolution to the
Scientific Revolution"}
{ "index": { "_index": "book-index", "_id": "3" } }
{ "id": "1003", "category": "Science Fiction", "price": 30,
"title_text": "The Three-Body Problem", "content_text": "Human
civilization and the laws of cosmic sociology"}
```

Check the write results through GET /book-index/_search. The vectors have been automatically generated.

Vector Search
Vector Search (Single-Vector and Multi-Vector)

Last updated: 2026-08-12 18:13:52

This document introduces the key points of k-nearest neighbor (kNN) search. For more details, refer to the related document.

Single-Vector Search

This is the most basic vector search method, which performs a similarity search on a single vector field.

```
GET /book-index/_search
{
   "knn": {
      "field": "title_vector",
      "query_vector": [0.1, 0.2, 0.3 ...],
      "k": 10,  // Number of top k results to return
      "num_candidates": 100  // Number of candidate vectors to search on
each shard
  }
  "fields": ["id", "title_text", "category"],
  "_source": false,
}
```

Parameter description:

- field: name of the vector field to search
- query_vector: query vector. Its dimension must match the field definition.
- k: number of the most similar documents to return
- num_candidates: number of candidate vectors considered on each shard. A larger value improves accuracy but reduces performance.

You can also directly input the query text to generate the query vector through online inference (same below):

```
GET /book-index/_search
```

```
{
  "knn": {
    "field": "title_vector",
    "query_vector_builder": {
      "text_embedding": {
          "model_id": "model-bge-base",
          "model_text": "Sapiens: A Brief History of Humankind"
      }
    },
    "k": 10,  // Number of top k results to return
    "num_candidates": 100  // Number of candidate vectors to search on
each shard
  },
  "fields": ["id", "title_text", "category"],
  "_source": false
}
```

Multi-Vector Search

ES supports searching multiple vector fields simultaneously and merging the results.

```
GET /book-index/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "knn": {
            "field": "title_vector",
            "query_vector": [0.1, 0.2, 0.3 ...],
            "k": 5,
            "num_candidates": 50,
            "boost": 0.8
          }
        },
        {
          "knn": {
            "field": "content_vector",
            "query_vector": [0.1, 0.2, 0.3, ...],
```

```
            "k": 5,
            "num_candidates": 50,
            "boost": 0.2
          }
        }
      ]
    }
  }
}
```

Features:

- Two search methods are executed in parallel.
- The results are merged and sorted by score.
- You can use the boost parameter to adjust the weight of each field.
- When the same document appears in different search results, the highest score is used.

Hybrid Search (Pre-Filtering/Post-Filtering/Parallel Search)

Last updated: 2026-08-12 18:13:52

This document introduces the key points of k-nearest neighbor (kNN) search. For more details, refer to the related document.
Hybrid search combines the advantages of vector search and traditional text search. You can flexibly select suitable combinations based on business scenarios to deliver more precise search results that meet specific recall requirements.

Pre-Filtering

Search method: Filters are applied prior to vector search. Vector similarity calculation is only performed on documents that meet the conditions.
Scenarios: Suitable for scenarios with clear filtering conditions, such as book category filtering and e-commerce product category filtering.

```
GET book-index/_search
{
    "knn":{
        "field":"title_vector",
        "query_vector": [0.1, 0.2, 0.3 ...],
        "k":10,
        "num_candidates": 100,
        "filter":{  // Pre-filtering conditions
            "term":{
                "category": "History"
            }
        }
    }
}
```

Post-Filtering

Search method: Vector search is executed first, followed by filtering on the results.
Scenarios: Suitable for scenarios that require a broad search first, followed by filtering to narrow down the results.

Note: Post-filtering may filter out a large number of results, leading to an insufficient number of returned documents. Therefore, it is recommended to set a larger k value to ensure that enough results are returned. In the following sample code, the execution effect is the same whether the filter clause is placed before or after the must clause. In both cases, post-filtering is applied.

```
GET book-index/_search
{
    "query":{
        "bool":{
            "must":[
                {
                    "knn":{
                        "field":"content_vector",
                        "query_vector": [0.1, 0.2, 0.3 ...],
                        "k":100,
                        "num_candidates": 500
                    }
                }
            ],
            "filter":[ // Post-filtering. The execution effect is the
same whether the filter clause is placed before or after the must
clause.
                {
                    "range": {
                      "price": {
                        "gte": 15,
                        "lte": 25
                      }
                    }
                }
            ]
        }
    },
    "size": 10
}
```

Parallel Search

Search method: Vector search and text search are executed in parallel. Their results are merged by score, and the weights can be adjusted using boost.
Scenarios: Suitable for scenarios where both semantic similarity and keyword matching are equally important.

```
GET book-index/_search
{
  "query":{
    "bool":{
      "should":[
        {
          "range": {
            "price": {
              "gte": 15,
              "lte": 25
            }
          }
        },
        {
          "knn":{
            "field":"title_vector",
            "query_vector": [0.1, 0.2, 0.3 ...],
            "k":2,
            "num_candidates": 50
          }
        }
      ]
    }
  }
}
```

Hybrid Search (RRF Fusion Ranking)

Last updated：2026-08-12 18:13:52

During hybrid search, the scoring mechanisms of text search and vector search differ significantly. The basic idea of Reciprocal Rank Fusion (RRF) is to assign a weight to the ranking result of each system, where the weight is the reciprocal of its rank. Specifically, for each item in the ranked list of each system, the RRF algorithm calculates a score, which is the sum of the reciprocals of that item's rank across all lists. Then, all items are reranked based on this score to generate the final fused ranking list. RRF is built into ES, requiring no external model services. As a lightweight algorithm, it can be used as one of the fusion ranking algorithms for the coarse ranking phase.

Syntax 1: Retriever Method (Officially Available Since Elastic Has Introduced Version 8.16)

For details about the retriever method, see Retrievers.

> **Note:**
> Tencent Cloud Elasticsearch Service (ES) has independently developed support for the RRF algorithm since version 8.16. In terms of syntax, it replaces the following rrf with rank_fusion. Elastic has introduced RRF since version 9.

```
GET /book-index/_search
{
  "retriever": {
    "rrf": { // For Tencent Cloud ES 8.16, use rank_fusion here; for
later versions, use rrf.
      "retrievers": [
        {
          "retriever": {
            "knn": {
              "field": "title_vector",
              "query_vector": [0.1, 0.2, 0.3 ...],
              "k": 10,
              "num_candidates": 100
            }
          },
          "weight": 0.8
        },
```

```
        {
          "retriever": {
            "standard": {
              "query": {
                "match": {
                  "title_text": "Human"
                }
              }
            }
          },
          "weight": 0.2
        }
      ],
      "rank_window_size": 50,
      "rank_constant": 20
    }
  }
}
```

Syntax 2: Traditional Method (Deprecated, Not Recommended)

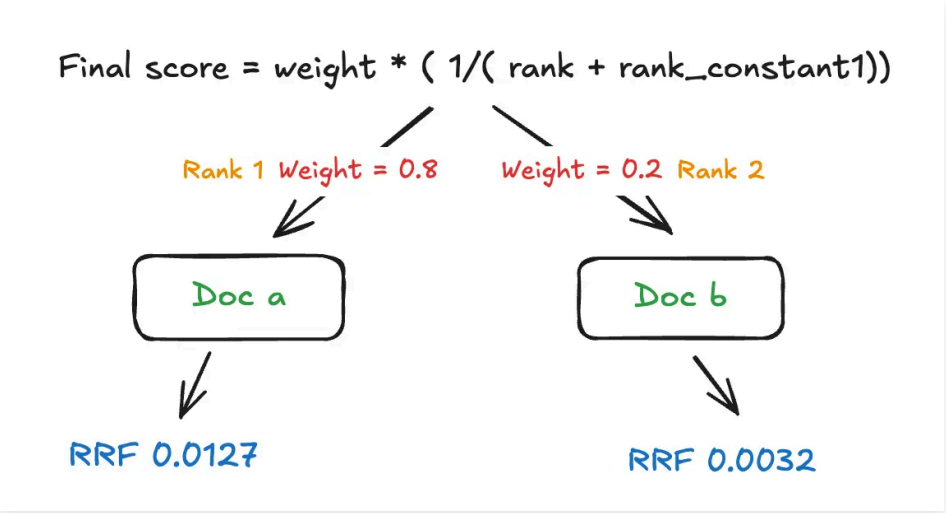
```
GET /book-index/_search
{
    "knn":{
        "field":"content_vector",
        "query_vector": [0.5, 0.6, 0.7 ...],
        "k":10,
        "num_candidates": 100
    },
    "query":{
        "match":{
            "title_text":{
                "query": "Human"
            }
        }
    },
    "rank":{
        "rrf":{
            "rank_window_size":50,
```

```
            "rank_constant":20
        }
    },
    "size": 10
}
```

Parameter description:

- rank_window_size: number of results taken from each query to participate in fusion. The default value is 100.
- rank_constant: constant that controls ranking. A smaller value has a greater impact on the ranking. The default value is 60.

The calculation formula for RRF is as follows:



Vector Pagination Search

Last updated：2026-08-12 18:13:52

This document introduces basic implementation methods for pagination search. For more details on pagination search, you can refer to the related document.

Pagination Search (Real-Time)

search_after is an efficient, real-time deep pagination solution provided by ES. Unlike traditional from+size pagination, whose performance drops sharply when processing large amounts of data, and unlike the Scroll API primarily used for offline exporting, search_after uses the sort values of the last result from the previous page as a "cursor" to fetch the next page, enabling high-performance deep pagination.

Principle:

- On the first query, you should specify one or more unique or high-cardinality fields for sorting (such as _id).
- From the return results, capture the sort values of the last document.
- On the next query, pass these sort values as the search_after parameter, and ES will start searching directly after that "cursor."

Strengths:

- High performance: Avoids re-sorting and re-traversing the first N results as in from-based pagination. Query time remains essentially constant regardless of page depth.
- Real-time: Queries are always based on the latest index data, without the need to maintain a data snapshot context as in scroll.
- Resource-friendly: No long-term resource usage on the server side. Resources are released after the query completes.
- Suitable for deep pagination: Standard method for searching data beyond tens of thousands of pages.

In addition, the pagination results in the following example may be unstable. To ensure consistency and stability throughout the pagination process, search_after should be used together with the Point-in-Time (PIT) API. PIT creates a snapshot of a data view, keeping the index status unchanged during pagination and preventing disordered or missing results caused by data changes. For details, you can refer to the related document.

```
//Search_after
GET /book-index/_search
{
    "knn": {
      "field": "title_vector",
      "query_vector": [0.1, 0.2, 0.3 ...],
```

```
      "k": 2,  // Number of results to return
      "num_candidates": 2  // Limit the number of candidate results
returned by the search node.
    },
    "sort": [
        {"id": "asc"},
        {"price": "asc"}
    ]
}

GET /book-index/_search
{
    "knn": {
      "field": "title_vector",
      "query_vector": [0.1, 0.2, 0.3],
      "k": 2,
      "num_candidates": 2
    },
    "search_after": ["1002","20"], // Pass in the sort values from the
previous step.
    "sort": [
        {"id": "asc"},
        {"price": "desc"}
    ]
}
```

Pagination Search (Offline)

The Scroll API is a deep pagination mechanism designed by ES for searching large volumes of data in a single pass (such as full exporting and batch offline processing). Its core concept is to create a data snapshot and then fetch data in batches using a scrollable "cursor" (scroll_id) until all results are traversed.

Core features and scenarios:

- Non-real-time: Scroll creates a snapshot of the index on the first query, and all subsequent pages are based on this snapshot. Data changes during this period do not affect the scroll results.
- Sequential traversal: Primarily used for sequential traversal from start to end. Jumping to arbitrary pages is not supported.
- Resource usage: The search context needs to be maintained in the cluster until the timeout, consuming memory and compute resources.

- Typical scenarios: Data migration, full backups, offline analysis, and tasks that need to process all or a large number of matching documents in an index.

The following is a complete example of the scroll pagination process.

Step 1: Initializing the Scroll Query

```
//Search Scroll
POST book-index/_search?scroll=1m
{
  "size": 20, // Number of documents per page
  "knn": {
      "field": "title_vector",
      "query_vector": [0.1, 0.2, 0.3],
      "k": 100,  // The value should cover the total number of documents
you want to scroll through.
      "num_candidates": 500
  }
}
```

scroll=5m: Sets the search context's time to live to 5 minutes. This time should be sufficient for you to process one batch of data and fetch the next.
size: Specifies the number of documents returned per scroll (the batch size).
Response: In addition to returning the first batch of hits, the response body also contains a _scroll_id, which is key for subsequent pagination.

Step 2: Using the scroll_id to Fetch Subsequent Batches

Use the _scroll_id returned from the previous step to fetch the next batch of results.

```
// Pass in the scroll_id obtained from the previous step.
POST /_search/scroll
{
  "scroll" : "1m",
  "scroll_id" : "XXXXXXXXXX"  // Replace scroll_id with the value
returned from the previous step.
}
```

- Each call returns the next batch of size documents and a new _scroll_id (use the latest one even if it appears identical).
- Repeat this step until the returned hits array is empty.

Step 3: Clearing the Scroll Context (Very Important)

After processing is complete, you should proactively release resources.

```
DELETE /_search/scroll
{
  "scroll_id": "XXXXXXXXXX"  // Replace scroll_id with the value
returned from the previous step.
}
```

Vector Quantization

Last updated：2026-08-12 18:13:52

This document introduces the key points of vector quantization search. You can refer to the related document for more details.

1. What Is Quantization

When ES vector search scales to **hundreds of millions or even billions of vectors**, two challenges need to be addressed: high memory usage and slow computation speed. ES's vector quantization is an important optimization policy at this stage. It effectively combines the speed advantage of quantization with the precision advantage of raw vectors, striking a balance between performance and accuracy. **Quantization** is essentially a lossy compression technique. The core idea is to convert high-precision vector data (typically 32-bit floating point, float) into a low-precision representation (such as 8-bit integer, int8). It can significantly reduce memory usage and improve query speed with minimal recall loss.

2. Changes in Memory and Storage Usage with Quantization

If you specify quantization in the index mapping settings, ES generates both the original precision data storage (assuming float32) and an additional quantized precision data storage for the float32 data during index creation. Therefore, for float32 vectors, int8, int4, and bbq quantization can **reduce memory usage** by 4x, 8x, and 32x respectively, but will also reduce vector precision and **increase disk space usage** (by 25%, 12.5%, and 3.125%, respectively). For example, when int8 quantization is used on 40 GB of float vectors, the quantized vectors will consume an additional 10 GB of disk space, bringing the total disk usage to 50 GB, but the memory usage will be reduced to 10 GB.

3. Coarse Ranking and Oversampling with Quantization

To address the precision loss caused by quantization, ES vector search is divided into two phases: approximate coarse ranking and exact fine ranking:

- **Coarse ranking phase:** ES uses the quantized query vector to search on the Hierarchical Navigable Small World (HNSW) graph index built on quantized data. After completing the search on each shard, it obtains a list of num_candidates document IDs with approximate scores.
- **Fine ranking phase:** By **configuring rescore_vector to specify an oversample factor**, ES reads the raw float32 vectors from disk for the top k*oversample documents out of the num_candidates documents returned by each shard. It then performs high-precision similarity computation between the original query vector and these original document vectors, selecting the top k results.

Clearly, the oversampling mechanism combines the performance and memory advantages of approximate search using quantized vectors with the accuracy of rescoring the best candidates using raw vectors. In practice, int8 typically does not require oversampling; int4 can achieve higher precision and recall with 1.5x to 2x oversampling; bbq typically requires oversampling, and 3x to 5x oversampling is usually sufficient.

4. Recall Assessment with Quantization

Based on our practical experience, **if the recall without quantization is 99%, int8 quantization yields a recall of about 96—97%, and bbq quantization yields about 90—94%**. You should be aware that the recall with quantization has a high correlation with data scale. The larger the dataset, the smaller the recall loss from quantization, and vice versa. Therefore, you should validate the final recall results on as large a dataset as possible.

5. Example

```
// Mapping settings (vectors using int8 quantization)
PUT /my-quantized-index
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text"
      },
      "title_vector": {
        "type": "dense_vector",
        "dims": 768,
        "index": true,
        "similarity": "cosine",
        "index_options": {
          "type": "int8_hnsw", // Enable the 8-bit scalar quantization
HNSW index.
          "m": 16,
          "ef_construction": 100
        }
      }
    }
  }
}


// Vector search (oversampling)
GET my-quantized-index/_search
{
  "knn": {
    "field": "title_vector",
```

```
    "query_vector": [0.15, 0.50, ..., 0.05], // Query vector
    "k": 10,                  // Number of top k results to return
    "num_candidates": 100,    // The value should be greater than or
equal to k.
    "rescore_vector": {
      "oversample": 2.0 // Oversampling factor
    }
  }
}
```

Special Note on Simplifying Writes and Searches with the Semantic Field

Last updated：2026-08-12 18:13:52

This document introduces the key points of semantic search. You can refer to the related document for more details.
The semantic field uses natural language directly for writes and queries, providing a simpler experience compared to k-nearest neighbor (kNN) queries. You only need to specify an inference service created based on a machine learning node or an inference endpoint to easily complete reads and writes, without creating an ingest pipeline or worrying about underlying model and inference details (such as dimensions, m, and ef).

Creating an Inference Endpoint

Create the model service deployed on a machine learning node as an inference endpoint (see here for more details on creating inference endpoints).

```
PUT _inference/text_embedding/bge-base-zh-service
{
  "service": "elasticsearch",
  "service_settings": {
    "model_id": "bge-base-zh",
    "num_allocations": 1,
    "num_threads": 1
  }
}
```

Creating an Index

The semantic field uses the built-in .elser-2-elastic inference endpoint (EIS service) from Elastic by default, which is a sparse vector embedding model. You can explicitly specify your own created inference endpoint:

```
PUT semantic_search_index
{
    "mappings":{
        "properties": {
            "content_vector":{
                "type":"semantic_text",
```

```
                "inference_id":"bge-base-zh-service",  // Specify the
inference endpoint for writes and queries.
            },
            "content":{
                "type":"text",
                "copy_to":"content_vector"
            }
        }
    }
}
```

Writing Data

During writes, the model is automatically called to generate vector data:

```
PUT semantic_search_index/_doc/1
{
    "content":"Using semantic search simplifies data ingestion
statements."
}

PUT semantic_search_index/_doc/2
{
    "content":"Using semantic search simplifies data query statements."
}
```

Performing Semantic Search

Perform a match search with the query in natural language, as shown in the example below:

```
GET semantic_search_index/_search
{
  "query": {
    "match": {
      "content_vector": {
        "query": "What is semantic search ?"
      }
    }
  }
```

```
}
```

Special Note on Flexibly Supporting a Variable Number of Vectors in a Single Field

Last updated：2026-08-12 18:13:52

This document introduces the key points of nested search. You can refer to the related document for more details.
ES allows a single field to support a vector array through nested, with a variable number of vectors. This feature is very useful. Let us look at a video search example.

Example Background

Suppose you need to perform vector search on videos. The challenge is that each video has a different number of extracted frames. You can solve this using nested fields.
Example: A video record has id, title, content, and image fields, where image stores the vector data of frames extracted from the video. Each video has n images, and n is not fixed.

Creating an Index

Specify image as a nested field when the mappings are created:

```
// id, title, and content are text fields.
// num in image is the image number, such as the frame number
(optional).
// emb in image is the image embedding data; image = [image1_emb,
image2_emb, image3_emb, ..., imagen_emb].

PUT /image_embeddings
{
  "mappings": {
    "properties": {
      "id": {
        "type": "keyword"
        },
      "title": {
        "type": "text"
        },
      "content": {
```

```
        "type": "text"
        },
      "image": {
        "type": "nested",
        "properties": {
          "num": {"type": "keyword"},
          "emb": {
              "type": "dense_vector",
              "dims": 5,
              "index_options": {
                "type": "int8_hnsw"
              },
              "similarity": "cosine"
          }
        }
      }
    }
  }
}
```

Writing Data

Wrap multiple vector groups in an array format, as shown below:

```
POST /image_embeddings/_doc/1
{
  "id": "book_001",
  "title": "Sunny day",
  "content": "On this windy day, I tried to hold your hand",
  "image": [
    {"num": "0", "emb": [0.1,0.2,0.3,0.4,0.5]},
    {"num": "1", "emb": [0.6,0.7,0.8,0.9,1.0]},
    {"num": "2", "emb": [0.2,0.3,0.4,0.5,0.6]}
  ]
}

POST /image_embeddings/_doc/2
{
  "id": "book_002",
  "title": "Heading north",
```

```
  "content": "Heading north, I tried to hold your hand",
  "image": [
    {"num": "0", "emb": [0.1,0.2,0.3,0.4,0.5]},
    {"num": "1", "emb": [0.6,0.7,0.8,0.9,1.0]}
  ]
}

POST /image_embeddings/_doc/3
{
  "id": "book_003",
  "title": " Nunchaku",
  "content": "Heng Heng Ha Hei",
  "image": [
    {"num": "0", "emb": [0.1,0.2,0.3,0.4,0.5]},
    {"num": "1", "emb": [0.6,0.7,0.8,0.9,1.0]},
    {"num": "2", "emb": [0.1,0.2,0.3,0.4,0.5]},
    {"num": "3", "emb": [0.6,0.7,0.8,0.9,1.0]},
    {"num": "4", "emb": [0.1,0.2,0.3,0.4,0.5]},
    {"num": "5", "emb": [0.6,0.7,0.8,0.9,1.0]}
  ]
}
```

Performing Vector Search

Query data. The syntax is the same as the hybrid search described earlier. The score of a nested field uses max as the final score.

```
GET book-index/_search
{
  "retriever": {
    "rrf": {
      "retrievers": [
        {
          "retriever": {
            "knn": {
              "field": "image.emb",
              "query_vector": [0.1, 0.2, 0.3,0.4,0.5],
              "k": 5,
              "num_candidates": 50
            }
```

```
          },
          "weight": 0.8
        },
        {
          "retriever": {
            "standard": {
              "query": {
                "match": {
                  "title": "Sunny day"
                }
              }
            }
          },
          "weight": 0.2
        }
      ],
      "rank_window_size": 50,
      "rank_constant": 20
    }
  }
}
```

Special Note on Using script_score Flexibly and with Caution for Brute-Force Vector Search

Last updated: 2026-08-12 18:13:53

ES function_score allows you to modify the scores of documents searched by a query. This is especially useful when the scoring function is computationally expensive, and you only need to compute scores for a filtered set of documents. script_score is a type of function_score that supports using scripts to provide custom scoring for returned documents.

Using script_score with User Profiles and Click Behavior Data for Scoring

ES's script_score query is a powerful advanced feature popular among ES developers in text search scenarios. It allows you to use scripts to customize document scoring, combining text relevance with business metrics for complex sorting logic. For example, you can combine user profiles and click behavior data to better optimize sorting results for more accurate search and recommendations, bringing great flexibility and practicality to your business implementations.

```
GET /_search
{
  "query": {
    "function_score": {
      "query": {
        "match": { "message": "elasticsearch" }
      },
      "script_score": {
        "script": {
          "source": "Math.log(2 + doc['my-int'].value)"
        }
      }
    }
  }
}
```

Using script_score for Brute-Force Vector Search with Caution

However, when script_score is used for vector search (see here), it performs **brute-force scanning** (even if you set the index type to hnsw in the index mapping), rather than the approximate k-nearest neighbor (kNN)

search defined in your index settings. The script reads the stored title_vector field value for every document within the range and calls built-in vector functions (such as cosineSimilarity) to compute against the passed-in query vector parameter. Vector search performance will drop dramatically. Therefore, unless for small datasets (< 100,000) or scenarios requiring exact results, we generally do not recommend using script_score for vector search.

```
GET my-index/_search
{
  "query": {
    "script_score": {
      "query" : {
        "bool" : {
          "filter" : {
            "term" : {
              "status" : "published"
            }
          }
        }
      },
      "script": {
        // Call the built-in function to compute cosine similarity.
        // title_vector is the vector in the document.
        // params.query_vector is the query vector passed in from an
external source.
        // + 1.0 ensures the score is positive (cosine similarity ranges
from -1 to 1).
        "source": "cosineSimilarity(params.query_vector, 'title_vector')
+ 1.0",
        "params": {
          "query_vector": [4, 3.4, -0.2]
        }
      }
    }
  }
}
```

Elasticsearch Vector Search Performance Tuning

Last updated: 2026-08-12 18:13:53

In the era of rapid AI development, ES is widely used for vector search, multimodal search, and knowledge base building. Its unique hybrid text-vector search capability, which balances recall and search accuracy, is favored by a broad range of developers. This document focuses on tuning techniques drawn from practical experience to help you better leverage the performance advantages of ES vector search.

Configuration Planning

1. Reasonable Cluster Configuration Assessment

To ensure the performance of ES vector search, a reasonable cluster configuration is essential. **The key is to ensure sufficient memory**. If memory is inadequate and searches frequently hit the disk, performance can degrade by more than 10 times. For cluster configuration estimation for vector search, see ES Vector Cluster Configuration Assessment.

2. Reasonable Index Planning

You need to evaluate your business data volume and future growth in advance, and plan your indexing and sharding accordingly. If the data volume is large, avoid putting everything in a single index, which would cause every search to scan the entire index. It is recommended to **split indexes** by category, such as department or product type. Both oversized shards and an excessive number of shards can affect read and write performance. The following are practical recommendations for shard settings:

- A single shard is recommended to be **20 GB–50 GB** in size. You can use this to initially determine the number of shards for your index.
- **The number of shards should be as close as possible to the number of data nodes**. If you have a large number of shards, it is recommended that you set the number of shards to **an integer multiple of the number of data nodes** to facilitate even distribution of shards across the data nodes.
- The total number of shards for all indexes on a single node should **not exceed 1,000**. The total number of shards in the cluster should be kept **below 30,000**.

Increasing replicas can improve query throughput (QPS), as searches can run in parallel on both primary and replica shards.

- Recommendation: In scenarios with low write pressure and high query concurrency, increase the number of replicas appropriately.

```
// Index settings
PUT /my_index
```

```
{
  "settings": {
    "index": {
      "number_of_shards": 10, // Shard settings. Reasonably estimate the
number of shards based on future growth.
      "number_of_replicas": 1 // Replica settings. Set it based on high
availability and concurrency requirements.
    }
  },
  "mappings": {
    "properties": {
     ...
  }
}
```

3. Test Set Preparation and Iterative Tuning

Vector search tuning is an iterative process of trial and optimization. You can prepare a high-quality test set containing query terms, known relevant documents (positive samples), and irrelevant documents (negative samples), define target requirements for recall (the proportion of positive samples retrieved out of all positive samples) and precision (the proportion of positive samples in the query results), and use a "hypothesis—verification—iteration" approach to gradually find the optimal configuration for your business.

Vector Index Mapping Settings

The following is an example of typical vector index mapping settings:

```
// Mapping settings
PUT /my-index
{
  "mappings": {
    "properties": {
      "category": {
        "type": "keyword"
      },
      "title": {
        "type": "text"
      },
      "title_vector": {
        "type": "dense_vector",
```

```
        "dims": 768, // Vector dimensions
        "similarity": "cosine",  // Vector similarity algorithm
        "index_options": {
          "type": "hnsw",// Index algorithm
          "m": 16,  // Maximum number of connections per node in the
Hierarchical Navigable Small World (HNSW) graph
          "ef_construction": 100  // Number of candidate neighbors
examined for each new node during HNSW construction, which affects the
quality and speed of index creation
        }
      }
    }
  }
}
```

- **dims – Vector dimensions:** Higher dimensions contain richer information and yield higher search accuracy, but also incur higher storage and computation costs. You can start by testing recall/precision at 384 or 768 dimensions. If the results are unsatisfactory, increase the dimensions; if satisfactory, try reducing them.
- **index setting:** The default value is true. If set to false, k-nearest neighbor (kNN) search will not be performed, and only brute-force scanning (script_score) can be performed.
- **type:** Default values vary by version (hnsw in 8.13, int8_hnsw in 8.16, and bbq_hnsw in 9.1.3). You can configure this parameter by referring to the following information based on your vector scale:
  - hnsw: It is recommended for scenarios where the number of vectors is less than 100 million.
  - int8_hnsw: It is recommended for scenarios where the number of vectors is between 100 million and 2 billion.
  - bbq_hnsw: It is recommended for scenarios where the number of vectors is between 1 billion and 10 billion.
  - bbq_disk: It is recommended for scenarios where the number of vectors is between 1 billion and 100 billion. (diskbbq is supported in ES 9.2 and later, and the cloud edition is expected to be released in Q1.)
- **m:** The default value is 16, which is the maximum number of connections per node in the HNSW graph. A larger m improves recall and query speed but increases indexing time and memory usage. Start with 16 for most scenarios. If you require extremely high recall and can accept longer indexing time and larger index size, you can try increasing it to 32 or higher.
- **efConstruction:** The default value is 100. It represents the number of candidate neighbors examined when connections are established for each new node during HNSW index creation. Increasing this value

improves index quality and recall but extends the index creation time. You can start with 100. If your data distribution is complex or you require high precision, you can set it to 200 or higher.

Bulk Vector Writing

1. Rationale

For vector search scenarios, bulk writing is strongly recommended. Bulk writing is a core optimization technique for high-throughput ES scenarios. By reducing network interactions, optimizing disk I/O, and lowering indexing overhead, it can significantly improve data writing efficiency and ensure cluster stability.

2. Bulk Write Settings

When using bulk writes, you can start testing with a batch size of about 500–1,000 documents, monitor cluster CPU and memory load, and gradually increase the batch size until performance no longer improves or requests start being rejected.

```
// Bulk write
POST /_bulk
{ "index" : { "_index" : "my-index", "_id" : "1" } }
{ "category":"cat", "title": "Lazy cat", "title_vector": [0.1, 0.2, ...
] }
{ "index" : { "_index" : "my-index", "_id" : "2" } }
{ "category":"dog", "title": "Walking dog", "title_vector": [0.3, 0.4,
... ] }
//... More data ...
```

3. Other Adjustable Parameters (Optional)

- **Adjust the refresh interval**

  refresh_interval defaults to 1s, making newly written data visible for search. However, frequent refreshing generates a large number of small segments, which affects vector index creation and query performance. Therefore, it is recommended to set refresh_interval to a larger value (such as 30s or 60s) or –1 during bulk data import, and restore it after the import is complete. Note: When refresh_interval is –1, newly written data will not be searchable until a manual refresh is performed or the next automatic refresh occurs after restoration.
- **Temporarily disable replicas**

  When a large volume of data is written, you can also set number_of_replicas to 0 first, and restore the number of replicas after the write is complete.

```
// Index settings
```

```
PUT /my_index
{
  "settings": {
    "index": {
      "number_of_shards": 10, // Shard settings. Reasonably estimate the
number of shards based on future growth.
      "number_of_replicas": 0, // Temporarily disable replicas during
initial bulk writing, and then restore afterward.
      "refresh_interval": -1 // Disable automatic refresh to improve
write performance.
    }
  },
  "mappings": {
    "properties": {
     ...
  }
}
```

Vector Search Parameter Settings

The following is a simple vector search query statement:

```
// Vector search - Recall only a small number of non-text fields.
GET /my-index/_search
{
  "size" : 3,
  "knn": {
      "field": "title_vector",
      "query_vector": [-5, 9, -12, ...],
      "k": 10, // Number of top k results to return
      "num_candidates": 100  // Number of candidate vectors to search on
each shard
  }
  "fields": ["category"],
  "_source": false
}

// Vector search - When recalling a large number of fields or text
fields
GET /my-index/_search
```

```
{
  "size" : 3,
  "knn": {
      "field": "title_vector",
      "query_vector": [-5, 9, -12, ...],
      "k": 10, // Number of top k results to return
      "num_candidates": 100  // Number of candidate vectors to search on
each shard
  }
  "_source": {
    "excludes": ["title_vector"] // Exclude vector fields.
    // "excludes": ["*_vector"] // Use wildcards, such as excluding all
fields ending with "vector".
    // "_source": ["doc_id", "title"] // Explicitly list the business
fields to return.
    // "_source": false // Do not return raw document data. After
obtaining the document _id, query another database such as MySQL for
details.
  }
}
```

- **k**

  Number of top results to recall at query time. Set it as needed.
- **num_candidates**

  Specifies the number of candidate vectors to search on each shard. This value should be greater than or equal to k. Increasing num_candidates improves recall (especially with quantization or filters), but increases query latency. It is generally recommended to set it to 5 to 10 times the value of k to balance recall and performance.
- **Avoid returning vector fields**

  Vector fields are typically large. If you do not need to return vector data, refer to the following recommendations:
  - If you need to recall only a small number of non-text fields, such as IDs (such as keyword, numeric, or other field types with doc_values enabled) or explicitly stored fields (store: true), you can use "fields" to explicitly specify the fields to recall. ES will query doc values or stored values directly, avoiding the overhead of parsing _source for better performance.
  - If you need to recall a large number of fields or text fields, you can use_source: false, _source: exclude, or _source: ["field1", "field2"] to exclude vector fields, thereby reducing network transfer and serialization overhead.

Vector Quantization and Oversampling

When ES vector search scales to **hundreds of millions or even billions of vectors**, the challenges of high memory usage and slow computation speed need to be addressed. Striking a balance between cost, performance, and accuracy requires an important optimization policy at this stage: vector quantization. For a detailed introduction to vector quantization and oversampling policies, see Vector Quantization.
The following is a code example of vector quantization and oversampling:

```
// Mapping settings (vectors using int8 quantization)
PUT /my-quantized-index
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
      },
      "title_vector": {
        "type": "dense_vector",
        "dims": 768,
        "index": true,
        "similarity": "cosine",
        "index_options": {
          "type": "int8_hnsw", // Enable the 8-bit scalar quantization
HNSW index.
          "m": 16,
          "ef_construction": 100
        }
      }
    }
  }
}

// Vector search (oversampling)
GET my-quantized-index/_search
{
  "knn": {
    "field": "title_vector",
    "query_vector": [0.15, 0.50, ..., 0.05], // Query vector
    "k": 10,                  // Number of top k results to return
```

```
    "num_candidates": 100,    // The value should be greater than or
equal to k.
    "rescore_vector": {
      "oversample": 2.0 // Oversampling factor
    }
  }
}
```

File System Cache Warm-Up

1. Background

ES relies on the operating system's page cache to accelerate reading of on-disk index files. By default, index files are loaded into the cache only when accessed. This can cause a problem: when the host operating system restarts, the page cache is cleared, and search performance may degrade significantly while the cache is being "warmed up" again, because the system needs to frequently read data from disk.

2. Solution

By setting index.store.preload, you can proactively preload specified key data files into memory when the index is opened, thereby avoiding costly disk I/O during subsequent searches. This is especially useful for frequently searched "hot" indexes.

> **Note:**
>
> 1. If the file system cache capacity is insufficient to hold all data, preloading data into the page cache too early on too many indexes or files can slow down searches. Use with caution.
> 2. If you use a quantized index, only preload the relevant quantized values and index structures, such as the HNSW graph. Preloading raw vectors (vec) is unnecessary and may be counterproductive, as preloading raw vectors can cause the operating system to evict important index structures from the cache.

3. File Types Related to Vector Search

For which index files index.store.preload actually loads, refer to the file extension descriptions below:

File Extension	File Purpose
vex	Stores HNSW graph structure files.
vec	All non-quantized vector values. Includes all element types: float, byte, and bit.

veq	Quantized vectors for quantized indexes: int4 or int8.
veb	Binary vectors for quantized indexes: bbq.
vem, vemf, vemq, vemb	Metadata, usually very small and does not require preloading.

4. Preloading Settings

index.store.preload is a static setting, meaning it can only be specified at index creation time or in the configuration file, and cannot be dynamically modified after index creation. For non-quantized scenarios, we generally recommend simply setting it to ve*. For quantized scenarios, you can specify which file types to preload individually (for example, by doing so, .vex and .vec are excluded from preloading).
We generally recommend also setting mmapfs, which maps index files directly into the process's virtual memory address space. Thereafter, ES can read and write files as if accessing ordinary memory (actual data loading is handled by the operating system via page fault interrupts). Using mmapfs together with preload achieves efficient queries and a smooth experience without cold starts, but increases node restart time.

```
// Configure preloading at index creation time.
PUT /my-preloaded-index
{
  "settings": {
    "index.store.preload": ["ve*"], // Preload
    //"index.store.preload": ["vex", "vec"], // Preload, or specify it
granularly.
    "index.store.type": "mmapfs" // Map index files to the process's
virtual memory address space.
  },
  "mappings": {
    ...
  }
}
```

If you want this to apply to all new indexes created on the cluster, you can set it in the global configuration file config/elasticsearch.yml:

```
index.store.preload: ["ve*"] // Preload specific data files into the
page cache.
index.store.type: mmapfs // Map index files to the process's virtual
memory address space.
```

For existing indexes, you should close them before modifying this setting:

```
// 1. Close the index.
POST /my-existing-index/_close

// 2. Update index settings.
PUT /my-existing-index/_settings
{
  "index.store.preload": ["ve*"] // Preload specific data files into the
page cache.
  "index.store.type": "mmapfs" // Map index files to the process's
virtual memory address space.
}

// 3. Reopen the index. The preload operation is triggered during
opening, which will be slower.
POST /my-existing-index/_open
```

Reduction of the Number of Index Segments

ES shards are composed of segments, which are internal storage elements within an index. For approximate kNN search, ES stores the vector values of each segment as a separate HNSW graph, so kNN search should examine each segment. Although kNN search parallelization significantly speeds up searches across multiple segments, kNN search can still be several times faster when the number of segments is small. By default, ES periodically merges smaller segments into larger ones through background merging. If this is insufficient, you can take the following explicit steps to reduce the number of index segments.

1. Increasing the Maximum Segment Size

ES provides many adjustable parameters to control the merge process. An important one is index.merge.policy.max_merged_segment, which controls the maximum size of segments created during merging. Increasing this value can reduce the number of segments in the index. This value defaults to 5 GB, which may be too small for high-dimensional vectors. It is recommended to increase it to 10 GB or 20 GB to help reduce the number of segments. This is a static setting, so for existing indexes, you need to close the index before configuring it:

```
// 1. Close the index.
POST /my-index/_close

// 2. Update index settings.
```

```
PUT /my-index/_settings
{
    "merge.policy.floor_segment": "300mb",    // Minimum segment size
    "merge.policy.max_merged_segment": "10g" // Maximum segment size
}

// 3. Reopen the index.
POST my-index/_open
```

2. Creating Large Segments During Bulk Indexing

A common practice is to perform the initial bulk writing and index creation first. In addition to relying on periodic background segment merging, you can also adjust index settings to encourage ES to create larger initial segments:

- Ensure that no searches are performed during the bulk upload, and set index.refresh_interval to –1. This prevents refresh operations and avoids generating additional segments.
- Allocate a large index buffer to ES so it can receive more documents before flushing. You can set indices.memory.index_buffer_size to 10% of the heap size, which is usually sufficient for larger heap sizes such as 32 GB. To allow the full index buffer to be used, you should also set index.translog.flush_threshold_size to be smaller than indices.memory.index_buffer_size.

3. Forcing a Segment Merge (Force Merge)

As mentioned earlier, vector indexes are created on each segment. The more segments there are, the more graphs need to be searched at query time, resulting in worse performance. It is recommended to perform a force merge after bulk writing data, or on indexes where data is no longer frequently updated. For a read–only index, merge to one segment: POST /my–index/_forcemerge?max_num_segments=1. Note: Force merge is a resource–intensive operation and should be performed during off–peak hours.

Pre–Filter Optimization for Improved Hybrid Search Performance

The following is a typical pre–filter search:

```
// Pre-Filter search
GET  /my-index/_search
{
    "knn":{
        "field":"title_vector",
        "query_vector": [-5, 9, -12, ...],
        "k": 10,
        "num_candidates": 100,
```

```
        "filter":{
            "term":{
                "category":"cat"
            }
        }
    }
}
```

In open-source ES, pre-filtering performs scalar filtering first, obtaining all qualifying DocIDs via the inverted index chain. It then performs a kNN query, checking each neighboring vector against the DocID set generated by scalar filtering until the top N results are obtained. In this case, if the scalar filtering result set is very large, for example, a query for gender "male" could yield hundreds of millions of results, query performance will degrade significantly.
To address the performance issue of pre-filtering in hybrid search, **Tencent Cloud ES has developed proprietary pre-filter optimization,** which can be enabled via optimize_prefilter_enable. This is a dynamic parameter and does not require reopening the index.

```
// Pre-Filter optimization
PUT /my-index/_settings
{
    "index.query.knn.optimize_prefilter_enable": "true"    // Enable pre-
filter optimization to improve pre-filter performance.
}
```

GPU-Powered Model Inference

Vector generation (embedding) requires substantial computation. The community edition of ES supports machine learning nodes dedicated to vector model inference, but does not yet support GPU inference.
The good news is that Tencent Cloud ES has supported GPU inference on machine learning nodes since version 8.16, which can dramatically accelerate vector inference speed. **Inference performance is 30 times higher than that of CPU**, and **the price-performance ratio is over 10 times better than that of CPU**.
Therefore, for large-scale vector generation, it is recommended to use Tencent Cloud ES machine learning nodes for GPU inference. This is done by enabling machine learning nodes when purchasing an ES cluster and selecting a GPU instance type. For existing clusters, you can enable machine learning nodes by adjusting the configuration.

Special Note on Using script_score with Caution for Vector Search

ES's script_score query is a powerful advanced feature popular among ES developers in text search scenarios. It allows you to use scripts to customize document scoring, combining text relevance with business metrics for complex sorting logic. However, when script_score is used for vector search, it performs **brute-force search** (even if the index type is set to hnsw in the index mapping). Unless working with small datasets (< 100,000) or scenarios requiring exact results, we generally do not recommend using script_score for vector search. For details, see Special Note on Using script_score Flexibly and with Caution for Brute-Force Vector Search.

```
// Example of brute-force vector search using script_score
GET my-index/_search
{
  "query": {
    "script_score": {
      "query" : {
        "bool" : {
          "filter" : {
            "term" : {
              "status" : "published"
            }
          }
        }
      },
      "script": {
        // Call the built-in function to compute cosine similarity.
        // title_vector is the vector in the document.
        // params.query_vector is the query vector passed in from an
external source.
        // + 1.0 ensures the score is positive (cosine similarity ranges
from -1 to 1).
        "source": "cosineSimilarity(params.query_vector, 'title_vector')
+ 1.0",
        "params": {
          "query_vector": [4, 3.4, -0.2]
        }
      }
    }
  }
}
```