TopK Operation

The TopK operation retrieves the most relevant items from your dataset using three different retrieval methods: semantic similarity, full-text search, or LLM-based comparison. It provides a clean, specialized interface for retrieval tasks where you need to find and rank the best matching documents based on specific criteria.

Overview

TopK is designed for retrieval and ranking use cases such as finding relevant documents for a query, filtering large datasets to the most important items, implementing retrieval-augmented generation (RAG) pipelines, and building recommendation systems. Unlike general sampling operations, TopK focuses specifically on retrieving the best matches according to your chosen method and criteria.

The operation supports three distinct retrieval methods, each optimized for different use cases. The **embedding method** uses semantic similarity to find conceptually related documents, making it ideal when meaning matters more than exact words. The **full-text search (FTS) method** employs the BM25 algorithm for keyword-based retrieval, perfect for when specific terms are important. The **LLM compare method** leverages a language model to rank documents based on complex criteria that require reasoning and multi-factor comparisons.

Configuration

Core Parameters

Parameter	Type	Description	Required
`method`	`"embedding"` | `"fts"` | `"llm_compare"`	Retrieval method to use	Yes
`k`	`int` or `float`	Number of items to retrieve (float = percentage)	Yes

Parameter	Type	Description	Required
`keys`	`list[str]`	Document fields to use for matching/comparison	Yes
`query`	`str`	Query or ranking criteria (Jinja templates supported for `embedding` and `fts` only)	Yes

Method-Specific Parameters

Parameter	Type	Methods	Description	Default
`embedding_model`	`str`	`embedding`, `llm_compare`	Model for embeddings	`"text-embedding-3-small"`
`model`	`str`	`llm_compare`	LLM model for comparisons	Required for `llm_compare`
`batch_size`	`int`	`llm_compare`	Batch size for LLM ranking	`10`
`stratify_key`	`str` or `list[str]`	`embedding`, `fts`	Keys for stratified retrieval	`None`

Examples

Semantic Search with Embeddings

When you need to find documents based on meaning rather than exact keywords, the embedding method excels. This example finds support tickets semantically similar to payment processing issues:

```yaml
- name: find_relevant_tickets
  type: topk
  method: embedding
  k: 5
  keys:
    - subject
    - description
    - customer_feedback
  query: "payment processing errors with international transactions"
  embedding_model: text-embedding-3-small
```

Keyword Search with FTS

For cases where specific terms matter, such as searching a product catalog, the FTS method provides fast, accurate keyword matching without API costs:

```
- name: search_products
  type: topk
  method: fts
  k: 20
  keys:
    - product_name
    - description
    - category
    - tags
  query: "wireless noise cancelling headphones bluetooth"
```

Complex Ranking with LLM Compare

When you need to rank items based on multiple factors or subjective criteria, the LLM compare method allows you to specify complex ranking logic. Note that this method requires consistent criteria across all documents and doesn't support Jinja templates:

```
- name: screen_resumes
  type: topk
  method: llm_compare
  k: 10
  keys:
    - skills
    - experience
    - education
  query: |
    Rank candidates based on their fit for a Senior Backend Engineer role
requiring:
    - 5+ years Python experience
    - Distributed systems expertise
    - Strong knowledge of PostgreSQL and Redis
    - Experience with microservices architecture
    - Leadership experience is a plus

    Prioritize hands-on technical experience over academic credentials.
  model: gpt-4o
  batch_size: 5
```

Dynamic Queries with Templates

The embedding and FTS methods support Jinja templates for dynamic queries that adapt based on input data. This enables personalized search experiences:

```yaml
- name: personalized_search
  type: topk
  method: embedding
  k: 10
  keys:
    - content
    - tags
  query: |
    {{ input.user_preferences }}
    Focus on {{ input.topic_of_interest }}
    Exclude anything related to {{ input.blocked_topics }}
```

Stratified Retrieval

Both embedding and FTS methods support stratification, which ensures you retrieve top items from each group. This is useful for maintaining diversity in results:

```yaml
- name: recommendations_by_category
  type: topk
  method: fts
  k: 3  # Get top 3 from each category
  keys:
    - product_name
    - description
  query: "premium quality bestseller"
  stratify_key: category
```

Common Patterns

Single-Document RAG Pipeline

A retrieval-augmented generation pipeline for answering questions about a single document typically retrieves the most relevant chunks, then synthesizes them into a coherent answer using reduce:

```yaml
# Step 1: Retrieve most relevant document chunks
- name: retrieve_context
  type: topk
  method: embedding
  k: 5
  keys: [content]
  query: "{{ input.user_question }}"

# Step 2: Generate comprehensive answer from all retrieved chunks
- name: generate_answer
  type: reduce
  reduce_key: user_question  # Group by the question
  prompt: |
    Based on the following document excerpts, provide a comprehensive answer
to the question.

    Question: {{ inputs[0].user_question }}

    Retrieved context from document:
    {% for chunk in inputs %}
    - {{ chunk.content }}
    {% endfor %}

    Synthesize the information from all excerpts into a single, coherent
answer.
  output_schema:
    answer: string
```

Multi-Stage Filtering

For complex retrieval tasks, you might combine multiple TopK operations with different methods, progressively refining your results:

```yaml
# Cast a wide net with keyword search
- name: initial_search
  type: topk
  method: fts
  k: 100
  keys: [title, content]
  query: "machine learning"

# Refine with semantic search
- name: refine_results
  type: topk
  method: embedding
  k: 20
  keys: [title, content]
  query: "practical applications of deep learning in healthcare"

# Final ranking with LLM
- name: final_ranking
  type: topk
  method: llm_compare
  k: 5
  keys: [title, abstract, impact_factor]
  query: "Rank by potential clinical impact and implementation feasibility"
  model: gpt-4o
```

Performance Considerations

Each retrieval method has different performance characteristics that should guide your choice. The **FTS method** is the fastest and has no API costs since it uses local BM25 scoring, making it ideal for high-volume or cost-sensitive applications. The **embedding method** requires API calls to generate embeddings for both documents and queries, but once computed, similarity matching is very fast. It provides excellent semantic understanding but at a moderate cost. The **LLM compare method** has the highest cost and slowest performance due to multiple LLM calls, but offers unmatched flexibility for complex ranking criteria.

When optimizing performance, consider preprocessing embeddings offline for the embedding method, using FTS for initial filtering before applying more expensive methods, and carefully tuning the batch_size parameter for LLM compare to balance speed and accuracy.

Implementation Details

Embedding Method

The embedding method converts both your documents and query into high-dimensional vectors using the specified embedding model (defaulting to OpenAI's text-embedding-3-small). It then calculates cosine similarity between the query vector and each document vector, returning the k documents with highest similarity scores. The method handles text normalization and truncation automatically, and when stratification is enabled, it performs this process independently for each stratum.

FTS Method

The full-text search method uses the BM25 algorithm, the same ranking function used by search engines like Elasticsearch. Documents are tokenized and lowercase-normalized, with special characters removed. The BM25 scoring function considers term frequency (with saturation to prevent overweighting repeated terms), inverse document frequency (giving more weight to rare terms), and document length normalization. The implementation uses the rank-bm25 library for efficient scoring.

LLM Compare Method

The LLM compare method delegates to the rank operation, which implements sophisticated comparison algorithms from human-powered sorting research. It starts with an initial ordering using embeddings, then applies sliding windows of documents for pairwise comparison by the LLM. The LLM evaluates documents in batches (controlled by batch_size) and produces rankings based on the specified criteria. This method requires consistent ranking criteria across all documents, which is why Jinja templates are not supported—the LLM needs to compare all documents using the same criteria for fair ranking.

Error Handling

The TopK operation is designed to handle edge cases gracefully. If k exceeds the number of available documents, it returns all available items rather than failing. When specified keys are missing from some documents, it uses whatever fields are available. For the FTS method, documents with empty text after normalization are handled appropriately, and for the embedding method, API failures include automatic retries with exponential backoff.