Package ‘ragnar’

May 30, 2025

Title Retrieval-Augmented Generation (RAG) Workflows

Version 0.1.0

Description Provides tools for implementing Retrieval-Augmented Generation (RAG) workflows with Large Language Models (LLM). Includes functions for document processing, text chunking, embedding generation, storage management, and content retrieval. Supports various document types and embedding providers ('Ollama', 'OpenAI'), with 'DuckDB' as the default storage backend. Integrates with the 'ellmer' package to equip chat objects with retrieval capabilities. Designed to offer both sensible defaults and customization options with transparent access to intermediate outputs. For a review of retrieval-augmented generation methods, see Gao et al. (2023) ``Retrieval-Augmented Generation for Large Language Models: A Survey" <doi:10.48550/arXiv.2312.10997>.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.2

Depends R (>= 4.3.0)

Imports DBI, duckdb (>= 1.2.2), glue, rlang (>= 1.1.0), dplyr, httr2, rvest, stringi, tibble, vctrs, tidyr, xml2, methods, S7, reticulate (>= 1.42.0), commonmark, blob, cli, curl, withr, dotty

Suggests pandoc, ellmer, knitr, rmarkdown, stringr, dbplyr, testthat (>= 3.0.0), paws.common, shiny

Config/Needs/website tidyverse/tidytemplate, rmarkdown

Config/testthat/edition 3

URL `http://ragnar.tidyverse.org/`, `https://github.com/tidyverse/ragnar`

VignetteBuilder knitr

BugReports `https://github.com/tidyverse/ragnar/issues`

NeedsCompilation yes

Author Tomasz Kalinowski [aut, cre],
Daniel Falbel [aut],
Posit Software, PBC [cph, fnd] (ROR: <`https://ror.org/03wc8by49`>)

Maintainer Tomasz Kalinowski <`tomasz@posit.co`>

Repository CRAN

Date/Publication 2025-05-30 09:30:08 UTC

Contents

`embed_bedrock` *Embed text using a Bedrock model*

Description

Embed text using a Bedrock model

Usage

```
embed_bedrock(x, model, profile, api_args = list())
```

Arguments

`x`	x can be: • A character vector, in which case a matrix of embeddings is returned. • A data frame with a column named `text`, in which case the dataframe is returned with an additional column named `embedding`.

	• Missing or `NULL`, in which case a function is returned that can be called to get embeddings. This is a convenient way to partial in additional arguments like `model`, and is the most convenient way to produce a function that can be passed to the `embed` argument of `ragnar_store_create()`.
`model`	Currently only Cohere.ai and Amazon Titan models are supported. There are no guardarails for the kind of model that is used, but the model must be available in the AWS region specified by the profile. You may look for available models in the Bedrock Model Catalog
`profile`	AWS profile to use.
`api_args`	Additional arguments to pass to the Bedrock API. Dependending on the `model`, you might be able to provide different parameters. Check the documentation for the model you are using in the Bedrock user guide.

Value

If `x` is missing returns a function that can be called to get embeddings. If `x` is not missing, a matrix of embeddings with 1 row per input string, or a dataframe with an 'embedding' column.

See Also

`embed_ollama()`

`embed_ollama` *Embedd Text*

Description

Embedd Text

Usage

```
embed_ollama(
  x,
  base_url = "http://localhost:11434",
  model = "all-minilm",
  batch_size = 10L
)

embed_openai(
  x,
  model = "text-embedding-3-small",
  base_url = "https://api.openai.com/v1",
  api_key = get_envvar("OPENAI_API_KEY"),
  dims = NULL,
  user = get_ragnar_username(),
  batch_size = 20L
)
```

Arguments

`x`	x can be: • A character vector, in which case a matrix of embeddings is returned. • A data frame with a column named `text`, in which case the dataframe is returned with an additional column named `embedding`. • Missing or `NULL`, in which case a function is returned that can be called to get embeddings. This is a convenient way to partial in additional arguments like `model`, and is the most convenient way to produce a function that can be passed to the `embed` argument of `ragnar_store_create()`.
`base_url`	string, url where the service is available.
`model`	string; model name
`batch_size`	split `x` into batches when embedding. Integer, limit of strings to include in a single request.
`api_key`	resolved using env var `OPENAI_API_KEY`
`dims`	An integer, can be used to truncate the embedding to a specific size.
`user`	User name passed via the API.

Value

If `x` is a character vector, then a numeric matrix is returned, where `nrow = length(x)` and `ncol = <model-embedding-size>`. If `x` is a data.frame, then a new `embedding` matrix "column" is added, containing the matrix described in the previous sentence.

A matrix of embeddings with 1 row per input string, or a dataframe with an 'embedding' column.

Examples

```
text <- c("a chunk of text", "another chunk of text", "one more chunk of text")
## Not run:
text |>
  embed_ollama() |>
  str()

text |>
  embed_openai() |>
  str()

## End(Not run)
```

`markdown_segment` *Segment markdown text*

Description

Segment markdown text

Usage

```
markdown_segment(
  text,
  tags = c("h1", "h2", "h3", "h4"),
  trim = FALSE,
  omit_empty = FALSE
)

markdown_frame(text, frame_by = c("h1", "h2", "h3"), segment_by = NULL)
```

Arguments

`text`	Markdown string
`tags, segment_by`	A character vector of html tag names, e.g., `c("h1", "h2", "h3", "pre")`
`trim`	logical, trim whitespace on segments
`omit_empty`	logical, whether to remove empty segments
`frame_by`	Character vector of tags that will become columns in the returned dataframe.

Value

A named character vector. Names will correspond to `tags`, or `""` for content in between tags.

Examples

~~~
md <- r"---(

# Sample Markdown File

## Introduction

This is a sample **Markdown** file for testing.

### Features

- Simple **bold** text
- _Italicized_ text
- `Inline code`
- A [link](https://example.com)
- ‘Curly quotes are 3 bytes chars.’ Non-ascii text is fine.

This is a paragraph with <p> tag.

This next segment with code has a <pre> tag

```r
hello_world <- function() {
  cat("Hello, World!\n")
}
```
~~~

```
A table <table>:

  | Name  | Age | City      |
  |-------|----:|-----------|
  | Alice |  25 | New York  |
  | Bob   |  30 | London    |


## Conclusion

Common tags:

- h1, h2, h3, h4, h5, h6: section headings
- p: paragraph (prose)
- pre: pre-formatted text, meant to be displayed with monospace font.
  Typically code or code output
- blockquote: A blockquote
- table: A table
- ul: Unordered list
- ol: Ordered list
- li: Individual list item in a <ul> or <ol>


)---"
markdown_segment(md) |> tibble::enframe()
markdown_segment(md |> trimws()) |> tibble::enframe()
markdown_segment(md, c("li"), trim = TRUE, omit_empty = TRUE) |> tibble::enframe()
markdown_segment(md, c("table"), trim = TRUE, omit_empty = TRUE) |> tibble::enframe()
markdown_segment(md, c("ul"), trim = TRUE, omit_empty = TRUE) |> tibble::enframe()
```

ragnar_chunk *Chunk text*

Description

Functions for chunking text into smaller pieces while preserving meaningful semantics. These functions provide flexible ways to split text based on various boundaries (sentences, words, etc.) while controlling chunk sizes and overlap.

Usage

```
ragnar_chunk(
  x,
  max_size = 1600L,
  boundaries = c("paragraph", "sentence", "line_break", "word", "character"),
  ...,
  trim = TRUE,
  simplify = TRUE
)

ragnar_segment(x, boundaries = "sentence", ..., trim = FALSE, simplify = TRUE)

ragnar_chunk_segments(x, max_size = 1600L, ..., simplify = TRUE, trim = TRUE)
```

Arguments

`x`	A character vector, list of character vectors, or data frame containing a `text` column.
`max_size`	Integer. The maximum number of characters in each chunk. Defaults to `1600`, which typically is approximately 400 tokens, or 1 page of text.
`boundaries`	A sequence of boundary types to use in order until `max_size` is satisfied. Valid values are `"sentence"`, `"word"`, `"line_break"`, `"character"`, `"paragraph"`, or a `stringr_pattern` object like `stringr::fixed()`.
`...`	Additional arguments passed to internal functions. tokenizer to use `tokens` instead of characters as the count (not fully implemented yet)
`trim`	logical, whether to trim leading and trailing whitespace from strings. Default `TRUE`.
`simplify`	Logical. If `TRUE`, the output is simplified. If `FALSE`, returns a vector that has the same length as `x`. If `TRUE`, character strings are `unlist()`ed, and dataframes are `tidyr::unchop()`ed.

Details

Chunking is the combination of two fundamental operations:

- identifying boundaries: finding character positions where it makes sense to split a string.
- extracting slices: extracting substrings using the candidate boundaries to produce chunks that match the requested `chunk_size` and `chunk_overlap`

`ragnar_chunk()` is a higher-level function that does both, identifies boundaries and extracts slices.

If you need lower-level control, you can alternatively use the lower-level functions `ragnar_segment()` in combination with `ragnar_chunk_segments()`.

`ragnar_segment()`: Splits text at semantic boundaries.

`ragnar_chunk_segments()`: Combines text segments into chunks.

For most usecases, these two are equivalent:

```
x |> ragnar_chunk()
x |> ragnar_segment() |> ragnar_chunk_segments()
```

When working with data frames, these functions preserve all columns and use `tidyr::unchop()` to handle the resulting list-columns when `simplify = TRUE`.

Value

- For character input with `simplify = FALSE`: A list of character vectors
- For character input with `simplify = TRUE`: A character vector of chunks
- For data frame input with `simplify = FALSE`: A data frame with the same number of rows as the input, where the `text` column transformed into a list of chararacter vectors.
- For data frame input with `simplify = TRUE`: Same as a data frame input with `simplify=FALSE`, with the `text` column expanded by `tidyr::unchop()`

Examples

```
# Basic chunking with max size
text <- "This is a long piece of text. It has multiple sentences.
         We want to split it into chunks. Here's another sentence."
ragnar_chunk(text, max_size = 40) # splits at sentences

# smaller chunk size: first splits at sentence boundaries, then word boundaries
ragnar_chunk(text, max_size = 20)

# only split at sentence boundaries. Note, some chunks are oversized
ragnar_chunk(text, max_size = 20, boundaries = c("sentence"))

# only consider word boundaries when splitting:
ragnar_chunk(text, max_size = 20, boundaries = c("word"))

# first split at sentence boundaries, then word boundaries,
# as needed to satisfy `max_chunk`
ragnar_chunk(text, max_size = 20, boundaries = c("sentence", "word"))

# Use a stringr pattern to find semantic boundaries
ragnar_chunk(text, max_size = 10, boundaries = stringr::fixed(". "))
ragnar_chunk(text, max_size = 10, boundaries = list(stringr::fixed(". "), "word"))


# Working with data frames
df <- data.frame(
  id = 1:2,
  text = c("First sentence. Second sentence.", "Another sentence here.")
)
ragnar_chunk(df, max_size = 20, boundaries = "sentence")
ragnar_chunk(df$text, max_size = 20, boundaries = "sentence")

# Chunking pre-segmented text
segments <- c("First segment. ", "Second segment. ", "Third segment. ", "Fourth segment. ")
ragnar_chunk_segments(segments, max_size = 20)
ragnar_chunk_segments(segments, max_size = 40)
ragnar_chunk_segments(segments, max_size = 60)
```

`ragnar_find_links` *Find links on a page*

Description

Find links on a page

Usage

```
ragnar_find_links(
  x,
  depth = 0L,
  children_only = TRUE,
  progress = TRUE,
  ...,
  url_filter = identity
)
```

Arguments

`x`	URL, HTML file path, or XML document. For Markdown, convert to HTML using `commonmark::markdown_html()` first.
`depth`	Integer specifying how many levels deep to crawl for links. When `depth` > `0`, the function will follow child links (links with `x` as a prefix) and collect links from those pages as well.
`children_only`	Logical or string. If `TRUE`, returns only child links (those having `x` as a prefix). If `FALSE`, returns all links found on the page. Note that regardless of this setting, only child links are followed when `depth` > `0`.
`progress`	Logical, draw a progress bar if `depth` > `0`. A separate progress bar is drawn per recursion level.
`...`	Currently unused. Must be empty.
`url_filter`	A function that takes a character vector of URL's and may subset them to return a smaller list. This can be useful for filtering out URL's by rules different them `children_only` which only checks the prefix.

Value

A character vector of links on the page.

Examples

```
## Not run:
ragnar_find_links("https://r4ds.hadley.nz/base-R.html")
ragnar_find_links("https://ellmer.tidyverse.org/")
ragnar_find_links("https://ellmer.tidyverse.org/", depth = 2)
ragnar_find_links("https://ellmer.tidyverse.org/", depth = 2, children_only = FALSE)
ragnar_find_links(
  paste0("https://github.com/Snowflake-Labs/sfquickstarts/",
         "tree/master/site/sfguides/src/build_a_custom_model_for_anomaly_detection"),
  children_only = "https://github.com/Snowflake-Labs/sfquickstarts",
  depth = 1
)

## End(Not run)
```

---

ragnar_read *Read a document as Markdown*

Description

ragnar_read() uses markitdown to convert a document to markdown. If frame_by_tags or split_by_tags is provided, the converted markdown content is then split and converted to a data frame, otherwise, the markdown is returned as a string.

Usage

```
ragnar_read(x, ..., split_by_tags = NULL, frame_by_tags = NULL)
```

Arguments

x	file path or url.
...	passed on markitdown.convert.
split_by_tags	character vector of html tag names used to split the returned text
frame_by_tags	character vector of html tag names used to create a dataframe of the returned content

Value

Always returns a data frame with the columns:

- origin: the file path or url
- hash: a hash of the text content
- text: the markdown content

If split_by_tags is not NULL, then a tag column is also included containing the corresponding tag for each text chunk. "" is used for text chunks that are not associated with a tag.

If frame_by_tags is not NULL, then additional columns are included for each tag in frame_by_tags. The text chunks are associated with the tags in the order they appear in the markdown content.

Examples

```
file <- tempfile(fileext = ".html")
download.file("https://r4ds.hadley.nz/base-R.html", file, quiet = TRUE)

# with no arguments, returns a single row data frame.
# the markdown content is in the `text` column.
file |> ragnar_read() |> str()

# use `split_by_tags` to get a data frame where the text is split by the
# specified tags (e.g., "h1", "h2", "h3")
file |>
  ragnar_read(split_by_tags = c("h1", "h2", "h3"))

# use `frame_by_tags` to get a dataframe where the
# headings associated with each text chunk are easily accessible
file |>
  ragnar_read(frame_by_tags = c("h1", "h2", "h3"))

# use `split_by_tags` and `frame_by_tags` together to further break up `text`.
file |>
  ragnar_read(
    split_by_tags = c("p"),
    frame_by_tags = c("h1", "h2", "h3")
  )

# Example workflow adding context to each chunk
file |>
  ragnar_read(frame_by_tags = c("h1", "h2", "h3")) |>
  glue::glue_data(r"--(
    ## Excerpt from the book "R for Data Science (2e)"
    chapter: {h1}
    section: {h2}
    content: {text}

    )--") |>
  # inspect
  _[6:7] |> cat(sep = "\n~~~~~~~~~~~\n")

# Advanced example of postprocessing the output of ragnar_read()
# to add language to code blocks, markdown style
library(dplyr, warn.conflicts = FALSE)
library(stringr)
library(rvest)
library(xml2)
file |>
  ragnar_read(frame_by_tags = c("h1", "h2", "h3"),
              split_by_tags = c("p", "pre")) |>
  mutate(
    is_code = tag == "pre",
    text = ifelse(is_code, str_replace(text, "```", "```r"), text)
  ) |>
  group_by(h1, h2, h3) |>
```

```
  summarise(text = str_flatten(text, "\n\n"), .groups = "drop") |>
  glue::glue_data(r"--(
    # Excerpt from the book "R for Data Science (2e)"
    chapter: {h1}
    section: {h2}
    content: {text}

    )--") |>
  # inspect
  _[9:10] |> cat(sep = "\n~~~~~~~~~~~\n")
```

`ragnar_read_document` *Read an HTML document*

Description

Read an HTML document

Usage

```
ragnar_read_document(
  x,
  ...,
  split_by_tags = frame_by_tags,
  frame_by_tags = NULL
)
```

Arguments

`x`	file path or url, passed on to `rvest::read_html()`, or an `xml_node`.
`...`	passed on to `rvest::read_html()`
`split_by_tags`	character vector of html tag names used to split the returned text
`frame_by_tags`	character vector of html tag names used to create a dataframe of the returned content

Value

If `frame_by_tags` is not `NULL`, then a data frame is returned, with column names `c("frame_by_tags", "text")`.

If `frame_by_tags` is `NULL` but `split_by_tags` is not `NULL`, then a named character vector is returned.

If both `frame_by_tags` and `split_by_tags` are `NULL`, then a string (length-1 character vector) is returned.

Examples

```
file <- tempfile(fileext = ".html")
download.file("https://r4ds.hadley.nz/base-R.html", file, quiet = TRUE)

# with no arguments, returns a single string of the text.
file |> ragnar_read_document() |> str()

# use `split_by_tags` to get a named character vector of length > 1
file |>
  ragnar_read_document(split_by_tags = c("h1", "h2", "h3")) |>
  tibble::enframe("tag", "text")

# use `frame_by_tags` to get a dataframe where the
# headings associated with each text chunk are easily accessible
file |>
  ragnar_read_document(frame_by_tags = c("h1", "h2", "h3"))

# use `split_by_tags` and `frame_by_tags` together to further break up `text`.
file |>
  ragnar_read_document(
    split_by_tags = c("p"),
    frame_by_tags = c("h1", "h2", "h3")
  )

# Example workflow adding context to each chunk
file |>
  ragnar_read_document(frame_by_tags = c("h1", "h2", "h3")) |>
  glue::glue_data(r"--(
    ## Excerpt from the book "R for Data Science (2e)"
    chapter: {h1}
    section: {h2}
    content: {text}

    )--") |>
    # inspect
    _[6:7] |> cat(sep = "\n~~~~~~~~~~~\n")

# Advanced example of postprocessing the output of ragnar_read_document()
# to wrap code blocks in backticks, markdown style
library(dplyr, warn.conflicts = FALSE)
library(stringr)
library(rvest)
library(xml2)
file |>
  ragnar_read_document(frame_by_tags = c("h1", "h2", "h3"),
                       split_by_tags = c("p", "pre")) |>
  mutate(
    is_code = tag == "pre",
    text = ifelse(is_code,
                  str_c("```", text, "```", sep = "\n"),
                  text)) |>
  group_by(h1, h2, h3) |>
```

```
    summarise(text = str_flatten(text, "\n"), .groups = "drop") |>
    glue::glue_data(r"--(
      # Excerpt from the book "R for Data Science (2e)"
      chapter: {h1}
      section: {h2}
      content: {text}

      )--") |>
      # inspect
      _[9:10] |> cat(sep = "\n~~~~~~~~~~~\n")

  # Example of preprocessing the input to ragnar_read_document()
  # to wrap code in backticks, markdown style
  # same outcome as above, except via pre processing instead of post processing.
  file |>
    read_html() |>
    (\(doc) {
      # fence preformatted code with triple backticks
      for (node in html_elements(doc, "pre")) {
        xml_add_child(node, "code", "```\n", .where = 0)
        xml_add_child(node, "code", "\n```")
      }
      # wrap inline code with single backticks
      for (node in html_elements(doc, "code")) {
        if (!"pre" %in% xml_name(xml_parents(node))) {
          xml_text(node) <- str_c("`", xml_text(node), "`")
        }
      }
      doc
    })() |>
    ragnar_read_document(frame_by_tags = c("h1", "h2", "h3")) |>
    glue::glue_data(r"--(
      # Excerpt from the book "R for Data Science (2e)"
      chapter: {h1}
      section: {h2}
      content: {text}

      )--") |> _[6]
```

`ragnar_register_tool_retrieve`

Register a 'retrieve' tool with ellmer

Description

Register a 'retrieve' tool with ellmer

Usage

```
ragnar_register_tool_retrieve(
```

```
    chat,
    store,
    store_description = "the knowledge store",
    ...
  )
```

Arguments

`chat`	a `ellmer::Chat` object.
`store`	a string of a store location, or a `RagnarStore` object.
`store_description`	Optional string, used for composing the tool description.
`...`	arguments passed on to `ragnar_retrieve()`.

Value

`chat`, invisibly.

Examples

```
system_prompt <- stringr::str_squish("
  You are an expert assistant in R programming.
  When responding, you first quote relevant material from books or documentation,
  provide links to the sources, and then add your own context and interpretation.
")
chat <- ellmer::chat_openai(system_prompt, model = "gpt-4o")

store <- ragnar_store_connect("r4ds.ragnar.duckdb", read_only = TRUE)
ragnar_register_tool_retrieve(chat, store)
chat$chat("How can I subset a dataframe?")
```

`ragnar_retrieve` *Retrieve chunks from a* `RagnarStore`

Description

`ragnar_retrieve()` is a thin wrapper around `ragnar_retrieve_vss_and_bm25()` using the recommended best practices.

Usage

```
ragnar_retrieve(store, text, top_k = 3L)
```

Arguments

store	A RagnarStore object or a `dplyr::tbl()` derived from it. When you pass a `tbl`, you may use usual dplyr verbs (e.g. `filter()`, `slice()`) to restrict the rows examined before similarity scoring. Avoid dropping essential columns such as `text`, `embedding`, `origin`, and `hash`.
text	A string to find the nearest match too
top_k	Integer, the number of nearest entries to find *per method*.

Value

A dataframe of retrieved chunks. Each row corresponds to an individual chunk in the store. It always contains a column named `text` that contains the chunks.

Pre-filtering with dplyr

The store behaves like a lazy table backed by DuckDB, so row-wise filtering is executed directly in the database. This lets you narrow the search space efficiently without pulling data into R.

See Also

Other ragnar_retrieve: `ragnar_retrieve_bm25()`, `ragnar_retrieve_vss()`, `ragnar_retrieve_vss_and_bm25()`

Examples

```
# Basic usage
store <- ragnar_store_create(
  embed = \(x) ragnar::embed_openai(x, model = "text-embedding-3-small")
)
ragnar_store_insert(store, data.frame(text = c("foo", "bar")))
ragnar_store_build_index(store)
ragnar_retrieve(store, "foo")

# More Advanced: store metadata, retrieve with pre-filtering
store <- ragnar_store_create(
  embed = \(x) ragnar::embed_openai(x, model = "text-embedding-3-small"),
  extra_cols = data.frame(category = character())
)

ragnar_store_insert(
  store,
  data.frame(
    category = "desert",
    text = c("ice cream", "cake", "cookies")
  )
)

ragnar_store_insert(
  store,
  data.frame(
    category = "meal",
    text = c("steak", "potatoes", "salad")
  )
)

ragnar_store_build_index(store)

# simple retrieve
ragnar_retrieve(store, "carbs")

# retrieve with pre-filtering
dplyr::tbl(store) |>
  dplyr::filter(category == "meal") |>
  ragnar_retrieve("carbs")
```

`ragnar_retrieve_bm25` *Retrieves chunks using the BM25 score*

Description

BM25 refers to Okapi Best Matching 25. See doi:10.1561/1500000019 for more information.

Usage

```
ragnar_retrieve_bm25(store, text, top_k = 3L)
```

Arguments

`store`	A `RagnarStore` object or a `dplyr::tbl()` derived from it. When you pass a `tbl`, you may use usual dplyr verbs (e.g. `filter()`, `slice()`) to restrict the rows examined before similarity scoring. Avoid dropping essential columns such as `text`, `embedding`, `origin`, and `hash`.
`text`	A string to find the nearest match too
`top_k`	Integer, maximum amount of document chunks to retrieve

Details

The supported methods are:

- **cosine_distance**: Measures the dissimilarity between two vectors based on the cosine of the angle between them. Defined as $1 - cos(\theta)$, where $cos(\theta)$ is the cosine similarity.
- **cosine_similarity**: Measures the similarity between two vectors based on the cosine of the angle between them. Ranges from -1 (opposite) to 1 (identical), with 0 indicating orthogonality.
- **euclidean_distance**: Computes the straight-line (L2) distance between two points in a multidimensional space. Defined as $\sqrt{\sum(x_i - y_i)^2}$.
- **dot_product**: Computes the sum of the element-wise products of two vectors.
- **negative_dot_product**: The negation of the dot product.

Value

A dataframe of retrieved chunks. Each row corresponds to an individual chunk in the store. It always contains a column named `text` that contains the chunks.

Pre-filtering with dplyr

The store behaves like a lazy table backed by DuckDB, so row-wise filtering is executed directly in the database. This lets you narrow the search space efficiently without pulling data into R.

See Also

Other ragnar_retrieve: `ragnar_retrieve()`, `ragnar_retrieve_vss()`, `ragnar_retrieve_vss_and_bm25()`

Examples

```
# Basic usage
store <- ragnar_store_create(
  embed = \(x) ragnar::embed_openai(x, model = "text-embedding-3-small")
)
ragnar_store_insert(store, data.frame(text = c("foo", "bar")))
ragnar_store_build_index(store)
ragnar_retrieve(store, "foo")

# More Advanced: store metadata, retrieve with pre-filtering
store <- ragnar_store_create(
  embed = \(x) ragnar::embed_openai(x, model = "text-embedding-3-small"),
  extra_cols = data.frame(category = character())
)

ragnar_store_insert(
  store,
  data.frame(
    category = "desert",
    text = c("ice cream", "cake", "cookies")
  )
)

ragnar_store_insert(
  store,
  data.frame(
    category = "meal",
    text = c("steak", "potatoes", "salad")
  )
)

ragnar_store_build_index(store)

# simple retrieve
ragnar_retrieve(store, "carbs")

# retrieve with pre-filtering
dplyr::tbl(store) |>
```

```
  dplyr::filter(category == "meal") |>
  ragnar_retrieve("carbs")
```

`ragnar_retrieve_vss` *Uses vector similarity search*

Description

Computes a similarity measure between the query and the documents embeddings and uses this similarity to rank the documents.

Usage

```
ragnar_retrieve_vss(
  store,
  text,
  top_k = 3L,
 method = c("cosine_distance", "cosine_similarity", "euclidean_distance", "dot_product",
    "negative_dot_product")
)
```

Arguments

`store`	A `RagnarStore` object or a `dplyr::tbl()` derived from it. When you pass a `tbl`, you may use usual dplyr verbs (e.g. `filter()`, `slice()`) to restrict the rows examined before similarity scoring. Avoid dropping essential columns such as `text`, `embedding`, `origin`, and `hash`.
`text`	A string to find the nearest match too
`top_k`	Integer, maximum amount of document chunks to retrieve
`method`	A string specifying the method used to compute the similarity between the query and the document chunks embeddings store in the database.

Details

The supported methods are:

- **cosine_distance**: Measures the dissimilarity between two vectors based on the cosine of the angle between them. Defined as $1 - cos(\theta)$, where $cos(\theta)$ is the cosine similarity.
- **cosine_similarity**: Measures the similarity between two vectors based on the cosine of the angle between them. Ranges from -1 (opposite) to 1 (identical), with 0 indicating orthogonality.
- **euclidean_distance**: Computes the straight-line (L2) distance between two points in a multi-dimensional space. Defined as $\sqrt{\sum(x_i - y_i)^2}$.
- **dot_product**: Computes the sum of the element-wise products of two vectors.
- **negative_dot_product**: The negation of the dot product.

Value

A dataframe of retrieved chunks. Each row corresponds to an individual chunk in the store. It always contains a column named `text` that contains the chunks.

Pre-filtering with dplyr

The store behaves like a lazy table backed by DuckDB, so row-wise filtering is executed directly in the database. This lets you narrow the search space efficiently without pulling data into R.

See Also

Other ragnar_retrieve: `ragnar_retrieve()`, `ragnar_retrieve_bm25()`, `ragnar_retrieve_vss_and_bm25()`

Examples

```
# Basic usage
store <- ragnar_store_create(
  embed = \(x) ragnar::embed_openai(x, model = "text-embedding-3-small")
)
ragnar_store_insert(store, data.frame(text = c("foo", "bar")))
ragnar_store_build_index(store)
ragnar_retrieve(store, "foo")

# More Advanced: store metadata, retrieve with pre-filtering
store <- ragnar_store_create(
  embed = \(x) ragnar::embed_openai(x, model = "text-embedding-3-small"),
  extra_cols = data.frame(category = character())
)

ragnar_store_insert(
  store,
  data.frame(
    category = "desert",
    text = c("ice cream", "cake", "cookies")
  )
)

ragnar_store_insert(
  store,
  data.frame(
    category = "meal",
    text = c("steak", "potatoes", "salad")
  )
)

ragnar_store_build_index(store)

# simple retrieve
ragnar_retrieve(store, "carbs")

# retrieve with pre-filtering
dplyr::tbl(store) |>
```

```
  dplyr::filter(category == "meal") |>
  ragnar_retrieve("carbs")
```

`ragnar_retrieve_vss_and_bm25`   *Retrieve VSS and BM25*

Description

Runs `ragnar_retrieve_vss()` and `ragnar_retrieve_bm25()` and get the distinct documents.

Usage

```
ragnar_retrieve_vss_and_bm25(store, text, top_k = 3, ...)
```

Arguments

`store`	A `RagnarStore` object or a `dplyr::tbl()` derived from it. When you pass a `tbl`, you may use usual dplyr verbs (e.g. `filter()`, `slice()`) to restrict the rows examined before similarity scoring. Avoid dropping essential columns such as `text`, `embedding`, `origin`, and `hash`.
`text`	A string to find the nearest match too
`top_k`	Integer, the number of entries to retrieve using **per method**.
`...`	Forwarded to `ragnar_retrieve_vss()`

Value

A dataframe of retrieved chunks. Each row corresponds to an individual chunk in the store. It always contains a column named `text` that contains the chunks.

Pre-filtering with dplyr

The store behaves like a lazy table backed by DuckDB, so row-wise filtering is executed directly in the database. This lets you narrow the search space efficiently without pulling data into R.

Note

The results are not re-ranked after identifying the unique values.

See Also

Other ragnar_retrieve: `ragnar_retrieve()`, `ragnar_retrieve_bm25()`, `ragnar_retrieve_vss()`

Examples

```
# Basic usage
store <- ragnar_store_create(
  embed = \(x) ragnar::embed_openai(x, model = "text-embedding-3-small")
)
ragnar_store_insert(store, data.frame(text = c("foo", "bar")))
ragnar_store_build_index(store)
ragnar_retrieve(store, "foo")

# More Advanced: store metadata, retrieve with pre-filtering
store <- ragnar_store_create(
  embed = \(x) ragnar::embed_openai(x, model = "text-embedding-3-small"),
  extra_cols = data.frame(category = character())
)

ragnar_store_insert(
  store,
  data.frame(
    category = "desert",
    text = c("ice cream", "cake", "cookies")
  )
)

ragnar_store_insert(
  store,
  data.frame(
    category = "meal",
    text = c("steak", "potatoes", "salad")
  )
)

ragnar_store_build_index(store)

# simple retrieve
ragnar_retrieve(store, "carbs")

# retrieve with pre-filtering
dplyr::tbl(store) |>
  dplyr::filter(category == "meal") |>
  ragnar_retrieve("carbs")
```

`ragnar_store_build_index` *Build a Ragnar Store index*

Description

A search index must be built before calling `ragnar_retrieve()`. If additional entries are added to the store with `ragnar_store_insert()`, `ragnar_store_build_index()` must be called again to rebuild the index.

Usage

```
ragnar_store_build_index(store, type = c("vss", "fts"))
```

Arguments

`store`	a `RagnarStore` object
`type`	The retrieval search type to build an index for.

Value

`store`, invisibly.

`ragnar_store_connect` *Connect to* `RagnarStore`

Description

Connect to `RagnarStore`

Usage

```
ragnar_store_connect(
  location = ":memory:",
  ...,
  read_only = FALSE,
  build_index = FALSE
)
```

Arguments

`location`	string, a filepath location.
`...`	unused; must be empty.
`read_only`	logical, whether the returned connection can be used to modify the store.
`build_index`	logical, whether to call `ragnar_store_build_index()` when creating the connection

Value

a `RagnarStore` object.

ragnar_store_create *Create and connect to a vector store*

Description

Create and connect to a vector store

Usage

```
ragnar_store_create(
  location = ":memory:",
  embed = embed_ollama(),
  embedding_size = ncol(embed("foo")),
  overwrite = FALSE,
  ...,
  extra_cols = NULL,
  name = NULL
)
```

Arguments

`location`	filepath, or `:memory:`
`embed`	A function that is called with a character vector and returns a matrix of embeddings. Note this function will be serialized and then deserialized in new R sessions, so it cannot reference to any objects in the global or parent environments. Make sure to namespace all function calls with `::`. If additional R objects must be available in the function, you can optionally supply a `carrier::crate()` with packaged data. It can also be `NULL` for stores that don't need to embed their texts, for example, if only using FTS algorithms such as `ragnar_retrieve_bm25()`.
`embedding_size`	integer
`overwrite`	logical, what to do if `location` already exists
`...`	Unused. Must be empty.
`extra_cols`	A zero row data frame used to specify additional columns that should be added to the store. Such columns can be used for adding additional context when retrieving. See the examples for more information. `vctrs::vec_cast()` is used to consistently perform type checks and casts when inserting with `ragnar_store_insert()`.
`name`	A unique name for the store. Must match the `^[a-zA-Z0-9_-]+$` regex. Used by `ragnar_register_tool_retrieve()` for registering tools.

Value

a `DuckDBRagnarStore` object

Examples

```
# A store with a dummy embedding
store <- ragnar_store_create(
  embed = \(x) matrix(stats::runif(10), nrow = length(x), ncol = 10),
)
ragnar_store_insert(store, data.frame(text = "hello"))

# A store with a schema. When inserting into this store, users need to
# provide a `area` column.
store <- ragnar_store_create(
  embed = \(x) matrix(stats::runif(10), nrow = length(x), ncol = 10),
  extra_cols = data.frame(area = character()),
)
ragnar_store_insert(store, data.frame(text = "hello", area = "rag"))

# If you already have a data.frame with chunks that will be inserted into
# the store, you can quickly create a suitable store with:
chunks <- data.frame(text = letters, area = "rag")
store <- ragnar_store_create(
  embed = \(x) matrix(stats::runif(10), nrow = length(x), ncol = 10),
  extra_cols = vctrs::vec_ptype(chunks),
)
ragnar_store_insert(store, chunks)
```

`ragnar_store_insert` *Insert chunks into a* `RagnarStore`

Description

Insert chunks into a `RagnarStore`

Usage

```
ragnar_store_insert(store, chunks)
```

Arguments

`store`	a `RagnarStore` object
`chunks`	a character vector or a dataframe with a `text` column, and optionally, a pre-computed `embedding` matrix column. If `embedding` is not present, then `store@embed()` is used. `chunks` can also be a character vector.

Value

`store`, invisibly.

`ragnar_store_inspect` *Launches the Ragnar Inspector Tool*

Description

Launches the Ragnar Inspector Tool

Usage

```
ragnar_store_inspect(store, ...)
```

Arguments

`store`	A `RagnarStore` object that you want to inspect with the tool.
`...`	Passed to `shiny::runApp()`.

Value

`NULL` invisibly

`ragnar_store_update` *Inserts or updates chunks in a* `RagnarStore`

Description

Inserts or updates chunks in a `RagnarStore`

Usage

```
ragnar_store_update(store, chunks)
```

Arguments

`store`	a `RagnarStore` object
`chunks`	a character vector or a dataframe with a `text` column, and optionally, a pre-computed `embedding` matrix column. If `embedding` is not present, then `store@embed()` is used. `chunks` can also be a character vector.

Details

`chunks` must be a data frame containing `origin` and `hash` columns. We first filter out chunks for which `origin` and `hash` are already in the store. If an `origin` is in the store, but with a different `hash`, we all of its chunks with the new chunks. Otherwise, a regular insert is performed.

This can help spending less time computing embeddings for chunks that are already in the store.

Value

`store`, invisibly.

`read_as_markdown` *Convert files to markdown*

Description

Convert files to markdown

Usage

```
read_as_markdown(x, ..., canonical = FALSE)
```

Arguments

`x`	A filepath or url. Accepts a wide variety of file types, including PDF, PowerPoint, Word, Excel, Images (EXIF metadata and OCR), Audio (EXIF metadata and speech transcription), HTML, Text-based formats (CSV, JSON, XML), ZIP files (iterates over contents), Youtube URLs, and EPubs.#'
`...`	Passed on to `MarkItDown.convert()`
`canonical`	logical, whether to postprocess the output from MarkItDown with `commonmark::markdown_commonmark(`

Value

A single string of markdown

Examples

```
# convert html
read_as_markdown("https://r4ds.hadley.nz/base-R.html") |>
  substr(1, 1000) |>
  cat()

read_as_markdown("https://r4ds.hadley.nz/base-R.html", canonical = TRUE) |>
  substr(1, 1000) |>
  cat()

# convert pdf
pdf <- file.path(R.home("doc"), "NEWS.pdf")
read_as_markdown(pdf) |> substr(1, 1000) |> cat()
## alternative:
# pdftools::pdf_text(pdf) |> substr(1, 2000) |> cat()

# convert images to markdown descriptions using OpenAI
jpg <- file.path(R.home("doc"), "html", "logo.jpg")
if (Sys.getenv("OPENAI_API_KEY") != "") {
  # if (xfun::is_macos()) system("brew install ffmpeg")
  reticulate::py_require("openai")
  llm_client <- reticulate::import("openai")$OpenAI()
  read_as_markdown(jpg, llm_client = llm_client, llm_model = "gpt-4.1-mini")
  # # Description:
  # The image displays the logo of the R programming language. It features a
  # large, stylized capital letter "R" in blue, positioned prominently in the
  # center. Surrounding the "R" is a gray oval shape that is open on the right
  # side, creating a dynamic and modern appearance. The R logo is commonly
  # associated with statistical computing, data analysis, and graphical
  # representation in various scientific and professional fields.
}

# Alternative approach to image conversion:
if (
  Sys.getenv("OPENAI_API_KEY") != "" &&
    rlang::is_installed("ellmer") &&
    rlang::is_installed("magick")
) {
  chat <- ellmer::chat_openai(echo = TRUE)
  chat$chat("Describe this image", ellmer::content_image_file(jpg))
}
```

Index