Chapter 8 - Semantic Search and Retrieval-Augmented Generation

One of the things language models enabled is the ability to search using semantic meaning instead of just having to use keyword matching. This is called ***semantic search***.

While the development of generation models allowed for fluent and readable responses, these responses are not always completely factual. To address this, one of the ways developed to provide knowledge to the language model is by using **RAG (Retrieval-Augmented Generation)**.

Overview of Semantic Search and RAG

The top **three** broad categories of language models used for search are:

1. **Dense Retrieval**
2. **Reranking**
3. **RAG**

Below, we will go through each of these categories one by one and in-depth.

Dense Retrieval

Dense Retrieval systems rely on the concept of *embeddings* (discussed in depth in the early chapters). Using embeddings, we can turn searching into retrieving the nearest neighbors.

> **Recall**
>
> Remember that the float values of an embedding represent their properties and their coordinates in an embedding space.
>
> This embedding space could have a hundred or more dimensions. The closer the embeddings of texts are, **the closer their meanings are.**

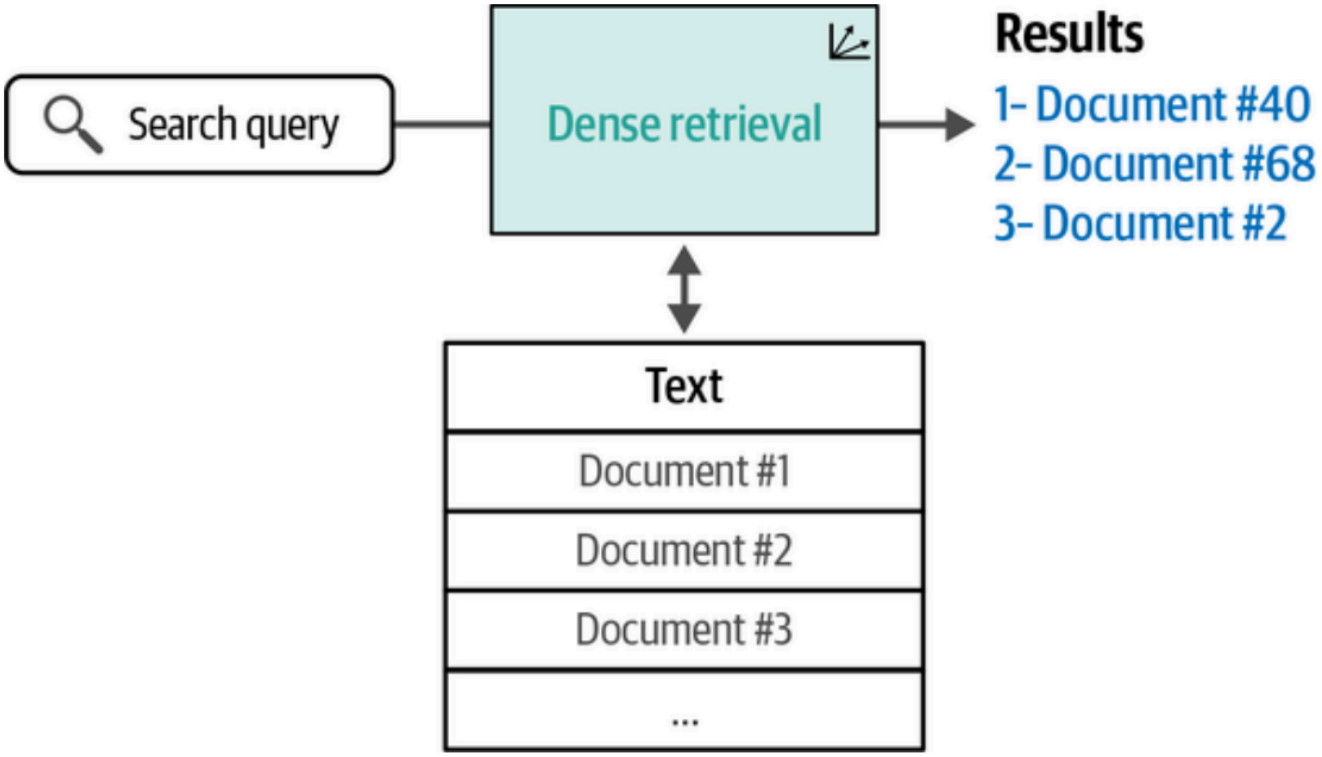

Figure 8-1. Dense retrieval is one of the key types of semantic search, relying on the similarity of text embeddings to retrieve relevant results.

Reranking

Search systems often contain multiple steps. A **reranking** language model is one of these steps.

Reranking language models are tasked with *scoring the relevance of a subset of results* against the query. See the diagram below.

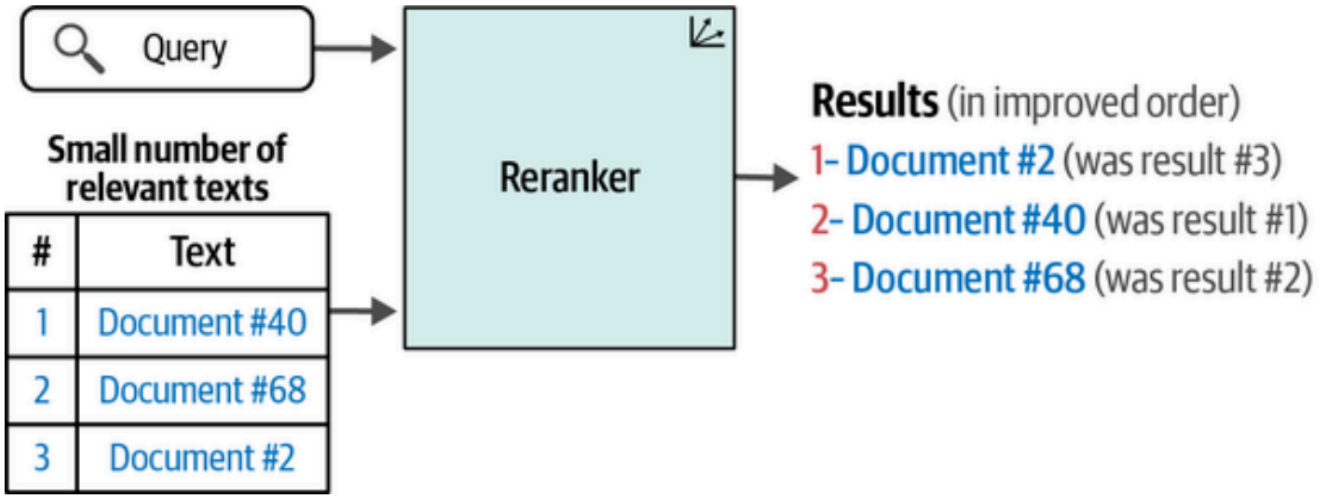

Figure 8-2. Rerankers, the second key type of semantic search, take a search query and a collection of results, and reorder them by relevance, often resulting in vastly improved results.

RAG

The growing LLM capabilities have led to a new type of search systems that include a model that generates an answer in response to a query.

Generative search is a subset of a broader type of category of systems called **RAG systems.**

RAG Systems incorporate search capabilities to reduce hallucinations, increase factuality and ground the generation model on a specific dataset.

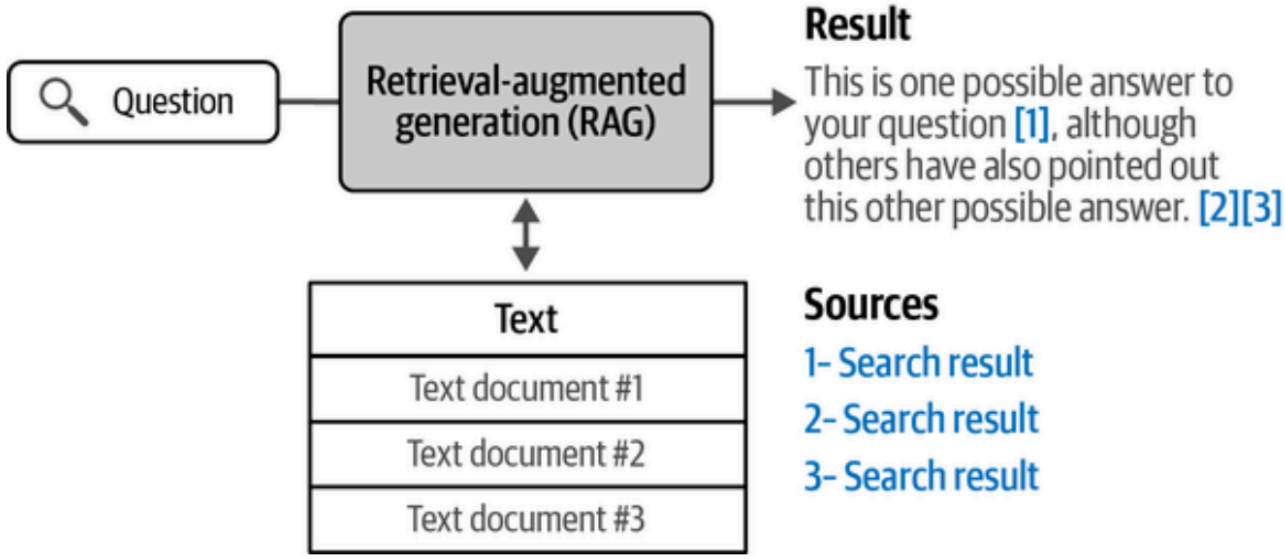

Figure 8-3. A RAG system formulates an answer to a question and (preferably) cites its information

Semantic Search with Language Models

Dense Retrievals

As briefly mentioned earlier in the text, we can think of the values in an embedding as points in a space. This is illustrated below.

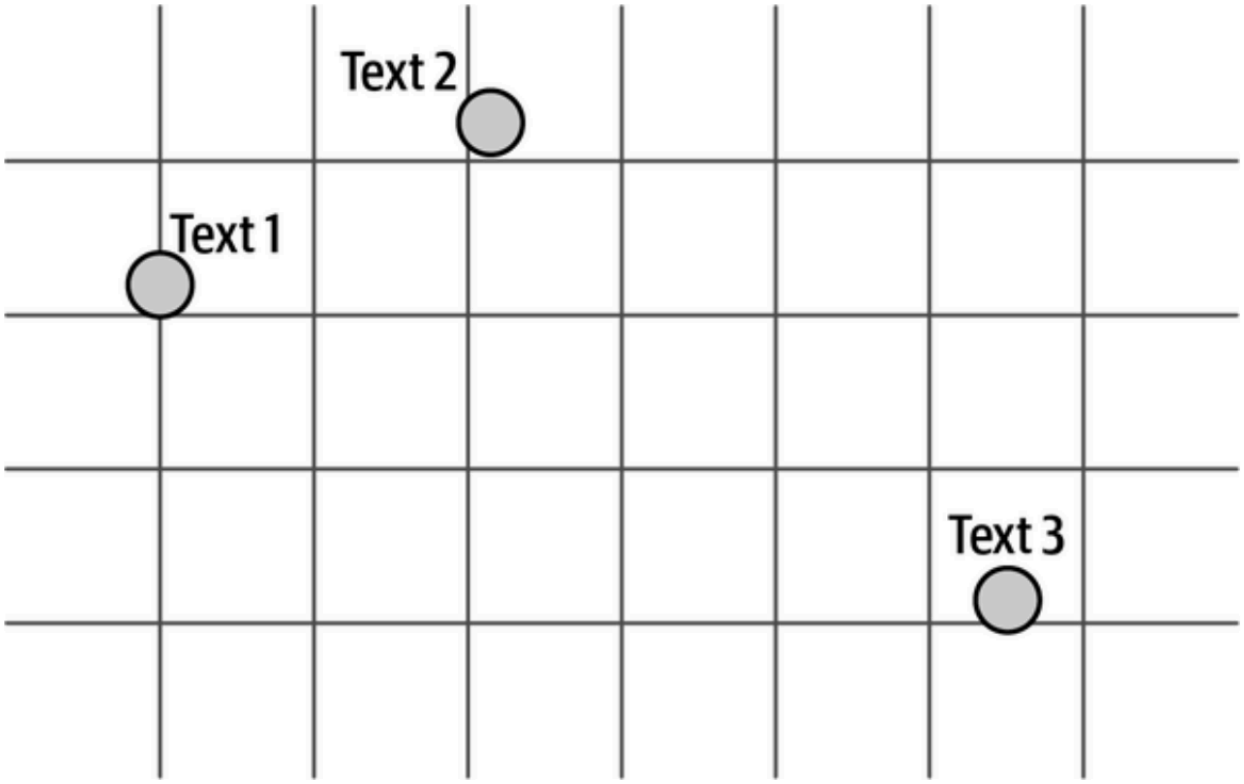

Figure 8-4. The intuition of embeddings: each text is a point and texts with similar meaning are close to each other.

Example, consider the illustration below:

- Should text 3 even be returned as a result? The answer depends on the nature of the system and the choice of you - the designer. Limits and thresholds are used to filter out *irrelevant* results.
- Are query and its best result always semantically similar (and are close to each other in the embedding space)? Not necessarily. This is why language models need to be *trained* on question-answer pairs to become better at retrieval.

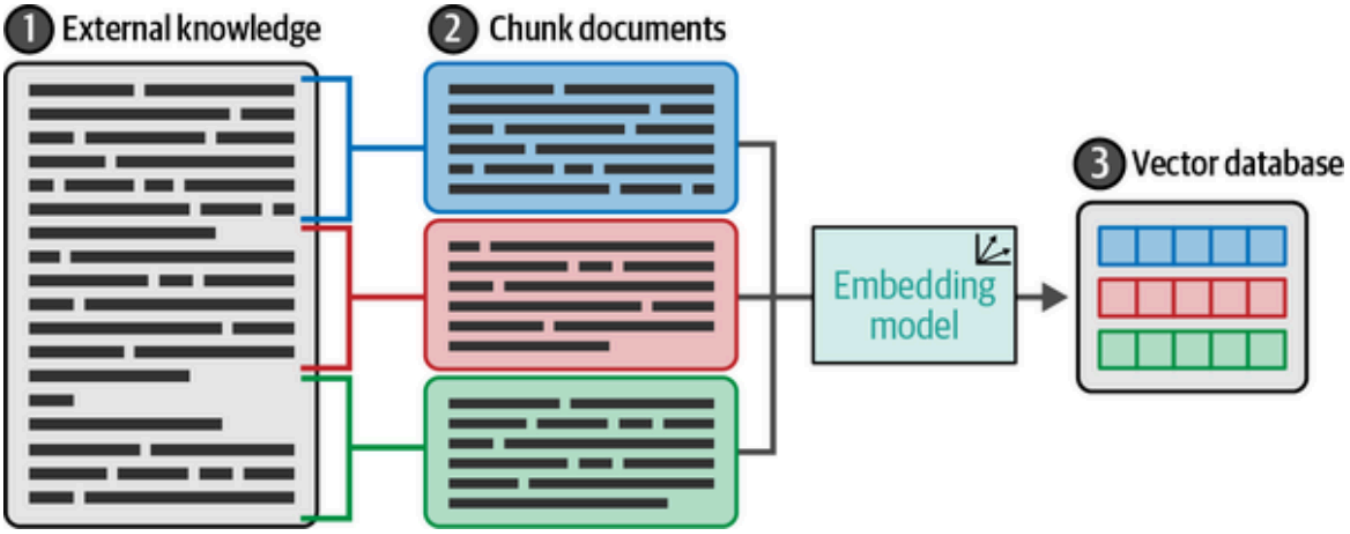

Figure 8-6. Convert some external knowledge base to a vector database. We can then query this vector database for information about the knowledge base.

Dense Retrieval Example

Here, we'll test and use embeddings to search for text.

Particularly, we will:

1. Get the text we want to make searchable and apply some processing to chunk it into sentences.
2. Embed the sentences.
3. Build the search index.
4. Search and see the results.

```
import cohere
import numpy as np
import pandas as pd
from tqdm import tqdm

# Paste your API key here. Remember not to share it publicly.
api_key = ''

# Create and retrieve a Cohere API key fromos.cohere.ai
co = cohere.Client(api_key)
```

Let's use the first section of the article on the film Interstellar.

Getting the text archive and chunking it

```
text = """
Interstellar is a 2014 epic science fiction film co-written,
directed, and produced by Christopher Nolan.
It stars Matthew McConaughey, Anne Hathaway, Jessica Chastain,
Bill Irwin, Ellen Burstyn, Matt Damon, and Michael Caine.
Set in a dystopian future where humanity is struggling to
survive, the film follows a group of astronauts who travel
through a wormhole near Saturn in search of a new home for
mankind.
Brothers Christopher and Jonathan Nolan wrote the screenplay,
which had its origins in a script Jonathan developed in 2007.
Caltech theoretical physicist and 2017 Nobel laureate in
Physics[4] Kip Thorne was an executive producer, acted as a
scientific consultant, and wrote a tie-in book, The Science of
Interstellar.
Cinematographer Hoyte van Hoytema shot it on 35 mm movie film in
the Panavision anamorphic format and IMAX 70 mm.
Principal photography began in late 2013 and took place in
Alberta, Iceland, and Los Angeles.
Interstellar uses extensive practical and miniature effects and
the company Double Negative created additional digital effects.
Interstellar premiered on October 26, 2014, in Los Angeles.
In the United States, it was first released on film stock,
expanding to venues using digital projectors.
The film had a worldwide gross over $677 million (and $773
million with subsequent re-releases), making it the tenth-highest
grossing film of 2014.
It received acclaim for its performances, direction, screenplay,
musical score, visual effects, ambition, themes, and emotional
weight.
It has also received praise from many astronomers for its
scientific accuracy and portrayal of theoretical astrophysics.
Since its premiere, Interstellar gained a cult following,[5] andnow is
regarded by many sci-fi experts as one of the best
science-fiction films of all time.
Interstellar was nominated for five awards at the 87th Academy
Awards, winning Best Visual Effects, and received numerous other
accolades"""


# Split into a list of sentences
texts = text.split('.')

# Clean up to remove empty spaces and new lines
texts = [t.strip(' \n') for t in texts]
```

Let's now embed the texts. We will use Cohere to get a vector embedding for each text.

Embedding the text chunks

```
# Get the embeddings
response = co.embed(
    texts=texts,
    input_type="search_document",
).embeddings

embeds = np.array(response)
print(embeds.shape)
```

This outputs `(15, 4096)`, which indicates that we have 15 vectors, each one of size 4,096.

Building the search index

Before we can search, we need to build a **search index**. An index stores the embeddings and is *optimized to quickly retrieve the **nearest neighbors***

```
import faiss
dim = embeds.shape[1]
index = faiss.IndexFlatL2(dim)
print(index.is_trained)
index.add(np.float32(embeds))
```

Search the index

After building the search index, we can now use it to search the dataset (Wikipedia article for Interstellar)

We embed the query and present its embedding to the index, which will retrieve the most similar sentence.

Below is the search function:

```
def search(query, number_of_results=3):

    # 1. Get the query's embedding
    query_embed = co.embed(texts=[query],
input_type="search_query",).embeddings[0]

    # 2. Retrieve the nearest neighbors
    distances , similar_item_ids = index.search(np.float32([query_embed]),
```

```
number_of_results)

    # 3. Format the results
    texts_np = np.array(texts) # Convert texts list to numpy for
    easier indexing
    results = pd.DataFrame(data={'texts': texts_np[similar_item_ids[0]],
'distance': distances[0]})

    # 4. Print and return the results
    print(f"Query:'{query}'\nNearest neighbors:")
    return results
```

We are now ready to write a query and search the texts:

```
query = "how precise was the science"
results = search(query)
results
```

This outputs:

```
Query: 'how precise was the science'
Nearest neighbors:
```

	texts	distance
0	It has also received praise from many astronomers for its scientific accuracy and portrayal of theoretical astrophysics	10757.379883
1	Caltech theoretical physicist and 2017 Nobel laureate in Physics[4] Kip Thorne was an executive producer, acted as a scientific consultant, and wrote a tie-in book, The Science of Interstellar	11566.131836
2	Interstellar uses extensive practical and miniature effects and the company Double Negative created additional digital effects	11922.833008

The **first** result has the **least distance**, and so is the **most similar** to the query.

Realize that because we used the word 'science', if we used *keyword matching*, there would not be any result.

However, since we're using *semantic search*, it returned the results we see (containing '**scientific**..', '**theoretical astrophysics**', etc.). We can see how useful semantic search can be.

Let's see how the **BM25** (one of the
leading lexical search methods) algorithm performs.

```
from rank_bm25 import BM25Okapi
from sklearn.feature_extraction import _stop_words
import string

def bm25_tokenizer(text):
    tokenized_doc = []
```

```
    for token in text.lower().split():
        token = token.strip(string.punctuation)
        if len(token) > 0 and token not in _stop_words.ENGLISH_STOP_WORDS:
            tokenized_doc.append(token)
    return tokenized_doc

tokenized_corpus = []
for passage in tqdm(texts):
    tokenized_corpus.append(bm25_tokenizer(passage))

bm25 = BM25Okapi(tokenized_corpus)

def keyword_search(query, top_k=3, num_candidates=15):
    print("Input question:", query)

    ##### BM25 search (lexical search) #####
    bm25_scores = bm25.get_scores(bm25_tokenizer(query))
    top_n = np.argpartition(bm25_scores, -num_candidates)[-num_candidates:]

    bm25_hits = [{'corpus_id': idx, 'score': bm25_scores[idx]} for idx in
top_n]

    bm25_hits = sorted(bm25_hits, key=lambda x: x['score'], reverse=True)

    print(f"Top-3 lexical search (BM25) hits")
    for hit in bm25_hits[0:top_k]:
        print("\t{:.3f}\t{}".format(hit['score'],
texts[hit['corpus_id']].replace("\n", " ")))
```

Now when we search for the same query, we get a different set of results from the dense retrieval search:

```
keyword_search(query = "how precise was the science")
```

Results:

```
Input question: how precise was the science
Top-3 lexical search (BM25) hits
    1.789 Interstellar is a 2014 epic science fiction film
co-written, directed, and produced by Christopher Nolan
    1.373 Caltech theoretical physicist and 2017 Nobel
laureate in Physics[4] Kip Thorne was an executive producer,
acted as a scientific consultant, and wrote a tie-in book, The
Science of Interstellar
```

```
    0.000 It stars Matthew McConaughey, Anne Hathaway,
Jessica Chastain, Bill Irwin, Ellen Burstyn, Matt Damon, and
Michael Caine
```

Note that the first result does not really answer the question despite it sharing the word “science” with the query.

Caveats of dense retrieval

Dense retrieval *always* returns the results, even when the documents don't contain the answer. It doesn't have a built-in no answer. An **example** is provided below:

```
Query: 'What is the mass of the moon?'
Nearest neighbors:
```

	texts	distance
0	The film had a worldwide gross over $677 million (and $773 million with subsequent re-releases), making it the tenth-highest grossing film of 2014	1.298275
1	It has also received praise from many astronomers for its scientific accuracy and portrayal of theoretical astrophysics	1.324389
2	Cinematographer Hoyte van Hoytema shot it on 35 mm movie film in the Panavision anamorphic format and IMAX 70 mm	1.328375

In cases like this, one possible heuristic is to set a **threshold level**.

A lot of search systems (e.g., search engines) present the user with the best info they can get and let the user decide if it's relevant or not. They also track information such as whether the

user clicked on a result, or was satisfied by it, to improve future versions of the search system.

Another caveat of dense retrieval is when a user wants to find an exact match for a specific phrase. **That's a case that's perfect for keyword matching.** That's also why a hybrid search system (that combines keyword matching and dense retrieval) is advised instead of relying solely on dense retrieval.

Dense retrieval systems also find it challenging to work properly other than the ones they are trained on

Summary:

- **No built-in relevance detection**
  - Always returns results even when documents don't contain the answer
  - All results get similarity scores regardless of actual relevance
  - Solution: Set distance thresholds, track user clicks to improve system
- **Poor at exact phrase matching**
  - Struggles when users need specific keyword/phrase matches
  - Solution: Use hybrid search (semantic + keyword search like BM25)
- **Domain mismatch problems**
  - Model trained on Wikipedia/internet data fails on specialized domains (legal, medical, etc.)
  - Solution: Include domain-specific data in training, or fine-tune for target domain
- **Text chunking challenges**
  - Hard to handle answers spanning multiple sentences
  - Design decision: How to chunk long texts optimally?
  - Trade-off between chunk size and information completeness

Key takeaway: Dense retrieval works best combined with other techniques (hybrid search, thresholding, domain adaptation) rather than used alone.

Chunking Long Texts

As we've seen in early chapters, every model has a context size limit. This is a limitation of **Transformer LMs**. This means we cannot input very long texts.

One way to solve this is by using *chunking*.

An illustration of chunking is provided below.

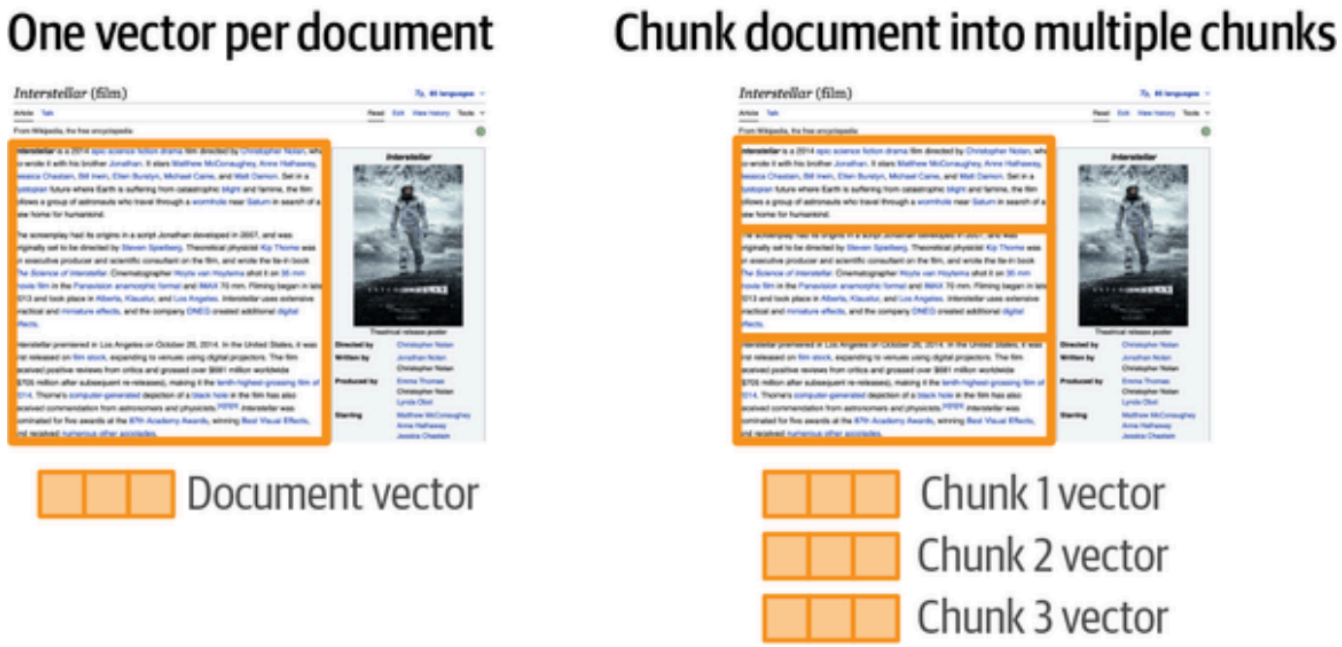

Figure 8-7. It's possible to create one vector representing an entire document, but it's better for longer documents to be split into smaller chunks that get their own embeddings.

One vector per document

When doing this, there are some possibilities:

1. Embedding only a **representative part of the text** (e.g., title, beginning of text). This means embedding only the **title, or beginning of the document**. This method may cause in a lot of loss of information.
2. Embedding the documents in **chunks**, **embedding those chunks**, and then **aggregating (averaging) those chunks into a single vector**. A downside is that it results in a highly compressed vector that loses a lot of the information of the document.

This approach can satisfy some information needs, but not others. This depends on whether the search is for a specific piece of information or a semantic concept.

Multiple vectors per document

This is basically **chunking the document into smaller pieces**, and **embedding those chunks, without aggregating them**.

Because it has full coverage of the text, it leads to a more expressive search index.

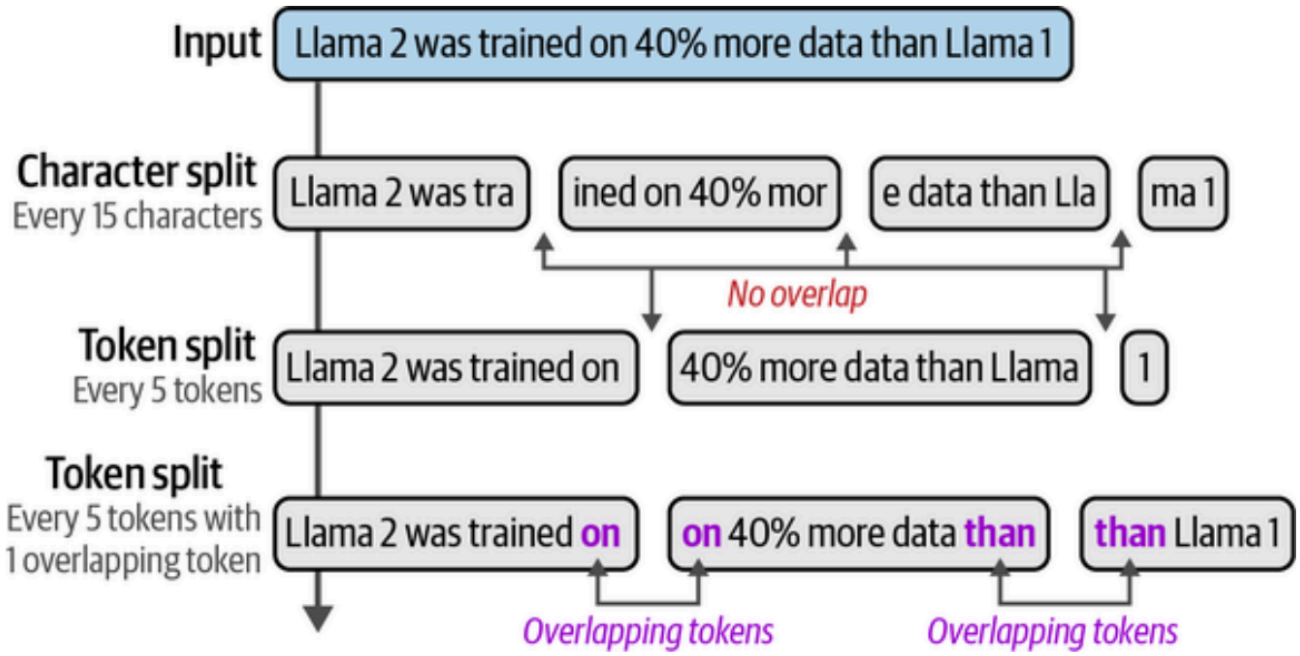

Figure 8-8. Several chunking methods and their effects on the input text. Overlapping chunks can be important to prevent the absence of context.

The best way of chunking a long text will depend on the types of texts and queries your system anticipates.

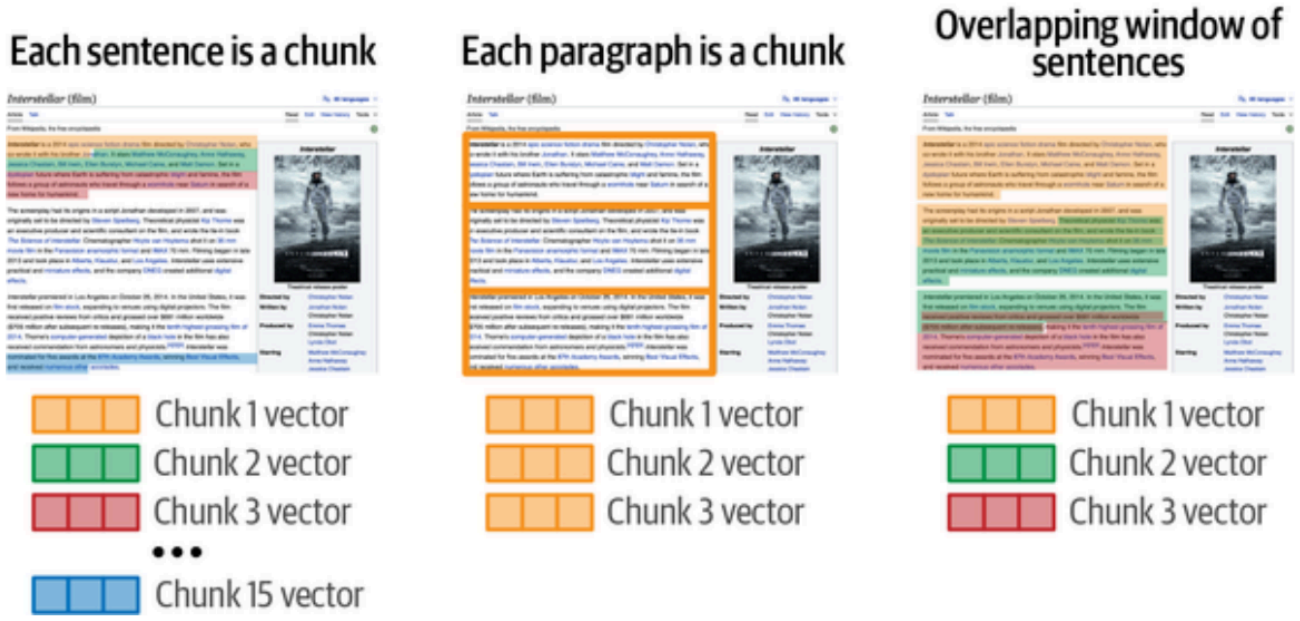

Figure 8-9. A number of possible options for chunking a document for embedding.

Was of chunking include:

1. **Each sentence is a chunk.**
   The issue is that it may not capture the overall context of a document.
2. **Each paragraph is a chunk.**
   Great if the text already is made of paragraphs (each with its own subcontext within the longer document). Otherwise, it would just be chunking of 3-8 sentences.

3. **Some chunks derive meaning from the text around them.**
   Context can be incorporated by:
   - adding the title of the document to the chunk
   - adding some overlap between chunks (words that chunks share in the beginning and the last part of each chunk).

A visual example of **chunking is provided below**:

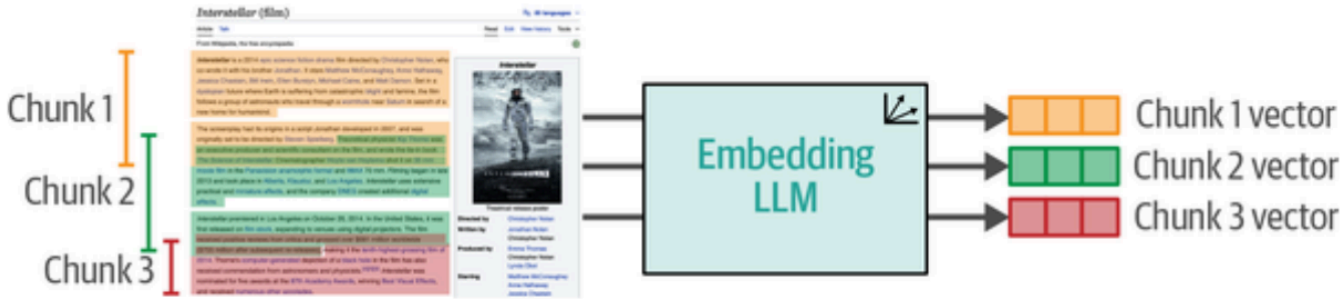

Figure 8-10. Chunking the text into overlapping segments is one strategy to retain more of the context around different segments.

Nearest neighbor search versus vector databases:

Once the query is embedded, we can start looking for results of a query.

We can do this by finding the nearest neighbors. This can be done by calculating the distances between the query and the archive.

An efficient approach is to use **nearest neighbor search libraries** like Annoy or FAISS. This allows you to retrieve results in milliseconds.

Another way to find the nearest neighbors is by using **vector databases** like Weaviate or Pinecone.

Vector databases allows deletion and modifications of the database without having to rebuild the entire database.

They also provide ways to filter your search or customize it in ways beyond merely vector distances.

Fine-tuning embedding models for dense retrieval

As we've seen before, we can fine-tune a language model to perform a specific task.

For embedding models, **they are fine-tuned to generate better embeddings for retrieval.** What this means is that **the model learns to pull relevant**

query-document pairs closer together in embedding space, while pushing irrelevant pairs farther apart.

To do this, the fine-tuning process requires **training data composed of queries paired with relevant documents (positive examples) and queries paired with irrelevant documents (negative examples).**

By training on this data, **the embedding model learns what "relevance" means for your specific task or domain.** The result is that the search system retrieves more relevant documents because the embeddings better capture semantic similarity in a way that's optimized for your use case.

Below are visual representations of when an embedding model is fine-tuned and when it is not fine-tuned.

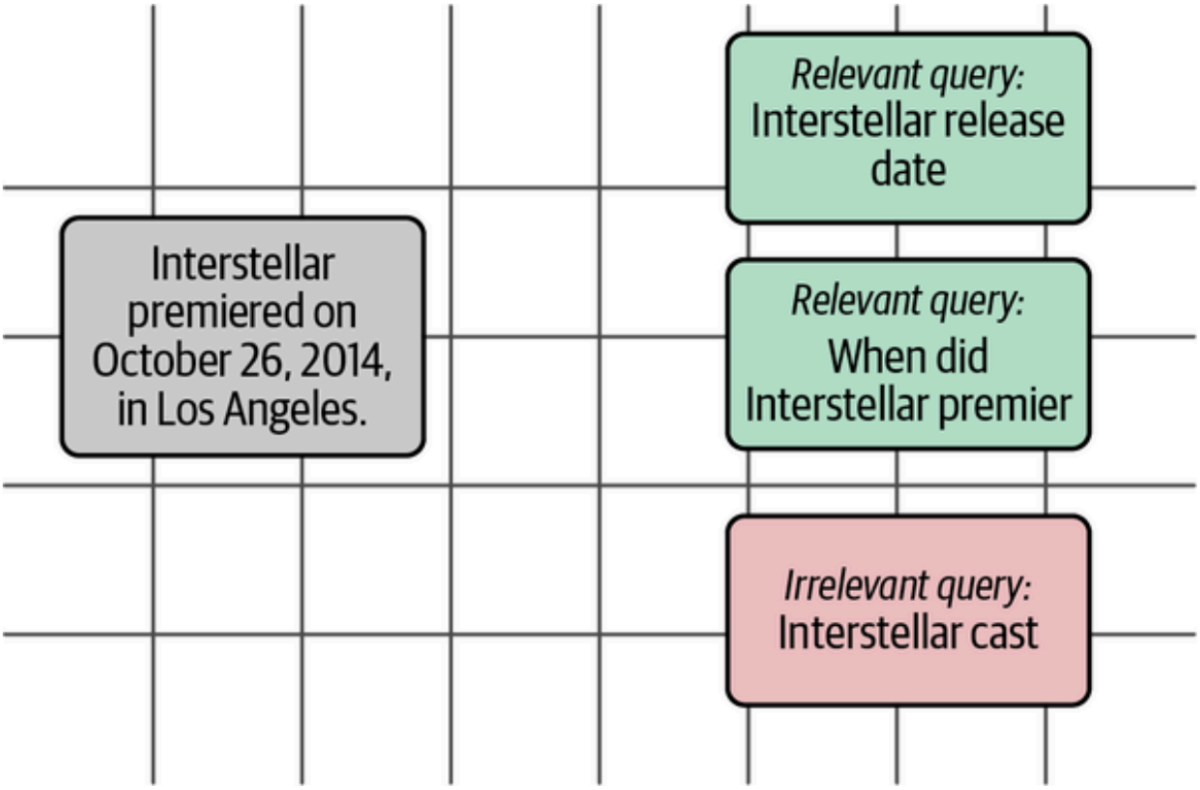

Figure 8-12. Before fine-tuning, the embeddings of both relevant and irrelevant queries may be close to a particular document.

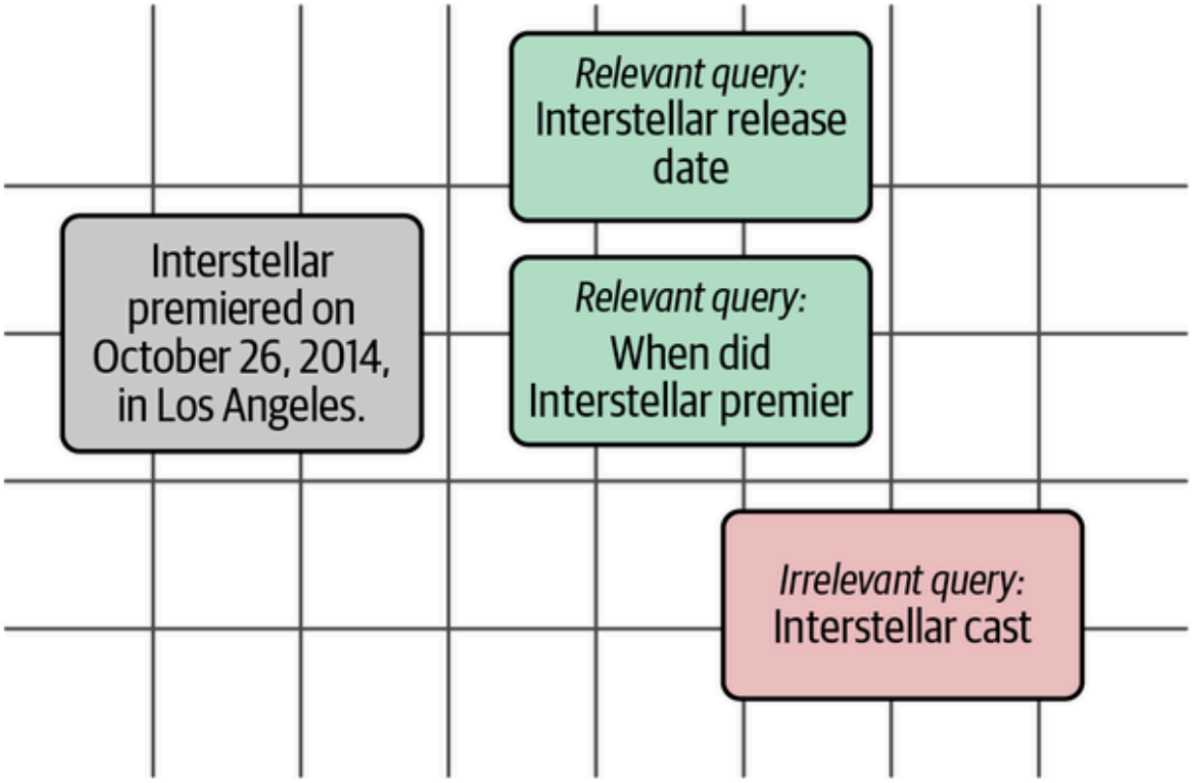

Figure 8-13. After the fine-tuning process, the text embedding model becomes better at this search task by incorporating how we define relevance on our dataset using the examples we provided of relevant and irrelevant documents.

Reranking

Adding reranking can significantly improve search results using BERT-like models.

Reranking is tasked with *changing the order of the search results based on relevance to the search query.*

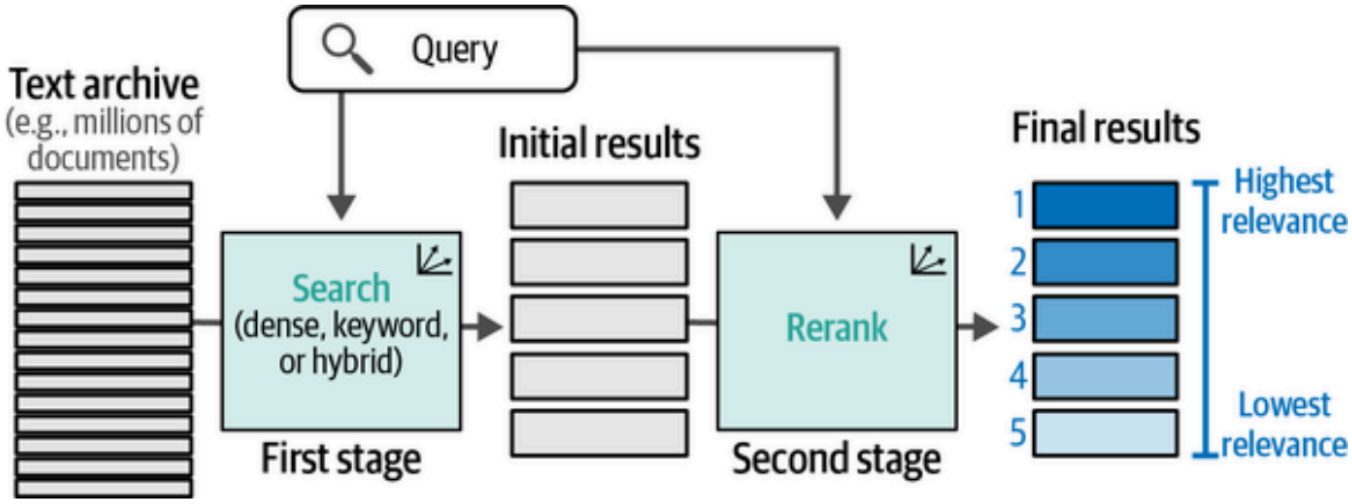

Figure 8-14. LLM rerankers operate as part of a search pipeline with the goal of reordering a number of shortlisted search results by relevance.

Reranking example

A **reranker** takes in the search query and a number of search results, and returns the optimal ordering of these documents so *the most relevant ones to the query are higher in ranking.*

In the following example, we will use Cohere's Rerank endpoint to start using a reranker.

```
query = "how precise was the science"

results = co.rerank(query=query, documents=text, top_n=3,
return_documents=True)

results.results
```

```
for idx, result in enumerate(results.results):
print(idx, result.relevance_score , result.document.text)
```

Output:

```
0 0.1698185 It has also received praise from many astronomers
for its scientific accuracy and portrayal of theoretical
astrophysics
1 0.07004896 The film had a worldwide gross over $677 million
(and $773 million with subsequent re-releases), making it the
tenth-highest grossing film of 2014
2 0.0043994132 Caltech theoretical physicist and 2017 Nobel
laureate in Physics[4] Kip Thorne was an executive producer,
acted as a scientific consultant, and wrote a tie-in book, The
Science of Interstellar
```

Here, you can see that the reranker shows higher confidence for the first result.

Let's change our keyword search function so that it retrieves a list of the top 10 results, then use rerank to choose the top 3 from the top 10.

```
def keyword_and_reranking_search(query, top_k=3,
num_candidates=10):
print("Input question:", query)

##### BM25 search (lexical search) #####
bm25_scores = bm25.get_scores(bm25_tokenizer(query))
top_n = np.argpartition(bm25_scores, -num_candidates)[-num_candidates:]
bm25_hits = [{'corpus_id': idx, 'score': bm25_scores[idx]} for idx in top_n]
bm25_hits = sorted(bm25_hits, key=lambda x: x['score'], reverse=True)
```

```
print(f"Top-3 lexical search (BM25) hits")for hit in bm25_hits[0:top_k]:
print("\t{:.3f}\t{}".format(hit['score'],
texts[hit['corpus_id']].replace("\n", " ")))

#Add re-ranking
docs = [texts[hit['corpus_id']] for hit in bm25_hits]
print(f"\nTop-3 hits by rank-API ({len(bm25_hits)} BM25 hits re-ranked)")
results = co.rerank(query=query, documents=docs, top_n=top_k,
return_documents=True)

# print(results.results)
for hit in results.results:

# print(hit)
print("\t{:.3f}\t{}".format(hit.relevance_score,
hit.document.text.replace("\n", " ")))
```

Let's send our query and see the top 10 results of the keyword search. We then pass these results to the reranker.

Results:

```
Input question: how precise was the science
Top-3 lexical search (BM25) hits
1.789 Interstellar is a 2014 epic science fiction film co-written, directed,
and produced by Christopher Nolan
1.373 Caltech theoretical physicist and 2017 Nobel laureate in
Physics[4] Kip Thorne was an executive producer, acted as a
scientific consultant, and wrote a tie-in book, The Science of
Interstellar
0.000 Interstellar uses extensive practical and miniature
effects and the company Double Negative created additional
digital effects

Top-3 hits by rank-API (10 BM25 hits re-ranked)
0.004 Caltech theoretical physicist and 2017 Nobel laureate in
Physics[4] Kip Thorne was an executive producer, acted as a
scientific consultant, and wrote a tie-in book, The Science of
Interstellar
0.004 Set in a dystopian future where humanity is struggling to
survive, the film follows a group of astronauts who travel
through a wormhole near Saturn in search of a new home for
mankind
0.003 Brothers Christopher and Jonathan Nolan wrote the
```

```
screenplay, which had its origins in a script Jonathan
developed in 2007
```

In this example, we see the top result of the keyword search are not included in the top 3 results of the reranker. This example shows us a glimpse of how using reranking can improve search quality.

How reranking models work

One popular way of building LLM search rerankers is to present the query and each result to an LLM working as a *cross-encoder.*

So the query and each result retrieved are presented to the LLM at the same time separately so the LLM can can view all the pairs before assigning a relevance score. **This score determines the new order of the results.**

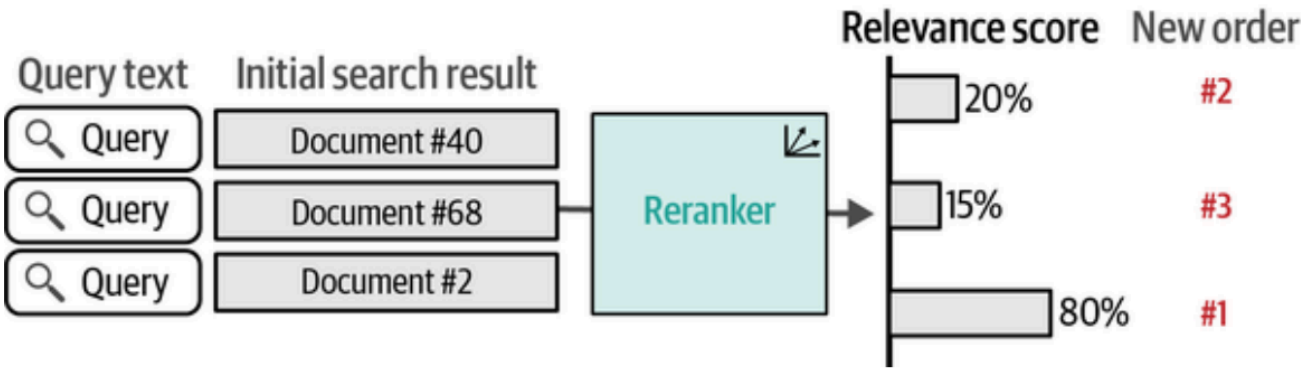

Figure 8-15. A reranker assigns a relevance score to each document by looking at the document and the query at the same time.

Retrieval Evaluation Metrics

Semantic search is evaluated using metrics from the **Information Retrieval (IR)** field.

MAP (Mean Average Precision)

Evaluating search systems needs **3 major components:**

- **a text archive**
- **a set of queries**
- **relevance judgments indicating which documents are relevant for each query.**

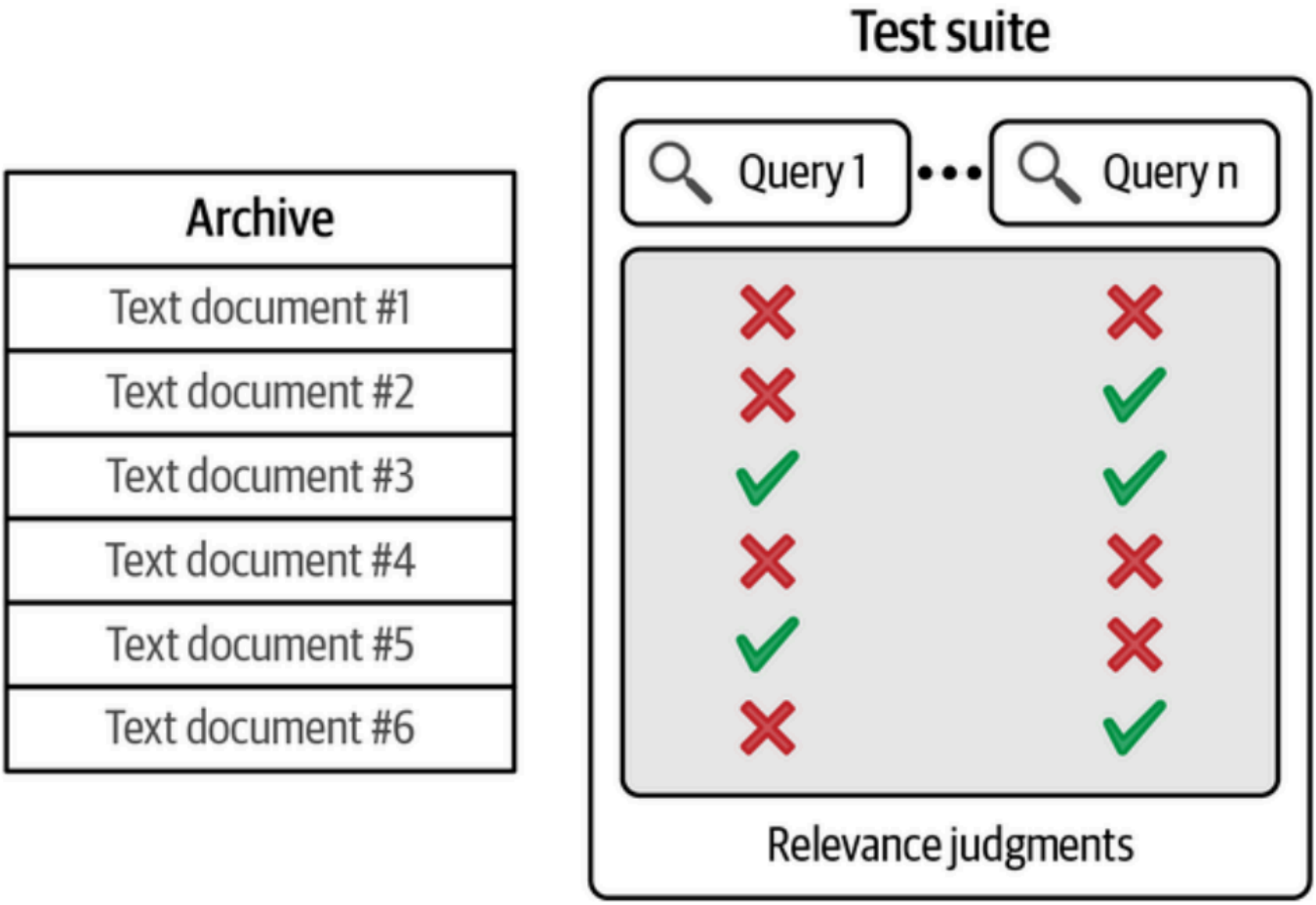

Figure 8-16. To evaluate search systems, we need a test suite including queries and relevance judgments indicating which documents in our archive are relevant for each query.

For each query, all text documents have relevance labels. These labels are either relevant/not relevant. The labels for each text document is called **relevance judgments**.

Relevance judgments serve as ground truth and are used to see the performance of a search system.

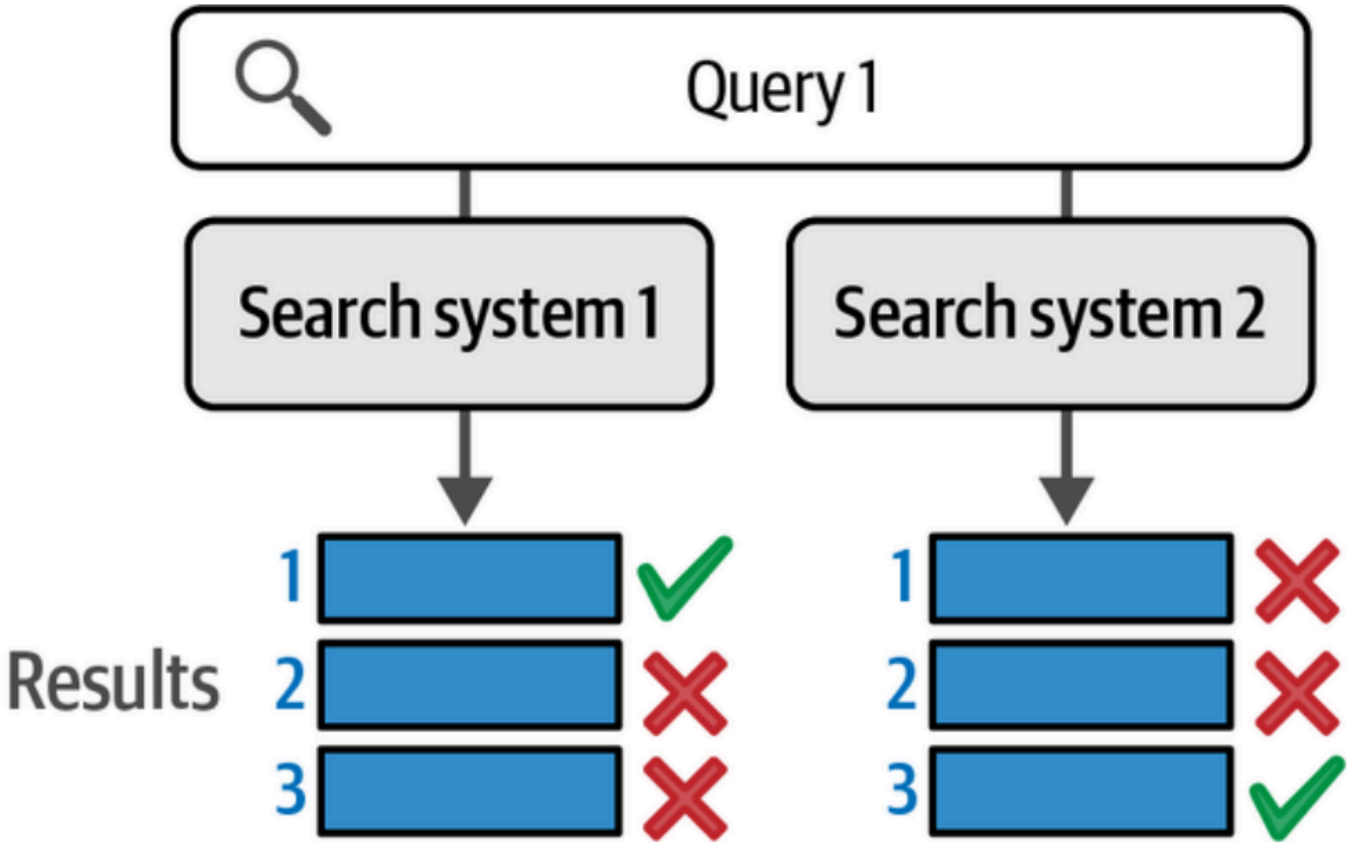

Figure 8-19. We need a scoring system that rewards system 1 for assigning a high position to a relevant result—even though both systems retrieved only one relevant result in their top three result

In this particular case in the illustration above, it is straightforward to know which search system performs better. But what if we need to quantify how much better a search system performs than the other one.

For this we can use **mean average precision (MAP)**.

Mean average precision is a measure that is able to quantify this distinction.

For example, it might score system 1's results as 1 and system 2's as 0.3. We'll look at how to calculate average precision for a single query, then how to combine scores across all queries in the test suite.

Scoring a single query with average precision

We can start by looking at a query that only has one relevant document in the test suite.

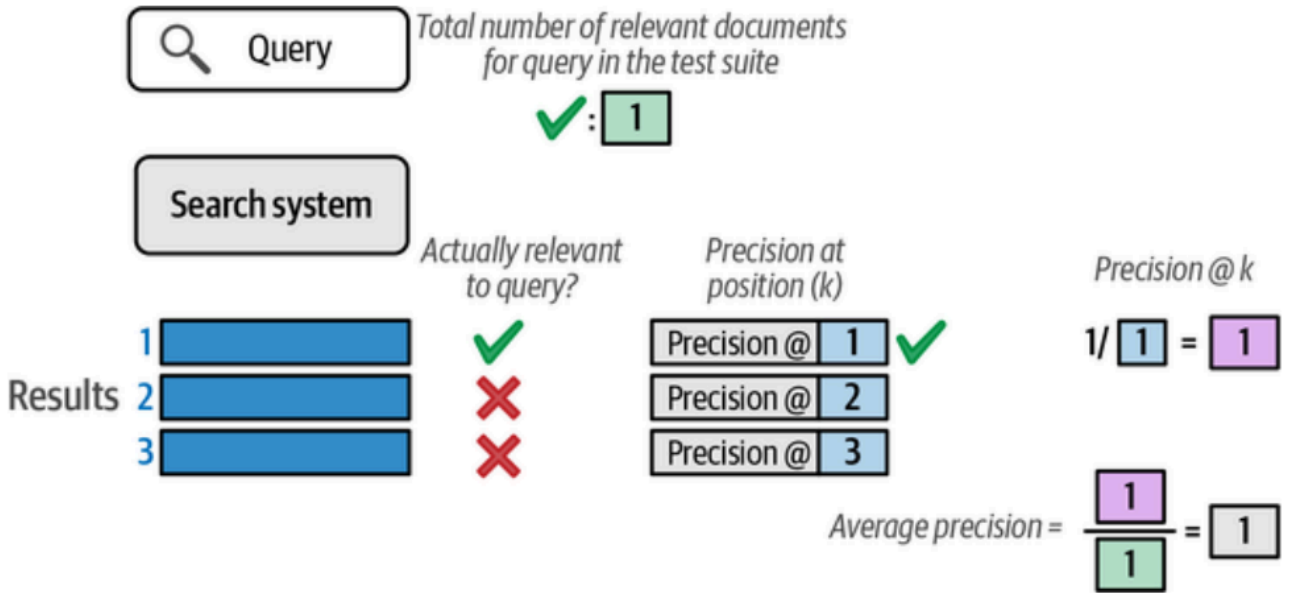

Figure 8-20. To calculate mean average precision, we start by calculating precision at each position, starting with position 1.

In this case, the search system placed the relevant result (the only available one for this query) at the top. This gives the system a perfect score of 1.

Looking at the calculation: we're at position 1, we have a relevant result, so precision at position 1 is 1.0 (it's calculated as the number of relevant results at position 1, divided by the position we're currently looking at, which is 1/1 = 1.0).

Since we're only scoring relevant documents, we can ignore the scores of nonrelevant documents and stop the calculation here.

What if the system placed the only relevant result at position 3? How would that affect the score?

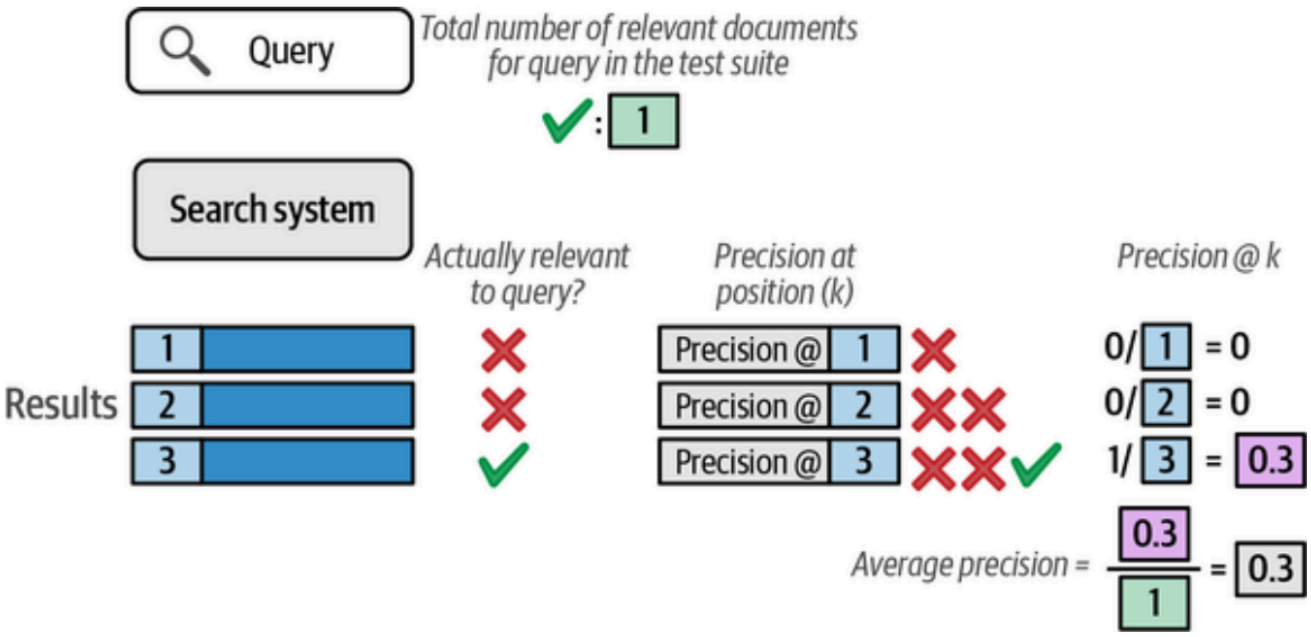

Figure 8-21. If the system places nonrelevant documents ahead of a relevant document, its precision score is penalized.

In this case, the system would get penalized. Looking at position 3, we have 1 relevant result out of 3 positions, so precision at position 3 = 1/3 = 0.3.

The average precision basically calculates ***how well the system places each result in order (ranks).*** This is why this metric is helpful in knowing whether a system works well and returns the best results in the right order.

Now let's look at a query with **more than one relevant document**.

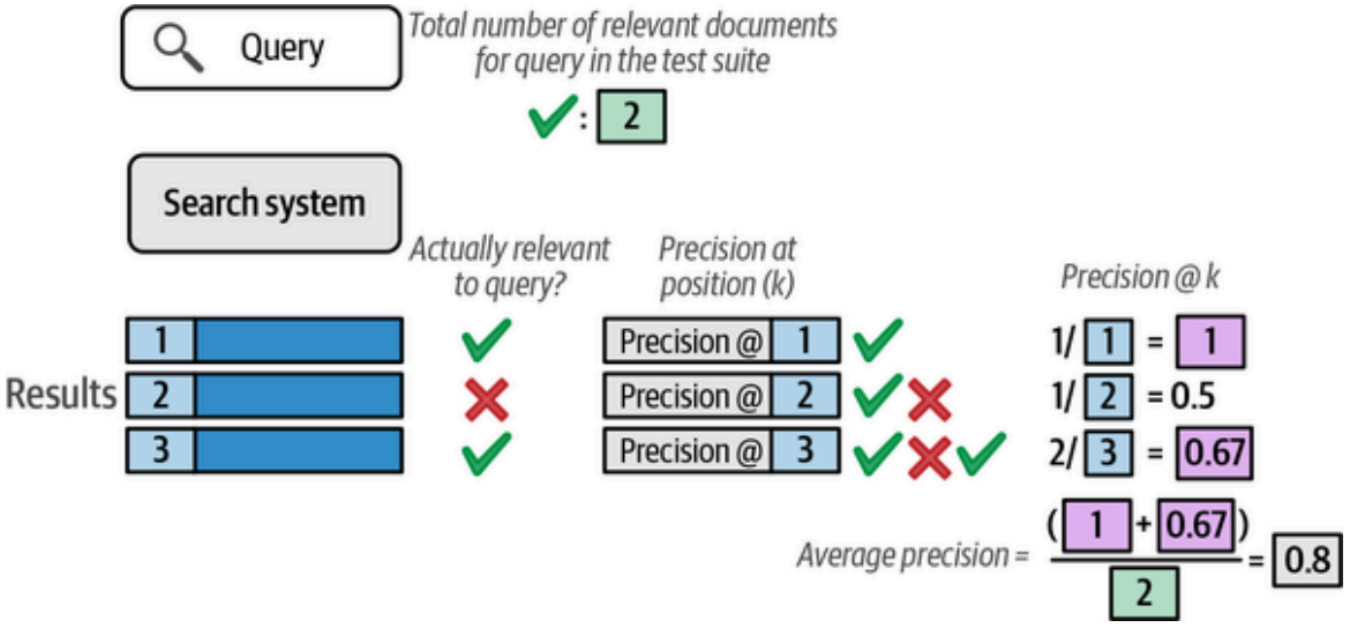

Figure 8-22. Average precision of a document with multiple relevant documents considers the precision at k results of all the relevant documents.

In this example, we have 2 relevant documents for the query. Let's say they appear at positions 1 and 3.

For position 1: precision@1 = 1/1 = 1.0
For position 3: precision@3 = 2/3 = 0.67 (because we've now found 2 relevant docs in 3 positions)

Average precision = (1.0 + 0.67) / 2 = 0.83

We're calculating the average of the precision values at each position where there is a relevant document.

Scoring across multiple queries with mean average precision

Now that we understand precision at k and average precision, we can use a metric that scores a search system against **all the queries in the test suite**.

As discussed earlier, the metric is called **mean average precision (MAP)**.

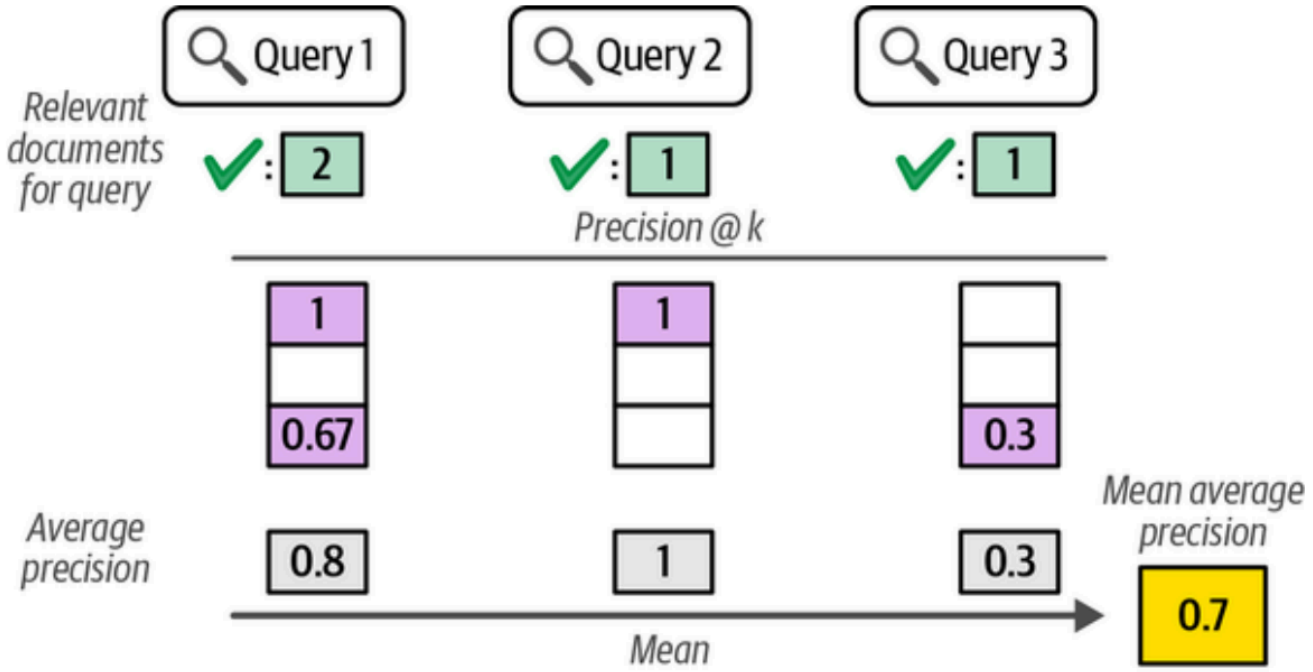

Figure 8-23. The mean average precision takes into consideration the average precision score of a system for every query in the test suite. By averaging them, it produces a single metric that we can use to compare a search system against another.

Now we have a single metric we can use to compare different search systems. If system A has **MAP** of 0.7 and system B has **MAP** of 0.5, we can confidently say system A performs better on our test suite.

Another commonly used metric for search systems is **normalized discounted cumulative gain (nDCG)**.

Unlike MAP that classifies between binary *relevant vs not relevant,* with **nDCG** can label documents as more relevant than another in the test suite and scoring.

Retrieval-Augmented Generation (RAG)

Due to the rapid increase in usage and application of LLMs, users have started using it to get factual answers.

It is true that these models have been trained on large amounts of data, and can answer many questions correctly. LLMs, however, can also answer questions confidently incorrectly.

To address this, one of the ways used is using RAG, or **Retrieval-Augmented Generation**.

RAG systems apply the search capabilities discussed previously to generate responses. This makes the responses be based on actual existing data, in addition to, or instead of just the training data.

RAG also enables users to access and '*chat with*' documents. This can be helpful for private or personal data the LLM wouldn't have access to otherwise.

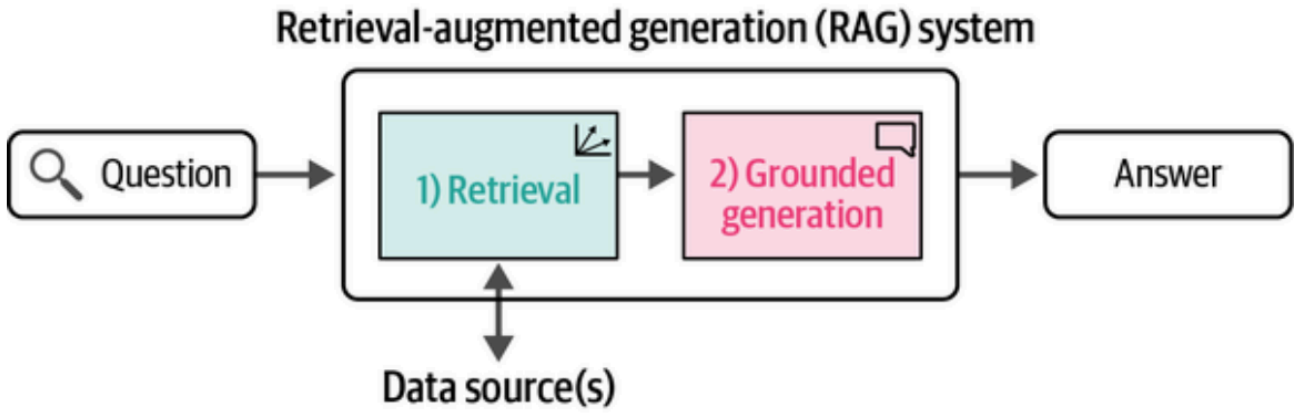

Figure 8-24. A basic RAG pipeline is made up of a search step followed by a grounded generation step where the LLM is prompted with the question and the information retrieved from the search step.

A basic RAG pipeline is made up of two steps:

1. **Retrieval step** - search for relevant information
2. **Grounded generation step** - the LLM is prompted with the question AND the information retrieved from the search step

This generation step is called **grounded generation** because the retrieved relevant information we provide the LLM establishes a certain context that grounds the LLM in the relevant domain.

From Search to RAG

The way to turn a search system to a **RAG System** is by adding an LLM to the end of the search pipeline. The query is sent to the LLM, and ask it to respond to the question given the retrieved information (search results).

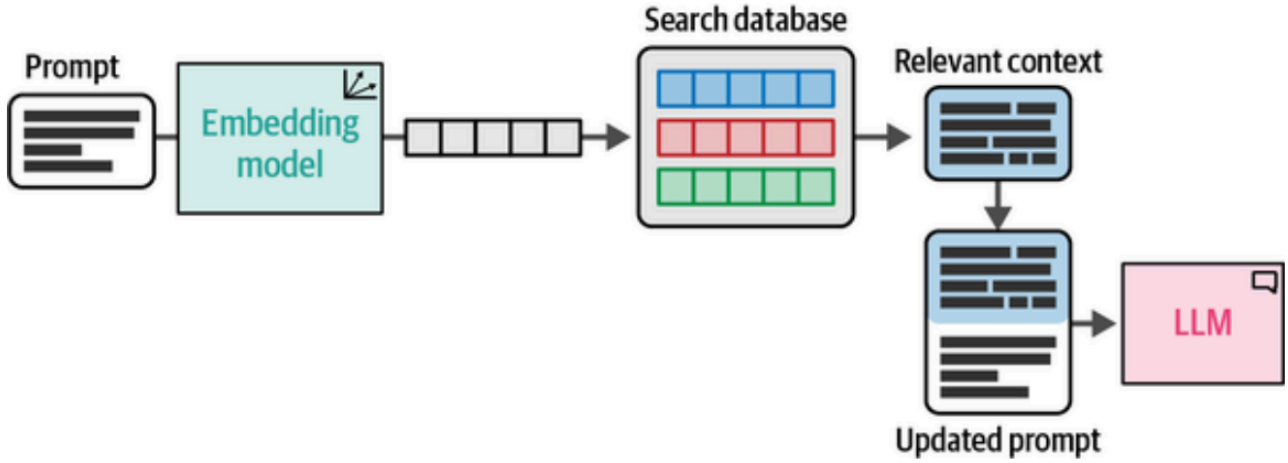

Figure 8-26. Find the most relevant information to an input prompt by comparing the similarities between embeddings. The most relevant information is added to the prompt before giving it to the LLM.

Example: Grounded Generation with an LLM API

Below is an example of one way to add a *grounded generation step* after search results to create a RAG system.

We'll use **embedding search** to retrieve the **top documents**, then we'll pass those to the `co.chat` endpoint along with the questions to provide a grounded answer:

```
query = "income generated"

# 1- Retrieval
# We'll use embedding search. But it's better if we do hybrid
results = search(query)

# 2- Grounded Generation
docs_dict = [{'text': text} for text in results['texts']]
response = co.chat(
    message = query,
    documents=docs_dict
)
print(response.text)
```

This outputs:

```
The film generated a worldwide gross of over $677 million, or $773 million
with subsequent re-releases.
```

This output has a second part:

```
citations=[ChatCitation(start=21, end=36, text='worldwide gross',
document_ids=['doc_0']), ChatCitation(start=40, end=57,text='over $677
million', document_ids=['doc_0']),
ChatCitation(start=62, end=103, text='$773 million with
subsequent re-releases.', document_ids=['doc_0'])]
documents=[{'id': 'doc_0', 'text': 'The film had a worldwide gross over $677
million (and $773 million with subsequent re-releases), making it the tenth-
highest grossing film of 2014'}]
```

As you can see, the start and end positions in `ChatCitation()` cited from a particular document using `document_ids`.

Example: RAG with Local Models

Let's now replicate this with local models.

Because of the much smaller size of the model, it will not be able to do as well as the previous one.

You'll need:

- A **generation model** (we'll use `Phi-3-mini-4k`)
- An **embedding model** (we'll use `BAAI/bge-small-en-v1.5`)
- A **vector database** (FAISS)
- A **RAG prompt template** with `{context}` and `{question}` variables

Just like in previous chapters, we start by downloading the model:

```
!wget https://huggingface.co/microsoft/Phi-3-mini-4k-instruct-
gguf/resolve/main/Phi-3-mini-4k-instruct-fp16.gguf
```

Here's the code:

```
from langchain import LlamaCpp
from langchain.embeddings.huggingface import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain import PromptTemplate
from langchain.chains import RetrievalQA
```

```
# Load the generation model
llm = LlamaCpp(
    model_path="Phi-3-mini-4k-instruct-fp16.gguf",
    n_gpu_layers=-1,
    max_tokens=500,
    n_ctx=2048,
    seed=42,
    verbose=False
)
```

Loading the embedding model

```
# Load the embedding model
embedding_model = HuggingFaceEmbeddings(
    model_name='thenlper/gte-small'
)
```

We can then use the embedding model to set up the **vector database**:

```
# Create a local vector database
db = FAISS.from_texts(texts, embedding_model)
```

The RAG Prompt

To create a prompt template, we would need to create an additional input variable `context` that can provide the LLM with the retrieved documents:

```
# Create a prompt template
# This is where we communicate the relevant documents to the LLM
template = """<|user|>
Relevant information:
{context}

Provide a concise answer the following question using the
relevant information provided above:
{question}<|end|>
<|assistant|>"""

prompt = PromptTemplate(
    template=template,
    input_variables=["context", "question"]
)
```

```
# RAG pipeline
rag = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type='stuff',
    retriever=db.as_retriever(),
    chain_type_kwargs={"prompt": prompt},
    verbose=True
)
```

Now you can call the model and ask it a question:

```
rag.invoke('Income generated')
```

Result:

```
The Income generated by the film in 2014 was over $677 million worldwide.
This made it the tenth-highest grossing film of that year. It should be
noted, however, this figure includes both initial ticket sales as well as
any subsequent re-releases. With these additional releases, total earnings
surged to approximately $773 million. The release format transitionedfrom
traditional film stock projection in theaters to digital projectors once it
was expanded to various venues in the United States. This shift might have
contributed to wider audience reach and potentially higher grossing figures
over time. However, specific data on how this affected total earnings isn't
provided in the information above.
```

As always, we can adjust the prompt to control the model's generation (e.g., answer length and tone).

Advanced RAG Techniques

There are several additional techniques to improve the performance of RAG systems.

Query rewriting

If the RAG system is a chatbot, **the RAG implementation we tried to build earlier would likely struggle** with the search step if a question is too verbose, or refers to context in previous messages in the conversation.

This is why it's a good idea to use an LLM to rewrite the query into one that's most optimal to get the most relevant results.

Below is an example of a **verbose**:

User Question: "We have an essay due tomorrow. We have to write about some animal. I love penguins. I could write about them. But I could also write about dolphins. Are they animals? Maybe. Let's do dolphins. Where do they live for example?"

This should actually be rewritten into a query like:

Query: "Where do dolphins live"

In this example, the *rewriter* omitted many parts of the original question that do not really help find the relevant document, and could cause incorrect information to be retrieved.

This rewriting can be done through a prompt (or through an API call). Cohere's API, for example, has a dedicated query-rewriting mode for `co.chat`.

Multi-query RAG

The next improvement is to extend the query rewriting to be able to search multiple queries if more than one is needed to answer a specific question.

For example:
User Question: "Compare the financial results of Nvidia in 2020 vs. 2023"

While there could be a single document that answers this query directly, **it is better and safer to use 2 separate queries for a question about two different years.**

Query 1: "Nvidia 2020 financial results"
Query 2: "Nvidia 2023 financial results"

We then present the top results of both queries to the model for grounded generation.

An additional small improvement here is to also give the query rewriter the option to determine if no search is required and if it can directly generate a confident answer without searching.

Multi-hop RAG

A more advanced question may require a series of sequential queries.

Take for example:

User Question: "Who are the largest car manufacturers in 2023? Do they each make EVs or not?"

To answer this, the system must first search for:
Step 1, Query 1: “largest car manufacturers 2023”

Then after it gets this information (the result being Toyota, Volkswagen, and Hyundai), it should ask follow-up questions:

Step 2, Query 1: “Toyota Motor Corporation electric vehicles”
Step 2, Query 2: “Volkswagen AG electric vehicles”
Step 2, Query 3: “Hyundai Motor Company electric vehicles”

Query routing

This technique is giving the model the ability to search multiple data sources.

We can, for example, specify for the model that if it gets a question about HR, it should search the company's HR information system (e.g., Notion) but if the question is about customer data, it should search the customer relationship management (CRM) (e.g., Salesforce).

Agentic RAG

Advanced RAG systems behave more like agents, capable of using multiple tools and making complex decisions.

RAG Evaluation

As a new technology, the ways to evaluate RAG are still being developed. For example, the paper "*Evaluating verifiability in generative search engines*" (2023) runs human evaluations on different generative systems.

It evaluates results **using four criteria:**

1. **Fluency**: Whether the generated text is fluent and cohesive.
2. **Perceived utility**: Whether the generated answer is helpful and informative.
3. **Citation recall**: The proportion of generated statements about the external world that are fully supported by their citations.
4. **Citation precision**": The proportion of generated citations that support their associated statements.

While human evaluation is always preferred, there are approaches that attempt to automate these evaluations by having a capable LLM act as a judge (called **LLM-as-a-judge**) and score the different generations along the different criteria.

Ragas is a software library that does exactly this. It also scores some additional useful metrics like:

1. **Faithfulness**: Whether the answer is consistent with the provided context
2. **Answer relevance**: How relevant the answer is to the question

The Ragas documentation site provides more details about the formulas to actually calculate these metrics.