Query/Answer Strategies Comparison

"The search for the perfect language is the story of a dream and of a series of failures."
— Umberto Eco, *The Search for the Perfect Language*

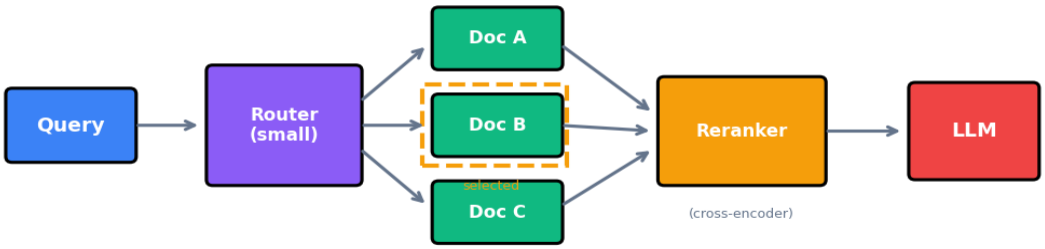
Historical note: Leibniz's *characteristica universalis* assigned prime numbers to primitive concepts — "animal" = 2, "rational" = 3 — so that composite concepts became products: "human" = 2 × 3 = 6. Reasoning reduced to divisibility tests. Modern embeddings pursue the same dream: mapping meaning to vectors where semantic relationships become geometric operations (cosine similarity). The difference is that Leibniz required *a priori* enumeration of all primitive ideas, while embeddings learn structure statistically from data.

OpenRouter API Token:

```
k-or-v1-c2b3fed13934605c6b75e9962fbef4b867bf2da694d1661f5d88229cda0da3c
```

Acronyms & Terms:
• **RAG** — Retrieval-Augmented Generation: combining retrieval with LLM generation
• **BM25** — Best Matching 25: classic keyword-based ranking algorithm (TF-IDF variant)
• **RAPTOR** — Recursive Abstractive Processing for Tree-Organized Retrieval
• **ColPali** — Vision-language model for visual document embeddings (based on PaliGemma)
• **LLM** — Large Language Model
• **VLM** — Vision-Language Model
• **Cross-encoder** — Reranker model that scores query-document pairs jointly

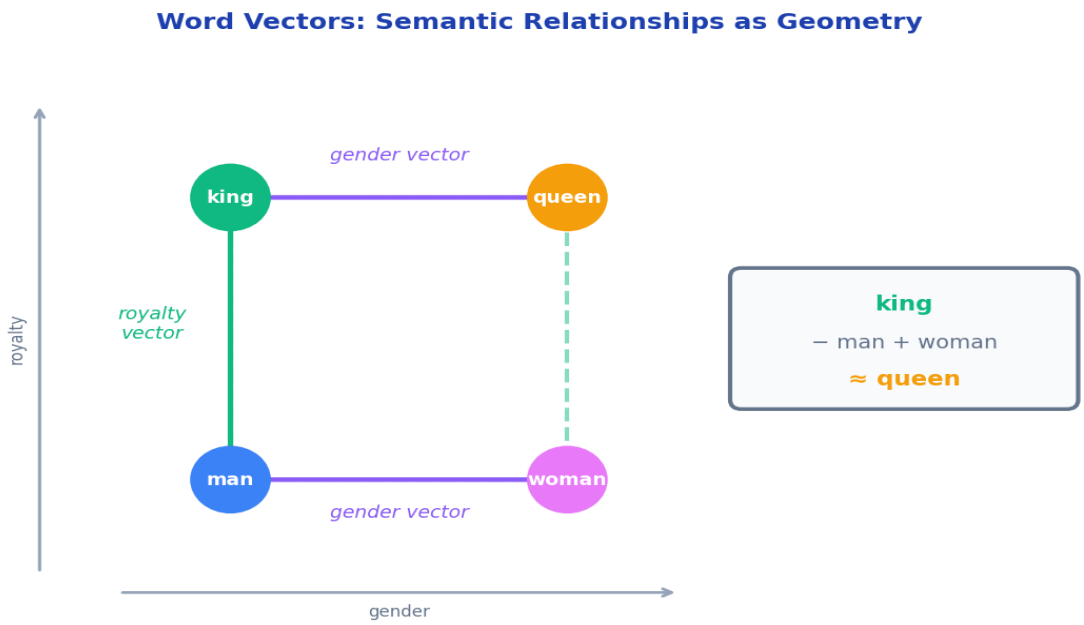
Strategy 1: Small Router Model



Routes query to most relevant document(s) using a lightweight classifier, then reranks before LLM.

Understanding Word Vectors

Word embeddings map words to vectors in high-dimensional space, where **semantic relationships become geometric relationships**. The classic example: the vector from "man" to "woman" captures the concept of gender — and this same vector connects "king" to "queen".

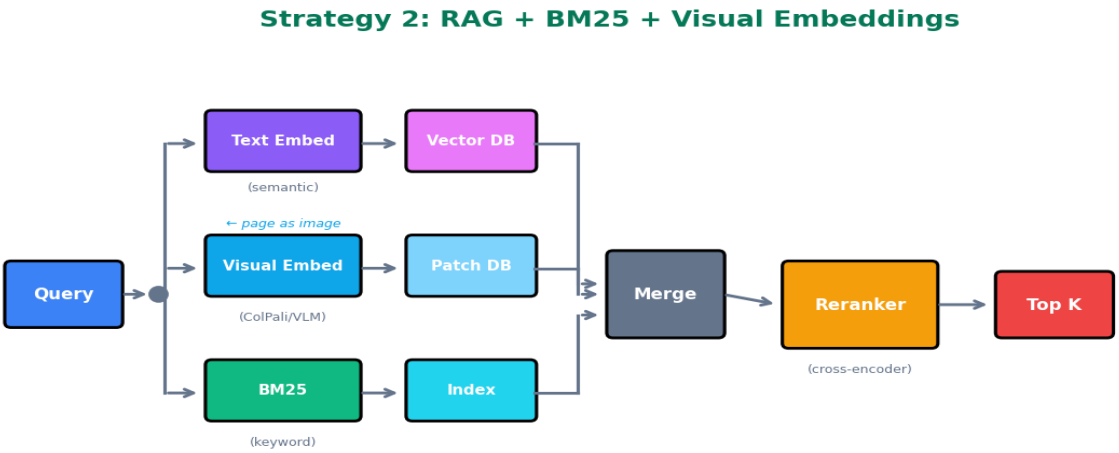


The famous equation: king – man + woman ≈ queen

This works because embeddings encode *relational structure*. The "royalty" dimension is preserved while the "gender" dimension shifts. This is why vector similarity (cosine distance) can capture meaning — semantically related words cluster together, and analogies become vector arithmetic.

From words to documents: RAG systems extend this idea. Instead of single words, we embed chunks or entire pages, then find the most similar vectors to a query. The same geometric intuition applies — relevant documents are "nearby" in embedding space.

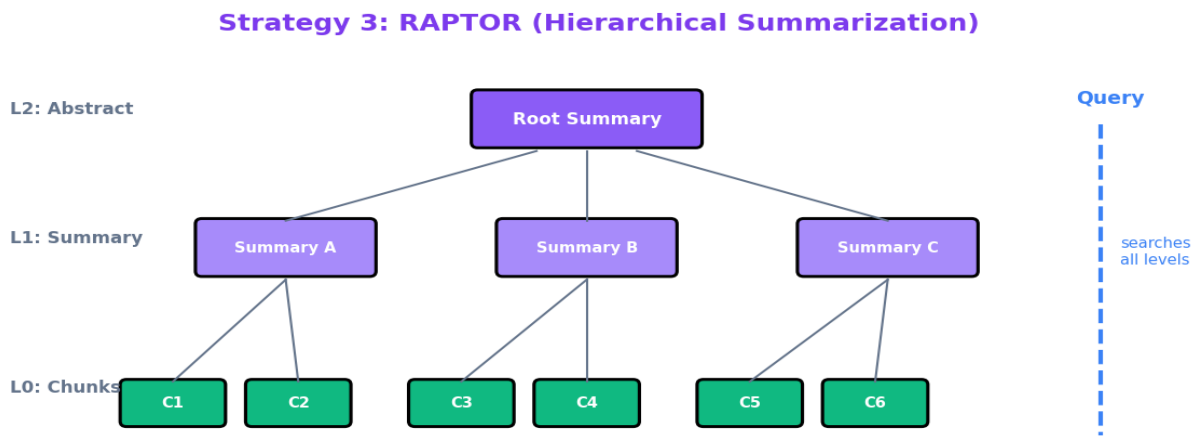
Strategy 2: RAG + BM25 + Visual Embeddings



Three parallel paths: text embeddings (semantic), visual embeddings (ColPali — page as image), and BM25 (keyword). Results merged and reranked.

Strategy 3: RAPTOR

"The defect of all symbolic systems lies in their inability to account for context." — Umberto Eco



Builds a hierarchical tree of summaries. Query searches all levels — chunks, summaries, and abstract root.

Strengths & Weaknesses Comparison

Strategy	Strengths	Weaknesses	Best For
1. Router (Small Model)	• Very fast latency • Low compute cost • Simple to implement • Easy to debug	• Loses within-doc granularity • May need retraining • Binary doc selection	Clearly separable document topics
2. RAG + BM25 (Hybrid)	• Semantic + keyword match • Handles diverse queries • No preprocessing • Easy to update index	• Chunking artifacts • Flat hierarchy • May miss context • Reranker adds latency	Mixed query types, general retrieval
3. RAPTOR (Hierarchical)	• Multi-hop reasoning • Handles abstraction • Searches all levels • Good for synthesis	• High preprocessing cost • Summary drift risk • Complex to maintain • Slow index updates	Conceptual queries, cross-doc synthesis
4. ColPali (Visual Embed)	• Preserves layout/tables • No OCR errors • Works on scanned docs • Captures visual context	• Higher compute cost • Larger index size • Less mature tooling • Query needs VLM	Complex layouts, forms, invoices, scanned PDFs

Code Examples

ChromaDB — Simple Vector Store

```
import chromadb

client = chromadb.Client()  # in-memory

collection = client.create_collection("docs")
collection.add(
    documents=["doc1", "doc2"],
    ids=["1", "2"]
)

results = collection.query(
    query_texts=["search term"],
    n_results=2
)
```

txtai — RAG with OpenRouter

```
import os
from txtai import Embeddings, LLM, RAG

os.environ["OPENROUTER_API_KEY"] = "your-key-here"

# LLM via OpenRouter (use litellm/ prefix)
llm = LLM("litellm/openrouter/meta-llama/llama-3.3-70b-instruct")

# Simple generation
response = llm("What is RAG?")
print(response)
```

RAPTOR — Hierarchical Retrieval with OpenRouter

```
import os
os.environ["OPENROUTER_API_KEY"] = "your-key"

from raptor import (
    BaseSummarizationModel, BaseQAModel, BaseEmbeddingModel,
    RetrievalAugmentationConfig, RetrievalAugmentation
)
from sentence_transformers import SentenceTransformer
from litellm import completion

# Embedding (local, free)
class LocalEmbedding(BaseEmbeddingModel):
    def __init__(self):
        self.model = SentenceTransformer("all-MiniLM-L6-v2")

    def create_embedding(self, text):
        return self.model.encode(text)

# Summarizer via OpenRouter
class OpenRouterSummarizer(BaseSummarizationModel):
    def summarize(self, context, max_tokens=150):
        resp = completion(
            model="openrouter/meta-llama/llama-3.3-70b-instruct",
            messages=[{"role": "user", "content": f"Summarize concisely:\n\n{context}"}],
            max_tokens=max_tokens
        )
        return resp.choices[0].message.content

# QA via OpenRouter
class OpenRouterQA(BaseQAModel):
    def answer_question(self, context, question):
        resp = completion(
            model="openrouter/meta-llama/llama-3.3-70b-instruct",
            messages=[{
                "role": "user",
                "content": f"Answer based on context only.\n\nContext: {context}\n\nQuestion: {question}"
```

```
            }]
        )
        return resp.choices[0].message.content

# Setup
config = RetrievalAugmentationConfig(
    summarization_model=OpenRouterSummarizer(),
    qa_model=OpenRouterQA(),
    embedding_model=LocalEmbedding()
)
RA = RetrievalAugmentation(config=config)

# Index document
text = open("your_doc.txt").read()
RA.add_documents(text)

# Query
answer = RA.answer_question("What are the key points?")
print(answer)
```