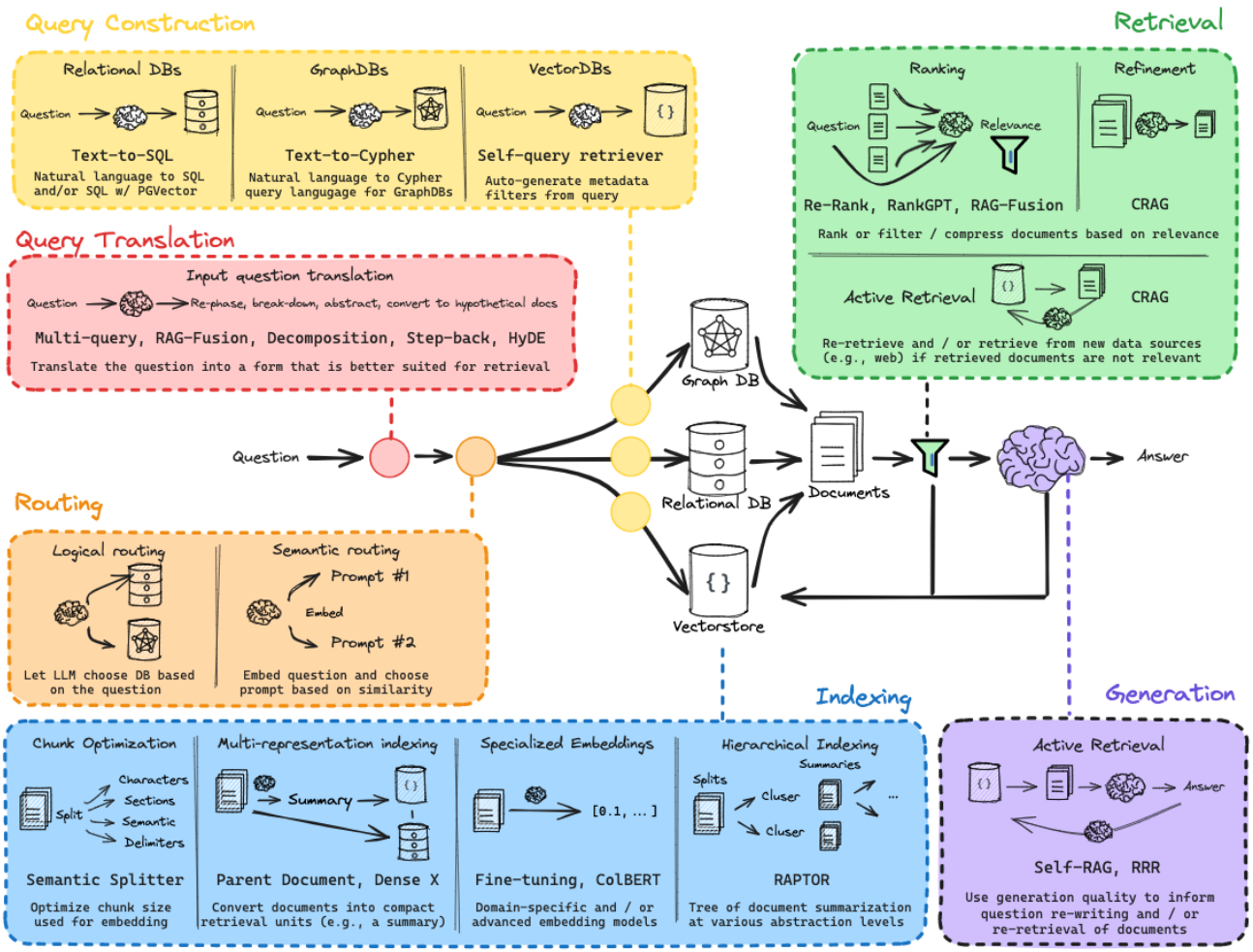
RAG for Scratch——langchain

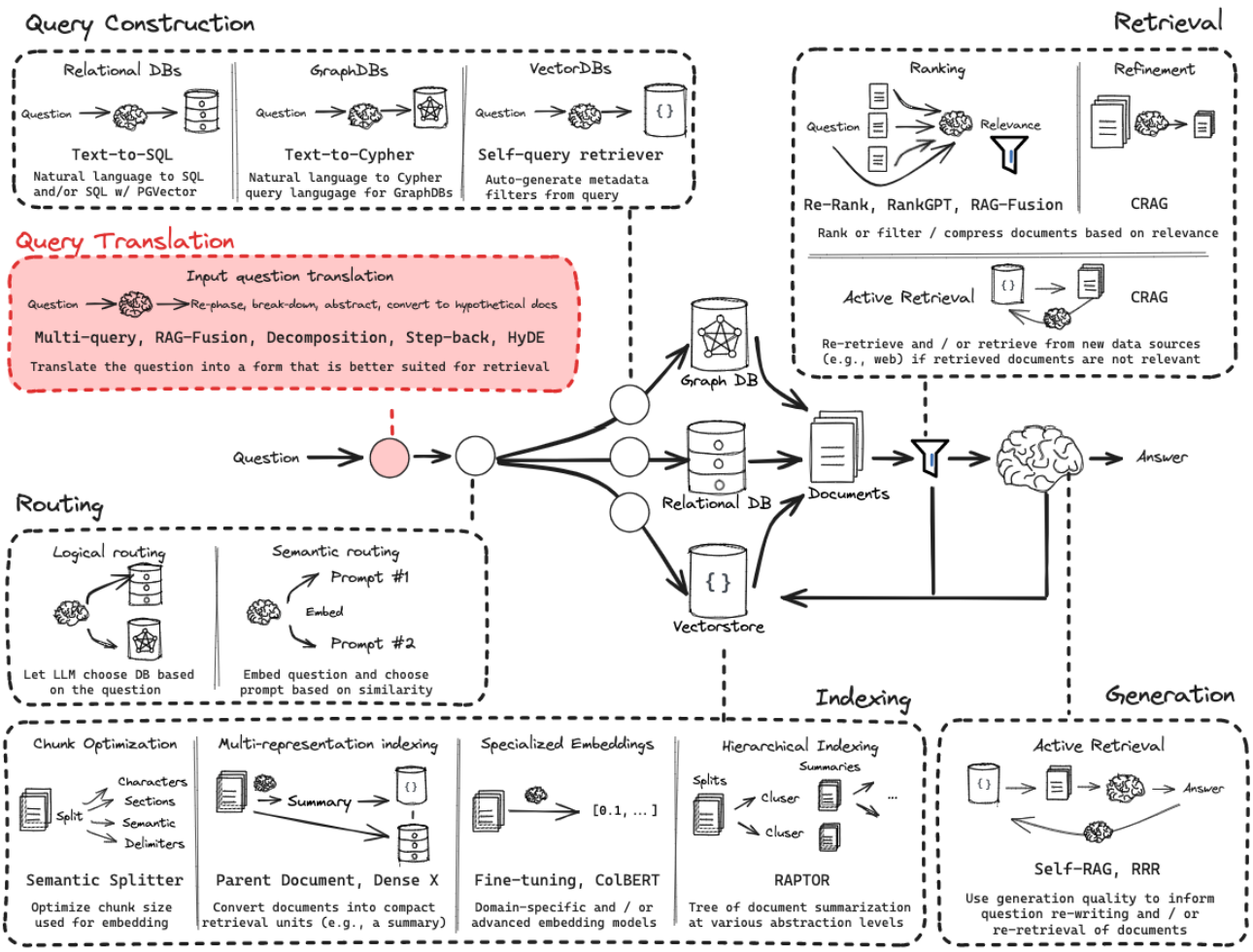
代码：[langchain-ai/rag-from-scratch (github.com)](https://github.com/langchain-ai/rag-from-scratch)

视频：[RAG From Scratch - YouTube](https://www.youtube.com/playlist?list=PLfaIDFEXuae2LXbO1_PKyVJiQ23ZztA0x)

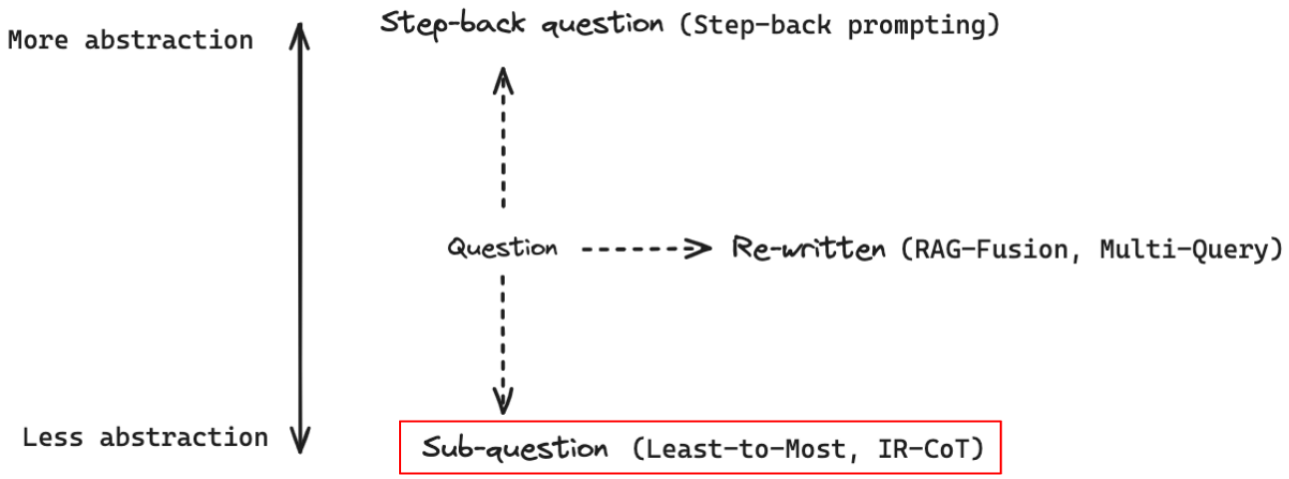


RAG方法通过将大型语言模型与外部数据源连接起来，帮助语言模型在特定任务上获得更高的相关性和准确性。上图是rag的pipeline。

query translation



query translation在rag pipeline的第一阶段，目的是把输入的query变成一种更易于检索的形式，因为用户的query可能会描述得比较模糊。



要达到这个目的有上图的几种方式：

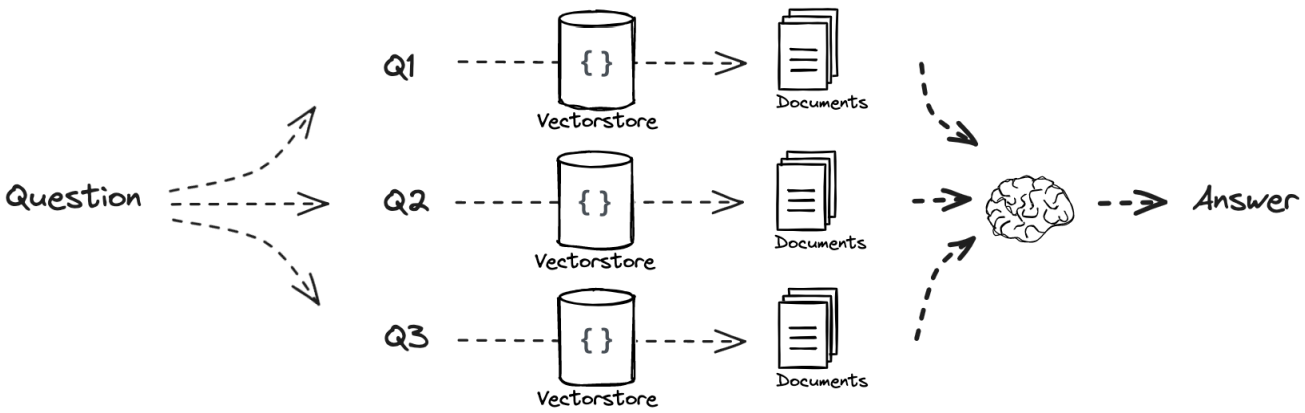
1、re-written：用多种方式、从多种角度重写query

2、sub-question：把query分解成多个query

3、step-back question：让query变得更加抽象、概括

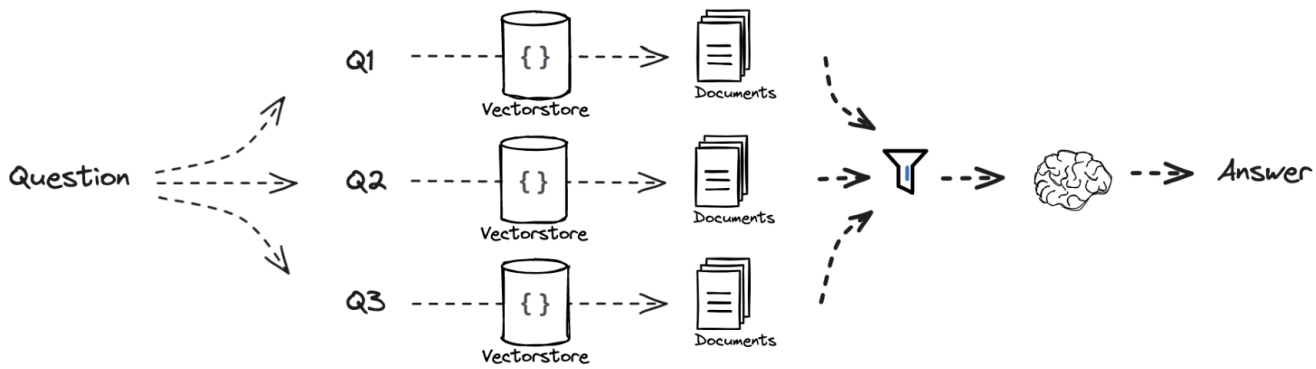
1 Multi Query Retriever

Multi Query Retriever是一种利用大型语言模型生成多个不同视角的查询，以自动化和改进基于距离的向量数据库检索的方法。通过合并多个查询的结果（求并集），它可以提供一个更全面、更丰富的文档集合，从而提高检索的效果和准确性。




```
def get_unique_union(documents: list[list]):
    """ Unique union of retrieved docs """
    flattened_docs = [dumps(doc) for sublist in documents for doc in sublist]
    # 获取唯一文档
    unique_docs = list(set(flattened_docs))
    return [loads(doc) for doc in unique_docs]
```

2 RAG-Fusion



与Multi Query Retriever唯一的区别是，对于检索到的Doc进行了互惠排名融合（Reciprocal Rank Fusion，RRF）算法，基本思想是通过对文档在各个查询结果中的排名进行加权融合，得到一个新的综合排名。

当你想要只取检索到的前x个doc时，这种方法会很方便。

```
def reciprocal_rank_fusion(results: list[list], k=60):
    """ RRF算法
        k: 参数，默认值为60。用于平滑排名得分，使得较高的排名对得分的影响更大，但不会过于极端。 """

    # 存储每个唯一文档的融合得分
    fused_scores = {}
```

```
    for docs in results:
        for rank, doc in enumerate(docs):
            doc_str = dumps(doc)
            if doc_str not in fused_scores:
                fused_scores[doc_str] = 0
            previous_score = fused_scores[doc_str]
            # 使用RRF公式更新文档的得分
            fused_scores[doc_str] += 1 / (rank + k)

    # 根据融合得分对文档进行降序排序:
    reranked_results = [
        (loads(doc), score)
        for doc, score in sorted(fused_scores.items(), key=lambda x: x[1], reverse=True)
    ]

    return reranked_results
```

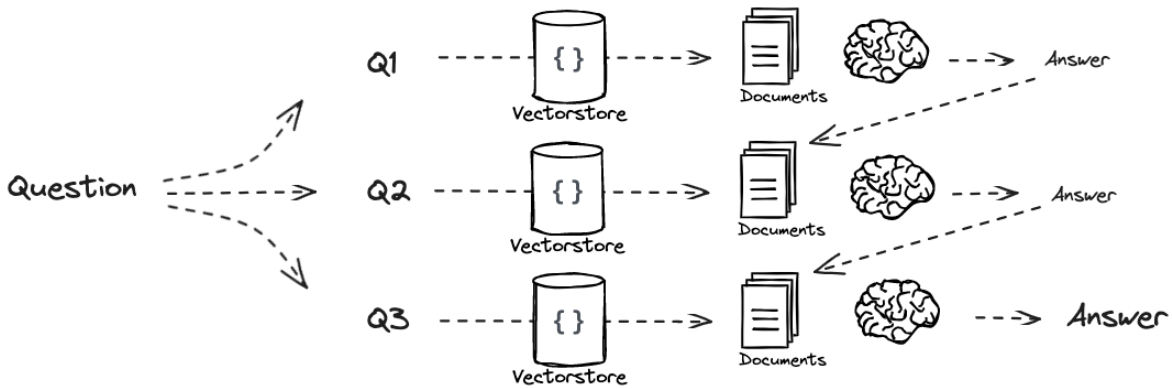
RRF公式用于计算每个文档的得分，公式如下：$\text{Score} = \frac{1}{\text{rank}+k}$

其中，`rank`是文档在某个查询结果列表中的排名（从0开始），`k`是平滑参数。

3 Decomposition

该策略将原始query拆解成多个子问题，主要有两种实现方式：

1. **顺序解决（Sequentially Solved）**：
    - 每个子问题的答案可能会影响后续子问题的提问和解答过程。类似于一个逐步推理的过程，通过前面的答案一步步缩小和明确接下来的问题。



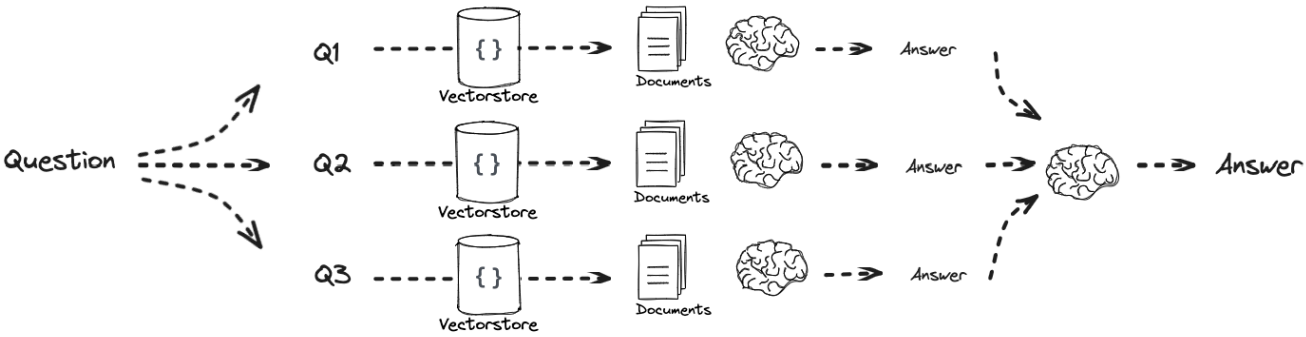
关键代码：

```
q_a_pairs = ""
for q in questions:

    rag_chain = (
    {"context": itemgetter("question") | retriever,
     "question": itemgetter("question"),
     "q_a_pairs": itemgetter("q_a_pairs")}
    | decomposition_prompt
    | llm
    | StrOutputParser())
```

```
    answer = rag_chain.invoke({"question":q,"q_a_pairs":q_a_pairs})
    q_a_pair = format_qa_pair(q,answer)
    q_a_pairs = q_a_pairs + "\n---\n"+  q_a_pair
    # 通过累积`q_a_pairs`，之前的问题和答案不断地加入到背景信息中，使得LLM在处理当前问题时能参考之前问答的内容
```

2. **独立回答及合并结果（Independently Answered Followed by Consolidation）**：
    - 每个子问题得到的答案不会影响其他子问题的解答。just将所有子问题的答案进行合并，从而得出一个全面的最终答案。




```
def format_qa_pairs(questions, answers):
    formatted_string = ""
    for i, (question, answer) in enumerate(zip(questions, answers), start=1):
        formatted_string += f"Question {i}: {question}\nAnswer {i}: {answer}\n\n"
    return formatted_string.strip()

context = format_qa_pairs(questions, answers)
```

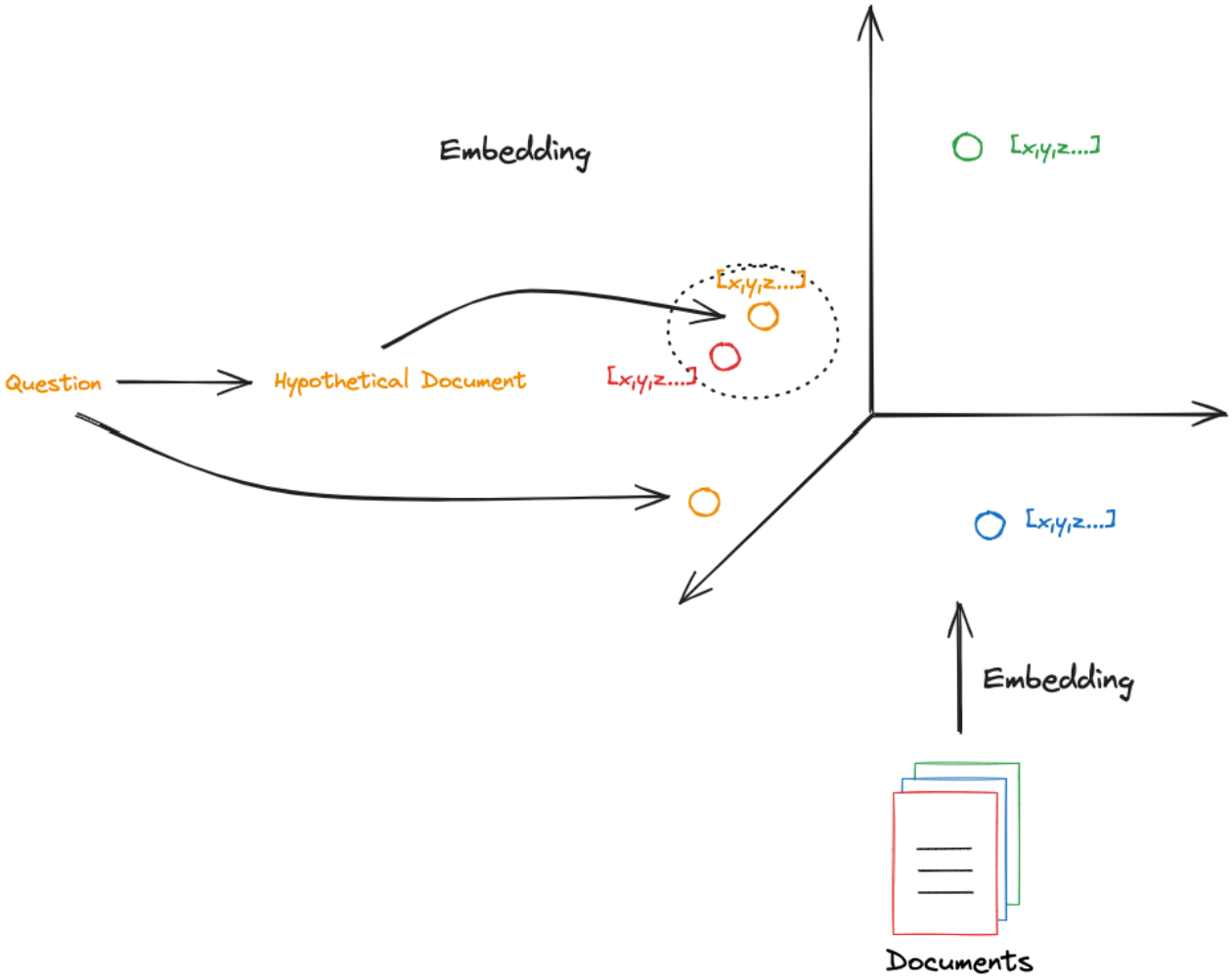
4 Step Back

“Step-back prompting”（反向推进提示），基于“chain-of-thought reasoning”（思维链推理）来进行。从一个具体的问题出发，通过给一些few-shot的方式，生成一个更高层次、更抽象的问题，以便于检索到相关文档。这在需要额外背景信息或基本概念理解的情况下尤其有用。

例子 “Why did the stock price of Company XYZ drop in Q3 2023?”（2023年第三季度XYZ公司股价为何下跌？）

通过 Step-back prompting 方法，可以生成一个更高层次的问题：**“What are the common reasons for a company's stock price to drop?”**（公司股价下跌的一般原因是什么？）

5 HyDE



考虑到其他的方法都是通过计算query和doc的相似度来检索文档的，但实际上，query和doc有很多不同之处。HyDE（Hypothetical Document Embeddings）根据用户输入的query生成一些假设的doc，将这些doc转换为向量形式，利用这些向量从一个索引（database/index）中检索相关的doc。

例子

用户提问：“What are the health benefits of green tea?”（绿茶的健康益处是什么？）

HyDE 方法就会生成一个或多个假设文档，例如：

- 一篇关于绿茶中含有的抗氧化剂及其对健康好处的文档。
- 一篇讨论绿茶对心血管健康的影响的文档。