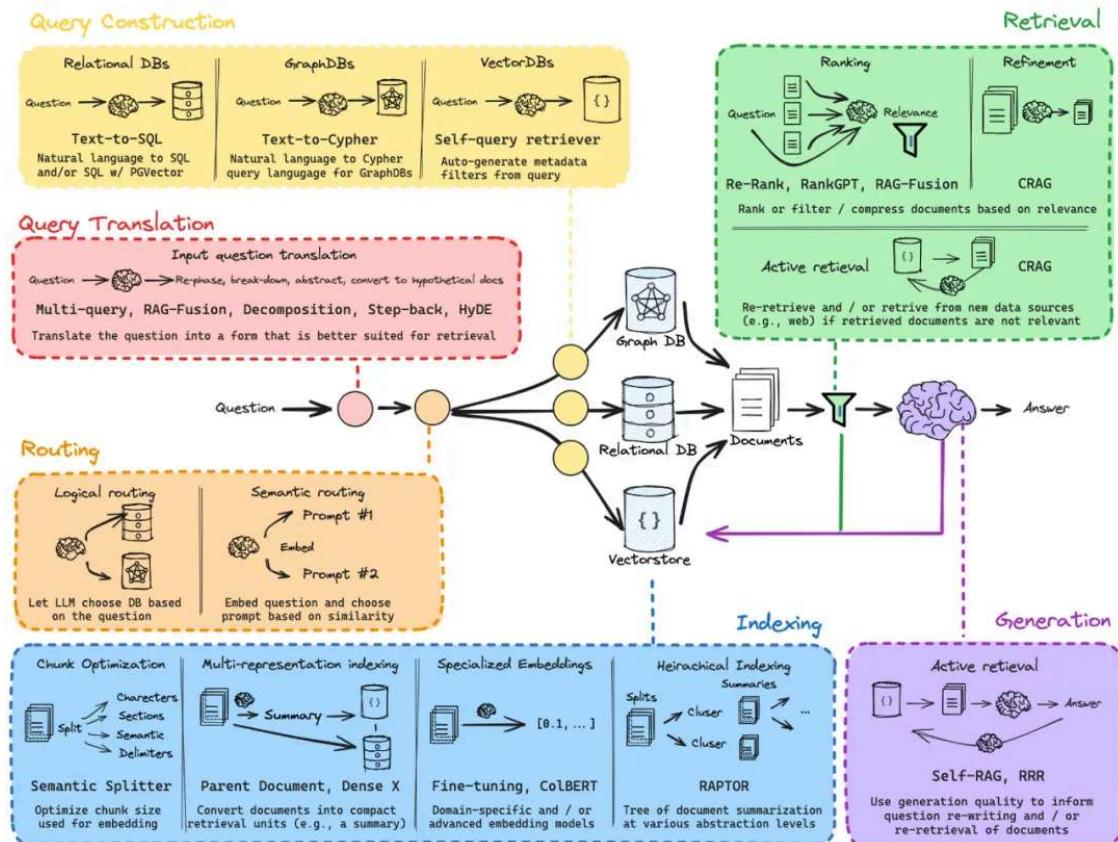


探索提升RAG系统问答质量的技术路线

原创 南七无名式 PyTorch研习社 2025年02月02日 11:00 安徽



Query Translation

Query Translation 将用户的自然语言查询转换为更适合检索和生成的形式。在这个过程中，系统将原始查询转化成一种或多种可以提升信息检索效果的形式，确保系统能够更有效地从不同的数据源中提取相关信息。

对于具有挑战性的检索任务，用户问题的措辞可能不太恰当。Query Translation 是指将用户的原始问题重新表达，使其更适合检索过程，提高检索的相关性和准确性。

在 RAG 系统中，直接使用用户的原始问题进行检索可能会遇到以下问题：

- **查询过于模糊或复杂**，数据库难以直接匹配相关内容。
- **信息缺失**，例如用户只输入“DeepSeek-R1 的优势？”而没有明确上下文。
- **不同数据库的适配性**，原始查询可能需要调整才能适应不同的数据源（如结构化数据、文档、向量数据库等）。

Query Translation 包括以下几个技术或方法：

Multi-query：通过生成多个查询，系统可以扩大检索范围并提高准确性。例如，针对同一个问题生成几个不同的查询，增加从多个角度检索到相关信息的可能性，提高召回率。

- 原始查询：“如何提高 DeepSeek-R1 的推理能力？”
- 生成的多个查询：
 - a. “DeepSeek-R1 的推理能力受哪些因素影响？”

- b. “如何优化提示 (prompt) 来增强 DeepSeek-R1 的推理能力? ”
- c. “有哪些方法可以提升 DeepSeek-R1 在数学推理任务上的表现? ”

RAG-Fusion: 指将多个查询的结果进行融合，选择最相关的信息。通过将多个查询结果合并，可以得到更加丰富和精确的答案。

Multi-query 主要是生成多个查询，而 *RAG-Fusion* 侧重于如何融合这些查询的检索结果（例如 *Reciprocal Rank Fusion*）。

Decomposition: 将复杂问题拆解成多个简单问题，然后分别检索每个子问题的答案。这样可以避免处理复杂问题时可能遇到的信息检索困难。

- 原始查询：“比较 Transformer 和 RNN 在文本摘要任务上的优缺点。”
- 分解后的查询：
 1. “Transformer 在文本摘要任务上的优点是什么？”
 2. “RNN 在文本摘要任务上的优点是什么？”
 3. “Transformer 和 RNN 在文本摘要任务上的对比研究有哪些？”

Step-back: 如果当前查询无法获得足够相关的文档，系统可以先查询更高层次的信息，再细化检索。适用于探索性查询和不确定性较高的问题。

- 用户问题：“李白的诗风如何？”
- 可能的 Step-back 查询：
 1. “李白的代表作品有哪些？”
 2. “李白的诗风如何在《将进酒》中体现？”
 3. “李白的诗风受哪些文学流派影响？”

HyDE: Hypothetical Document Embeddings，这是一种更为复杂的技术，根据用户查询生成假设性的文档形式，将其嵌入，并在检索中使用它们，从而提高检索效果。通过这种转换，系统可以更好地理解用户意图并找到最相关的信息。

简而言之，Query Translation 通过一系列技术方法将原始问题转化成更易于检索的查询形式，优化检索过程并提升最终的答案质量。

Query Translation 的优势

- ✓ 提高召回率：通过 Multi-query、RAG-Fusion 等方法确保检索更全面。
- ✓ 提升查询精度：Step-back 和 HyDE 方法优化查询表达，提高相关性。
- ✓ 增强复杂查询的可操作性：查询分解可以让复杂问题更容易匹配数据库内容。

QA：Multi-query 和 RAG-Fusion 都首先根据用户的原始查询生成多个相关的查询，那它们有什么区别呢？

Multi-query 和 *RAG-Fusion* 都是处理和扩展用户查询的技术，尽管它们看起来有些相似，但在具体实现和目标上有所不同。

Multi-query

Multi-query 的核心思想是根据用户的一个查询生成多个不同的查询。这些查询可以从不同角度、不同维度或者用不同的表达方式来探讨同一个问题。这样做的目的是扩展检索范围，增加找到相关信息的机会。

- 目标：扩大检索的覆盖面，确保系统在多个可能的查询方式下都能获取到相关信息。
- 实现：系统会生成多个查询并分别发送到检索模块，获取每个查询的结果。
- 举例：假设用户询问“如何提高机器学习模型的准确性？”系统可能生成以下几个查询：
 - “提高机器学习模型准确性的技巧”
 - “机器学习精度优化方法”
 - “如何训练更精确的机器学习模型”

RAG-Fusion

RAG-Fusion 是一种将多个查询的结果融合在一起的技术。它不仅是生成多个查询，还涉及如何处理这些查询的结果，并将它们合并，以得到一个更为精确和丰富的答案。

- 目标：通过融合多个查询的结果，获取更加精确、全面的信息，避免单一查询可能带来的信息缺失。
- 实现：多个查询的结果会被整合并过滤，以确保最终输出的结果更具相关性和准确性。RAG-Fusion 不仅关注检索，还关注如何对多个结果进行加权或排序，从中提取最有价值的信息。
- 举例：基于上面的查询，系统可能会将不同查询得到的结果合并（例如从“提高机器学习模型准确性的技巧”中提取到的一条技巧与“机器学习精度优化方法”中提取到的优化方法合并），最后生成一个融合的、更加丰富的回答。

关键区别

- Multi-query 主要关注生成多个不同的查询以扩大检索范围，而 RAG-Fusion 则关注如何将这些不同查询的结果有效地融合在一起，得到一个最终的答案。
- Multi-query 是检索层面的扩展，RAG-Fusion 更多是在生成层面上的整合。

总结来说，Multi-query 着眼于如何增加查询的多样性，而 RAG-Fusion 则是在多个查询的结果合并之后，通过一定的技术手段融合它们，提供一个更准确、更具深度的回答。

QA：RAG-Fusion 中的文档融合方法：RRF

RRF (*Reciprocal Rank Fusion*) 是一种常用的融合方法，在信息检索领域尤其是在 RAG-Fusion 中被用于整合来自不同查询或信息源的检索结果。RRF 的基本思想是根据每个结果的排名 (rank) 来加权融合多个查询的结果。该方法尤其适用于处理多来源的检索结果，以提高最终答案的质量。

Demo：

<https://github.com/realyinchen/RAG/tree/main/QueryTranslation>

Routing

Routing 是指根据用户的查询和相关信息，智能地选择合适的数据源或查询处理方法。它决定了应该从哪一类数据库、向量存储或检索系统中获取信息。这一过程帮助系统更加高效和精准地定位相关数据源，从而提高最终结果的质量。

它包括逻辑路由和语义路由两种方式，前者依赖查询的结构化特征，后者通过查询的语义理解来进行匹配。Routing 技术的引入，可以提升检索效率，确保系统能够精确地为用户提供最相关的信息。

1. Logical Routing

Logical Routing 是基于预定义的规则或者查询的结构化特征来选择合适的数据源或处理路径。其基本思路是根据用户查询的类型或格式，自动决定使用哪个数据源。

- 工作原理：系统通过分析查询的类型或内容，推测查询所需的信息源。例如，如果查询涉及结构化数据，系统可能会选择从关系数据库（如 SQL 数据库）中获取数据；如果查询涉及图数据，系统可能会选择从图数据库（如 Neo4j）中获取数据。
- 示例：
 - 如果用户查询“找出所有销售量最高的商品”，系统会判断这属于结构化查询，选择从关系数据库中获取数据。
 - 如果查询是“查找与某个用户关系最密切的其他用户”，系统可能会选择图数据库，因为图数据库适合处理这类社交关系数据。

2. Semantic Routing

Semantic Routing 则基于查询的语义或内容相似性来决定如何路由。也就是说，系统会分析用户查询的实际语义，并通过查询与预定义的“提示词”或嵌入（Embedding）相似度来选择处理路径。这

种方法更多依赖自然语言处理 (NLP) 技术，通过嵌入向量的方式来实现高效的语义匹配。

- 工作原理：在语义路由中，查询会被转换为向量表示，系统将该向量与存储在数据库中的预定义提示词或模板进行比对。然后，系统根据语义相似度来确定最合适的处理路径或数据源。
- 示例：
 - 如果用户的查询是“推荐给我一些科技书籍”，系统会通过语义理解，识别出这是与书籍推荐相关的查询，因此可以将其路由到包含书籍数据的向量数据库或推荐系统。
 - 如果查询涉及情感分析，系统则可以将查询路由到专门处理情感分析的模块或模型。

Routing 的优势

- ✓ 智能分流，提高检索效率：通过智能路由，系统可以避免无效的查询路径，直接跳转到最相关的数据源或模型，提高响应速度。
- ✓ 精准匹配：根据查询的内容和类型，能够选择最匹配的处理方法，确保最终返回的结果具有较高的相关性和质量。
- ✓ 适应多模态数据：可以处理结构化数据、非结构化数据和语义搜索任务。

Demo：

<https://github.com/realyinchen/RAG/tree/main/Routing>

Query Construction

Query Construction 是指将用户的自然语言查询转换为适用于不同数据库（关系型数据库、图数据库、向量数据库）的结构化查询语句或带有适当过滤条件的查询表示。这样，整个 RAG 体系能够更智能地处理复杂查询，并从不同类型的数据源中高效检索信息。

1. 关系型数据库 (Relational DBs)

在查询关系型数据库时，Query Construction 需要将自然语言转换为 SQL 语句，以便查询结构化数据。

对于用户问题：“列出销售量大于 1000 的产品。”会生成以下 SQL 查询：

```
1 SELECT * FROM products WHERE sales > 1000;
```

如果数据库支持向量搜索（如 PostgreSQL + PGVector），SQL 查询可能还会包含 ANN（近似最近邻）搜索，如：

```
1 SELECT * FROM products ORDER BY embedding <-> '[query_embedding]' LIMIT 10;
```

这里的 `embedding <-> '[query_embedding]'` 表示使用向量距离进行排序，找到最相关的产品。

2. 图数据库 (Graph DBs)

对于存储实体关系（如社交网络、知识图谱）的图数据库，Query Construction 需要将自然语言转换为 Cypher 查询（Neo4j 的查询语言）或其他图查询语言，如 Gremlin（适用于 JanusGraph 等）。

对于用户问题：“查找与用户 123 关系最密切的用户。”会生成以下 Cypher 查询：

```
1 MATCH (u:User)-[:FRIEND_WITH]-(f:User)
2 WHERE u.id = 123
3 RETURN f
4 ORDER BY f.interaction_score DESC
5 LIMIT 5;
```

- MATCH (u:User)-[:FRIEND_WITH]-(f:User) 找到与用户 123 直接相连的用户
- ORDER BY f.interaction_score DESC 按照交互得分降序排序，确保找到最相关的用户。

3. 向量数据库 (Vector DBs)

向量数据库用于存储和检索高维向量（通常来自文本、图像或音频的嵌入向量）。在 Query Construction 过程中，自动生成元数据过滤条件 (Metadata Filters)：结合用户查询内容，自动附加元数据过滤条件以减少无关的搜索结果，提高搜索精度。

- 纯粹的向量相似性搜索可能会返回一些无关的信息，比如书籍、博客文章等，而元数据过滤能保证只检索论文类的文档。
- 过滤条件还能提高检索效率，因为数据库可以利用索引加速查询，而不需要对整个数据库进行向量计算。

对于用户问题：“找一些关于 Transformer 论文的摘要。”，则常规向量搜索：

```
1 vector_db.search(query_embedding, top_k=10)
```

这里 query_embedding 是通过 BERT 或 OpenAI Embeddings 等模型计算得到的向量表示。

结合元数据过滤的查询：

```
1 vector_db.search(
2     query_embedding,
3     top_k=10,
4     filters={"category": "paper", "year": {"$gte": 2020}}
5 )
```

这里 filters={"category": "paper", "year": {"\$gte": 2020}} 说明：

- 只查找 category=paper 的文档，排除其他类别的内容；
- 仅搜索 2020 年及之后的论文，避免返回过时的信息。

Query Construction 的优势

- ✓ **更精准的查询：**通过合适的查询构造方法，确保不同数据源返回高质量的结果。
- ✓ **支持多种数据库类型：**能够适配结构化数据、关系型数据和向量数据。
- ✓ **自动元数据过滤：**在向量数据库查询时自动附加适当的过滤条件，减少噪声，提高相关性。
- ✓ **高效查询执行：**合理构造查询语句，使得数据库能更快地响应，提高检索效率。

Demo：

<https://github.com/realyinchen/RAG/tree/main/QueryConstruction>

Indexing

Indexing 是指将文档组织和优化，以提高检索效率和相关性。在 RAG 系统中，索引过程影响着数据的存储方式、查询匹配效果以及最终的答案质量。

Indexing 主要解决数据组织和优化问题，提升 RAG 系统的查询效率和召回质量：

1. **Chunk Optimization** (语义切分)：优化查询匹配精度。
2. **Multi-representation Indexing** (多表征索引)：支持关键词+向量检索。
3. **Specialized Embeddings** (特定领域嵌入)：针对专业知识优化向量检索。
4. **Hierarchical Indexing** (层级索引)：提高大规模文档检索的效率。

1. Chunk Optimization (文本切片优化)

文档通常是长篇文本，直接索引完整文档可能导致：

- **查询匹配不精准**：一个 10 页的论文可能只含有 2 句话与查询相关，但向量检索可能会返回整个论文，使答案不够聚焦。
- **计算开销大**：如果索引粒度太大，查询时需要处理的文本过多，影响检索速度。

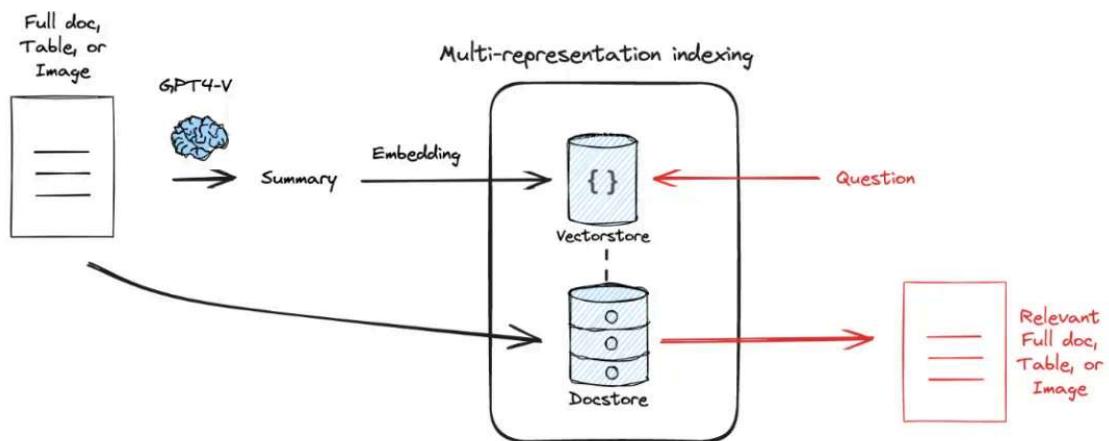
比如使用 Semantic Splitter (语义切分器) 按照语义单元而不是固定字符数切分文本，使得每个切片 (chunk) 都包含完整的信息。从而可以：

- 提高查询匹配精度，避免语义断裂。
- 降低计算开销，减少不必要的文本传输。

2. Multi-representation Indexing (多表征索引)

通过结合多种表示（例如，文档的摘要、块或完整内容），来在检索和生成过程中取得最佳效果。它通过使用不同层次和粒度的表示（简洁的摘要与详细的文档）来弥补单一表示可能存在的不足。

Proposition Indexing 是 Multi-representation 的一种实现方法：



1. 我们使用 LLM 对原始文本进行总结形成摘要 (Summary)；
 2. 对摘要进行嵌入处理，保存到 Vector Store 中；同时也将原始文本保存到 Doc Store 中；
 3. 检索时，根据用户的 Question 嵌入去 Vector Store 中检索对应的摘要，然后返回原始的文本；
- 对长上下文的 LLM 来说更加友好，因为甚至可以将整篇原始文档输入 LLM 进行回答。

3. Specialized Embeddings (特定领域嵌入)

通用向量嵌入（如 OpenAI Embeddings）在一些特定领域（如法律、医学）可能表现不佳，原因是：

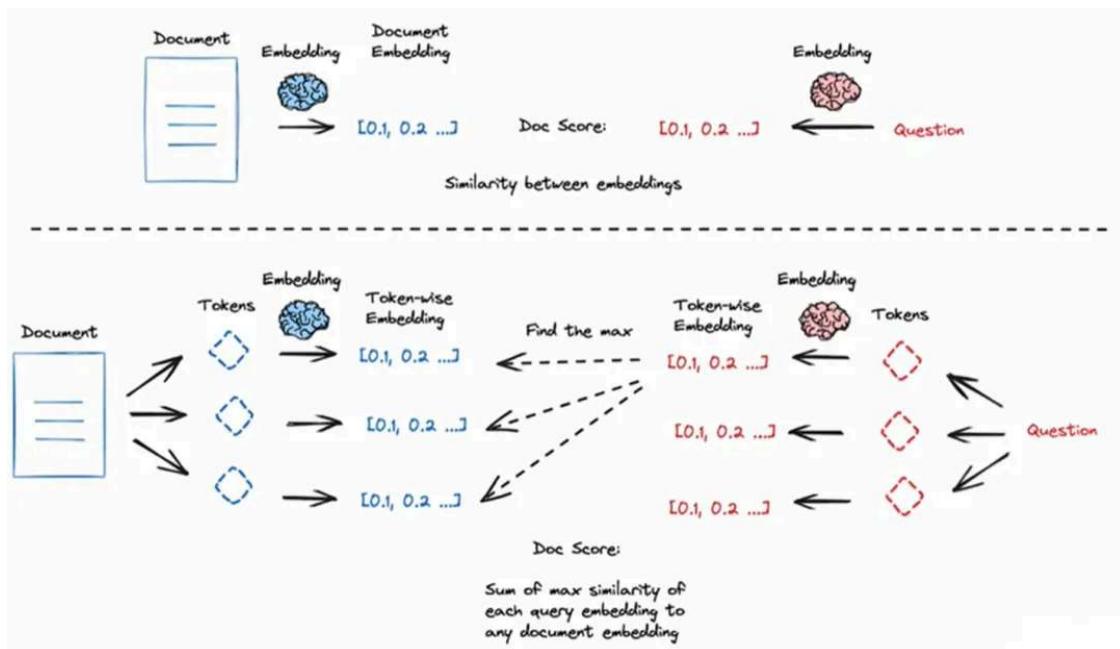
- 预训练数据中缺乏领域知识，导致向量表达偏离专业概念。
- 语义相似度不一定适合特定任务，比如在法律文本中，“判例 A”和“判例 B”可能需要特别区分，而普通 NLP 可能会认为它们相似。

我们可以使用领域数据对嵌入模型进行**Fine-tuning**（微调）或使用专门的检索优化模型（如 ColBERT）。从而可以：

- 针对专业领域优化，提高搜索精度。
- 避免通用模型的语义偏差，更适合专业知识库。

到目前为止，我们所学的各种 RAG 技巧都是基于匹配文档（块）和问题的语义相似度来实现的，更具体地说：我们直接对整篇文档或者文档块与用户问题进行嵌入处理，比较处理之后的嵌入向量之间的相似程度。

但是有人说用户提出的问题可能是各种各样的，如果仅仅将整篇文档或者文档块压缩成一个向量表示，那么这未免太过于笼统了。



ColBERT (Contextualized Late Interaction over BERT)，基于 BERT，实现了细粒度的 token 级嵌入相似性计算。

以下是具体的步骤和原理：

1. Token 级别相似度计算

对用户问题 (Query) 和文档 (Document) 分别生成 token 嵌入，记为：

- 查询嵌入： $\mathbf{Q} = \{\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_m\}$, 其中 m 是查询的 token 数。
- 文档嵌入： $\mathbf{D} = \{\mathbf{d}_1, \mathbf{d}_2, \dots, \mathbf{d}_n\}$, 其中 n 是文档的 token 数。

计算查询中每个 token 嵌入 \mathbf{q}_i 和文档中每个 token 嵌入 \mathbf{d}_j 的相似度，通常使用 **余弦相似度**：

$$s_{ij} = \cos(\mathbf{q}_i, \mathbf{d}_j) = \frac{\mathbf{q}_i \cdot \mathbf{d}_j}{\|\mathbf{q}_i\| \|\mathbf{d}_j\|}$$

得到一个 $m \times n$ 的相似度矩阵 S , 其中每个值 s_{ij} 表示查询 token i 和文档 token j 的相似度。

2. 最大池化 (MaxSim) 操作

为了从 token 级别相似度汇总到查询级别：

1. 对于查询中的每个 token i , 找到与文档中 token 的最大相似度：

$$\text{MaxSim}_i = \max_{j \in [1, n]} s_{ij}$$

这表示查询中第 i 个 token 最相关的文档 token 的相似度。

2. 汇总所有查询 token 的最大相似度作为查询与文档的整体相似度：

$$\text{Score}_{Q,D} = \sum_{i=1}^m \text{MaxSim}_i$$

这种方式捕获了查询中每个 token 的最强匹配，同时忽略了不匹配的 token，从而提升了文档级别的匹配准确性。

3. 选择最相关的文档

对所有候选文档 D_1, D_2, \dots, D_k 分别计算 Score_{Q,D_i} ，然后根据得分从高到低排序，选择得分最高的文档作为最终检索结果。

CoBERT 的开源实现：

<https://github.com/stanford-futuredata/CoBERT>

关于具体的代码案例，可以参考：

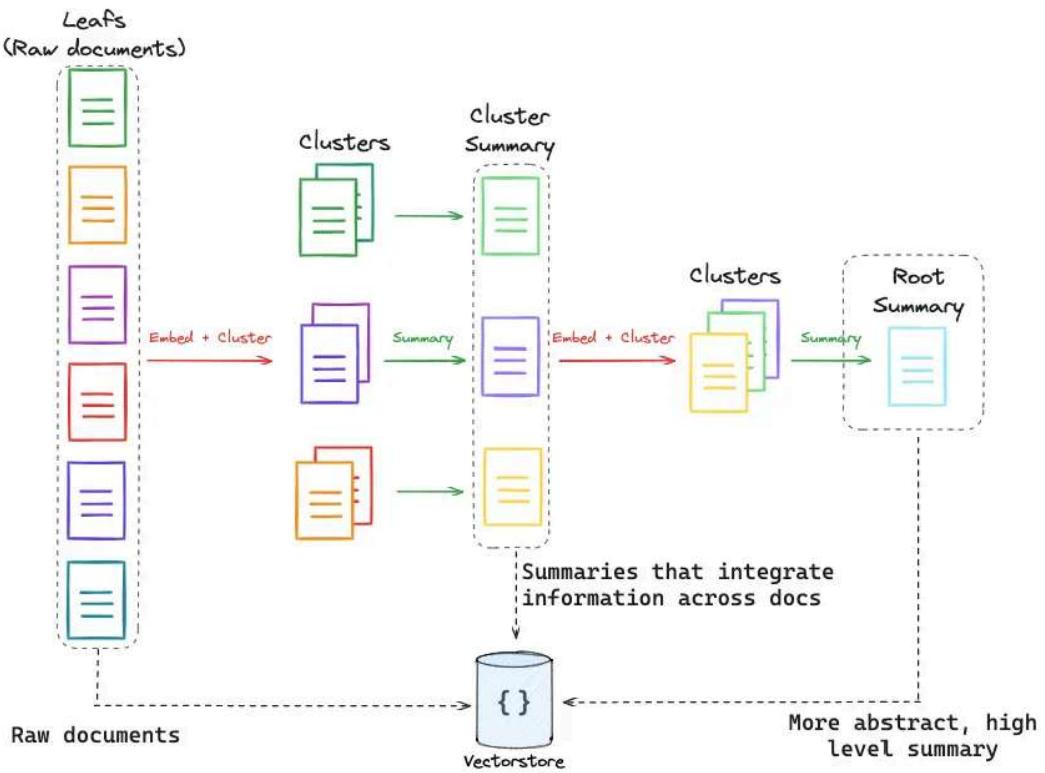
<https://python.langchain.com/docs/integrations/retrievers/ragatouille>

4. Hierarchical Indexing (层级索引)

对于大型文档（如书籍、论文集合），直接索引所有文本会导致：

- **检索结果过多**，难以找到真正相关的信息。
- **计算成本高**，每次查询都要处理大量数据。

RAPTOR (Recursive Abstractive Processing for Tree-Organized Retrieval) 是一种将文档组织成分层结构的索引技术，它能够有效地实现递归式的信息抽象和检索。这种方法以“树”的形式组织文档数据，叶节点代表原始文档或者文档块，树的高层节点则代表这些文档经过抽象后的总结。RAPTOR 的目标是通过分层的方式，提升信息检索的效率，同时能够适应不同粒度的查询需求。



Demo:

<https://github.com/realyinchen/RAG/tree/main/Indexing>

Retrieval & Generation

🔍 Retrieval & Generation —— RAG 的双引擎 🚀

Retrieval 和 Generation 是互补的两个核心环节，共同决定最终答案的质量。一个好的 RAG 系统，既要确保检索到最相关的信息，又要让生成模型能够充分利用这些信息，生成高质量、可信的回答。

📌 Retrieval & Generation 的核心目标

- ✓ **Retrieval**: 找到最相关的外部信息，为 LLM 生成答案提供支持。
- ✓ **Generation**: 基于检索到的内容，生成连贯、准确、有事实依据的回答。

● Retrieval: 找到最相关的信息

Retrieval 负责从外部知识库（如 VectorDB、SQL 数据库、文件系统）中检索到与用户查询最相关的信息，并将其提供给 LLM 进行生成。

📌 核心目标：

- **高召回率 (Recall)**：确保不会遗漏重要信息。
- **高精准度 (Precision)**：减少无关或冗余的内容。
- **可扩展性 (Scalability)**：支持大规模数据检索，提高系统效率。

◆ 核心技术

◆ 1. RAG Fusion

通过不同角度生成多个查询，提高召回率，并使用 Reciprocal Rank Fusion (RRF) 融合检索结果。

✓ 优势：

- 适用于复杂问题，避免遗漏关键信息。
- 结合多个查询方式，提高搜索的全面性。

◆ 2. Re-Rank (重排序)

召回初步结果后，使用更强的排序方法（如 Cross-Encoder、RankGPT）进行二次排序，确保最相关的文档排在前面。

✓ 优势：

- 提高文档相关性，减少 LLM 处理无关信息的负担。
- 适用于开放领域问答，提高答案质量。

◆ 3. Hybrid Retrieval (混合检索)

结合向量搜索（Vector Search）、关键字搜索（BM25）、知识图谱（Knowledge Graph），以多模态方式找到最佳结果。

✓ 优势：

- 兼顾语义匹配和关键字匹配，减少错误召回。
- 适用于结构化 + 非结构化数据的场景。

◆ 4. Active Retrieval (主动检索)

让 LLM 评估初步检索结果，决定是否需要额外查询，以提高答案的完整性。

✓ 优势：

- 避免因信息不足导致的错误回答。
- 适用于需要多轮查询的复杂任务。

● Generation：基于检索信息生成高质量答案

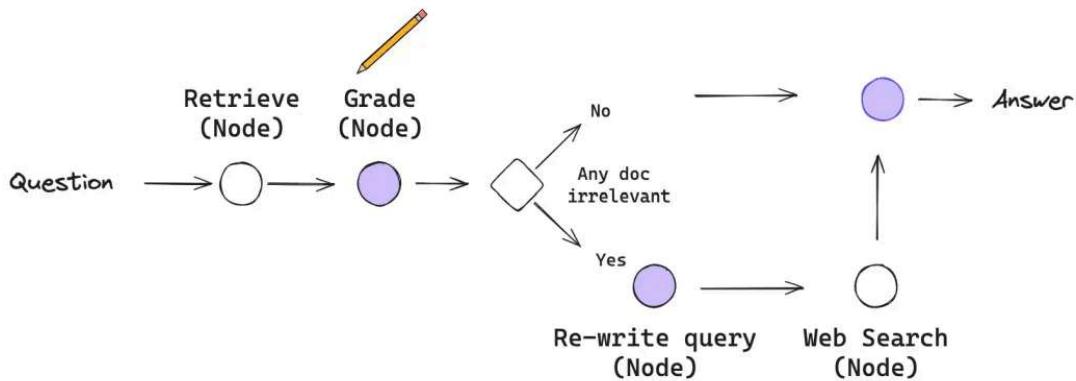
Generation 负责利用 Retrieval 提供的信息，生成准确、连贯、有事实依据的回答。

◆ 核心目标：

- 增强事实性（Fact-enhancement）：减少幻觉，提高答案可信度。
- 提高推理能力（Reasoning）：让 LLM 具备跨文档、多步骤推理能力。
- 提高可控性（Controllability）：确保答案符合特定格式或标准。

◆ 核心技术

◆ 1. CRAG (Corrective RAG, 纠正型 RAG)

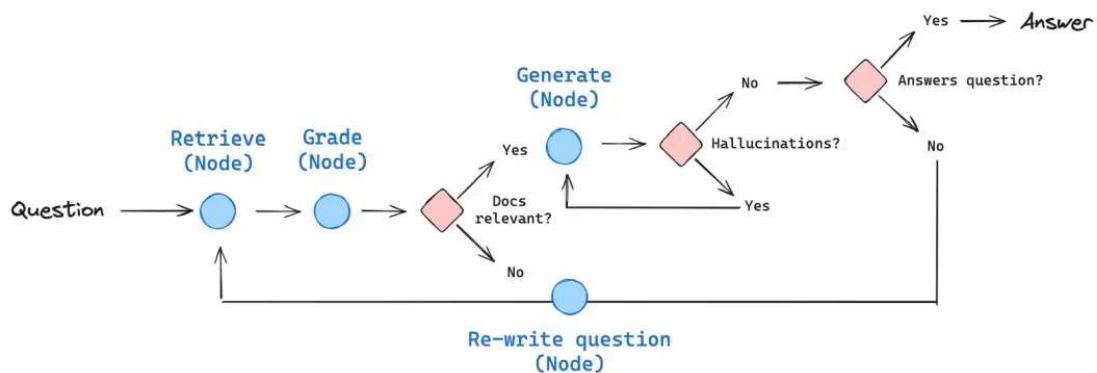


LLM 评估检索到的文档是否相关，并根据需要触发额外检索来补充检索数据。

优势：

- 减少错误答案，提高答案质量。
- 适用于法律、医学、金融等领域，确保答案符合行业标准。

◆ 2. Self-RAG (Self-Reflective RAG, 自我反思 RAG)

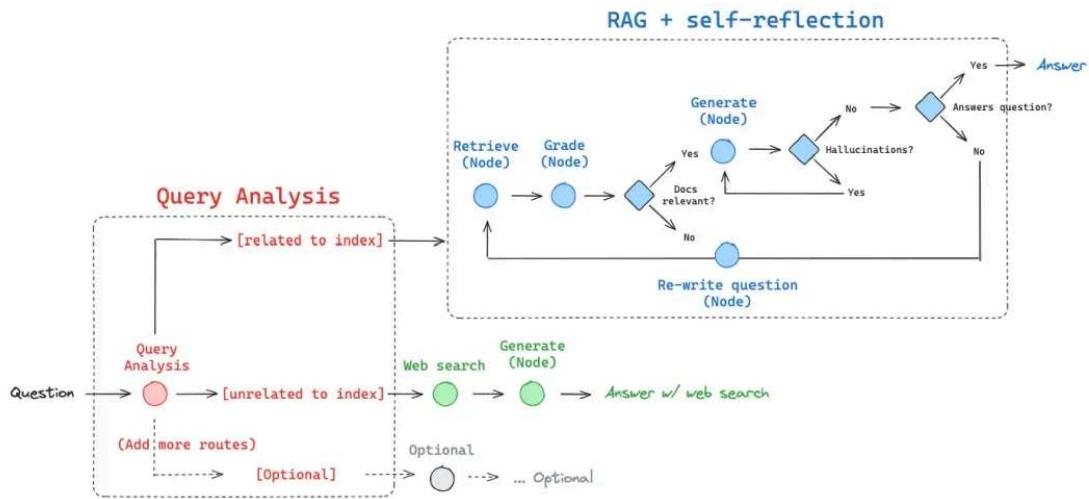


让 LLM 生成初步答案后，进行自我检查，并决定是否需要额外检索或调整答案结构。

优势：

- 提高生成的连贯性和合理性。
- 适用于开放领域问答，提高 LLM 的“自我意识”。

◆ 3. Adaptive-RAG (自适应 RAG)



系统根据查询的复杂度和要求自适应地选择检索和生成的方式。

✓ 优势:

- **提高效率**: 通过自适应地选择检索和生成的策略，减少不必要的计算，提升处理速度。
- **增强准确性**: 动态评估查询的需求，避免生成无关或低质量的答案。
- **灵活应对多样化查询**: 适应不同领域和复杂度的查询，确保答案的质量和适用性。
- **减少计算开销**: 针对简单问题，直接检索而不进行多轮生成，减少不必要的计算资源消耗。

未来趋势

- ❶ **混合检索 (Hybrid Retrieval)** : 结合向量搜索 + 关键字搜索 + 知识图谱，提高召回质量
- ❷ **自适应 (Adaptive RAG)** : LLM 根据需求动态决定是否检索，提高系统效率
- ❸ **端到端优化 (End-to-End RAG)** : 将检索与生成集成训练，提高整体效果
- ❹ **增强事实性 (Fact-Enhanced Generation)** : 结合检索结果进行交叉验证，减少幻觉

Demo:

<https://github.com/realyinchen/RAG/tree/main/AgenticRAG>



PyTorch研习社

打破知识壁垒，做一名知识的传播者

654篇原创内容

公众号

RAG 16

RAG · 目录

[上一篇 · RAG从入门到精通系列6：Retrieval（检索）](#)