This code implements **RAG Fusion**, a **Retrieval-Augmented Generation (RAG)** method that enhances information retrieval by combining multiple search results and reranking them using **Reciprocal Rank Fusion (RRF)**. Below is a **detailed breakdown** of the code, along with its **concepts, architecture, and improvements**.

1) Line-by-Line Code Explanation

1.1 Installing Dependencies

```
python
CopyEdit
!pip -q install langchain huggingftiktace_hub oken pypdf
!pip -q install google-generativeai chromadb
!pip -q install sentence_transformers
!pip install -U langchain-community
```

- Installs **LangChain**, **Google Generative AI SDK**, **ChromaDB** (vector store), **Hugging Face Sentence Transformers**, and **pypdf** (for PDF loading).
- `-q` flag suppresses the installation logs.

1.2 Text Wrapping Helper Function

```
python
CopyEdit
import textwrap
def wrap_text(text, width=90):
    lines = text.split('\n')
    wrapped_lines = [textwrap.fill(line, width=width) for line in
lines]
    return '\n'.join(wrapped_lines)
```

- This function **formats output text** to a specified width (90 characters).
- Useful for displaying responses from **LLMs** in a readable manner.

1.3 Setting Up API Keys for Google Gemini

```
python
CopyEdit
import os
from google.colab import userdata
GOOGLE_API_KEY = userdata.get('GOOGLE_API_KEY')
os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
```

- Fetches **Google API Key** from Colab's `userdata` storage and sets it as an environment variable.
- Used for **Google Generative AI (Gemini LLM)**.

1.4 Loading Google Gemini LLM

```
python
CopyEdit
%pip install --upgrade --quiet langchain-google-genai
from langchain_google_genai import ChatGoogleGenerativeAI
llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro")
```

- Installs **LangChain Google GenAI module**.
- Initializes **Gemini 1.5 Pro**, an advanced LLM from Google.

1.5 Testing LLM with a Simple Query

```
python
CopyEdit
result = llm.invoke("Write a ballad about LangChain")
print(result.content)
```

- Calls **Gemini LLM** to generate a ballad about **LangChain**.
- `invoke()` is used for synchronous text generation.

1.6 Loading Text Documents from Google Drive

```
python
CopyEdit
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.document_loaders import DirectoryLoader, TextLoader
from google.colab import drive
drive.mount('/content/drive')
data_path="/content/drive/MyDrive/English"
```

- **Mounts Google Drive** to access text documents.
- `DirectoryLoader` loads all `.txt` files from **`/MyDrive/English`**.

1.7 Splitting Text into Chunks

```
python
CopyEdit
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=100)
texts = text_splitter.split_text(raw_text)
```

- **Splits large text** into smaller **overlapping chunks** (500 tokens with 100 overlap).
- **Overlap** ensures context is maintained across chunks.

1.8 Converting Text into Embeddings

```
python
CopyEdit
from langchain.embeddings import HuggingFaceBgeEmbeddings
model_name = "BAAI/bge-small-en-v1.5"
embedding_function = HuggingFaceBgeEmbeddings(model_name=model_name,
encode_kwargs={'normalize_embeddings': True})
```

- Uses **BGE Embeddings** from Hugging Face.
- `normalize_embeddings=True` ensures **cosine similarity** is computed correctly.

1.9 Storing Embeddings in ChromaDB

```
python
CopyEdit
from langchain.vectorstores import Chroma
db = Chroma.from_texts(texts, embedding_function,
persist_directory="./chroma_db")
query = "Tell me about Universal Studios Singapore?"
db.similarity_search(query, k=5)
```

- **Stores embeddings in ChromaDB** (a fast **vector database**).
- Uses **similarity search** to find **top-5 relevant documents** for a query.

1.10 Creating a Retriever

```
python
CopyEdit
retriever = db.as_retriever()
retriever.get_relevant_documents(query)
```

- Converts ChromaDB into a **retriever** to fetch **relevant documents** dynamically.

1.11 Creating a Chat Prompt Chain

```
python
CopyEdit
from langchain.prompts import ChatPromptTemplate
template = """Answer the question based only on the following context:
{context}

Question: {question}
"""
prompt = ChatPromptTemplate.from_template(template)
```

- **Formats user query** along with **retrieved context**.
- Ensures the **LLM does not hallucinate** by answering **only based on retrieved data**.

1.12 Combining LLM & Retriever into a Chain

```
python
CopyEdit
from langchain.schema.runnable import RunnableLambda,
RunnablePassthrough
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)
```

- **Combines retriever, prompt, and LLM** into a **single query pipeline**.
- Uses **RunnablePassthrough** to pass data **unchanged**.

2) RAG Fusion - Enhancing Retrieval Quality

2.1 Generating Multiple Queries for Better Retrieval

```
python
CopyEdit
prompt = ChatPromptTemplate(input_variables=['original_query'],

messages=[SystemMessagePromptTemplate(prompt=PromptTemplate(input_vari
ables=[],template='You are a helpful assistant that generates multiple
search queries based on a single input query.')),

HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['ori
ginal_query'], template='Generate multiple search queries related to:
{question} \n OUTPUT (4 queries):'))])
```

- Instead of retrieving **documents for a single query**, we generate **multiple related queries**.
- This **improves recall** by fetching **more diverse** documents.

2.2 Reciprocal Rank Fusion (RRF)

```
python
CopyEdit
def reciprocal_rank_fusion(results: list[list], k=60):
    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
            fused_scores[doc_str] += 1 / (rank + k)

    reranked_results = [(loads(doc), score) for doc, score in
sorted(fused_scores.items(), key=lambda x: x[1], reverse=True)]
    return reranked_results
```

- **Ranks multiple retrieved document sets** using **Reciprocal Rank Fusion (RRF)**.
- RRF gives **higher weight** to documents appearing **higher in multiple results**.

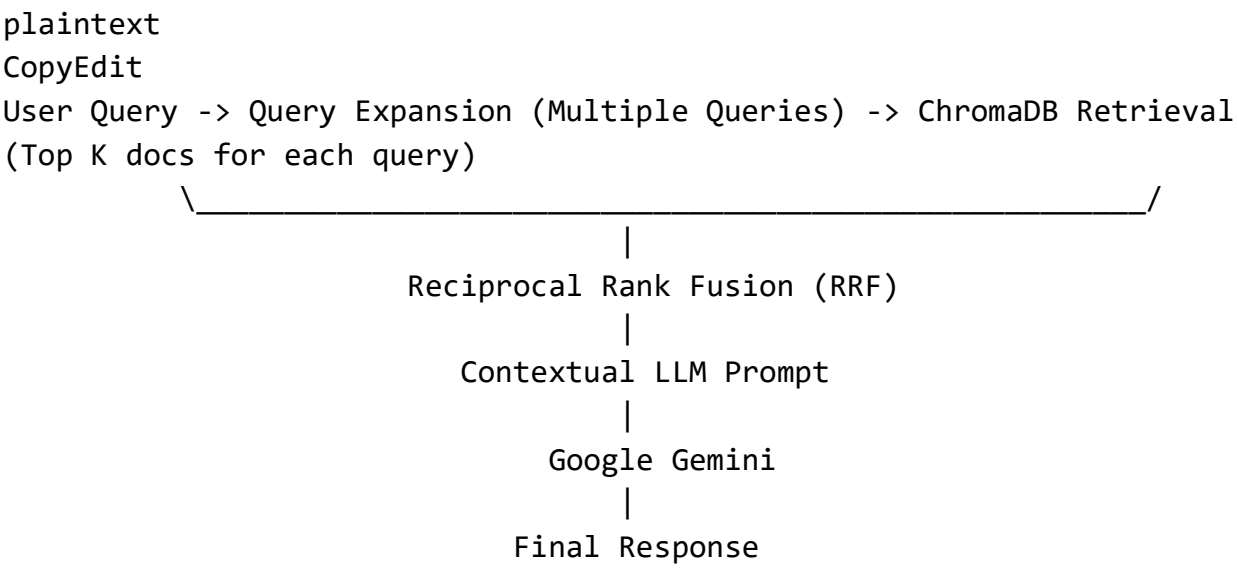
2.3 Full RAG Fusion Chain

```
python
CopyEdit
full_rag_fusion_chain = (
    {
        "context": ragfusion_chain,
        "question": RunnablePassthrough()
    }
    | prompt
    | llm
    | StrOutputParser()
)
```

- **Final Chain**: Uses **RAG Fusion retriever**, applies **prompt formatting**, and generates an answer via **LLM**.

3) Architecture Diagram

RAG Fusion with Reciprocal Rank Fusion (RRF)



- **Query Expansion:** Generates multiple search queries for **better document recall**.
- **Vector DB (Chroma):** Stores and retrieves **semantic embeddings**.
- **Reciprocal Rank Fusion (RRF):** Combines multiple results to **rank the best**.
- **LLM (Gemini):** Generates **final response** using **retrieved context**.

4) Missing Features / Possible Enhancements

✅ **Use Hybrid Search** (BM25 + Embeddings) for **better recall**.
✅ **Integrate LlamaIndex** for improved retrieval.
✅ **Optimize RRF Parameters** (k value) for **better ranking**.

Would you like a **Colab notebook** with this implementation? 🚀

################ Below is from DeepSeek ####################

1. **Code Explanation Line-by-Line**

Install Dependencies

```
python
Copy
!pip -q install langchain huggingface_hub token pypdf
!pip -q install google-generativeai chromadb
!pip -q install sentence_transformers
!pip install -U langchain-community
```

- Installs required libraries:
  - `langchain`: Framework for building LLM applications.
  - `huggingface_hub`: Access Hugging Face models.
  - `google-generativeai`: Google's Gemini models.
  - `chromadb`: Vector database for storing embeddings.
  - `sentence_transformers`: Text embedding models.

Text Wrapping Utility

```
python
Copy
def wrap_text(text, width=90):
    lines = text.split('\n')
    wrapped_lines = [textwrap.fill(line, width=width) for line in
lines]
    return '\n'.join(wrapped_lines)
```

- Formats text into 90-character wide lines for readability.

Google API Setup

```
python
Copy
import os
from google.colab import userdata
GOOGLE_API_KEY = userdata.get('GOOGLE_API_KEY')
os.environ["GOOGLE_API_KEY"] = GOOGLE_API_KEY
```

- Fetches the Google API key from Colab secrets and sets it as an environment variable for Gemini.

Initialize Gemini LLM

```
python
Copy
from langchain_google_genai import ChatGoogleGenerativeAI
llm = ChatGoogleGenerativeAI(model="gemini-1.5-pro")
result = llm.invoke("Write a ballad about LangChain")
print(result.content)
```

- Initializes the Gemini 1.5 Pro model for text generation and tests it with a sample query.

Load Documents

```
python
Copy
from langchain.document_loaders import DirectoryLoader
drive.mount('/content/drive')
data_path = "/content/drive/MyDrive/English"
loader = DirectoryLoader(data_path, glob="*.txt", show_progress=True)
docs = loader.load()
docs = docs[:50]  # Use first 50 documents for testing
```

- Mounts Google Drive, loads 50 text files from the specified directory, and truncates to 50 docs for efficiency.

Text Splitting

```
python
Copy
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500, chunk_overlap=100
)
texts = text_splitter.split_text(raw_text)
```

- Splits concatenated text into chunks of 500 tokens with 100-token overlap to preserve context.

Embeddings Setup

```
python
Copy
from langchain.embeddings import HuggingFaceBgeEmbeddings
embedding_function = HuggingFaceBgeEmbeddings(
    model_name="BAAI/bge-small-en-v1.5",
    encode_kwargs={'normalize_embeddings': True}
)
```

- Uses the `bge-small-en-v1.5` model to generate embeddings normalized for cosine similarity.

Vector Database (Chroma)

```
python
Copy
db = Chroma.from_texts(texts, embedding_function,
persist_directory="./chroma_db")
retriever = db.as_retriever()
```

- Creates a Chroma vector store from text chunks and initializes a retriever for similarity search.

RAG Chain

```
python
Copy
template = """Answer the question based only on the context:
{context}\nQuestion: {question}"""
prompt = ChatPromptTemplate.from_template(template)
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt | llm | StrOutputParser()
)
text_reply = chain.invoke("Tell me about Universal Studio Singapore")
```

- Defines a prompt template and chains together:
    - Retriever to fetch context.
    - Prompt formatting.
    - LLM for answer generation.
    - Output parsing.

RAG Fusion

```
python
Copy
# Query Expansion
prompt = ChatPromptTemplate(
    input_variables=['original_query'],
    messages=[SystemMessagePromptTemplate(...),
HumanMessagePromptTemplate(...)]
)
generate_queries = prompt | llm | StrOutputParser() | (lambda x:
x.split("\n"))

# Reciprocal Rank Fusion (RRF)
def reciprocal_rank_fusion(results: list[list], k=60):
    # Ranks documents across multiple queries by fusing scores
    ...

# Full RAG Fusion Chain
ragfusion_chain = generate_queries | retriever.map() |
reciprocal_rank_fusion
full_rag_fusion_chain = (
    {"context": ragfusion_chain, "question": RunnablePassthrough()}
    | prompt | llm | StrOutputParser()
)
```

1. **Query Expansion**: Generates 4 related queries from the original input.
2. **Reciprocal Rank Fusion (RRF)**: Combines results from multiple queries, prioritizing documents that rank highly across all.
3. **Integration**: Uses RRF results as context for the final LLM answer.

2. **Key Concepts**

Retrieval-Augmented Generation (RAG)

- Enhances LLMs by retrieving relevant documents to ground answers in external data.
- **Steps**:
  - **Indexing**: Chunk text, generate embeddings, store in a vector DB.
  - **Retrieval**: Fetch top-k documents for a query.
  - **Generation**: Feed context + query to LLM for answer.

RAG Fusion

- Improves RAG by:
  - **Query Expansion**: Generate multiple queries to cover diverse aspects.
  - **Reciprocal Rank Fusion (RRF)**: Merge results from all queries, prioritizing consensus.

Reciprocal Rank Fusion (RRF)

- Formula: `score += 1 / (rank + k)` for each document across query results.
- Balances relevance across multiple retrievals (lower `k` = more emphasis on top ranks).

Vector Database (Chroma)

- Stores embeddings and enables fast similarity search.
- `persist_directory` saves the index to disk for reuse.

Embeddings

- `BAAI/bge-small-en-v1.5`: Lightweight model optimized for retrieval tasks.
- Normalized embeddings ensure cosine similarity works correctly.

3. **Architecture Diagram**

```
Copy
User Query
   |
   ▼
[Query Generator] → Multiple Queries (Q1, Q2, Q3, Q4)
   |
```

```
   ▼
[Retriever → Vector DB] → Fetch docs for each query
   |
   ▼
[Reciprocal Rank Fusion] → Fused & Re-ranked Documents
   |
   ▼
[LLM (Gemini)] → Final Answer
```

4. **Missing Points & Improvements**
Missing in Code

1. **Error Handling**: No try/catch for API calls or file loading.
2. **Evaluation**: Metrics like precision/recall for retrieval quality.
3. **Hyperparameter Tuning**: No experimentation with k in RRF or chunk sizes.
4. **Document Diversity**: Assumes all text files are in English; no handling of other formats (PDFs, images).
5. **Security**: API key exposed in Colab (use environment variables in production).

Improvements

1. **Query Expansion**: Use LLM fine-tuned for query generation.
2. **Hybrid Search**: Combine keyword-based (BM25) and vector search.
3. **Metadata Filtering**: Filter documents by date/source during retrieval.
4. **Caching**: Cache frequent queries to reduce latency.
5. **User Feedback**: Add mechanisms to collect feedback on answer quality.

##

The provided code is implementing an advanced **Retrieval-Augmented Generation (RAG) Fusion** system using **LangChain, ChromaDB, Google Gemini LLM, and Reciprocal Rank Fusion (RRF)**. Below is a **detailed explanation** of the **concepts** related to the code.

📌 Key Concepts in the Code

1) Retrieval-Augmented Generation (RAG)

◇ What is RAG?

- **RAG** is a technique that combines **retrieval-based search** with **text generation**.
- Instead of relying only on **pre-trained knowledge**, it **retrieves relevant documents** from an **external knowledge base** and uses them as **context** for an LLM (like Gemini).
- This improves **accuracy**, prevents **hallucinations**, and provides **up-to-date** information.

◇ RAG Pipeline in the Code:

1. **User Query:** `"Tell me about Universal Studios Singapore"`
2. **Retrieve Relevant Documents:** From **ChromaDB** (vector database).
3. **Use Retrieved Context in LLM:** The model **answers** using only the retrieved **context**, avoiding incorrect information.

2) Query Expansion & RAG Fusion

◇ Why Expand Queries?

- **A single search query might miss relevant information.**
- Instead of retrieving documents for **just one query**, we generate **multiple related queries**.
- This helps **fetch a more diverse set of relevant documents**.

◇ Query Expansion in the Code:

```
python
CopyEdit
prompt = ChatPromptTemplate(input_variables=['original_query'],
```

```
messages=[SystemMessagePromptTemplate(prompt=PromptTemplate(input_vari
ables=[],template='You are a helpful assistant that generates multiple
search queries based on a single input query.')),

HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['ori
ginal_query'], template='Generate multiple search queries related to:
{question} \n OUTPUT (4 queries):'))])
```

- The **LLM generates multiple versions** of the user query.
- Example:
  - **Original Query:** `"Universal Studios Singapore"`
  - **Expanded Queries:**
    - `"What attractions are there in Universal Studios Singapore?"`
    - `"How much does a ticket to Universal Studios Singapore cost?"`
    - `"What are the best rides at Universal Studios Singapore?"`
    - `"Is Universal Studios Singapore good for families?"`

◇ Benefits of Query Expansion:

✅ **Improves Recall** → Retrieves a broader range of documents.
✅ **Handles Ambiguity** → Covers different interpretations of the question.
✅ **Increases Context Quality** → LLM gets **better input**, leading to **better answers**.

3) Document Retrieval & Vector Databases

◇ Why Use Vector Databases?

- Text search traditionally relies on **keyword matching** (BM25).
- Instead, **vector databases** (like ChromaDB) **convert text into numerical vectors**.
- This allows **semantic search**, meaning it retrieves results even if the wording is different.

◇ ChromaDB in the Code:

```
python
CopyEdit
from langchain.vectorstores import Chroma
db = Chroma.from_texts(texts, embedding_function,
persist_directory="./chroma_db")
query = "Tell me about Universal Studios Singapore?"
db.similarity_search(query, k=5)
```

- Stores **embedded document chunks**.
- Performs **similarity search** to find the **top 5 most relevant documents**.

◇ How It Works:

1. **Convert Text into Vectors** (using `HuggingFaceBgeEmbeddings`).
2. **Store Vectors in ChromaDB**.
3. **Search for Similar Vectors** when querying.

◇ Why Use Semantic Search Instead of BM25?

BM25 (Keyword Search)	Vector Search (Semantic Search)
Matches **exact words**.	Matches **similar meaning**.
Misses synonyms.	Finds **similar concepts**.
Works well for structured text.	**Works well** for unstructured text.

4) Reciprocal Rank Fusion (RRF)

◇ Why RRF?

- **Combines multiple ranked lists** into a **single, improved ranking**.
- Ensures that **documents appearing higher in multiple results** get a **higher final rank**.
- Prevents the **"missing document problem"** caused by poor single-query retrieval.

◇ RRF Implementation:

```
python
CopyEdit
def reciprocal_rank_fusion(results: list[list], k=60):
    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
            fused_scores[doc_str] += 1 / (rank + k)
```

- **Each document** gets a **score based on its rank** in each result set.
- **Lower rank = higher score contribution**.
- **Final ranking = sum of all scores**.

◇ Example:

Two Queries Retrieve These Results

Rank	Query 1 Results	Query 2 Results
1	Doc A	Doc C
2	Doc B	Doc A
3	Doc C	Doc B

After Applying RRF, Final Rank is:

1. **Doc A** (appears at rank **1 and 2** → high score).
2. **Doc C** (appears at rank **1 and 3**).
3. **Doc B** (appears at rank **2 and 3**).

◇ Why Use RRF?

✅ **Fuses results from multiple query searches.**
✅ **Prevents missing good documents from single-query failures.**
✅ **More diverse retrieval, leading to better LLM responses.**

5) Full RAG Fusion Pipeline

◇ Final Query Processing Steps:

1. **Expand the query** into multiple related queries.
2. **Perform vector search** for each query.
3. **Use RRF** to **rerank** the retrieved documents.
4. **Pass the best-ranked documents** to the **LLM** as context.
5. **LLM generates the final answer**.

◇ Final Chain Implementation:

```
python
CopyEdit
full_rag_fusion_chain = (
    {
        "context": ragfusion_chain,
        "question": RunnablePassthrough()
    }
    | prompt
    | llm
    | StrOutputParser()
)
```

- **Combines:**
  - ✅ RAG Fusion (retriever)
  - ✅ Prompt Formatting
  - ✅ LLM (Google Gemini)
  - ✅ String Output Parsing

📌 Architecture Diagram

```
plaintext
```

```
CopyEdit
User Query -> Query Expansion (Multiple Queries) -> ChromaDB Retrieval
(Top K docs for each query)
          \_____________________________________________________/
                                |
                    Reciprocal Rank Fusion (RRF)
                                |
                       Contextual LLM Prompt
                                |
                            Google Gemini
                                |
                          Final Response
```

📌 Advantages of RAG Fusion

☑ More Accurate Retrieval

- **Query expansion ensures broader search coverage.**
- **Multiple retrievals reduce missing relevant documents.**

☑ Prevents LLM Hallucination

- **LLM answers only based on retrieved context.**
- **No incorrect or "made-up" facts.**

☑ Optimized Ranking

- **Uses RRF to combine multiple result lists.**
- **Ensures the most relevant documents are ranked higher.**

📌 What Could Be Improved?

🚀 **Hybrid Search** (BM25 + Embeddings) for even better recall.
🚀 **Fine-tune RRF parameters** (k value) for optimal ranking.
🚀 **Integrate External APIs** for real-time fact retrieval.

Would you like **a working Colab notebook** with this setup? 🚀