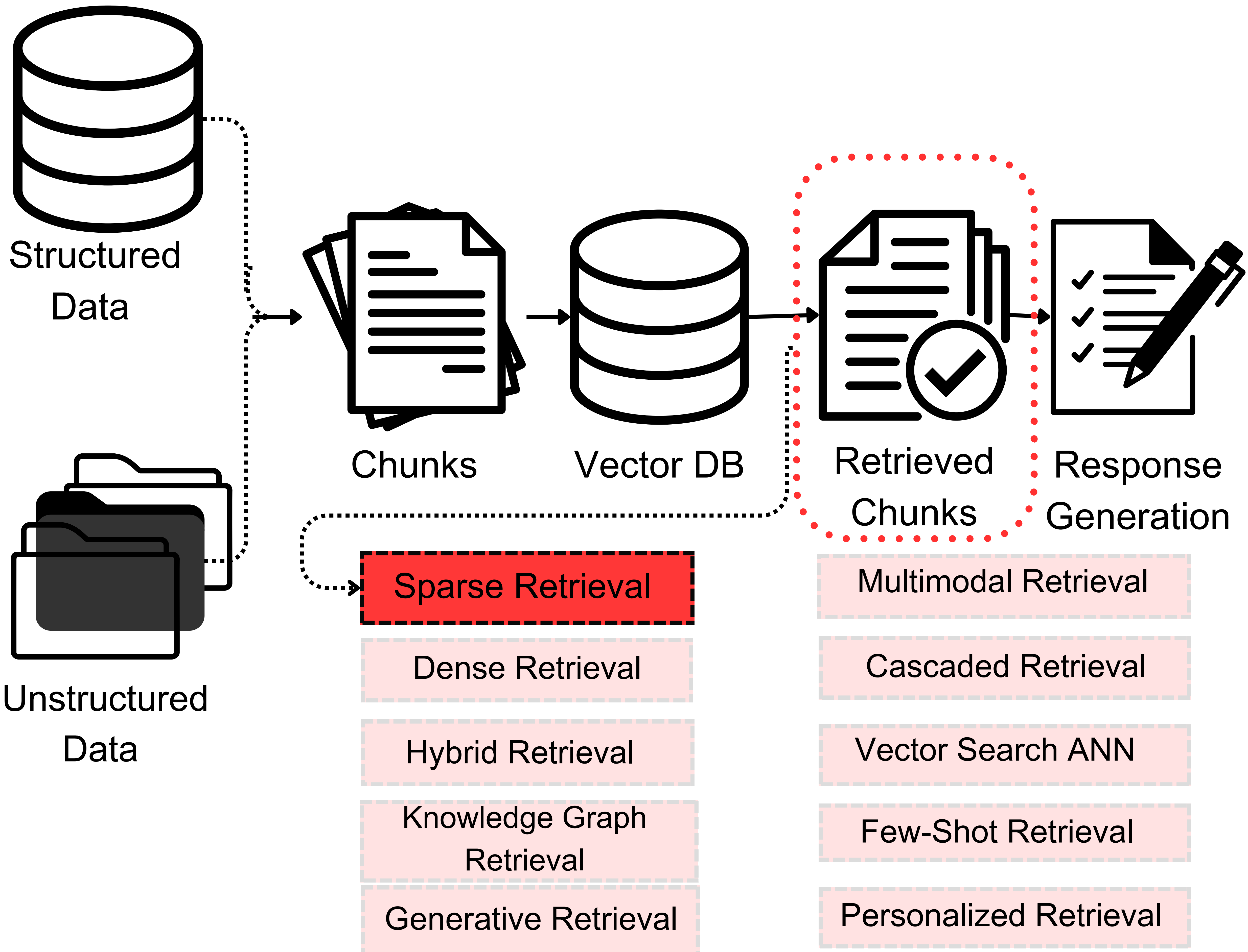
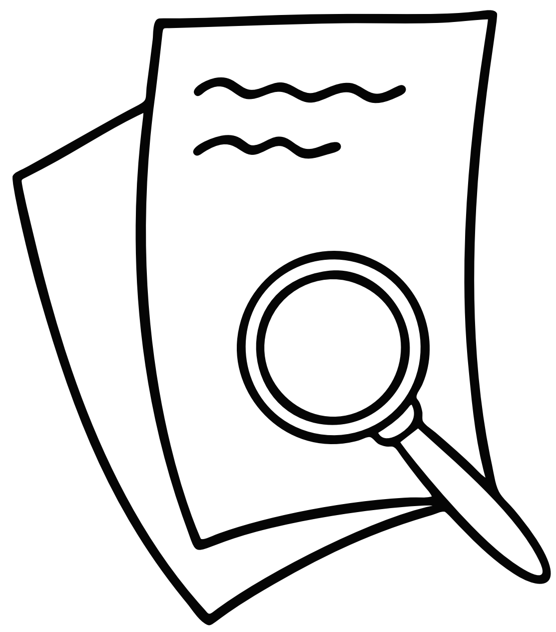
Different Types of

Retrieval

in RAG System



Information Retrieval



Information Retrieval (IR) is the process of finding relevant information from large collections of unstructured data like text, documents, or multimedia.

How it works?

1️⃣ A user query is submitted (e.g., a question or keywords).
2️⃣ The system searches through available data (e.g., documents or embeddings).
3️⃣ Results are ranked and displayed based on relevance.

Why is it important?

1️⃣ Powers RAG-based systems .
2️⃣ Enables context-aware conversational AI.
3️⃣ Supports knowledge-intensive tasks.

Key Components

🔍 Retrieval Models: How the system retrieves data (e.g., Sparse, Dense).
📊 Ranking Algorithms: Ensuring the best results appear first.
🔄 Relevance Feedback: Continuous improvement based on user interaction.

Sparse

Retrieval

Relies on **exact keyword matching** to retrieve relevant documents.

Popular algorithms

- **TF-IDF** (Term Frequency-Inverse Document Frequency)
- **BM25**

How it works?

1. Counts term frequency in the document.
2. Scores based on the importance of terms (e.g., rare terms have higher weight).
3. Returns documents that match the query terms exactly.

Sparse

Retrieval

✅ Pros	❌ Cons
✅ Simple and interpretable. ✅ Low computational cost. ✅ Great for structured or small datasets.	❌ Struggles with synonyms or semantic meaning (e.g., "car" ≠ "automobile"). ❌ Limited to exact term overlap.

Examples →

- Legal document searches.
- Library catalogs.
- FAQ-based retrieval systems.

Process

Query → Keyword Matching → Scoring → Ranked Results

Sparse Retrieval - Best Practices

Optimize Preprocessing → Remove stop words and use stemming or lemmatization.

Choose the Right Algorithm → Use BM25 for weighted keyword matching over plain TF-IDF.

Use Hybrid Retrieval → Combine sparse (keyword matching) and dense (semantic search) for better performance.

Technical Insights

How TF-IDF scores work

Term Frequency (TF) — Counts the occurrences of a word in a document.

Inverse Document Frequency (IDF): — Penalizes common terms (e.g., "the", "is").

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$$

How BM25 improves upon TF-IDF

- Includes term saturation to avoid over-penalizing long documents.
- Allows tuning with parameters like k1 (term frequency scaling) & b (length normalization).

Sparse Retrieval in Python- TF-IDF

Step1

Step 1: Tokenization

First, we need to tokenize the documents.

```python
documents = [
    "The cat sat on the mat",
    "The dog sat on the log",
    "The cat chased the dog"
]

# Tokenize the documents
tokenized_documents = [doc.lower().split() for doc in documents]
print(tokenized_documents)
```

```
[['the', 'cat', 'sat', 'on', 'the', 'mat'], ['the', 'dog', 'sat', 'on', 'the', 'log'],
```

Step2

Step 2: Calculate Term Frequency (TF)

Next, we calculate the term frequency for each term in each document.

```python
from collections import Counter

# Calculate term frequency for each document
tf = [Counter(doc) for doc in tokenized_documents]
print(tf)
```

Python

```
[Counter({'the': 2, 'cat': 1, 'sat': 1, 'on': 1, 'mat': 1}), Counter({'the': 2, 'dog': 1, 'sat': 1, 'on': 1, 'log': 1}
```

Step3

Step 3: Calculate Inverse Document Frequency (IDF)

Now, we calculate the inverse document frequency for each term across all documents.

```python
import math

# Calculate document frequency for each term
df = Counter()
for doc in tokenized_documents:
    df.update(set(doc))

# Calculate IDF for each term
idf = {term: math.log(len(documents) / df[term]) for term in df}
print(idf)
```

```
{'on': 0.4054651081081644, 'cat': 0.4054651081081644, 'the': 0.0, 'sat': 0.4054651081081644, 'mat': 1.0986122886681098,
```

Sparse Retrieval in Python - Tf-idf

Step4

Step 4: Calculate TF-IDF

Multiply TF by IDF for each term in each document.

```
# Calculate TF-IDF for each document
tf_idf = []
for doc_tf in tf:
    doc_tf_idf = {term: freq * idf[term] for term, freq in doc_tf.items()}
    tf_idf.append(doc_tf_idf)
print(tf_idf)
```

```
[{'the': 0.0, 'cat': 0.4054651081081644, 'sat': 0.4054651081081644, 'on': 0.4054651081081644, 'mat': 1.0986122886681098},
```

Step5

Step 5: Query Matching

```
# Tokenize the query
query = "cat sat"
tokenized_query = query.lower().split()

# Calculate term frequency for the query
query_tf = Counter(tokenized_query)

# Calculate TF-IDF for the query
query_tf_idf = {term: freq * idf.get(term, 0) for term, freq in query_tf.items()}
print(query_tf_idf)

# Calculate cosine similarity between query and each document
def cosine_similarity(doc_tf_idf, query_tf_idf):
    dot_product = sum(doc_tf_idf.get(term, 0) * query_tf_idf.get(term, 0) for term in query_tf_idf)
    doc_magnitude = math.sqrt(sum(value ** 2 for value in doc_tf_idf.values()))
    query_magnitude = math.sqrt(sum(value ** 2 for value in query_tf_idf.values()))
    if doc_magnitude == 0 or query_magnitude == 0:
        return 0.0
    return dot_product / (doc_magnitude * query_magnitude)

# Calculate similarity for each document
similarities = [cosine_similarity(doc_tf_idf, query_tf_idf) for doc_tf_idf in tf_idf]
print(similarities)

# Find the most relevant document
most_relevant_doc_index = similarities.index(max(similarities))
print(f"The most relevant document is: {documents[most_relevant_doc_index]}")
```

Output

```
{'cat': 0.4054651081081644, 'sat': 0.4054651081081644}
[0.4397863279651823, 0.21988431639825912, 0.23135443112611218]
The most relevant document is: The cat sat on the mat
```

Sparse Retrieval in Python - Tf-idf vs BM25

Tf-idf

BM25

```python
# Combine the above code into a single script
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np
from rank_bm25 import BM25Okapi

corpus = [
    "The quick brown fox jumps over the lazy dog.",
    "Never jump over the lazy dog quickly.",
    "A quick brown dog outpaces a quick fox.",
    "Lazy dogs are not quick to jump over."
]

# TF-IDF Retrieval
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(corpus)
query = "quick fox"
query_vector = tfidf_vectorizer.transform([query])
cosine_similarities = np.dot(tfidf_matrix, query_vector.T).toarray().flatten()

print("TF-IDF Scores:")
for i, score in enumerate(cosine_similarities):
    print(f"Document {i+1}: {score}")

# BM25 Retrieval
tokenized_corpus = [doc.split(" ") for doc in corpus]
bm25 = BM25Okapi(tokenized_corpus)
tokenized_query = query.split(" ")
bm25_scores = bm25.get_scores(tokenized_query)

print("\nBM25 Scores:")
for i, score in enumerate(bm25_scores):
    print(f"Document {i+1}: {score}")
```

Output

```
TF-IDF Scores:
Document 1: 0.402260298067896
Document 2: 0.0
Document 3: 0.684442866742917
Document 4: 0.16614037331333648

BM25 Scores:
Document 1: 0.9329649710463678
Document 2: 0.0
Document 3: 0.19735198611503196
Document 4: 0.13814639028052236
```

CONGRATULATIONS

You have reached the end, now

If you want to help your network

REPOST THIS

Sarveshwaran R

https://github.com/DataSphereX/Retrieval-Strategies