Docs

# Document Retrieval for Language Models

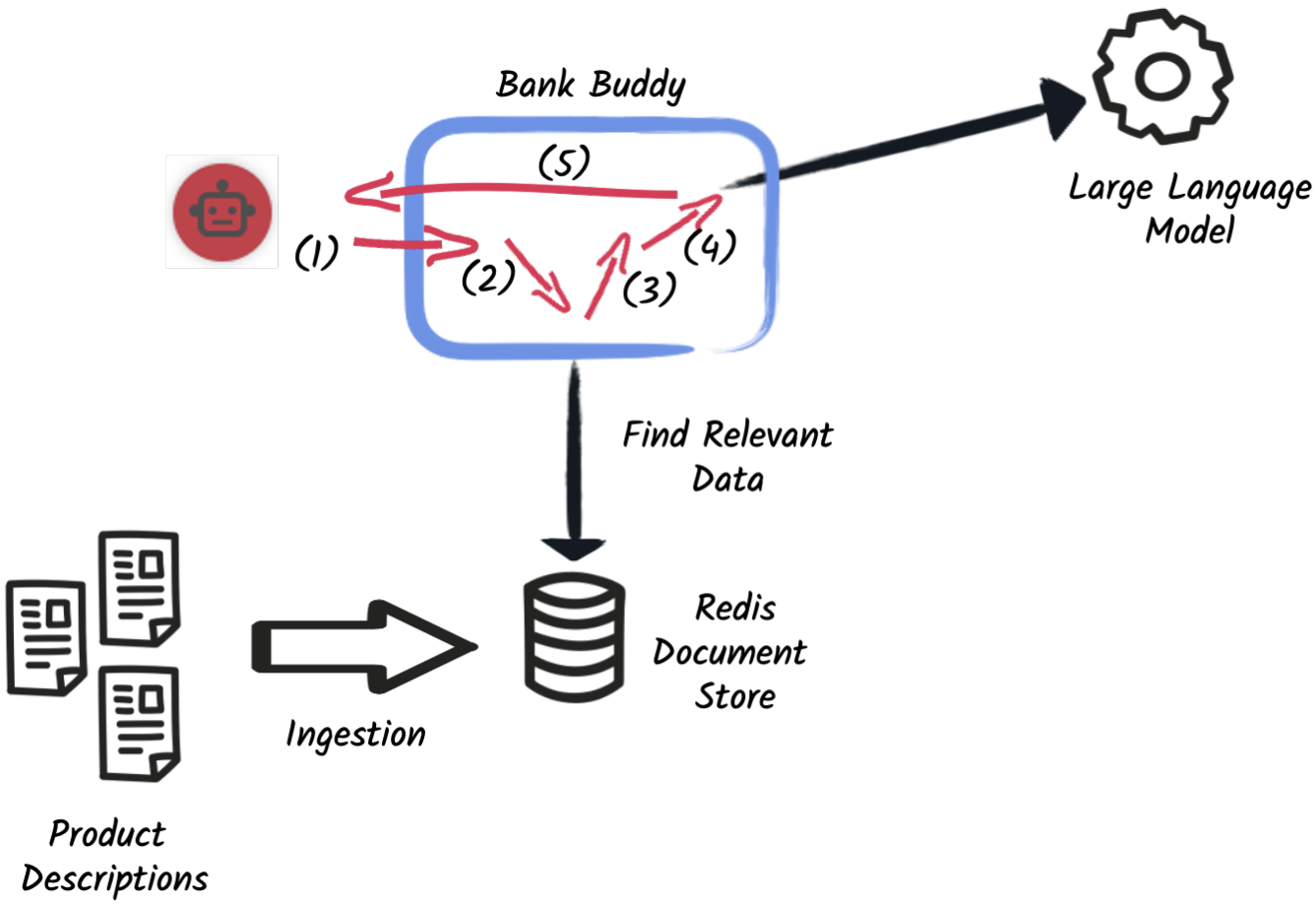
Many applications involving Large Language Models (LLMs) often require user-specific data beyond their training set, such as CSV files, data from various sources, or reports. To achieve this, the process of Retrieval Augmented Generation (RAG) is commonly employed.

## Understanding the Data Ingestion Journey

Before delving into the RAG process, it's crucial to delineate two distinct phases:

*The Data Ingestion Process*
: Involves data collection, cleaning, transformation, and adding metadata, resulting in a set of vectorized documents stored in a database.

*The RAG Process*
: Before engaging with the LLM, it seeks relevant documents in the database and passes them for model augmentation.



## Unveiling Embeddings

Embeddings denote data representations, typically in a lower-dimensional space (arrays of floats), preserving essential characteristics of the original data. In the realm of language or text, word embeddings represent words as numerical vectors in a continuous space.

TIP

*Why Use Vectors/Embeddings?*

Efficiently comparing documents to user queries during the RAG process, a crucial step in finding relevant documents, relies on computing similarity (or distance) between documents and queries. Vectorizing documents significantly speeds up this process.

## The Ingestion Process

The ingestion process varies based on the data source, operating as a one-shot or continuous process, possibly within a data streaming architecture where each message is treated as a document.

To illustrate document creation, the following code exemplifies creating a Document from a text file:

```
package io.quarkiverse.langchain4j.samples;

import static
dev.langchain4j.data.document.loader.FileSystemDocumentLoader.loadDocument;

import java.io.File;

import dev.langchain4j.data.document.Document;
import dev.langchain4j.data.document.parser.TextDocumentParser;

public class DocumentFromTextCreationExample {

    Document createDocument(File file) {
        return loadDocument(file.toPath(), new TextDocumentParser());
    }
}
```

A more complex scenario involves creating a Document from a CSV line:

```
package io.quarkiverse.langchain4j.samples;

import java.util.HashMap;
```

```
import java.util.List;
import java.util.Map;

import dev.langchain4j.data.document.Document;
import dev.langchain4j.data.document.Metadata;

public class DocumentCreationExample {

    Document createFromCSVLine(List<String> headers, List<String> line,
            List<String> columnsToIncludeInMetadata) {
        Map<String, String> metadata = new HashMap<>();
        var content = new StringBuilder();
        for (int i = 0; i < headers.size(); i++) {
            var columnName = headers.get(i);
            var value = line.get(i).trim();

            if (columnName.trim().isEmpty()) {
                continue;
            }

            if (columnsToIncludeInMetadata.contains(columnName)) {
                metadata.put(columnName, value);
            }
            // We compute a Text format for the CSV line: key: value, key: value,
...
            content.append(columnName).append(": ").append(value).append(", ");
        }
        // The \n is added to the end of the content
        return new Document(content.append("\n").toString(),
Metadata.from(metadata));
    }

}
```

Following document creation, the documents need to be ingested. The Quarkus LangChain4j extension offers *ingestor* components for database storage. For instance, quarkus-langchain4j-redis stores data in a Redis database, while quarkus-langchain4j-chroma uses a Chroma database.

The following code demonstrates document ingestion in a Redis database:

```
package io.quarkiverse.langchain4j.samples;

import static dev.langchain4j.data.document.splitter.DocumentSplitters.recursive;
```

```
import java.util.List;

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

import dev.langchain4j.data.document.Document;
import dev.langchain4j.model.embedding.EmbeddingModel;
import dev.langchain4j.store.embedding.EmbeddingStoreIngestor;
import io.quarkiverse.langchain4j.redis.RedisEmbeddingStore;

@ApplicationScoped
public class IngestorExampleWithRedis {

    /**
     * The embedding store (the database).
     * The bean is provided by the quarkus-langchain4j-redis extension.
     */
    @Inject
    RedisEmbeddingStore store;

    /**
     * The embedding model (how is computed the vector of a document).
     * The bean is provided by the LLM (like openai) extension.
     */
    @Inject
    EmbeddingModel embeddingModel;

    public void ingest(List<Document> documents) {
        EmbeddingStoreIngestor ingestor = EmbeddingStoreIngestor.builder()
                .embeddingStore(store)
                .embeddingModel(embeddingModel)
                .documentSplitter(recursive(500, 0))
                .build();
        // Warning - this can take a long time...
        ingestor.ingest(documents);
    }
}
```

Adjust the `documentSplitter` parameter based on the data structure. For instance, for CSV files with document representation separated by `\n`, `new DocumentByLineSplitter(500, 0)` is a recommended starting point.

## Retrieval Augmented Generation (RAG)

Once documents are ingested, they can augment the LLM's capabilities. The following code illus-

trates the creation of a `RetrievalAugmentor`:

```
package io.quarkiverse.langchain4j.samples;

import java.util.function.Supplier;

import jakarta.inject.Singleton;

import dev.langchain4j.model.embedding.EmbeddingModel;
import dev.langchain4j.rag.DefaultRetrievalAugmentor;
import dev.langchain4j.rag.RetrievalAugmentor;
import dev.langchain4j.rag.content.retriever.EmbeddingStoreContentRetriever;
import io.quarkiverse.langchain4j.redis.RedisEmbeddingStore;

@Singleton
public class RetrievalAugmentorExample implements Supplier<RetrievalAugmentor> {

    private final RetrievalAugmentor augmentor;

    RetrievalAugmentorExample(RedisEmbeddingStore store, EmbeddingModel model) {
        EmbeddingStoreContentRetriever contentRetriever =
EmbeddingStoreContentRetriever.builder()
                .embeddingModel(model)
                .embeddingStore(store)
                .maxResults(3)
                .build();
        augmentor = DefaultRetrievalAugmentor
                .builder()
                .contentRetriever(contentRetriever)
                .build();
    }

    @Override
    public RetrievalAugmentor get() {
        return augmentor;
    }

}
```

NOTE

This is the simplest example of retrieval augmentor, which only uses a `EmbeddingStoreContentRetriever` to retrieve documents from an embedding store to pass them directly to the LLM. A retrieval augmentor can use more sophisticated strategies to process queries, such as query compression, splitting a query into multiple queries and then routing them via different content retrievers (which may or may not be based on vector storage, but for example on a full-text search engine), using a scoring model to further filter the retrieved results, etc. For more information about advanced RAG strategies, refer to https://docs.langchain4j.dev/tutorials/rag/.

The example above is a CDI bean that implements `Supplier<RetrievalAugmentor>`. An alternative way to wire things up is to create a CDI bean that directly implements `RetrievalAugmentor`, for example via a CDI producer, and letting Quarkus auto-discover it (by not specifying the `retrievalAugmentor` parameter of the `@RegisterAiService` annotation).

The EmbeddingStoreContentRetriever necessitates a configured embedding store (Redis, Chroma, etc.) and an embedding model. Configure the maximum number of documents to retrieve (e.g., 3 in the example) and set the minimum relevance score if required.

Make sure that the number of documents is not too high (or document too large). More document you have, more data you are adding to the LLM context, and you may exceed the limit.

An AI service does not use a retrieval augmentor by default, one needs to be configured explicitly via the `retrievalAugmentor` property of `@RegisterAiService` and the configured `Supplier<RetrievalAugmentor>` is expected to be a CDI bean.

---