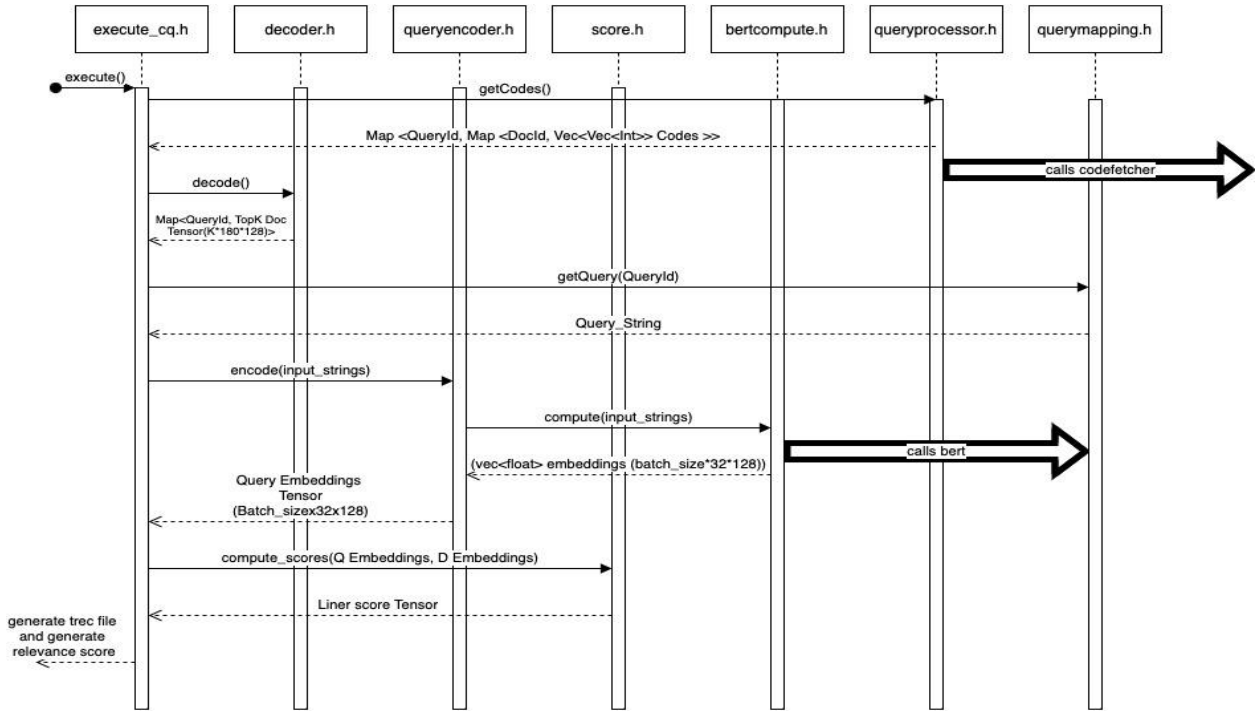
Developer Documentation

Code Architecture



codefetcher.cc and codefetcher.h

Purpose

These files implement the CodeFetcher class, responsible for fetching document embeddings from a document store and managing related data structures. It is utilized to retrieve compressed document embeddings from a document store.

Key Components

CodeFetcher class: This class manages the fetching and processing of embeddings.
Member variables: The class has various member variables to store file pointers, code data, and other information.

Constructors and Destructor: The class provides a constructor for initialization and a destructor to release resources.
get_codes function: This function retrieves document codes and their associated embeddings.
File I/O: The code fetches embeddings from binary files, processes document keys and offsets, and manages memory for document embeddings.

CodeFetcher Class
The CodeFetcher class is responsible for fetching document embeddings and managing related data structures. It is utilized to retrieve uncompressed embeddings from a document store. The CodeFetcher class also includes methods to read from multiple binary files, parse document key and offset information, and handle in-memory code storage.

Member Variables

- total_docs: An integer that keeps track of the total number of documents in the store.
- base_filename: A string containing the base filename for document storage.
- number_of_files: An integer specifying the number of binary files used for storage.
- key_offset_store: An unordered_map that stores document keys and their corresponding offsets.
- file_ptrs: A vector of file pointers to binary files containing embeddings.
- codes_store: An unordered_map that stores document codes for quick access.

Constructor

```C/C++
CodeFetcher::CodeFetcher()
```

The constructor initializes the CodeFetcher object and sets up data structures for fetching and processing document embeddings.

Destructor

```C/C++
CodeFetcher::~CodeFetcher()
```

The destructor releases allocated resources, including file pointers and data structures.

get_codes Function

```
C/C++
unordered_map<string, vector<vector<int>*>*>*
CodeFetcher::get_codes(vector<string>* document_ids)
```

This function retrieves document codes and their associated embeddings. It takes a vector of document IDs as input and returns an unordered map that maps document IDs to their respective embeddings.

document_ids: A pointer to a vector of strings containing document IDs for which codes and embeddings are to be retrieved.

Developers can utilize the CodeFetcher class to fetch document embeddings, read binary data, and store codes efficiently. The class offers the get_codes function to obtain embeddings for a list of document IDs, which is crucial for downstream processing in the project.

decoder.cc and decoder.h

Purpose:

These files implement the Decoder class, which is responsible for decoding compressed embeddings, loading codebooks, and processing embeddings. It plays a vital role in generating document approximations for user queries.

Key Components:

Decoder class: This class initializes and manages the decoding process.
Member variables: The class has member variables to store codebook data, static embeddings, and composition layer information.
Constructors and Destructor: The class provides a constructor for initialization and a destructor to release resources.
decode function: This function processes embeddings and generates document approximations.
Composition Layer: The code includes a composition layer for generating document approximations.

Member Variables

- vocab_size_: An integer representing the vocabulary size.
- dimension_size_: An integer specifying the dimension size for embeddings.
- pad_token_id_: An integer indicating the padding token ID.
- M_: An integer representing the number of codebooks.
- K_: An integer representing the number of codes.
- codebook_dim_: An integer specifying the dimension of the codebook.
- doc_maxlen_: An integer representing the maximum document length.
- static_embeddings: A pointer to a Torch tensor for storing static embeddings.
- non_contextual_embedding: An instance of a Torch Embedding model for non-contextual embeddings.
- codebook: A pointer to a Torch tensor for storing the codebook.
- composition_layer: An instance of a Torch Linear model used for composition layers.

Constructor

```
C/C++
Decoder::Decoder()
```

The constructor initializes the Decoder object, setting up various member variables and loading essential components, such as static embeddings, codebooks, and composition layers.

Destructor

```
C/C++
Decoder::~Decoder()
```

The destructor releases allocated resources, including the non_contextual_embedding and codebook data.

decode Function

```
C/C++
map<int, map<string, torch::Tensor>*>* Decoder::decode(unordered_map<int,
unordered_map<string, vector<vector<int>*>*>*>* fetched_codes)
```

This function is the core of the decoding process. It decodes document codes, generates document approximations, and returns a map of query IDs and their corresponding approximated embeddings.

fetched_codes: An unordered map containing query IDs mapped to document codes and their associated embeddings.

Decoding Process

The Decoder class decodes document codes, combines them with static embeddings, and applies composition layers to generate document approximations. This process is essential for contextual quantization and improving the retrieval of relevant documents.

Developers can use the Decoder class to decode embeddings and generate approximations for documents associated with user queries.

execute_cq.cc and execute_cq.h

Purpose

These files implement the ExecuteCQ class, which orchestrates the execution of a contextual quantization approach. It encompasses various components like decoding, encoding, scoring, and result output.

Member Variables

decoder_: An instance of the Decoder class responsible for decoding and generating document approximations.
query_encoder_: An instance of the QueryEncoder class used for encoding query strings.
score_: An instance of the Score class responsible for computing scores between queries and documents.
query_mapping_: An instance of the QueryMapping class for mapping query IDs to query strings.
query_processor_: An instance of the QueryProcessor class for processing queries and fetching document codes.

Constructor

```
C/C++
ExecuteCQ::ExecuteCQ()
```

The constructor initializes the ExecuteCQ object and sets up the necessary components for contextual quantization, such as the Decoder, QueryEncoder, Score, QueryMapping, and QueryProcessor.

Destructor

```
C/C++
ExecuteCQ::~ExecuteCQ()
```

The destructor releases allocated resources, including the instances of the components mentioned above.

execute Function

```
C/C++
void ExecuteCQ::execute()
```

This function is the core of the contextual quantization approach. It encodes queries, fetches document codes, generates document approximations, computes scores, and outputs results to a TREC file.

Execution Workflow

The ExecuteCQ class follows a well-structured workflow:

Query Encoding: Encodes a list of input strings by encoding them into a query tensor using the QueryEncoder. It prepares query strings for scoring.

Document Code Retrieval: Fetches the top-K documents and their approximations from the Decoder. Document codes and embeddings are retrieved.

Score Computation: Computes scores between the query tensor and document embeddings using the Score class. The scores are stored in a map.

Result Output: Results, including query IDs, document IDs, ranks, and scores, are written to a TREC file.

Developers can use the ExecuteCQ class to run the complete contextual quantization workflow, from query encoding to result output.

querymapping.cc and querymapping.h

Purpose

These files implement the QueryMapping class, which is responsible for mapping query IDs to their corresponding queries.

Member Variables

queryMapping: An unordered map that stores the query mapping, with query IDs as keys and query strings as values.

Constructor

```C/C++
QueryMapping::QueryMapping()
```

The constructor initializes the QueryMapping object, reads the query mapping from a file, and populates an unordered map with the query ID as the key and the query string as the value.

Destructor

```C/C++
QueryMapping::~QueryMapping()
```

The destructor releases allocated resources, including the query mapping data.

getQuery Function

```C/C++
string QueryMapping::getQuery(int queryId)
```

This function retrieves the query string associated with a given query ID.

queryId: An integer representing the query ID for which the query string is to be retrieved.
Query Mapping
The QueryMapping class plays a crucial role in contextual quantization by providing a mechanism to map query IDs to their corresponding query strings. This mapping is essential for result output and analysis.

Developers can use the QueryMapping class to access query strings based on query IDs, facilitating the output of results with human-readable query information.

Please let me know if you need more information or detailed documentation for specific functions or if you have any questions related to this code.

queryprocessor.cc and queryprocessor.h

Purpose:

These files implement the QueryProcessor class, which handles the processing of user queries and fetching document codes. It coordinates the retrieval and processing of codes for a batch of queries.

Member Variables

code_fetcher: An instance of the CodeFetcher class used for fetching document codes and embeddings.
queryResults: An unordered_map that stores query results, mapping query IDs to vectors of document IDs.

Constructor

```
C/C++
QueryProcessor::QueryProcessor()
```

The constructor initializes the QueryProcessor object, creates an instance of the CodeFetcher class, loads query results, and prepares for processing queries.

Destructor

```
C/C++
QueryProcessor::~QueryProcessor()
```

The destructor releases allocated resources, including the CodeFetcher instance and query result data.

getCodes Function

```
C/C++
unordered_map<int, unordered_map<string, vector<vector<int>*>*>*>*
QueryProcessor::getCodes(int offset)
```

This function retrieves document codes and their associated embeddings for a batch of queries. It takes an offset value to specify the starting point for batch processing and returns an unordered map containing query IDs mapped to document codes and embeddings.

offset: An integer specifying the offset that determines where the batch processing should begin.

The QueryProcessor class processes queries in batches by fetching codes and embeddings for multiple queries at once. It manages the interaction between query results and the CodeFetcher, making it a crucial component for the contextual quantization approach.

Developers can use the QueryProcessor class to fetch and process document codes for a specific set of queries, optimizing the overall workflow.