Project 4: DocuQuest

Due: 11:00 PM Tuesday, March 17

Revision: March 8, 10:45 PM

Table of Contents

Introduction

In this project you will build a program that lets you "chat with your documents." Imagine you have a bunch of documents and you wish to ask questions about their contents. We will use an LLM like ChatGPT to help us answer these questions. The important thing to know is that an LLM can sound confident even when it is wrong, especially if you ask it to answer from its own general knowledge. So your program will treat the LLM like a smart writer that needs to be given source material first, like an open-book quiz. The goal is this: Given a question from the user, find the right set of documents that contain relevant materials, then send these documents to the LLM along with the original question, then ask it to answer using only the text you provided.

We will be providing you with dozens of documents about fictitious archaeological sites which your program will have to answer questions about. Rather than sending all of the documents to the LLM to review (which is slow and expensive), we want to find a small set of relevant documents based on the user's question and send those to the LLM so it may answer using just the minimal necessary information required. To find the right

documents quickly, your program starts by building a special lookup table called an *inverted index* which is like a fancy STL map. To do so, your project will read every text file we provide and record, for all words across all files, which text files contain that word. It is similar to the index at the back of a textbook: “jade” might point to pages 12 and 47. In our project, your index will associate each word with the set of text files that contain that word. For example, after building your index you might have `jade` map to `{glyphmonkey_stelae.txt, marchalign_megaliths.txt}` and `sapphire` map to `{oasiscolumn_city.txt}`. This lets you answer “Which documents mention X?” instantly, instead of re-reading every file every time.

Now the key missing step: Users often ask questions using general words that might not appear in the documents. If the user asks, “What gems were discovered in archaeological sites mentioned in the documents?”, the documents might never use the exact word “gems.” They might list specific gems like "ruby” or “lapis lazuli” instead. So when the user poses a question, your program will first ask an LLM to suggest concrete search words that are likely to appear in text files, based on the user’s question. To do so, we'll send a request like this to the LLM: "The user is asking the following question: 'What gems were discovered in archaeological sites mentioned in the documents?' Please give me a set of search terms that I might find in documents which would identify those documents as being relevant for that question."

For the “gems” question, the LLM might suggest: “jade", "ruby", "sapphire", "emerald", and "lapis lazuli". Then your program could use the inverted index to retrieve all documents that contain any of those suggested phrases, gather the text of those documents, and send the user’s original question plus that gathered text back to the LLM to request a final answer.

This overall pattern is called retrieval-augmented generation (RAG): retrieve relevant context first, then generate an answer grounded in that context, which makes the final response much more likely to be based on the content of your documents instead of guesses or hallucinations.

In project #4, you will build a set of classes that work alongside our provided code (which knows how to talk to an LLM) to implement this Q&A system.

Once you finish, your code will work with our provided driver (main.cpp) to produce answers interactively.

Background Information

How we use an LLM in this project?

An LLM is a neural-network-based system that can follow instructions in a prompt and generate a response. In project 4, we will use it for two jobs:

- Term generation: Given the user's question, propose a small list of search terms that are likely to appear in relevant documents. We'll use those terms identified by the LLM to search for relevant documents that contain those terms.
- Answer generation: Given the user's question plus the full text of relevant documents (discovered via the search terms), produce a concise answer.

LLMs today have a critical limitation: They might hallucinate, meaning they sometimes produce confident-sounding statements that are not actually supported by evidence. One reason is that, by default, an LLM answers using patterns from its training data, and that "internal knowledge" can be incomplete, out of date, or simply the wrong fit for your specific question.

A reliable way to reduce this is to provide the model with the relevant source material directly in the prompt, like excerpts from the documents you want it to use. When the needed facts are in the context window (i.e., directly in the chat with the question being asked), the model can base its answer on that text instead of guessing. For this project, that means your second prompt should explicitly instruct the LLM to answer the user's question using only the provided documents, and to avoid introducing outside information. Writing a really effective "answer from sources only" prompt usually takes a bit of iteration and testing, so expect to experiment before you find wording that consistently produces strong results.

In a later section, we'll explain how you can interact with a cloud-based LLM and send it prompts and get back results.

What is an inverted index?

An inverted index is a data structure that maps each term (word) to the list of documents (and often positions) where that term appears. Instead of doing a linear search for terms across each document each time to answer every user question, before answering any questions, your program will build a data structure that looks like this:

- "temple" → {docs/marblegoddess_temple.txt, docs/artemisplendor_ruins.txt}

- "columns" → {docs/artemisplendor_ruins.txt, docs/oasiscolumn_city.txt}
- "jade" → {docs/glyphmonkey_stelae.txt, docs/marchalign_megaliths.txt, docs/marblegoddess_temple.txt}
- "sapphire"→ {docs/oasiscolumn_city.txt}
- …

So given a word like "jade", your program can quickly determine all documents that contain that word. Similarly, for a multi-word search like `["temple", "jade"]`, we can look up each word in our index separately, and then take the intersection of the documents found for each word to find documents that contain *all* of these words. For example, since temple was found in:

- docs/marblegoddess_temple.txt
- docs/artemisplendor_ruins.txt

And jade was found in:

- docs/glyphmonkey_stelae.txt
- docs/marchalign_megaliths.txt
- docs/marblegoddess_temple.txt

The only document we care about is one that contains *both* terms, which is docs/marblegoddess_temple.txt, so the index would return this as a hit for a search for `["temple", "jade"].`

Stuff We Provide

We are providing several source files for your use. You may use them but MUST NOT modify them.

- `provided.h` and `provided.cpp`: These contain base classes from which you must derive your classes. They also contain class "factory" functions as well as code to query an LLM hosted at openrouter.ai. More on these items below.
- `main.cpp`: This contains a simple driver program that you can use to test your classes.

Base Classes

We are giving you the following abstract base classes:

- `MultimapBase` (with nested public class `IteratorBase`)
- `TokenizerBase`
- `IndexBase`
- `AgentBase`

You must implement your classes (Multimap/Iterator, Tokenizer, Index, Agent) for this project by deriving them from our provided base classes. Failing to do so will make your project ungradable and you will receive a zero.

Factory Functions

A class factory is just a function whose job is to create an object for you and return it through a base-class pointer or reference. For example, instead of writing `Tokenizer t;` or `Tokenizer* p = new Tokenizer(...)` directly in your code, you will allocate a tokenizer object like this:

```
TokenizerBase* b = create_tokenizer();  // allocates a new object and returns it
...
delete b;
```

The key idea behind a class factory is you can ask it to construct a new object for you and return it. The returned object is guaranteed to be derived from the specified base class (e.g., TokenizerBase), but the code that uses the returned object does not need to know the exact concrete class that it is actually using (e.g., Tokenizer). It only knows "I'm getting an object back that behaves like a TokenizerBase." That forces you to write your code against the abstract interface defined in the base class, not the derived class.

We use factories so the classes you write can be swapped out easily during grading with our versions of the same classes. Say you wrote an Index class which needs to use your Tokenizer class, and further that your Tokenizer class has a nasty bug. Let's say your index class works like this:

```
class Index {
public:
  ...
private:
  string helper(const string& s) {
    TokenizerBase* b = create_tokenizer(); // uses the factory
```

```
    b->tokenize(s);
    ...
    delete b;
  }
};
```

While you're developing your project, the create_tokenizer() method will instantiate and return a Tokenizer object using your buggy Tokenizer class:

```
TokenizerBase* create_tokenizer() {
  Tokenizer* t = new Tokenizer; // instantiates your buggy tokenizer object
  return t;
}
```

However when we test your code, we can swap out your buggy Tokenizer class code for a version of the class that we wrote (that we know is 100% correct) and then test your Index class with our tokenizer:

```
TokenizerBase* create_tokenizer() {
  // instantiates our correct Tokenizer, also derived from TokenizerBase
  OurPerfectTokenizer* t = new OurPerfectTokenizer;
  return t;
}
```

During testing, our create_tokenizer() function will return an OurPerfectTokenizer object based on our class and use it to test your Index.

Let's say your Index code is 100% correct but you have a bug in your Tokenizer class. In this case, we can determine that your Index works properly and give you full credit for it by using your Index class with our OurPerfectTokenizer class. Without the factory, if your Tokenizer had a bug, then for some of our tests of Index, your Index class might behave incorrectly because the Tokenizer it uses behaves incorrectly, and it wouldn't be able to pass those tests.

Thus, in all of the classes you write, if they need to use one of your other classes, they MUST instantiate those other classes via the factories we provide in provided.h and provided.cpp.

- `TokenizerBase* create_tokenizer();`
- `IndexBase* create_index();`
- `AgentBase* create_agent(const IndexBase& index);`
- `MultimapBase* create_multimap();`

Furthermore, your Multimap, Tokenizer, Index, and Agent classes MUST NEVER refer directly to your other Multimap, Tokenizer, Index, and Agent types by those names. Rather, they must only refer to objects of those types via pointers and references to base classes: MultimapBase, TokenizerBase, IndexBase, and AgentBase.

LLM Query Function

We provide you with pre-written code that can connect to an LLM hosted by openrouter.ai in provided.h and provided.cpp:

```
bool query_llm(const std::string& category, const std::string& prompt,
               std::string& response);
```

This function:

1. Sends the user's prompt to a remote LLM service over the Internet.
2. Receives the LLM's response text back.
3. Stores that text in the *result* parameter and returns true on success or false on error.

Here's an example of how to use query_llm():

```
std::string answer;
if (query_llm("summarize",
              "What archaeological sites have jade columns?", answer)) {
    // answer now contains the LLM's response
}
```

In the provided query_llm() function, the *category* parameter specifies what task you are using the LLM for (this is to help us when we test your project). In our reference implementation, you MUST pass only one of two options:

- `"terms"` - this indicates that your prompt is using the LLM to generate search terms
- `"summarize"` - this indicates that your prompt includes the user's question along with the text from a set of relevant documents, and is using the LLM to summarize a final answer for the user

The *prompt* parameter is the prompt you want to send to the LLM, e.g., "What is the capital of California?"

This *response* parameter will be filled with the LLM's response: "The capital of California is Sacramento."

API keys

To authenticate to the LLM service, `query_llm()` reads an API key from a local file. By default, this will be a file named `.orkey` in the same folder where you run your program's executable file. You can change this (perhaps because you want to give a full path to the file if your program is not finding it) by calling the provided function `set_api_key_filename` with the path (e.g., `set_api_key_filename("C:/CS32/P4/.orkey")` or `set_api_key_filename("/Users/fred/.orkey")`. You would call this function once, before the first time you call `query_llm()`.

- We will send you an API key, which is a long string of digits/letters/symbols.
- You must put our provided API key on the first line in the key file, with nothing else in the file.

If the API key file is missing or empty, `query_llm()` will fail. The API key we provide has been loaded with enough credits for you to query the LLM several hundred times. If you run out of credits you will be responsible for any additional fees, so use your queries sparingly.

Document Files to Search

We will provide a directory full of 40 text documents for your program to index and answer questions from, e.g.:

> **cliffnoodle_dwellings.txt**
>
> Cliffnoodle Dwellings: Ancient Apartments in the American Southwest
> Tucked beneath massive sandstone overhangs in the canyons of southwestern Coloradonia, the Cliffnoodle cliff dwellings represent some of North America's best-preserved archaeological sites. Built and occupied by Ancestral Pueblonians between 600 and 1300 CE, these remarkable structures offer a window into a sophisticated ancient culture.
>
> …

These documents contain details about actual historical sites but we have changed the names of these sites (e.g., Machu Picchu -> Skibbledewidget). This prevents the LLM from answering questions about these sites based on its training data and limits its answers to data in the prompt.

Classes You Must Implement

You must provide implementations for the following four classes. Each one must be derived from our corresponding base class in `provided.h`.

1. `Multimap` (derived from `MultimapBase`) and its Iterator derived from IteratorBase
2. `Tokenizer` (derived from `TokenizerBase`)
3. `Index` (derived from `IndexBase`)
4. `Agent` (derived from `AgentBase`)

We will test your code by compiling it together with our provided source files. Your classes must match the required interfaces exactly and must not add other public members.

1. Multimap (Derived from MultimapBase)

You must implement a binary search tree-based multimap that stores a mapping from each string key to one or more unique string values. In this project, it is primarily used to map:

- term → one or more document-paths

Your multimap must support retrieving all values for a given key through a custom iterator.

Required public behavior

Your derived Multimap class must implement:

- Construction & destruction
  - You may rely on compiler-generated constructors/destructors if appropriate, but you must not leak memory.
- `void put(const std::string& key, const std::string& value)`
  - Associates a specified key with the specified value.
  - The multimap must not include duplicates, so if the key/value pair is already in the multimap, this function does nothing.
- `IteratorBase* get(const std::string& key) const`
  - Returns a pointer to a dynamically-allocated Iterator object (derived from `IteratorBase`) that can be used to iterate over all values associated with that key. The client is responsible for deleting the iterator object.
  - If the key does not exist in the multimap, return a pointer to an Iterator that is an "invalid iterator" as defined below (i.e., one that immediately reports no values).
- `bool empty() const`
  - Returns `true` if the multimap has no key/value pairs.
- `int size() const`

- Returns the total number of stored key/value pairs.

Iterator requirements

Your Iterator class must implement:

- Iterator(...)
  - You may implement one or more constructors, with zero or more parameters as you wish.
- ~Iterator()

  If relying on the compiler generated destructor is inappropriate, you must implement a destructor.
- `bool next(std::string& value)`
  - If the iterator points at an item in the multimap, then set the `value` parameter to that item's value, advance the iterator, and return `true`.
  - If no more values exist, return `false` without changing `value`.

We define an “invalid iterator” to be one for which calling `next()` immediately returns `false`. The iterator class does not need to iterate through the values associated with the specified key in any particular order.

Example Usage

```
MultimapBase* mm = create_multimap();

mm->put("carey", "silly");
mm->put("carey", "boring");
mm->put("carey", "silly");     // duplicate value: ignored
mm->put("carey", "awesome");

MultimapBase::IteratorBase* it = mm->get("carey");

std::string value;
while (it->next(value))
    std::cout << value << "\n";  // prints: boring, awesome, silly
                                 // not necessarily in that order
delete it;
delete mm;
```

Efficiency requirements

Your multimap implementation must be efficient enough to index a folder of many documents.

- `put()` and `get()` must run in O(log N+V) average time, where N is the number of distinct keys in the multimap and V is the average number of values associated with each key.

STL restrictions

When creating this class, you must adhere to the following STL restrictions:

- Your `Multimap` class must not use any associative STL container (`std::map`, `std::unordered_map`, `std::multimap`, `std::set`, etc.).
- You may use `std::string`, `std::vector`, and `std::list`, and STL algorithms and simple utilities (e.g. `std::pair`).

2. Tokenizer (Derived from TokenizerBase)

A tokenizer breaks a string into a sequence of “terms” suitable for indexing and searching.

Required public behavior

- `void tokenize(const std::string& input)`
  - Resets the tokenizer so we can tokenize the input string from the start.
- `bool next(std::string& token)`
  - Stores into the `token` parameter the next token from string that's being tokenized and returns `true`.
  - Returns `false` if no tokens remain.

Tokenizing rules

Your tokenizer must follow these rules exactly (so everyone’s index behaves the same):

- A token is a sequence of one or more alphanumeric characters.
  - Letters `A-Z`, `a-z`, and digits `0-9` are allowed inside tokens.
  - All other characters (e.g., spaces, tabs, punctuation) are considered delimiters and must not be included in a token.
- Alphabetic characters in tokens MUST be converted to lowercase.
- Digits are preserved as is.

Example:

- Input: `"Hello, CS32!!! Go 2 LA's top-rated UCLA."`
- Tokens produced (in order): `"hello"`, `"cs32"`, `"go"`, `"2"`, `"la"`, `"s"`, `"top"`, `"rated"`, `"ucla"`

3. Index (Derived from IndexBase)

Your `Index` class stores an *inverted index* of terms to documents. It must use your Multimap class in a meaningful way, via the create_multimap() factory. It may also use `std::vector`, `std::list` and `std::set` but not `std::map`, `std::unordered_map` or `std::multimap`.

Required public behavior

- `int build_index(const std::string& path)`
  - Indexes all files in the folder named by the specified `path` (you must use our provided `get_filenames(path)` helper function in provided.h to get a vector of the file names).
  - For each file, open the file and read its entire contents.
  - Tokenize the contents of each file using your Tokenizer.
  - For each token `t` in a file, insert an association between `t` and the name of the file into your index data structures.
  - Return the number of files indexed.
- `void add_doc(const std::string& doc_file)`
  - Open the file named by the parameter and read its entire contents.
  - Tokenize the contents of the file using your Tokenizer.
  - For each token `t` in the file, insert an association between `t` and the name of the file into your index data structures.
- `std::vector<std::string> query(const std::vector<std::string>& terms) const`
  - Return the document names that contain **all** terms in the `terms` vector.
  - The returned vector must contain no duplicate documents.
  - The order of documents in the returned vector does not matter.

Your `Index` must not “know” about or use concrete classes other than itself.

- It must obtain tokenizers from create_tokenizer() and store them using TokenizerBase*, not Tokenizer*.
- It must obtain a multimap from create_multimap() and store it using MultimapBase*, not Multimap*.

This lets us test your code with our implementations.

Efficiency expectations

- Building an index should be O(T * (log M + V)) on average, where T = total tokens processed, M = number of unique tokens, V = average number of unique values per token.
- Querying should be O(K * (log M + V)) where K is the number of tokens in the query. In addition, there may be intersection overhead to identify documents with *all* query terms.

4. Agent (Derived from AgentBase)

The Agent class is the "brain" of your document question-answering system. It orchestrates a technique called Retrieval-Augmented Generation (RAG) — a multi-step process where you first find and **retrieve** relevant documents, then ask an AI to **generate** answers to questions using the text in those documents.

Think of it like a research assistant: When you ask a question, they first search through a library to find relevant books, then read those books to give you an answer. The Agent class automates this process using a Large Language Model (LLM).

Here are the high-level steps your Agent class will take, given a question/query from the user:

1. Construct a prompt to ask an LLM to evaluate the user's question (e.g., "What tombs are there where gold was discovered?") and come up with one or more sets of search terms that would likely be found inside of documents about that topic (e.g., ["gold", "discovered"], ["gold", "tomb"], ["gold", "burial", "site"]).
2. Send the prompt to the LLM to get its recommended search terms.
3. Search the index to locate documents that have the terms. Then we load the text of those relevant documents into memory.
4. Construct a second prompt which poses the original user question along with the text loaded from the relevant documents, and asks the LLM to summarize an answer based on the question and the included source material.
5. Send the second prompt to the LLM and get its answer.
6. Return its answer to the user.

Agent Class Methods

Your Agent class must inherit from the AgentBase class (in `provided.h`). This base class defines the interface your class must implement. As with the other classes you write, your Agent class must only interact with other components through their base class types and must create them using factory functions.

Here are the methods you need to implement

Agent::Agent(const IndexBase& index)

Your Agent constructor receives a reference to an IndexBase object. A user of the Agent class creates and populates an index, and passes it to the constructor. (The Agent doesn't create the index.) Your constructor will need to save the index for later use by the `query()` method.

bool Agent::query(const string& question, std::string& answer)

The query() method is where the magic happens. It takes a user's question and produces an answer. Here's the high-level flow:

Step 1: Generate Search Terms

Goal: Convert the user's natural language question into search terms the index can search for.

1. Take your loaded terms template (from terms.txt; see below for more details).
2. Produce a copy of the terms template, replacing the {query} placeholder in the template with the text of the `question` parameter, which is the user's actual question (e.g., "Where was gold found?").
3. Send this filled-in prompt to the LLM using the provided query_llm() function.
4. Parse the LLM's response to extract one or more sets of search terms.

The LLM will ideally return something like:

gold discovered
golden objects
glittering artifacts buried

Each line represents a set of terms. Your query() function needs to:

- Split the LLM response into lines.
- Tokenize each line (using your Tokenizer) to get individual terms.
- Store these as groups of terms.

So the above becomes:

- ["gold", "discovered"]
- ["golden", "objects"]
- ["glittering", "artifacts", "buried"]

A document is considered a match if it matches all the terms in one or more of the groups of terms.

Step 2: Search the Index

Goal: Find documents that match your search terms.

For each group of search terms from Step 1:

1. Query the index with that group of terms.
2. The index returns the names of documents containing *all* terms in that group.

Collect all matching document names (the union of all the document name groups)

Example:

- Query ["gold", "discovered"] → returns {doc1.txt, doc3.txt}
- Query ["ancient", "artifacts"] → returns {doc2.txt, doc3.txt}
- Combined set of relevant documents: {doc1.txt, doc2.txt, doc3.txt}

What if no documents are found with the search terms? Your method must return false.

Step 3: Generate the Answer

Goal: Use the LLM to synthesize a final answer from the retrieved documents.

1. Given the documents identified in the previous step, load the contents of each matching document file into strings.
2. Take your loaded summarization template (from summarize.txt; see below for more details).
3. Produce a copy of the summarization template, replacing `{query}` in the template with the user's question, e.g., "Where was gold discovered?".
4. Suppose there were a sorted sequence of the names of all the relevant documents that were discovered in Step #2. From that sorted sequence, select the first 10 names (or all the names if there are fewer than 10).
5. With those alphabetically-ordered selected names, replace `{documents}` in the template with the *complete* contents of each of the files named, with a newline after each document's text, including after the last document's text.
6. Send this filled-in prompt string to the LLM using query_llm().
7. The LLM will respond with an answer. This response is your final answer — store it in the `answer` output parameter.

Return Value

- Return true if you successfully generated an answer.
- Return false if:
    - Prompts weren't loaded
    - No matching documents were found
    - An LLM query failed

bool Agent::load_prompts(const std::string& terms_file, const std::string& summarize_file)

This method loads your two template files into memory:

1. Read the contents of the file named by `terms_file` into a data member.
2. Read the contents of the fine named by `summarize_file` into a data member.
3. Return true if both files were successfully read, false otherwise

These loaded templates will be copied and customized each time query() is called.

What is an LLM Prompt?

A prompt is simply text you send to an LLM (like ChatGPT or Claude) to get a response. The quality of the response depends heavily on how well you craft your prompt. For example, if you ask an LLM:

> *What is the capital of France?*

It will respond with something like "Paris."

But prompts can be much more sophisticated. You can give the LLM detailed instructions, context, and formatting requirements. For example:

> *You are a helpful assistant. Given the following documents, answer the user's question. Use only information from the documents. If the answer isn't in the documents, say so.*
>
> *USER'S QUESTION: What gems were found at the Dumwiddle excavation?*
>
> *RELEVANT DOCUMENTS:*
> *The archaeologists discovered jade artifacts at the northern site...*
>
> *Ruby and emerald jewelry was uncovered in the royal tomb...*
>
> *Please answer in 200 words or less.*

Template Prompt Files

We will provide you with two text files that contain simple, pre-written LLM prompts for your Agent class to use to perform its question answering. These prompts have "placeholders" in them where data such as the user's question must be inserted. After loading these text files into memory, your Agent class will need to "fill in the blanks" in these prompts to customize them every time it answers a question, as described below.

terms.txt (Search Term Generation)

This text file contains simple instructions to the LLM to convert a user's natural language question into one or more sets of search terms that can be used to locate relevant documents in your index.

Here's what the terms.txt file might look like:

> *Given the following question provided by a user, generate up to 20 sets of search terms, one set per line. On each line, you must have 1 to 4 words that must all be present in a document for it to be considered relevant.*
>
> *Here is the user's question: {query}*
>
> *Now output each set of search terms on a separate line, just the words separated by spaces. Nothing else.*

Notice that the above file has a placeholder {query}. Your job in the Agent class will be to make a copy of the prompt text and replace every occurrence of {query} with the user's question, such as "Where was gold found?". So after customizing the prompt text for this question, it would look like this:

> *Given the following question provided by the user, generate up to 20 sets of search terms, one set per line. On each line, you must have 1 to 4 words that must all be present for a document for it to be relevant.*
>
> *Here is the user's question:* Where was gold found?
>
> *Now output each set of search terms on a separate line, just the words separated by spaces. Nothing else.*

Given such a prompt, the LLM might then respond with:

> gold found
> gold discovered
> buried gold

Each line represents a set of search terms that will be used to query your document index. For the example above, a document would be relevant if it contains both "gold" and "found" anywhere in the document, or it has both "gold" and "discovered", or both "buried" and "gold". But if a document simply has "gold" but not "found", "discovered" or "buried" then it would not be a valid match.

summarize.txt (Answer Synthesis)

The second prompt text file is meant to instruct the LLM to synthesize a final answer. It uses two placeholders:

- {query} — as before, this must be replaced with the user's question
- {documents} — replaced with the contents of relevant documents that contained search terms identified by the LLM using your earlier prompt

Here's what the summarize.txt file might look like:

> *Based on the following documents, please answer the user's question.*
>
> *USER'S QUESTION: {query}*
>
> *TEXT FROM RELEVANT DOCUMENTS:*
> *{documents}*
>
> *Please provide a concise answer based only on the documents above.*

After customizing the prompt text for this question, it would look like this:

> *Based on the following documents, please answer the user's question.*
>
> *USER'S QUESTION:* Where was gold found?
>
> *TEXT FROM RELEVANT DOCUMENTS:*
> *The Apadana, the great audience hall, could accommodate 10,000 people beneath a roof supported by 72 columns, each 60 feet tall. The stairway reliefs depict delegations from 23 nations bearing distinctive gifts: lions and gold from Africania, camels from the eastern regions, horses from the steppes. These carvings provide invaluable evidence for the ethnic diversity and material culture of the ancient world.*
>
> *The Sacred Cenote, a natural sinkhole 200 feet in diameter, served as a site for offerings and human sacrifices. Dredging has recovered thousands of objects including jade, gold, pottery, and human remains, many showing signs of violent death.*
>
> *The site was identified by treasure-hunter Heinrich Shovelpants in 1876, who famously telegraphed, "I have gazed upon the face of Agamemblob" upon discovering a golden death mask in a royal shaft grave. Though archaeologists now doubt this mask belonged to the legendary king who led forces against Troy, the finds confirmed that Mycenoodlean civilization was no mere myth.*
>
> *Please provide a concise answer based only on the documents above.*

Customizing Your Prompts

The quality of your system depends heavily on your prompt templates. Here are tips:

For terms.txt:

- Be specific about output format (e.g., one set of terms per line)
- Encourage synonyms and variations
- Tell the LLM to think about different terms that might be found in documents that relate to the user's question
- Limit the number of term sets (too many = slow, too few = might miss documents)

For summarize.txt:

- Instruct the LLM to use only information from the provided documents. This prevents "hallucination" (making up facts)
- Set a word limit to keep answers concise
- Ask for direct, factual answers

Experimentation is Key

The prompts in the `terms.txt` and summarize.txt files we provide are just starting points for you. Test your system with various questions and refine your prompts based on results. Consider:

- Does the LLM generate good search terms?
- Are the right documents being retrieved?
- Is the final answer accurate and well-sourced?

While your score on this project will NOT depend on the quality of your queries (since even good queries can produce bad results with an LLM), you should use this as a learning opportunity to get better at prompting LLMs.

Our Provided Driver Program (`main.cpp`)

Our `main.cpp` does the following:

1. Initializes an index object and uses it to index all of the provided documents.
2. Initializes an agent object, passing in that index.
3. Repeatedly asks the user for a question, calls the agent to answer it (using the LLM), and prints the agent's answer.

The string constant `DOCUMENT_DIRECTORY` in `main.cpp` is the name of the directory containing the documents that our test driver looks for; as given, it's a directory named `docs` in the same folder where you run your program's executable file. You may change that to a full path to the directory if your program is not finding it.

A typical run looks like:

```
Loading documents...
Indexed 42 documents.
Enter question (or 'quit' to exit): In what sites was gold or gems found?

Search terms from LLM:
gold sites
rubies found
diamonds found
gems buried
golden

Documents matching all search terms:
docs/goldenmask_citadel.txt
docs/stonewobble_enclosure.txt
docs/wibbleston_hoo.txt
docs/zippityscrolls.txt

Gold and gems were found in several significant archaeological sites.
At the Goldenmask Citadel in Mycenoodle, southern Greebleland, ancient
treasures including gold masks, jewelry, weapons, and ornaments were
discovered within the walls, with over 35 pounds of gold found in Shaft
Graves of Grave Circle A. This gold, along with other artifacts like
hunting-scene inlaid daggers, indicates a civilization rich in metals
and artistic craftsmanship. Additionally, recent excavations have
uncovered chamber tombs with golden treasures, further revealing the
citadel’s hidden riches. Further south, in Great Stonewobble,
Zimbloobia, the complex featured walls made of million granite blocks
and included a conical tower, housing evidence of gold trade and
sophisticated African kingdom. Notably, the site also yielded gold
ornaments, demonstrating the kingdom’s wealth and trade networks
reaching distant lands. Finally, at Wibbleston Hoo in Snufflefolk,
Blibberland, a ship burial was rediscovered holding a vast collection
of Wango-Saxon artifacts, such as a gold belt buckle and a ceremonial
whetstone, which included garnets and glass. This burial chamber
provided evidence of trade and the sophistication of the Wango-Saxon
society during the era often labeled as the "Dark Ages."
```

You may write your own test main (for your own development), but you will not turn it in.

Project Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

1. Do not modify `provided.h`, `provided.cpp`, or `main.cpp` (except possibly to change the `DOCUMENT_DIRECTORY` constant in `main.cpp`). We will not see your local changes to those files when grading.
2. Your derived classes must never directly refer to your other derived classes. They must refer to the provided base classes instead.

Incorrect (do not do this):

```
class Agent {
  const Index& m_index; // BAD
};
```

Correct:

```
class Agent {
  const IndexBase& m_index; // GOOD
};
```

3. Use the factories. If you need a tokenizer, call `create_tokenizer()`; do not directly construct a `Tokenizer` object in code.
4. LLMs are not deterministic. On your machine, answers may vary from run to run even for the same question. Our grading will not evaluate the *wording* of the LLM's final answer just in case it screws up; it will evaluate whether your program:
    - has correct versions of Multimap, Index, Tokenizer and Agent,
    - generates term queries correctly,
    - retrieves the correct documents based on terms,
    - generates summarization queries correctly,
    - handles failures and edge cases.

To support deterministic grading, we will replace the provided `query_llm()` with a test version that returns fixed outputs.

What to Turn In

Turn in a zip file containing these eight files. Note that no report is required. 🎉

- `multimap.h` and `multimap.cpp`
- `tokenizer.h` and `tokenizer.cpp`
- `index.h` and `index.cpp`
- `agent.h` and `agent.cpp`

None of `multimap.h`, `tokenizer.h`, `index.h`, and `agent.h` header files are allowed to include others in that list (e.g., `index.h` must not include `multimap.h`). They should include our `provided.h` file instead (which, for example, provides MultimapBase and IndexBase).

Each cpp file may include its corresponding header file (e.g., `multimap.cpp` may include `multimap.h`) and our `provided.h` file but none of the other header files in the above list (e.g., `index.cpp` must not include `multimap.h`).

Do not turn in:

- `provided.h`, `provided.cpp`, `main.cpp`
- any test drivers you write
- the `.orkey` file
- a report file

Grading

Correctness will be the primary factor. Do your Multimap, Tokenizer, Index, and Agent classes correctly exhibit the specified behavior for normal situations? Are they robust, dealing with unusual situations (empty inputs, missing files, LLM failures, strange formatting of the LLM's response to a "terms" category query) as specified, or if not mentioned in the spec, gracefully. Are there no memory leaks?

About 10% of your score will be for efficiency: Each correctly implemented function among `Multimap::put`, `Multimap::get`, `Index::build_index`, and `Index::query` that meets its stated big-O requirement on realistic inputs will contribute toward this 10%.

Partial credit: because we grade using base classes and factories, if a component of your solution is incomplete or incorrect, we may replace it with our correct version when we test your other components so that it doesn't cause them to fail.

Good luck, and have fun building your tiny document oracle!