File Structure

The file structure of my dictionary encoder is organized in a way that makes it easy to manage and understand the different components. In the **bin/** directory, I store the compiled executable files that are generated after building the project. The **data/** folder contains important input and output files, such as Column.txt (the raw data) and encoded_column.txt (the encoded version of the data). The **include/** directory holds header files like dictionary_encoder.h and multi_threading.h, which define the functions and structures used across the project. These headers are included in the source files to promote modularity and code reuse.

The **Makefile** outlines the build process for the project, specifying how the source code should be compiled and linked. In the **obj/** directory, I keep the object files (.o) that are compiled from the source code. These files are later linked to produce the final executable. The **README.md** file provides the necessary documentation, explaining how to set up and use the project.

Inside the **results/** directory, I store the results of performance tests and queries in text files such as encoding_results.txt, performance_results.txt, and query_results.txt. The **src/** directory contains all the source code files and is organized by functionality. The **encoding/** subdirectory includes files like dictionary_encoder.cpp and multi_threading.cpp for handling the dictionary encoding process. The **performance/** subdirectory contains performance_test.cpp, which is responsible for running encoding performance tests. The **query/** subdirectory has files like query_handler.cpp, query_utils.cpp, and vanilla_search.cpp, which handle the querying and searching within the encoded dictionary.

Finally, I have **temp_encoded.txt** and **temp_results.txt**, which are temporary files used for storing intermediate results during the encoding and testing process. This structure allows me to keep everything organized and makes it easier to maintain and expand the project in the future.

Encoding

The dictionary encoding process in my project works as follows: I start by reading the input data from a file, typically `Column.txt`, line by line, storing each line in a vector called `data`. This vector holds all the raw input values that need to be encoded. To improve performance, I use a multi-threaded approach to handle the dictionary encoding efficiently.

The function `dictionary_encode_multi_threaded()` divides the input data into chunks based on the number of threads specified. Each thread is assigned a subset of the data to process. The data is split into nearly equal parts, and each thread independently processes its assigned chunk by calling the function `process_data_chunk()`. Within this function, a local dictionary is created for each chunk, which maps each unique string in that chunk to a unique ID.

To ensure thread-safety when updating the global dictionary, I use a mutex. This prevents multiple threads from modifying the global dictionary at the same time, which could cause data corruption. Once all threads have processed their chunks and merged their local dictionaries into the global one, the dictionary is complete, with each distinct string in the dataset assigned a unique ID.

After constructing the dictionary, the `dictionary_encode()` function populates the `encoded_data` vector. This vector holds the numeric IDs that correspond to the original strings in the input data. The strings in the original data are replaced by their respective IDs from the dictionary.

The encoding results are written to the `encoding_results.txt` file. This includes the dictionary entries, with each string and its corresponding ID, as well as the frequency of occurrences for each unique string. Additionally, the function logs any duplicate entries, noting the indices where they appear in the original data.

The encoded data in the form of IDs is then written to the `encoded_column.txt` file, providing the compressed representation of the original data. Finally, the process is completed, and a message is displayed in the console, indicating that the dictionary encoding is finished and the results have been saved to the output files.

Query

In this implementation, I focused on optimizing the performance of string matching operations by utilizing advanced processor instructions that allow multiple string comparisons to be handled in parallel. These optimizations are particularly useful when working with large datasets, as they significantly reduce the time required for string searches.

The code handles two types of string queries: exact match and prefix match. For both types of queries, the solution is designed to compare multiple characters at once rather than one by one, which greatly improves efficiency. By processing multiple bytes in a single instruction, the search operations are sped up, especially when the dataset contains long strings or a high volume of data.

For exact match queries, the implementation first checks if the queried string and the stored data have the same length. If they do, it compares chunks of the strings in parallel, and if any mismatch is found within a chunk, the function returns false. This parallel comparison ensures that the check is performed much faster than a traditional, one-character-at-a-time comparison. Similarly, for prefix match queries, the solution compares the beginning portion of the string with the target prefix in parallel, efficiently determining if the string starts with the given prefix.

The approach ensures that, even for larger strings, the matching process remains fast. When comparing strings, the system processes 32 characters at once, handling large portions of the string with a single operation. This allows for quick exits when mismatches are detected, avoiding the need to scan every character individually. Additionally, if the string is smaller or if SIMD instructions are not applicable, the solution defaults to a standard string comparison method.

Results - Encoding and Query

Encoding

Encoding Tests

Threads	Time (ms)
1	4
2	11
4	3
8	3
16	3
20	2

The results from the encoding tests show an initial increase in time when moving from 1 thread to 2 threads, with the time rising from 4 milliseconds to 11 milliseconds. This increase may be due to the overhead of managing two threads, such as thread synchronization and context switching. While using multiple threads typically offers performance benefits, in this case, the extra complexity of managing parallel tasks might outweigh the gains in parallelization, leading to a slower overall execution time when the workload is split across two threads.

However, as the number of threads increases beyond 2, the time begins to decrease. At 4, 8, and 16 threads, the time stabilizes at 3 milliseconds, suggesting that the encoding process becomes more efficient as it can be divided into multiple parallel tasks. This indicates that the workload has a significant level of parallelism, and the system can handle more threads without introducing additional overhead. At 20 threads, the time further decreases to 2 milliseconds, suggesting that the system has effectively utilized its available resources, with the encoding process running faster as more threads are employed. This result indicates that the optimal number of threads for this encoding task is between 4 and 20, where the system maximizes parallel processing efficiency without excessive overhead.

Query

Query Performance Analysis

Query Type	Query Term	Indices	Time (µs)
Exact Match Vanilla	uyadc	357	22 µs
Exact Match Vanilla	ikjsyqehs	818	26 µs
Exact Match Vanilla	yafcyuc	958	26 µs
Exact Match Vanilla	odbeyulzt	135	23 µs
Exact Match Vanilla	wzmbpd	18	29 µs
Exact Match Vanilla	edwdyswjf	945	26 µs
Exact Match Vanilla	vbtjs	554	26 µs
Exact Match Vanilla	mzsirlq	93	26 µs
Exact Match Vanilla	qwlarkmbw	967	28 µs
Exact Match No SIMD	uyadc	357	3 µs
Exact Match No SIMD	ikjsyqehs	818	4 µs
Exact Match No SIMD	yafcyuc	958	4 µs
Exact Match No SIMD	odbeyulzt	135	3 µs
Exact Match No SIMD	wzmbpd	18	3 µs
Exact Match No SIMD	edwdyswjf	945	3 µs
Exact Match No SIMD	vbtjs	554	3 µs
Exact Match No SIMD	mzsirlq	93	3 µs
Exact Match No SIMD	qwlarkmbw	967	3 µs
Prefix Match Vanilla	uya	357	27 µs

Prefix Match Vanilla	ikj	818	30 µs
Prefix Match Vanilla	yaf	958	39 µs
Prefix Match Vanilla	odb	135	23 µs
Prefix Match Vanilla	wzm	18	26 µs
Prefix Match Vanilla	edw	945	38 µs
Prefix Match Vanilla	vbt	554	31 µs
Prefix Match Vanilla	mzs	93	33 µs
Prefix Match Vanilla	qwl	967	23 µs
Prefix Match No SIMD	uya	357	23 µs
Prefix Match No SIMD	ikj	818	24 µs
Prefix Match No SIMD	yaf	958	22 µs
Prefix Match No SIMD	odb	135	24 µs
Prefix Match No SIMD	wzm	18	22 µs
Prefix Match No SIMD	edw	945	25 µs
Prefix Match No SIMD	vbt	554	22 µs
Prefix Match No SIMD	mzs	93	22 µs
Prefix Match No SIMD	qwl	967	22 µs

Based on the query results, here's an analysis of the performance for both Exact Match and Prefix Match queries, distinguishing between the Vanilla (unencoded) approach and the SIMD-encoded versions.

Exact Match Queries

In the Exact Match category, we can clearly see a significant difference in performance between the Vanilla (unencoded) and No SIMD (encoded) versions. For all the queries tested, the Exact Match Vanilla approach consistently takes longer than the Exact Match No SIMD approach. For instance, the query `uyadc` takes 22 µs in the Vanilla approach, but it only takes 3 µs when SIMD is applied. This pattern holds true for all the other queries, with Vanilla taking more time than No SIMD. The speed improvement is substantial, ranging from 19 µs to 26 µs faster for the No SIMD implementation. The efficiency boost here is likely due to the SIMD optimizations that allow for parallel processing and more efficient data handling.

Prefix Match Queries

Similarly, the Prefix Match queries show a clear performance advantage for the No SIMD version. For instance, the query `uya` takes 27 µs in Vanilla, but only 23 µs when SIMD is used. Across all tested queries, the No SIMD approach performs faster than Vanilla, with time improvements ranging from 4 µs to 16 µs, depending on the query term. Even though the time differences are not as large as those observed in Exact Match queries, the No SIMD implementation still outperforms the Vanilla method. The reason for this improvement is likely related to the more efficient handling of prefixes using SIMD, even if the query term is not an exact match.

Summary

Overall, the **SIMD-encoded** versions (whether for Exact Match or Prefix Match) outperform the **Vanilla (unencoded)** versions by a significant margin. The **No SIMD** approach is also considerably faster than Vanilla, but the improvements are not as dramatic as with SIMD-encoded methods. SIMD encoding provides enhanced efficiency by allowing the system to process multiple operations in parallel, which is especially useful for large datasets or when queries require repeated operations. In summary, SIMD optimization consistently yields faster query processing times, particularly for exact matches, and the No SIMD-encoded results show that encoding optimizations, even without SIMD, are still much more efficient than the unencoded Vanilla approach.