Introduction

The purpose of this project is about Dictionary encoding, as this concept is a cornerstone in contemporary data analytics in the real world. This technique operates by mapping unique data values to compact, digit representations. By assigning each unique value in a dataset to a dictionary key and replacing occurrences of the value with the corresponding key, this method significan;y reduces storage requirements, particularly for datasets with lots of repetition or large files. This approach not only optimizes memory usage, but also improves performance in tasks such as data retrieval, query processing, and analytics, further proving it as a cornerstone of efficient data management.

This project focuses on implementing a high performance dictionary encoding designed for large text files. It supports three modes: dictionary encoding, vanilla pas-through, and SIMD instructions for query. The core of this project lies in its efficient multithreaded implementation, which leverages a producer-consumer model to process input files in chunks, dynamically distribute workloads across threads, and ensure thread-safe dictionary updates. Additional optimizations include the use of thread-local storage for minimizing contention and scalable task distribution via dynamic chunking. The system supports encoding data, querying values, and benchmarking performance under different configurations. This project aims to demonstrate the practical benefits of dictionary encoding in terms of both storage efficiency and query acceleration while providing a robust framework adaptable to various real-world datasets.

In this document, we will conduct experiments that benchmark the performance of dictionary encoding and its query modes against the vanilla baseline. The units for the performance are being measured in the execution time. In addition, we will also test how the number of threads affects the dictionary process and how SIMD integrations influences query performance. As a disclaimer, while my code is not the best, I have tried my best to get the results as best I can within the tight time frame I have.

Experimental Setup

There are 2 files that are primarily used throughout the project. When compiling the Dict_enc, you must use this:
g++ -std=c++17 -pthread -o {output name} {filename}
An example of compiling is what I used throughout this project:
g++ -std=c++17 -pthread -o test.out Dict_enc.cpp

The 1st file “Dict_enc.cpp” is the code that is primarily used to enter the input file and then encode the file into an output file using 2 modes: Dictionary encoding and vanilla. When trying to run the code, here’s the general syntax:
./test.out

As mentioned, you can choose 2 modes which are the dictionary and vanilla mode. In dictionary mode, unique strings in the input are assigned unique integer IDs and encoded for compression, while vanilla mode simply processes and writes lines to the output as-is. The code's structure is well-organized around a producer-consumer model. A producer thread reads the input file in chunks, dynamically adds them to a thread-safe queue, and notifies consumer threads. Multiple worker threads process the chunks, either encoding the data in dictionary mode or directly writing lines in vanilla mode. Each worker thread uses a thread-local dictionary to minimize contention, which is later merged into a global dictionary. To ensure scalability and safety, synchronization mechanisms like mutexes, condition variables, and atomic operations are used for dictionary updates and queue management. The dictionary encoding and encoded data are outputted in dictionary mode, while vanilla mode outputs unmodified lines. The program demonstrates the effectiveness of multithreading by dividing tasks dynamically among threads and measuring encoding/processing time for performance evaluation.

When compiling the 2nd one, use the statement below because SIMD requires the -mavx2 flag to compile:
g++ -mavx2 -o Q_test.out query.cpp

The 2nd code “query.cpp” is the code that primarily enables users to query an existing encoded column file. Once you enable query you can choose either option 1-6 where:

1. check whether one data item exists in the column, if it exists, return the indices of all the matching entries in the column
2. given a prefix, search and return all the unique matching data and their indices.

Here is the general syntax when running the code:

./Q_test.out query <encoded_file> <query_mode> <query_value>"

- <query_mode>: Enter the number which you want to run
    - 1: Exact Match (Query).
    - 2: Prefix Match (Query).
    - 3: Exact Match (Vanilla).
    - 4: Prefix Match (Vanilla).
    - 5: SIMD Exact Match.
    - 6: SIMD Prefix Match.
- <query_value>: is the key you wanna search/scan

As you can see, there's an <encoded_file>, which means you have to run the "Dict_enc.cpp" to get the encoded text file.

For in general the gist of this file is this implementation of a dictionary-based encoding system with functionality for querying encoded data using exact and prefix matches. The program reads an encoded file containing a dictionary and a sequence of encoded data, which it stores in memory using a global std::unordered_map for reverse lookups and a std::vector for the encoded data. The file is parsed by distinguishing between the dictionary and encoded data sections, with the dictionary mapping integers to strings. Users can query the loaded data via the query_encoded_file function, which supports two query modes: exact match and prefix match. Exact match queries find all indices of the encoded data corresponding to a specific string, while prefix match queries locate strings in the dictionary that start with a given prefix and list their indices. Timing functionality is incorporated using the std::chrono library to measure query performance. Additionally, SIMD operations via <immintrin.h> can be integrated for faster data processing in future optimizations, particularly to accelerate scanning operations or matching tasks, making the code well-suited for high-performance scenarios. The program is invoked via the command line with arguments specifying the command (query), file path, query mode, and query value. It ensures robust error handling for invalid inputs and missing files, making it efficient and user-friendly.

Performance and Analysis:

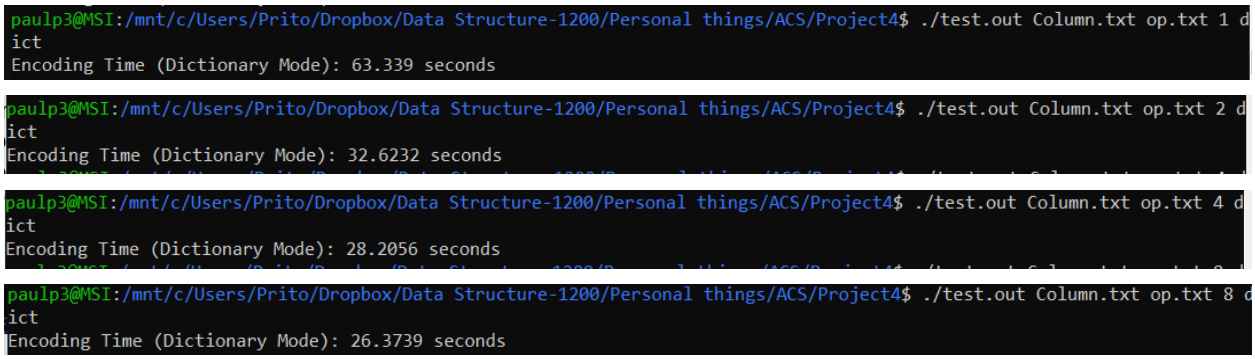
For this project, we are given a 1GB text file to use for our self-designed dictionary encoding and query methods as a way to test our code under heavy conditions. For our experiments, we will first test and analyze under the conditions for encoding during multi-trending against vanilla base. Then, the 2nd experiment will be the search/scan test for 3 algorithm comparison: Query with SIMD, without SIMD and vanilla base.

Encoding speed performance under different # of threads

For this experiment the encoding speed performance will be under the condition of different threads counts: 1, 2, 4, and 8 vs the vanilla base. As I'm sure you know what multithreading is as you have seen previous projects about multi-threading. The oversimplification of it is the parallelization of performing multiple tasks at once. This parallelization should therefore, significantly reduce the overall processing time.

Table 1: Performance for different # of threads

# of threads	1	2	4	8
Latency (s)	63.339	32.6232	28.2056	26.3739

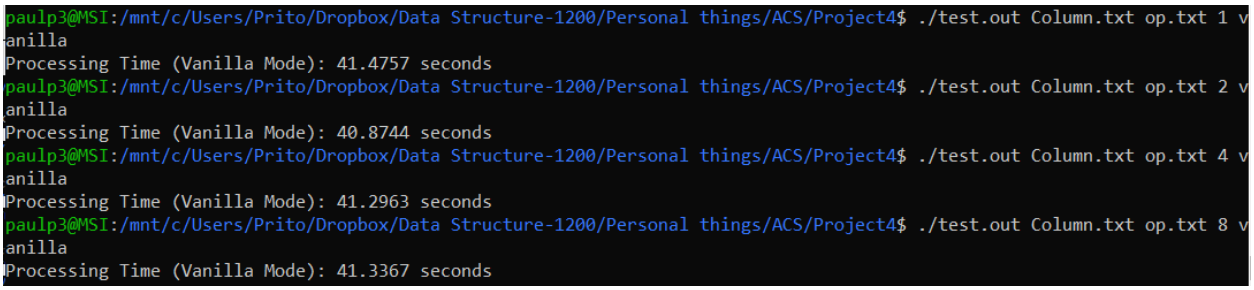


From the results of the latency for dictionary encoding on Table 1 demonstrates the efficiency of multithreading in reducing processing time. With an increasing number of threads, the latency significantly dresses showcasing the scalability of dictionary encoding. As such, this makes sense as we have mentioned that parallelism would different the latency required. The results highlight the effective parallelism of dictionary encoding.

Table 2: Performance on Vanilla base:

# of threads	1	2	4	8

Latency (s)	41.4757	40.8744	41.2963	41.3367



The performance for the vanilla baseline (Table 2) shows significantly less scalability compared to dictionary encoding. The latency remains relatively constant as the number of threads increases. This behavior suggests that the vanilla baseline is likely I/O-bound or lacks significant computational overhead, making it less amenable to parallel processing.

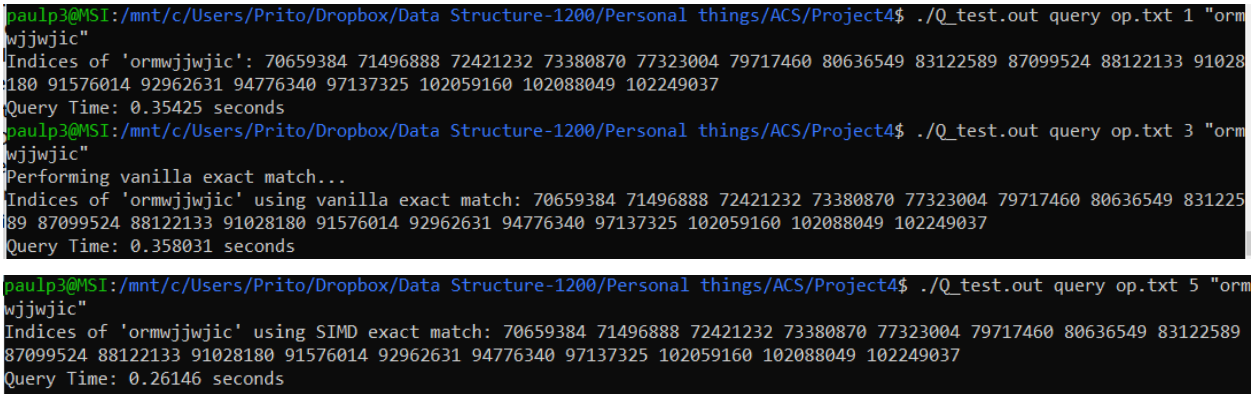
Comparison:
When comparing the two approaches, dictionary encoding shows a clear advantage in utilizing multi-threading to improve performance. While vanilla processing is simpler and incurs lower initial latency with fewer threads, it lacks the scalability and efficiency of dictionary encoding, which capitalizes on parallelism to handle larger datasets effectively.
These results validate the superior adaptability of dictionary encoding for multi-threaded environments, particularly for workloads requiring complex transformations or large-scale data handling.

Single data and prefix scan speed performance

As we know from previous projects, SIMD optimizations are usually the fastest algorithm to use for various cases. The algorithm can perform parallel comparisons with multiple characters at once, which significantly reduces the search/scan speed. In the Search Exact match experiment, the image that is shown is a test case where I will be searching for the word “ormwjjwjic” and seeing the indices on where it took place. For the Scan Prefix match experiment, I will not be including the images due to the fact that when I finish running the program, it will output a large text of the prefix match of both the word and the index. Therefore, if you wanna see an example, you can run the code yourself, but I did include a small example of what it could look like if your cursitoy.

Table 3: Speed performance for exact match:

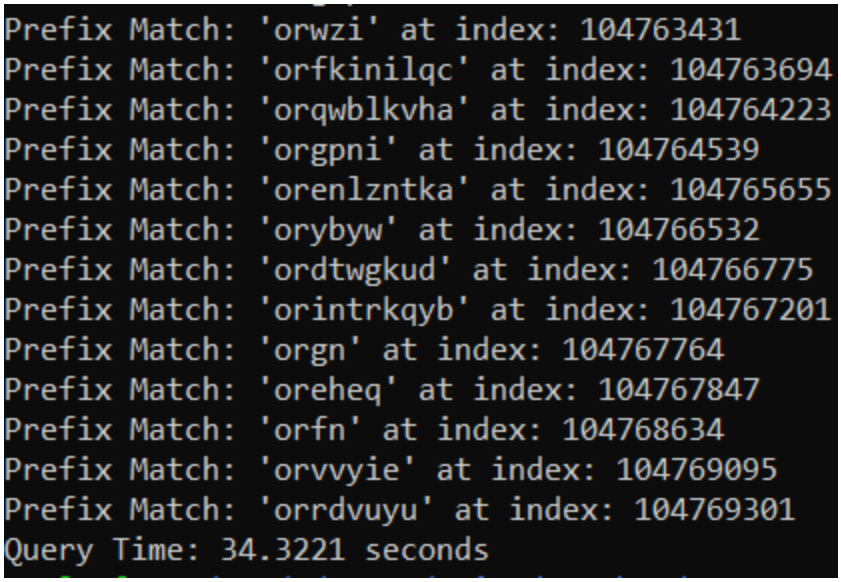
Search Exact	Query	Vanilla	Query + SIMD
Latency (s)	0.35425	0.358031	0.26146



Looking at the result from Table 3, we can the speed performance is almost similar but still really fast. The Query + SIMD implementation demonstrates a clear performance improvement over both Vanilla and Query methods. SIMD optimizations excel in exact match scenarios due to the ability to process multiple comparisons simultaneously, which reduces the computational overhead significantly. The Query being slight faster does affect my prediction that the rank would be Query + SIMD > Query > Vanilla, but the fact that Query is extremely close to Vanilla tells me that there is an issue in the timing implementation or just the way my code isn't efficient. The Vanilla method, being straightforward, is expected to provide baseline performance. The Query implementation, ideally, should perform slightly faster or equal to Vanilla.

Table 4: Speed performance for prefix match:

Scan Prefix	Query	Vanilla	Query + SIMD
Latency (s)	31.8686	34.4695	34.3221



From Table 4, the results surprised me as Query takes the 1st place in the better performance, then Query + SIMD, and then Vanilla. The minimal improvement from SIMD suggests that prefix matching is inherently less suited to SIMD optimizations. Prefix matching requires string operations (e.g., checking substrings), which are less efficiently handled by SIMD compared to integer comparisons in exact matches. Overall, since all 3 algorithms take over 30 seconds to find the prefixes compared to scan exact matches suggests that they were implemented inefficiencies, such as extra memory operations or complex logic that outweighed the benefits of SIMD, etc.

Conclusion and insight:

As I've already analyzed the results during the Performance and Analysis section, this section will be my concluding thoughts about this project. The speed performance for dictionary encoding across multithreading clearly proves its purpose that the more threads you have the faster performance you get. In addition, with my additional test of comparing it to the vanilla baseline, it has shown that multithreading is more effective than vanilla. As parallelism or pipeline is more effective than singular job runs. For the Single data dn prefix scan, while it is true that SIMD instructions are better for search/scan, my data have shown me that there's need to be improvement on the code. However, for the search exact match, it did prove its result, however, the scan definitely needs an improvement. Search should be faster than scan as shown from Table 3 and 4, but I didn't expect the differences to be huge. Since SEARCH, you are trying to find the exact match and located at all of the indexes, but for scan, it makes sense that it will take longer because a prefix can occur and so many different places that the time will be longer than search.