Adv. Computer Systems - Project 4

Running Instructions: **2**
Grading Criteria: **2**
Software Implementation 2
Encoding Functionality 2
Figure 1: DictionaryCodec.cpp encoding 3
Figure 2: test_encoding.cpp handling EncodedColumn output file 3
Query Operations 3
Figure 3: DictionaryCodec.cpp query 4
Multi-Threading Implementation 4
Figure 4: Multithreading test in test_encoding.cpp 4
SIMD Utilization 5
Figure 5: simdQueryItem 5
Figure 6: simdQueryPrefix 5
Performance and Analysis: 5
Vanilla Column Scan 5
Dictionary Encoding 5
Table 1: Encoding Time (ms) vs. Thread Count 6
Figure 7: Encoding Time (ms) vs. Thread Count 6
Query 6
Table 2: Time to complete Query Operations 7
Figure 8: Time to complete Query Operations 7
Documentation: 8
Readme Clarity 8
Experimental Setup and Analysis 8
Conclusion & Analysis 8

Running Instructions:

This assignment is split into 4 program files. DictionaryCodec.h and DictionaryCodec.cpp handle implementing the dictionary encoding and query logic, as well as setting up the dictionary encoding scheme. The encoding utilizes the variable num_threads, to define the number of threads used in encoding. The query operations include VanillaScan, Item Search (SIMD/no SIMD), and Prefix Scan (SIMD/No SIMD). Item Search(SIMD/No SIMD) and Prefix Scan (SIMD/No SIMD). The last two files are testbenches that handle encoding and query tests. test_encoding.cpp handles encoding benchmarks with a varying num_thread value (Testing 1,2,4 and 8 thread encoding), and test_query.cpp measures query speeds for the five query operations.

To run this assignment, download the 4 program files, as well as the given raw column data file (Column.txt, provided by the professor at https://drive.google.com/file/d/195XTg8HWDTILc1JlsGX6_jJ5PUhi9KDG/view?usp=drive_link). To compile the program, use the following two commands:

```
g++ -std=c++17 -O2 -msse4.2 -mavx2 -march=native -o test_encoding test_encoding.cpp DictionaryCodec.cpp
g++ -std=c++17 -O2 -msse4.2 -mavx2 -march=native -o test_query test_query.cpp DictionaryCodec.cpp
```

The program testbenches can now be run with ./test_encoding and ./test_query respectively.

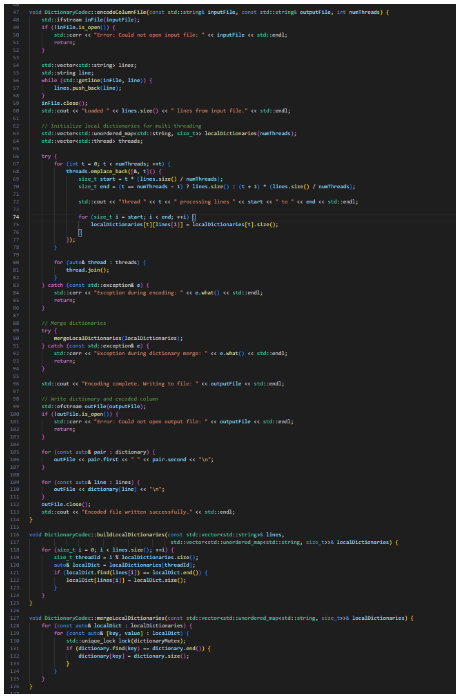
To modify the query string, adjust line 61 in test_query.cpp. The default implementation is [**std::string query = "spin";**], meaning the search query is the string “spin”.

Grading Criteria:

Software Implementation

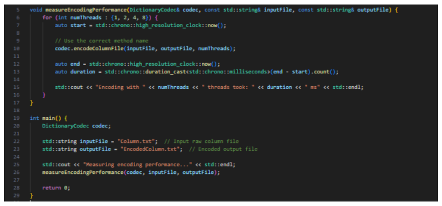
Encoding Functionality

Dictionary encoding is handled in DictionaryCodec.cpp and manages the encoding of the given Column.txt file. The encoded file is outputted as EncodedColumn.txt (as handled in test_query.cpp) and displays the dictionary and encoded data column.



```cpp
void DictionaryCodec::encodeColumnFile(const std::string& inputFile, const std::string& outputFile, int numThreads) {
    std::ifstream inFile(inputFile);
    if (!inFile.is_open()) {
        std::cerr << "Error: Could not open input file: " << inputFile << std::endl;
        return;
    }

    std::vector<std::string> lines;
    std::string line;
    while (std::getline(inFile, line)) {
        lines.push_back(line);
    }
    inFile.close();
    std::cout << "Loaded " << lines.size() << " lines from input file." << std::endl;

    // Initialize local dictionaries for multi-threading
    std::vector<std::unordered_map<std::string, size_t>> localDictionaries(numThreads);
    std::vector<std::thread> threads;

    try {
        for (int t = 0; t < numThreads; ++t) {
            threads.emplace_back([&, t]() {
                size_t start = t * (lines.size() / numThreads);
                size_t end = (t == numThreads - 1) ? lines.size() : (t + 1) * (lines.size() / numThreads);

                std::cout << "Thread " << t << " processing lines " << start << " to " << end << std::endl;

                for (size_t i = start; i < end; ++i) {
                    localDictionaries[t][lines[i]] = localDictionaries[t].size();
                }
            });
        }

        for (auto& thread : threads) {
            thread.join();
        }
    } catch (const std::exception& e) {
        std::cerr << "Exception during encoding: " << e.what() << std::endl;
        return;
    }

    // Merge dictionaries
    try {
        mergeLocalDictionaries(localDictionaries);
    } catch (const std::exception& e) {
        std::cerr << "Exception during dictionary merge: " << e.what() << std::endl;
        return;
    }

    std::cout << "Encoding complete. Writing to file: " << outputFile << std::endl;

    // Write dictionary and encoded column
    std::ofstream outFile(outputFile);
    if (!outFile.is_open()) {
        std::cerr << "Error: Could not open output file: " << outputFile << std::endl;
        return;
    }

    for (const auto& pair : dictionary) {
        outFile << pair.first << " " << pair.second << "\n";
    }

    for (const auto& line : lines) {
        outFile << dictionary[line] << "\n";
    }
    outFile.close();
    std::cout << "Encoded file written successfully." << std::endl;
}

void DictionaryCodec::buildLocalDictionaries(const std::vector<std::string>& lines,
                                             std::vector<std::unordered_map<std::string, size_t>>& localDictionaries) {
    for (size_t i = 0; i < lines.size(); ++i) {
        size_t threadId = i % localDictionaries.size();
        auto& localDict = localDictionaries[threadId];
        if (localDict.find(lines[i]) == localDict.end()) {
            localDict[lines[i]] = localDict.size();
        }
    }
}

void DictionaryCodec::mergeLocalDictionaries(const std::vector<std::unordered_map<std::string, size_t>>& localDictionaries) {
    for (const auto& localDict : localDictionaries) {
        for (const auto& [key, value] : localDict) {
            std::unique_lock lock(dictionaryMutex);
            if (dictionary.find(key) == dictionary.end()) {
                dictionary[key] = dictionary.size();
            }
        }
    }
}
```

Figure 1: DictionaryCodec.cpp encoding

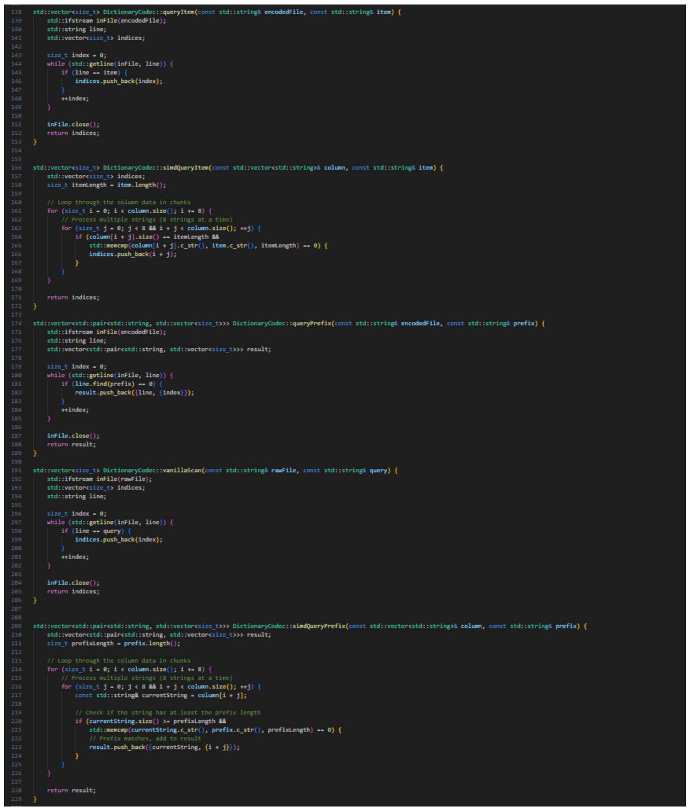


```cpp
void measureEncodingPerformance(DictionaryCodec& codec, const std::string& inputFile, const std::string& outputFile) {
    for (int numThreads : {1, 2, 4, 8}) {
        auto start = std::chrono::high_resolution_clock::now();

        // Use the correct method name
        codec.encodeColumnFile(inputFile, outputFile, numThreads);

        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();

        std::cout << "Encoding with " << numThreads << " threads took: " << duration << " ms" << std::endl;
    }
}

int main() {
    DictionaryCodec codec;

    std::string inputFile = "Column.txt";  // Input raw column file
    std::string outputFile = "EncodedColumn.txt";  // Encoded output file

    std::cout << "Measuring encoding performance..." << std::endl;
    measureEncodingPerformance(codec, inputFile, outputFile);

    return 0;
}
```

Figure 2: test_encoding.cpp handling EncodedColumn output file

Query Operations

There are five essential query operations, vanillaScan, queryItem (non SIMD single scan), queryPrefix (non SIMD prefix scan), simdQueryItem (SIMD single scan), simdQueryPrefix (SIMD prefix scan). The test_query.cpp testbench file handles measuring the timing for all of these operations, given a string query.

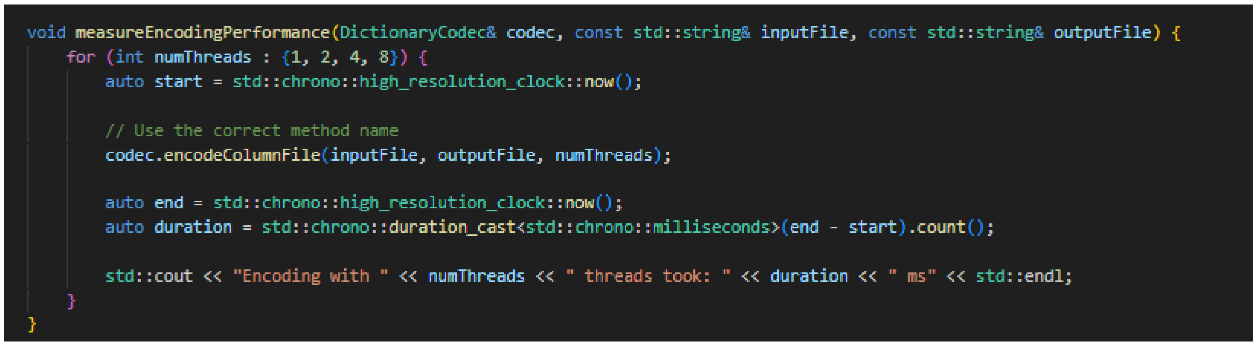


```cpp
std::vector<size_t> DictionaryCodec::queryItem(const std::string& encodedFile, const std::string& item) {
    std::ifstream inFile(encodedFile);
    std::string line;
    std::vector<size_t> indices;

    size_t index = 0;
    while (std::getline(inFile, line)) {
        if (line == item) {
            indices.push_back(index);
        }
        ++index;
    }

    inFile.close();
    return indices;
}


std::vector<size_t> DictionaryCodec::simdQueryItem(const std::vector<std::string>& column, const std::string& item) {
    std::vector<size_t> indices;
    size_t itemLength = item.length();

    // Loop through the column data in chunks
    for (size_t i = 0; i < column.size(); i += 8) {
        // Process multiple strings (8 strings at a time)
        for (size_t j = 0; j < 8 && i + j < column.size(); ++j) {
            if (column[i + j].size() == itemLength &&
                std::memcmp(column[i + j].c_str(), item.c_str(), itemLength) == 0) {
                indices.push_back(i + j);
            }
        }
    }

    return indices;
}

std::vector<std::pair<std::string, std::vector<size_t>>> DictionaryCodec::queryPrefix(const std::string& encodedFile, const std::string& prefix) {
    std::ifstream inFile(encodedFile);
    std::string line;
    std::vector<std::pair<std::string, std::vector<size_t>>> result;

    size_t index = 0;
    while (std::getline(inFile, line)) {
        if (line.find(prefix) == 0) {
            result.push_back({line, {index}});
        }
        ++index;
    }

    inFile.close();
    return result;
}

std::vector<size_t> DictionaryCodec::vanillaScan(const std::string& rawFile, const std::string& query) {
    std::ifstream inFile(rawFile);
    std::vector<size_t> indices;
    std::string line;

    size_t index = 0;
    while (std::getline(inFile, line)) {
        if (line == query) {
            indices.push_back(index);
        }
        ++index;
    }

    inFile.close();
    return indices;
}


std::vector<std::pair<std::string, std::vector<size_t>>> DictionaryCodec::simdQueryPrefix(const std::vector<std::string>& column, const std::string& prefix) {
    std::vector<std::pair<std::string, std::vector<size_t>>> result;
    size_t prefixLength = prefix.length();

    // Loop through the column data in chunks
    for (size_t i = 0; i < column.size(); i += 8) {
        // Process multiple strings (8 strings at a time)
        for (size_t j = 0; j < 8 && i + j < column.size(); ++j) {
            const std::string& currentString = column[i + j];

            // Check if the string has at least the prefix length
            if (currentString.size() >= prefixLength &&
                std::memcmp(currentString.c_str(), prefix.c_str(), prefixLength) == 0) {
                // Prefix matches, add to result
                result.push_back({currentString, {i + j}});
            }
        }
    }

    return result;
}
```

Figure 3: DictionaryCodec.cpp query

Multi-Threading Implementation

Multithreading is used within the Column File encoding to speed up the process. The number of lines to encode is equally divided amongst the number of threads specified. In this case, encoding was measured with 1,2,4, and 8 threads.

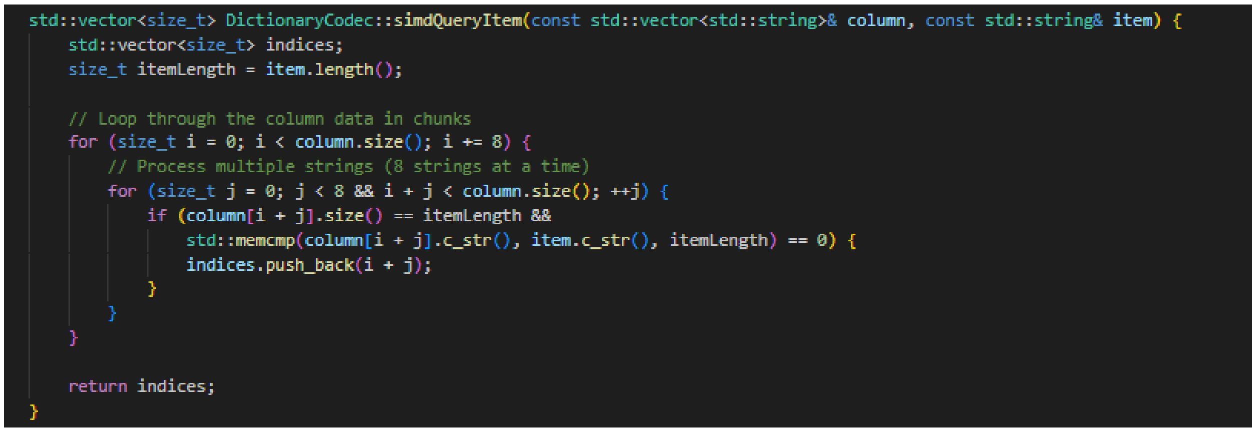


```cpp
void measureEncodingPerformance(DictionaryCodec& codec, const std::string& inputFile, const std::string& outputFile) {
    for (int numThreads : {1, 2, 4, 8}) {
        auto start = std::chrono::high_resolution_clock::now();

        // Use the correct method name
        codec.encodeColumnFile(inputFile, outputFile, numThreads);

        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();

        std::cout << "Encoding with " << numThreads << " threads took: " << duration << " ms" << std::endl;
    }
}
```

Figure 4: Multithreading test in test_encoding.cpp

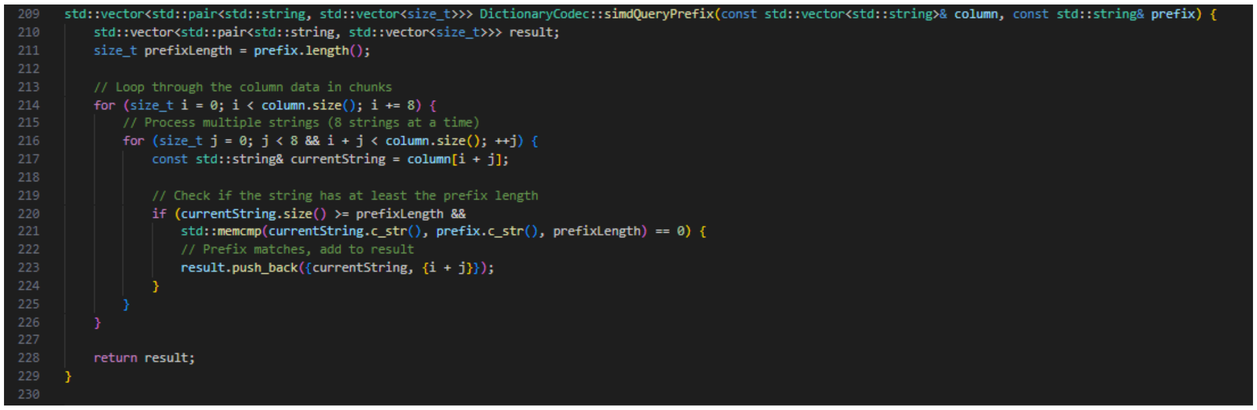
SIMD Utilization

As mentioned in the previous section, SIMD accelerates single-query and prefix-query searches. This program includes <immintrin.h> to handle SIMD operations.



```cpp
std::vector<size_t> DictionaryCodec::simdQueryItem(const std::vector<std::string>& column, const std::string& item) {
    std::vector<size_t> indices;
    size_t itemLength = item.length();

    // Loop through the column data in chunks
    for (size_t i = 0; i < column.size(); i += 8) {
        // Process multiple strings (8 strings at a time)
        for (size_t j = 0; j < 8 && i + j < column.size(); ++j) {
            if (column[i + j].size() == itemLength &&
                std::memcmp(column[i + j].c_str(), item.c_str(), itemLength) == 0) {
                indices.push_back(i + j);
            }
        }
    }

    return indices;
}
```

Figure 5: simdQueryItem



```cpp
std::vector<std::pair<std::string, std::vector<size_t>>> DictionaryCodec::simdQueryPrefix(const std::vector<std::string>& column, const std::string& prefix) {
    std::vector<std::pair<std::string, std::vector<size_t>>> result;
    size_t prefixLength = prefix.length();

    // Loop through the column data in chunks
    for (size_t i = 0; i < column.size(); i += 8) {
        // Process multiple strings (8 strings at a time)
        for (size_t j = 0; j < 8 && i + j < column.size(); ++j) {
            const std::string& currentString = column[i + j];

            // Check if the string has at least the prefix length
            if (currentString.size() >= prefixLength &&
                std::memcmp(currentString.c_str(), prefix.c_str(), prefixLength) == 0) {
                // Prefix matches, add to result
                result.push_back({currentString, {i + j}});
            }
        }
    }

    return result;
}
```

Figure 6: simdQueryPrefix

Performance and Analysis:

Vanilla Column Scan

The program uses a vanilla column scan (aka scanning without using the dictionary encoding scheme) as a baseline to compare the effect of encoded queries.

For the query string “spin”, the vanilla scan took 3510 milliseconds. This value is used in the query section later in the report.

Dictionary Encoding

Dictionary encoding was handled across four different thread counts- those being 1,2 4 and 8. A table and graph showing the thread count speed difference is shown below.

Table 1: Encoding Time (ms) vs. Thread Count

Thread Count	Encoding Time (ms)
1	94226
2	80589
4	75451
8	76140

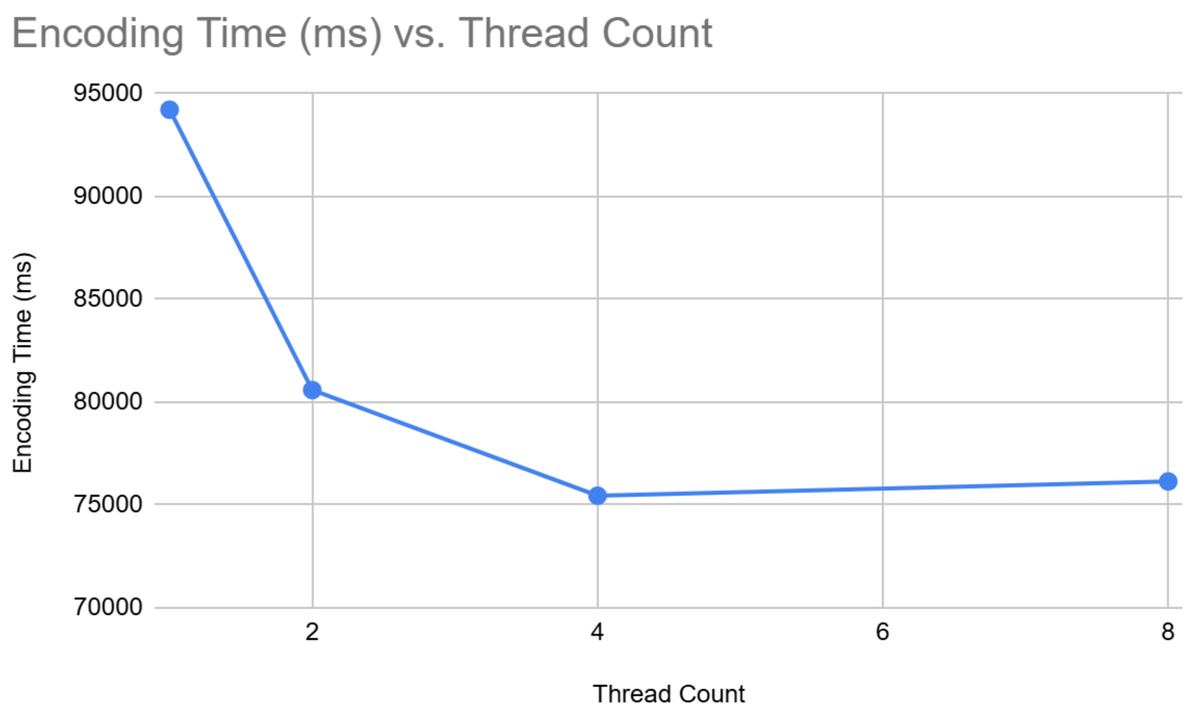


Figure 7: Encoding Time (ms) vs. Thread Count

The graph shows that as thread count increases, the time to encode decreases. This functionality works as expected. It is worth noting that there seems to be a bottleneck or diminishing return after 4 threads. 8 threads do not provide any additional performance benefit and seem to be the same as 4 threads (within the margin of error).

Query

For the query test, the query string “spin” was used to run 5 different operations, vanilla scan, item search (No SIMD), item search (SIMD), query scan (No SIMD), query scan (SIMD). A table and graph showing the time to complete these operations are included below.

Table 2: Time to complete Query Operations

Operation	Time to Complete (ms)
Vanilla Scan	3510
Item Search	2650
Item Search (SIMD)	289
Prefix Scan	3531
Prefix Scan (SIMD)	489

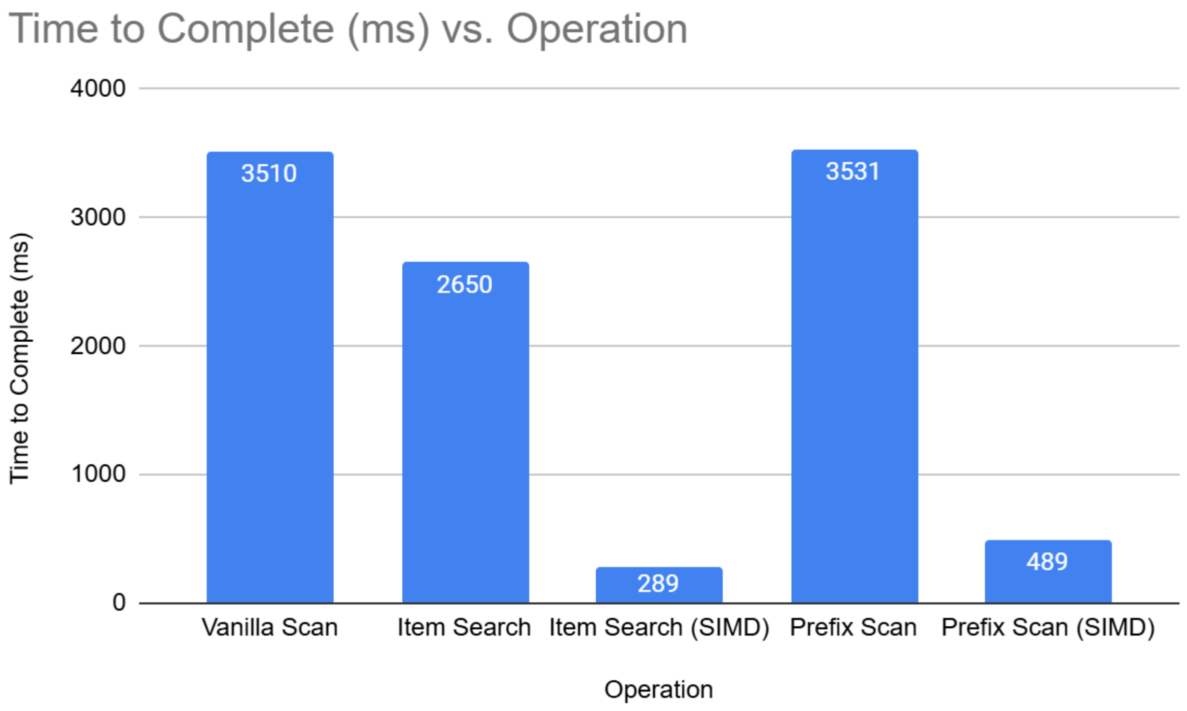


Figure 8: Time to complete Query Operations

As expected, dictionary encoding provided far faster results than the vanilla scan. Both item searches (SIMD and No-SIMD), provided much faster execution times than the non-encoded vanilla scan did, especially the SIMD implementation. Looking at the prefix scan, we see an expected rise in execution time, due to the new task of searching for a prefix through the entire data file (rather than stopping at the first find). SIMD once again provided a much faster execution time than the No-SIMD version.

Documentation:

Readme Clarity

Readme.txt provided in the repository.

Experimental Setup and Analysis

Experimental setup and analysis are provided in the “Performance and Analysis” portion of this report.

Conclusion & Analysis

The dictionary codec is based on compressing repeated data (encoding) and using indexing to make querying more efficient. In the context of large datasets, such as the one tested here, many values tend to repeat (e.g., string values of common phrases). A dictionary codec exploits this redundancy by doing two things. One, storing unique values in a dictionary, and two, encoding the original dataset as a list of indices pointing to the corresponding unique value in the dictionary.

Benefits of dictionary codec include spare/query efficiency, scalability to larger datasets, faster prefix and range queries, and parallelization by utilizing multiple threads. It is clear in this assignment that multithreaded encoding can improve performance but can diminish returns after a certain number of threads. Dictionary encoding (combined with SIMD optimizations) can greatly reduce execution times compared to a vanilla/raw scan.

In summary, the philosophy of dictionary codecs focuses on reducing data redundancy, improving memory efficiency, and speeding up data access, making it an excellent choice for large-scale data storage and retrieval, especially when combined with parallel and SIMD processing for further acceleration.