Course Project #4: Implementation of Dictionary Codec
17 November 2024

1 Introduction

Dictionary encoding is a fundamental technique used in data analytics systems to compress data with relatively low cardinality and accelerate search and scan operations. By replacing repeated data items with unique identifiers, dictionary encoding reduces data footprint and enables more efficient data processing. This project aims to implement a dictionary codec that supports encoding of raw column data, querying of the encoded data, and comparison with a baseline vanilla column scan. The implementation leverages multi-threading to improve encoding performance and utilizes Single Instruction Multiple Data (SIMD) instructions to speed up search and scan operations.

2 Software Implementation

2.1 Encoding Functionality

The dictionary encoder reads the raw column data from a file and performs dictionary encoding by scanning the data to build a dictionary of unique data items. Each unique data item is assigned a unique integer identifier (ID). The encoder then replaces each data item in the original data with its corresponding dictionary ID, producing an encoded data column. The encoded data column and the dictionary are stored in separate files for later use.

The encoding process supports multi-threaded implementation to enhance performance. The raw data is partitioned into chunks, and multiple processes are spawned using Python's `multiprocessing` module to process each chunk in parallel. During the dictionary building phase, each process extracts unique strings from its data chunk, and the results are combined to form the complete dictionary. In the encoding phase, each process replaces the data items in its chunk with the corresponding IDs from the dictionary.

2.2 Query Operations

The implementation provides functionalities to query the encoded column data. Users can check whether a specific data item exists in the column and retrieve the indices of all matching entries. Additionally, users can perform prefix searches to find all unique data items that start with a given prefix and obtain their indices.

To accelerate query operations, the implementation utilizes SIMD instructions through NumPy arrays to perform vectorized operations. This allows simultaneous comparison of multiple data items, significantly speeding up search and scan processes. The queries are implemented in two ways: one using standard iteration over the data (without SIMD) and the other using vectorized operations with NumPy (with SIMD).

2.3 Multi-threading Implementation

Multi-threading is employed during the encoding phase to improve performance by parallelizing the workload across multiple CPU cores. The raw data is divided into equal-sized chunks, and each chunk is processed independently by a separate process. This approach reduces the overall encoding time by taking advantage of concurrent execution. The number of processes (threads) can be adjusted based on the system's capabilities.

2.4 SIMD Utilization

SIMD instructions are used in the query operations to enhance performance. By converting the encoded data into NumPy arrays, the implementation leverages NumPy's optimized vectorized operations to perform comparisons across large datasets efficiently. This method reduces the number of iterations required and takes advantage of low-level hardware optimizations present in modern CPUs.

3 Experimental Results

3.1 Experimental Setup

The experiments were conducted on a system with the following specifications: [Specify CPU model and number of cores], [Specify amount of RAM], and [Specify operating system]. Due to resource constraints, a subset of the dataset containing 100,000 entries was used for testing.

3.2 Encoding Performance Under Different Thread Counts

The encoding performance was measured using different numbers of threads: 1, 2, 4, and 8. The times taken to build the dictionary and encode the data are presented in Table 1.

Table 1: Encoding Performance with Varying Thread Counts

Number of Threads	Dictionary Build Time (s)	Data Encode Time (s)
1	0.25	0.23
2	0.26	0.26
4	0.43	0.38
8	0.49	0.67

3.3 Query Performance

The performance of single data item search and prefix search was evaluated using the vanilla scan, dictionary encoding without SIMD, and dictionary encoding with SIMD. The results are summarized in Tables 2 and 3.

Table 2: Single Data Item Search Performance

Method	Occurrences Found	Time (s)
Vanilla Search	1	0.004747
Dictionary Search without SIMD	1	0.004097
Dictionary Search with SIMD	1	0.004451

Table 3: Prefix Search Performance

Method	Unique Items Found	Time (s)
Vanilla Prefix Search	14	0.018220
Dictionary Prefix Search without SIMD	14	0.074744
Dictionary Prefix Search with SIMD	14	0.018461

4 Analysis

4.1 Encoding Performance Analysis

The encoding performance results indicate that increasing the number of threads does not consistently improve encoding times for the dataset size used. With one thread, the dictionary build time was 0.25 seconds and the data encode time was 0.23 seconds. Using two threads yielded similar times, while using four and eight threads resulted in longer encoding times. This behavior can be attributed to the overhead associated with creating and managing multiple processes in Python's multiprocessing module. For small datasets, the overhead of process creation, inter-process communication, and data partitioning can outweigh the benefits of parallel processing. As the number of threads increases, the overhead becomes more significant, leading to diminished or negative returns in performance.

4.2 Query Performance Analysis

In the single data item search, the dictionary search without SIMD was slightly faster than the vanilla search, with times of 0.004097 seconds and 0.004747 seconds, respectively. The dictionary search with SIMD had a time of 0.004451 seconds. The marginal improvement in the dictionary searches can be attributed to the efficiency of integer comparisons over string comparisons. For the prefix search, the vanilla prefix search and the dictionary prefix search with SIMD had similar times of approximately 0.018 seconds. The dictionary prefix search without SIMD was significantly slower, taking 0.074744 seconds. This demonstrates that SIMD utilization effectively accelerates prefix searches by performing vectorized operations, reducing the time needed to compare multiple data items. The overhead of mapping IDs back to strings and the lack of vectorization in the non-SIMD dictionary search contribute to its slower performance.

5 Conclusion and Insights

The implementation of the dictionary codec successfully achieved the project's objectives, providing encoding functionality, query operations, and a comparison with a vanilla column scan. Multi-threading was employed to enhance encoding performance, and SIMD instructions were utilized to accelerate query operations. The experimental results show that multi-threading did not significantly improve encoding performance for the small dataset used. The overhead associated with process management in Python can negate the benefits of parallelization when dealing with limited data sizes. For larger datasets, it is expected that multi-threading would yield more substantial performance improvements. In the query operations, dictionary encoding provided a slight performance benefit in single data item searches due to faster integer comparisons. SIMD utilization significantly improved the performance of prefix searches, demonstrating its effectiveness in handling operations that involve comparing multiple data items simultaneously.

Possible optimizations include processing larger datasets to better leverage the benefits of multi-threading and SIMD. Additionally, exploring alternative approaches to multi-threading, such as using threading instead of multiprocessing or implementing asynchronous I/O, could reduce overhead for smaller datasets. Implementing lower-level SIMD instructions using libraries like NumPy's `numexpr` or writing performance-critical code in C or Cython may also enhance performance.
In conclusion, the project highlights the importance of considering dataset size and computational overhead when implementing performance optimizations. While multi-threading and SIMD can provide significant benefits, their effectiveness depends on the context and specific characteristics of the workload. Future work could involve testing the implementation with larger datasets and exploring additional optimizations to further improve performance.