Course Project #4: Implementation of Dictionary Codec

Charles Clarke
Advanced Computer Systems
11/14/2024

Introduction:

Dictionary encoding is an incredibly useful form of data compression which can reduce the storage space required for datasets by replacing repeated or similar values with unique symbols, or keys. Creating a dictionary in the form of a hash table or B tree can be very beneficial to the time it takes to search for items within the dictionary. In this implementation I make use of an unordered map in c++ in order to store all of the data. It provides average constant time complexity for insertions, deletions, and lookups which makes it ideal for any dictionary encoding tasks. Using this method, I was able to take the raw column data and encode it all into a separate file for further querying. The query feature I implemented allows users to locate the index at which an encoded value has been stored. As well as utilizing a prefix function to search for every data value which starts with the required prefix, alongside the index it's stored at. SIMD search was also included in order to speed up the query function. I took all of these features and tested them alongside a vanilla lookup, using no encoded dictionary or SIMD instructions. The results of these experiments can be seen in the following sections.

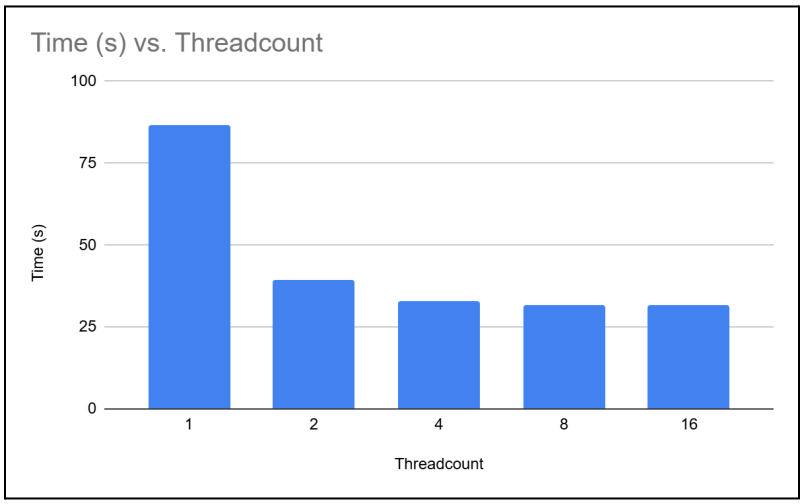
Code Structure:

My program uses a few key functions in order to handle all of the operations required by this implementation. First of all, I have encodeBatch and encodeFile which handle the encoding of the dictionary and allow for the use of multithreading. These functions require the input file name, the output file name, and the number of threads that will be used. These functions are able to dynamically divide the input file into chunks for each thread and so that the memory usage isn't incredibly high. I had an issue of high memory usage killing the program early in the testing. Next is the simd function which allows for the use of simd when querying the file. I had some difficulty getting this to work over the course of a couple days. Next is my queryFile function which handles all of the querying operations, exact match and prefix. Searching for an exact match is the default case, however a user can input "prefix" if they wish to activate this feature. This function simply takes in the encoded file and the query, then searches for the data using the hash table. This results in very quick lookup time due to the efficiency of the hash table and the encoded data file. Finally is the main function which simply handles all of the inputs given to the program via the command line and selects the correct functions to use for the operation.

Encoding:

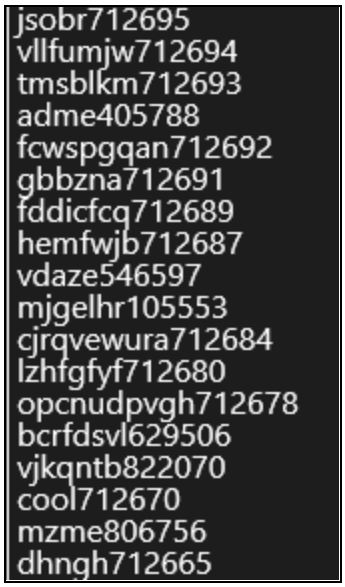
The dictionary encoding feature of my code wasn't too difficult to implement, however it was difficult to make a quick and efficient encoding feature. After many iterations, I created an encoding function that could completely encode the 1GB data file in about 30 seconds using 4 threads. I was more than happy with this result, based on how much raw column data there was. I've created a graph that can be seen below which relates thread count to encoding speed. The commands and terminal outputs which helped me obtain the data can also be seen below.

```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 1 on encode
Encoding completed in 86625 ms.
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 2 on encode
Encoding completed in 39514 ms.
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 4 on encode
Encoding completed in 33013 ms.
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 8 on encode
Encoding completed in 31671 ms.
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 16 on encode
Encoding completed in 31869 ms.
```



The graph above shows the relationship between time and thread count when encoding the raw column data. From this data 4 threads seems to be the optimal amount of multithreading for my system. This speeds up the encoding by a large margin over 1 or 2 threads, while not stressing the system nearly as much as 8 or 16 does.

The structure of the encoded file can also be seen below, which allows for much quicker lookup times than in a raw data file for a few reasons. First of all, the data is encoded within a dictionary using a unique integer ID. This speeds up the lookup time, because it's within a hash table, which provides much more efficient lookups. Also the encoded IDs are smaller than the raw string data, which can reduce memory usage significantly.



Here you can see just how encoded data is formatted, with a specific key placed next to each data item to assist in quick lookup using a hash table.

Querying:

Querying via the encoded dictionary turned out to be incredibly quick, whether SIMD was being used or not. The first step of the querying process is loading the hashed dictionary into memory so it can be used. The exact match feature simply looks up the item in question and returns whether it is within the dictionary or not, along with the index at which it is located in the encoded file. The prefix function allows for the search of a prefix and returns all the data values which start with that prefix, along with their indices as well. The output of the exact match feature can be seen below.

```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 4 on query "abcd"
Found abcd at indices: 760533
Query completed in 0 ms.
```

As seen in the terminal output, the exact match feature is super fast, because of how the dictionary has been implemented as a hash table. This means any exact match lookups will be O(1) which is very impressive. The prefix function takes a bit longer than the exact match, simply because there are more values most of the time. The output of this search can be seen below.

```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 4 on query "abcd"
prefix
Prefix matches for "abcd":
Found prefix "abcd" in abcd at indices: 760533
Found prefix "abcd" in abcdzodcjbpfrnsndrdhtt at indices: 613270
Found prefix "abcd" in abcdltd at indices: 43374
Found prefix "abcd" in abcdpvbn at indices: 352608
Query completed in 32 ms.
```

Here, there were four matches for the prefix of “abcd” including the exact match seen just above. This took just 32 ms to complete, which is incredibly quick when compared to how long it will take when scanning the raw data. This comparison will be seen later in the report.

I had a lot of trouble implementing SIMD over the course of this project, I just couldn’t get it to make a noticeable difference in the lookup time of various data items. The results of querying the data for an exact match with SIMD on and off can be seen below. There isn’t too much of a difference in time at all.

```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 4 on query "abcd"
Found abcd at indices: 1232288
Query completed in 0 ms.
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./dictionary_3 Column.txt output.txt 4 off query "abcd"
Found abcd at indices: 1232288
Query completed in 0 ms.
```

You can see above that with SIMD on or off for exact match, the query feature is almost instant, taking less than a millisecond. This makes sense because single instruction multiple data really improves lookup time when doing many operations, not just searching for a single value. This is why I was really only focused on the next test for my SIMD implementation. The results of different times when querying can be seen below. First will be the result of searching for the prefix “ab” with SIMD on.

```
Found prefix "ab" in abmv at indices: 1533629
Found prefix "ab" in abbebu at indices: 136131
Found prefix "ab" in abjqqscu at indices: 1632031
Found prefix "ab" in abof at indices: 1727085
Found prefix "ab" in abwhgtnipzwinrkicawnjrvtkjzwocgqeqsr at indices: 922020
Found prefix "ab" in abwh at indices: 1265900
Found prefix "ab" in abrwh at indices: 1365690
Found prefix "ab" in abkz at indices: 604690
Query completed in 51 ms.
```

This search resulted in a whole lot of data values found in just 51 ms with SIMD on. Compared to SIMD off which can be seen in the next image below.

```
Found prefix "ab" in abhggg at indices: 1342031
Found prefix "ab" in abmv at indices: 1533629
Found prefix "ab" in abbebu at indices: 136131
Found prefix "ab" in abjqqscu at indices: 1632031
Found prefix "ab" in abof at indices: 1727085
Found prefix "ab" in abwhgtnipzwinrkicawnjrvtkjzwocgqeqsr at indices: 922020
Found prefix "ab" in abwh at indices: 1265900
Found prefix "ab" in abrwh at indices: 1365690
Found prefix "ab" in abkz at indices: 604690
Query completed in 54 ms.
```

As seen above, there isn't much of a difference in time at all between the two tests. Turning SIMD on in my case only saved my system 3ms. I tried this test again with the prefix "a," in order to get a larger data set, and I received pretty similar results. With SIMD on, the search took a total of 258 milliseconds. When I turned SIMD off, it took 270 milliseconds. This is a very small difference once again. In the end it seems that SIMD did speed up the search process for multiple elements when looking for a specific prefix. However not by a large enough margin to make it worth it in my opinion.

Vanilla Comparison:

In order to determine just how much quicker the hash table lookup was than the vanilla line by line search method, I created a short program to perform the raw search. The program simply takes an input file and an exact match query, then performs the search by checking each line one by one. It took much longer for this program to find the string "abcd" within the raw data. As seen earlier in this report, the string "abcd" was found in less than a millisecond using the hash table dictionary method. However searching through the raw data took just about 200 milliseconds as seen in the output below.

```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320/Project4$ ./vanilla
Enter the file path: Column.txt
Enter the string to search: abcd
String found at line 878780: abcd
Time taken to search: 0.198261 seconds
```

The time it takes to find a value using this method will change depending on where it is within the data, as its time complexity is O(n). In this case it was over 200x faster to use the encoded hashed search method**.**

Conclusion:

In the end, this implementation of a dictionary codec was very enlightening for me. I learned a lot about just how useful hash tables and encoding could be for data storage. First of all the data needed to be encoded. This took my program a lot of time

up front in order to save a lot of time in the future when searching for data values. This is very similar to how it is in the real world too I believe. Efficient systems are tougher to implement than basic systems, however once implemented correctly, they can save vast amounts of time and resources in the long run. Depending on how many threads I used, I could encode the data in about 30 seconds, which isn't too bad considering just how many elements were in that column.txt file. As the number of threads increased, the returns definitely diminished as I've seen in other projects we've done. I think the optimal amount of usage for my system is 4 threads, as it provides a good encoding time without using too many resources. Next was querying, the lookup of elements is where this dictionary really saves users a lot of time. In my test I saw that the hash table lookup was 200x faster than the vanilla lookup of the same element. This ratio will always vary depending on where the data is located, but is a great example of just how much time can be saved using this method of hashing. I also implemented SIMD which allowed for a little bit of time saved when searching for items with a certain prefix, not so much for a single item, as only one instruction had to be executed anyways. Overall I learned just how complicated an efficient data storage system can be, and how it can be used to access data much quicker than by simply looking for it line by line.