Code Structure:

When designing the different algorithms to fulfill the project requirements, I had to be mindful of what language I had chosen, in this case python, and the constraints that gave me. Because python does not use hard data types like C, C++ or Java, I decided to work with the in-built byte data type. For each of the size tests, I used arrays of each byte size of eight character byte strings. For example, for the 64B test I used a single dimension list containing 256 eight character byte strings.

For the encoding and decoding functions, I utilized a global python dictionary to store the key-value pairs, assigning an ever incrementing value to every unique element found in the data. This would mean that if I had an original data of: [sun, mercury, venus, sun, mercury], it would be replaced with: [1, 2, 3, 1, 2].

From here, I worked on the individual benchmarking algorithms, testing the multithreading capability, read/write ratios and scanning times. For the user tests, I modeled the number of users using several concurrent threads, testing out one, two, and four users. I used the single user test as the basis to create the other two benchmarking functions, using the threading library within python to execute the concurrent operations. I used a list to store each of the threads, using two for loops to lock and unlock them once they finished their processes.

For the read/write tests, I reused the single user test code, since I realized that the read operation could be done using the decode function, and the write with the encode function. I specifically tested equal read/writes, one read/two writes, and two reads/one write. The one to one ratio test just reutilized the same code from the single user function, removing some of the extraneous print statements. The other two functions modified this by adding a single additional encode or decode operation to the first for loop, adding an additional thread to the list of threads.

For the scanning functions, I simply iterated through the list of encoded data, matching it with the key-value pairs to whether certain prefixes were found. I used the string .startswith() method to do the prefix matching operations, converting the byte strings to regular strings using the byte decode(“utf-8”) method. To do single element string matching, I only removed the startswith() method and simply checked each element using a simple if statement.

For data compression, I modified the encode and decode algorithms so that I could compress the values whenever I needed using a parameter flag. I used the zlib library to compress the values, storing them in the dictionary and a separate output list. I later called this using true/false values whenever needed.

Analysis:

One important thing to note about the timings gathered from each of the tests is that they are not necessarily the most precise estimates. This is because of python's innate code compile process and other related inconsistencies. Knowing this, we can then understand why the following data is the way it is. For every test, the smallest or shortest operation in theory usually led to the shortest performance time.

For the internal threads test, the number of threads had a direct affect on how much time the operations took to complete. This makes sense because the more concurrent threads you have, the more time it takes for you to fully lock and unlock each thread upon their completion. So we can say that there was a positive relationship between the number of threads and the amount of time to complete. The user tests followed a similar pattern, due to the fact that users were modeled using threads. However, less threads were used making the largest test time much smaller than the largest test for the internal threads.

The read/write ratio had slightly mixed results, with the only consistent theme being that the one to one read/write (r/w) ratio had the smallest performance time. The 1r/2w and 2r/1w had very similar results, with the 1r/2w taking slightly more time than the 2r/1w operation.

The data size test gave largely unclear results, providing results that went counter to the theory learned in class. The 8B test took more time than the 16B test, which made very little sense. It did make sense, however, that the 256B test took longer than both prior tests.

The search tests seemed to be the most consistent of all the tests, since the operation that utilized more iterations, namely the prefix scanning algorithm, took the most time. Meanwhile, the single scan test took about the same amount of time to find something that did exist in the data as it did to find something that did not exist.

The compression tests provided very simple results, with the compression operation being much faster than the operations without compression. This makes sense in theory since there is less extraneous data to parse through to reach the end of the algorithm.

Conclusion:

The most significant conclusions I found from this project were that the number of concurrent operations could dramatically affect performance time. This is largely due to each operation needing to wait for the prior ones to complete before starting their next process. Another conclusion was that compression can easily reduce the performance time if used properly. If not used properly it is easy to lose potentially important pieces of data. For languages like python, data size has a very small effect on the overall performance due to the implicit assumption on memory allocation that the language makes. For instance, it will always allocate a certain amount of memory space, specifically 14B, unless some threshold is passed. Another smaller conclusion is that read operations seem to be less intensive than write operations. This largely makes sense because with a read, all you need is to access the memory address and directly pull its value. With a write operation, you need to find the same address, and modify its contents using another address's values. If I were to optimize the code in this project, I could have used C or C++ the utilize exact memory sizes and more precise timing functions. Overall, I learned the value of utilizing data compression, concurrency support and precise data scanning.