Web Search Engine: Index Quantization

Haozhong Zheng
hz2675@nyu.edu
New York University

ABSTRACT

In this course, we've built a simple inverted index and used it for query. When the system processes a query, it must compute the impact scores using ranking functions such as BM25. A common solution for speeding up the process, is to precompute impact scores and store them in the index file to avoid computational overhead while querying. However, storing uncompressed floating point numbers takes up a lot of space, and generic data compression algorithms do not provide the desired compression rate. So, quantization methods become much more important. We read recent papers on data compression and quantization for this final project and attempted to implement primitive linear quantization, logarithmic quantization, and a new method called adaptive float. Finally, we visualized the results and ran several benchmarks to gain a better understanding of their performance.

KEYWORDS: Inverted Index, Linear Quantization, Logarithmic Quantization, Adaptive Float

1 INTRODUCTION

1.1 Background

The capacity of today's Internet is known to be growing exponentially, posing a significant challenge to web search engines. According to relevant statistics, the world's largest search engine, Google, has built more than ten trillion web pages for indexing. Typically, users want to wait no more than hundreds of milliseconds for a query result among the massive web pages; however, primitive information retrieval techniques are incapable of doing so.

There are numerous methods for increasing query speed, the most important of which are to design advanced query processing algorithms, efficient search engine architecture, and indexing structures. Because the impact scores computed by algorithms such as BM25 are independent of the user's query, developers would always store the precomputed impact scores among all types of refined indexing structures, to avoid computational overhead during querying. Given that the most advanced search engines usually have trillions of pages with even more terms, storing floating point numbers without any compression would take up so much space that it is not feasible. Furthermore, using standard compressors such as Gzip on floating-point-type impact scores produces mediocre results. If developers want to reduce space consumption, they should look into different floating point quantization methods.

In fact, there are so many ways to quantize data that researchers have been putting in a lot of effort to develop various solutions long before web search engines were invented. The most common application of quantization has been in the field of telecommunication, whereas the impact scores of web pages differ from those types of signal data. As a result, many quantization methods may not always perform well, necessitating comparative experiments in this project.

1.1 Project Overview

The web search engine course this semester consists of four assignments. The first task is to create a priority queue-based web crawler, which has nothing to do with this project. The second is building the inverted index data structure, and the third is query processing. This final project's code is based on and improved upon the previous two.

- Index Building: Instead of merging intermediate postings, the program employs the merging-subindexes algorithm. In addition, the inverted indexes and lexicon terms are arranged alphabetically. I divide the entire building process into two steps: The documents would be parsed and tokenized in the first step. Then it'd convert tokens to inverted indexes, build necessary data structures, and save them to disk. The final index would be built in the second step, which would involve performing an n-way merge. Var-byte compression is used in this program for both the document id and the term frequency.
- Query Processing: The program implements basic "Document-At-A-Time" conjunctive query and "Term-At-A-Time" disjunctive query. Also it supports complex query, which would process disjunctive query for multiple conjunctive terms or single term. For example, A complex query string *"microsoft, apple || band"* means *"(microsoft && apple) || (band)"* . This is implemented by some kind of polymorphism design.

The inverted index in the previous assignment stores the frequency of terms appearing in each document and uses var-byte compression. To convert frequency data into precomputed impact scores and then apply the quantization method, our code must be more generalizable, modular, and scalable. As a result, we significantly refactored the original code in this project, separated the redundant logic into different classes, and made extensive use of C++ template programming to allow derived classes to inherit the common logic. This makes it easier to extend the functionality of various inverted index data structures, such as quantization or compression.

2 RELATED WORKS

2.1 Search Engine

Information retrieval technology is widely used in a variety of applications in computer science, including search engines. Currently, the internet search engine market is nearly monopolized. Only Google, Microsoft Bing, Yahoo, Baidu, Yandex, and DuckDuckGo are among the top search engines. These commercial search engines, of course, are closed source. Apache Lucene is the most fully developed open source search engine. It is a Java search engine library supported by the Apache Software Foundation. There are also some open source search engines, such as the PISA system (Performant Indexes and Search for Academia), which is an open source library that implements text indexing and search, primarily for use in an academic setting. [1]

A comprehensive search engine system should include at least the following features.

- Documents Parsing: Parse and tokenize documents in order to obtain terms and create postings, where each term is assigned a unique document ID and each document is made up of a list of postings.
- Indexing: Once the parsing phase is complete, the postings can be used to build inverted index, which is a collection of terms, each of which contains a list of document IDs and frequencies or precomputed scores.
- Scoring: Because BM25 is a simple yet effective ranking function for bag-of-words queries, it is widely used in many systems.
- Index Compression: This includes not only compressing document IDs or frequencies, but also compressing and quantizing precomputed impact scores.

- Search: In addition to the basic index traversal strategy, the system should support dynamic pruning algorithms to improve query performance.

2.2 Data Compression

Computer and communication technologies have advanced rapidly over the last two decades, and the information explosion of the Internet era has highlighted the importance of data compression techniques. Data compression techniques are typically classified as lossless compression or lossy compression.

As the name implies, lossless compression is a process that results in no information loss. It is used in situations where there is no difference between the original data and the decompressed data. Because we should never change the original data unless there is a specific user requirement, the standard compression method would always use lossless compression.

The most well-known lossless compression algorithms are the LZ77 and LZ78 (Lempel-Ziv) algorithms, which were released in 1977 and 1978, respectively. Based on the LZ77 algorithm, a family of compression algorithms known as the LZ77 family has been developed. The Lempel-Ziv-Welch (LZW) algorithm, the Lempel-Ziv-Markov chain (LZMA) algorithm, the Lempel-Ziv-Oberhumer (LZO) algorithm, and the more recent LZ4 algorithm are examples of these. Furthermore, gzip, the most widely used gzip compression file format, is based on the DEFLATE algorithm, which is a combination of the LZ77 algorithm and Huffman coding. Additionally, a high performance data library like HDF5 would preprocess data before compression using Bitshuffle [2] or other algorithms.

In fact, many applications can accept a certain amount of information loss and do not require 100% accurate data recovery, in which case lossy compression techniques are used. Obviously, a lossy compression algorithm designed for a specific application should have a higher compression ratio than a generic lossless compression algorithm.

Lossy compression is commonly used to compress multimedia data such as audio, video, or images, which differs from the impact scores dealt with in this project. However, some generic lossy compression algorithms are now available for compressing floating-point numbers from large scale scientific data, and they have achieved very good results. The ZFP compressor [3] , for example, is capable of compressing integer or floating-point numbers in multiple dimensions. The SZ compressor, which has emerged in recent years, uses a hybrid Lorenzo prediction method called mean-integrated Lorenzo prediction as well as a linear regression method to achieve much higher prediction accuracy.[4][5][6] It currently has cutting-edge performance. In general, data compression is a very broad research area, with many papers [7] reviewing and comparing various state-of-the-art data compression techniques.

2.3 Quantization

Quantization is one of the lossy compression techniques, and it is the most fundamental and broadly accepted idea. In many lossy compression applications, one or more values from the original dataset need to be represented as one of a small collection of code words. The number of distinct values in a dataset is typically much greater than the number of code words available to represent them. Quantization is the process of representing a very large (possibly infinite) set of values with a very small subset of elements.

Quantization can also be divided into two types: scalar quantization, in which the quantizer's output values correspond one-to-one to the input values, and vector quantization, in which multiple input values are quantized into a single output value. The linear quantization and logarithmic quantization methods used in this project are scalar quantization methods. There are also adaptive

quantization methods that can adjust based on statistical indicators such as mean, variance, and probability density function (PDF). If the statistics probability model is already known, the Lloyd-Max algorithm is the best non-uniform quantization method. As deep neural network models have grown in size in recent years, papers quantizing the floating-point weights of neural networks have gradually dominated the field of quantization, so we also refer to these deep learning-related techniques.

The concept behind vector quantization is to group the input values and then quantize each group as a whole to obtain the corresponding code word. As a result, the primary task of vector quantization is actually related to clustering, thus related algorithms include a clustering step. Classical vector quantization methods include Linde-Buzo-Gray (LBG) algorithm (which is an extent of Lloyd-Max algorithm), pair-wise nearest neighbor (PNN) algorithm, the simulated annealing (SA) algorithm, and the fuzzy c-means clustering analysis (FCM) algorithm. This paper [8] compares the performance of several algorithms.

3 METHODS

3.1 Linear Quantization

If quantization is the simplest and most common idea among lossy compression techniques, then linear quantization (or uniform quantization) is the simplest and most common idea among quantization methods. It assumes that the data are uniformly distributed within a given interval, and that given the number of intervals, they can be divided into equal-length subintervals. The linear quantization is performed by mapping the input values to the index of some subinterval. If the interval is in the range *[min, max]* and the number of quantized bits is *B*, the interval will be divided into 2^B pieces, and the quantization equation is:

$$\mathrm{LinearQuant(x)} = \mathrm{Round}\left(\frac{\mathrm{x} - \min}{\max - \min} \cdot 2^{\mathrm{B}}\right) \cdots\cdots (1)$$

the de-quantization equation is:

$$\mathrm{LinearDeQuant(q)} = \min + \frac{q \cdot (max - min)}{2^B} \cdots\cdots (2)$$

3.2 Logarithmic Quantization

Analyzing the BM25 score distribution of the dataset in this project reveals that it is not uniformly distributed, so that linear quantization cannot get a good result. The weight parameters in deep neural networks are mostly concentrated around 0, and the larger the network, the more it conforms to this rule. Because this rule is similar to the non-uniformly distributed BM25 score, we can begin by experimenting with quantization methods designed for deep neural networks. QuanSeries [9] proposes improved logarithmic quantization with minimum accuracy loss when applied to parameters from large scale networks.

The traditional method of logarithmic quantization simply rounds the exponents, which has various of disadvantages. To improve it, the paper extends the exponents from integer domain to the real number field thus quantized weights now have a higher density.

Firstly, it generates an arithmetic progression via equation (3) called QuanSeries, this would be kept as a lookup table. Parameter R is a positive real number. In practice, R could be tuned based on the distribution of values. Secondly, the quantization step would use *argmin* method to find the indexes of nearest values in lookup table, and these indexes are the quantized results.

$$QuanSeries = \{\frac{-Rx}{2^B} \mid (x \in 0 \sim 2^B) \cap (x \in Z), B = bitwidth\} \cdots\cdots (3)$$

$$LogQuant(x) = Index(x) = \underset{i \in 0 \sim 2^B - 1}{\operatorname{argmin}} \mid QuanSeries[i] - log_2|x| \mid \cdots\cdots (4)$$

Given that quantized results are the indexes of lookup table, dequantization is to simply get the exponents from table and compute the power of 2.

$$LogDeQuant(q) = 2^{QuanSeries[q]} \cdots\cdots (5)$$

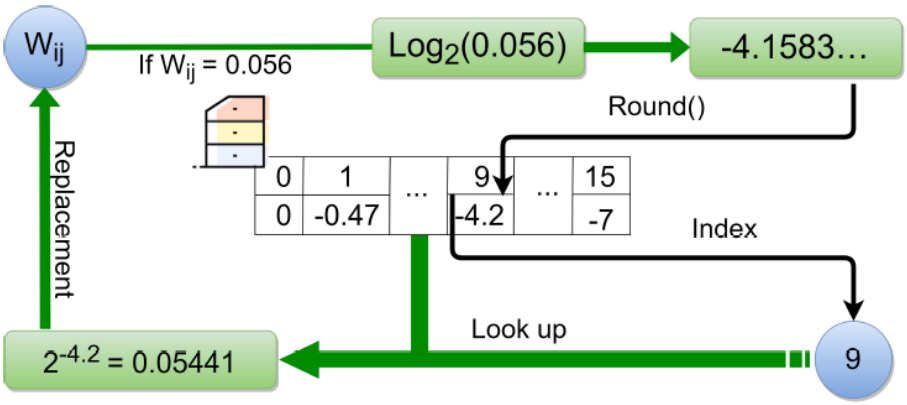

Figure 1 An overview of QuanSeries

3.3 Adaptive Float

Adaptive Float [10] is a floating-point quantization method that also designed for deep neural networks, and because it has some adaptive capabilities, we believe that it is likely to have good performance for quantization of impact scores as well.

The Adaptive Float number representation generally follows the IEEE 754 Standard floating-point format that includes a sign bit, exponent bit, and mantissa bit fields. At very low bit compression, the exponent representation becomes tricky. Thus, similar to integer quantization that uses a quantization scale (or step), we introduce a bias value, *exp_bias*, to dynamically shift the range of exponent values.

The code below describes the Adaptive Float quantization process. It specifies the total number of bits occupied by a number, *n*, as well as the number of bits occupied by the exponent bits, *e*. First, it determines the original data's sign digit *arr_sign* and absolute value *arr_abs*. Then, based on the maximum value of the absolute value, it finds *exp_bias* and *exp_max*, as well as the corresponding *value_min* and *value_max*, so that it can clip the values that are too large or too small in the data. A typical *exp_bias* should be negative, and by subtracting *exp_bias* from each exponent in the data, we get that the exponents are all positive, avoiding the problem of dealing with negative exponents for very small bits, thus simplifying the algorithm.

```
# Get Mantissa bits
m = n - e - 1
# Obtain sign and abs
arr_sign = sign(arr)
arr = abs(arr)
# Determine exp_bias and range
# Find normalized exp_max for max(arr) such that
# 2**exp_max < max(arr) < 2**exp_max+1
exp_bias = exp_max - (2**n_exp - 1)
```

```
value_min = 2**exp_bias*(1 + 2**(-m))
value_max = (2**exp_max)*(2 - 2**(-m))
# Handle unrepresentable values
arr[arr < 0.5*value_min] = 0
arr[(arr > 0.5*value_min) & (arr < value_min)] = value_min
arr[arr > value_max] = value_max
# get mantissa and exponent
mant, exp = frexp(arr)
# Quantize arr
q_exp = exp - exp_bias
q_mant = round(mant * (2**m-1))
```

4 IMPLEMENTATION

4.1 C++ Template Programming

We mentioned in section 1.2 that we significantly refactored the original code in this project to improve the program's scalability. For example, our code primarily consists of the classes listed below.

- Lexicon: Map a term to meta data of the index list
- Index: Includes the block-wise compressed data as well as the meta data of blocks.
- IndexForwardIter: Retrieve items from index list, decompress a block to the cache one at a time.
- IndexBackInserter: Append items to index list, compress a block from the cache when it's full.
- InputBuffer/OutputBuffer: Read/write data from/to disk in a sequential manner.

Obviously, when you need to save or read different types of data, you must derive different classes using C++ templates, and the shared logic in base class may also call the respective methods from derived classes, requiring some kind of polymorphism. When we talk about polymorphism, we usually mean runtime polymorphism, which requires the code to be specified by the virtual keyword, and there is a slight performance loss when calling different derived class methods at runtime via virtual function pointers. Therefore, in this project, we attempted to implement static polymorphism using a C++ programming style known as The Curiously Recurring Template Pattern (CRTP) to get rid of the runtime overhead.

```
template <typename T>
class Base {
public:
    friend class Derived;
    void doSomething() {
        T& derived = static_cast<T&>(*this);
        // use derived...
        derived.toBeCalledFromBase();
    }
};

class Derived :public Base<Derived> {
private:
    int x;
```

```
public:
    void toBeCalledFromBase() {
        x = ...;
    }
};
```

4.2 Memory Counter

During the inverted index building process, the input and output buffers should have their own memory limits. To accomplish this, the default memory allocation function must be "hooked" so that the exact amount of memory space allocated for specific objects can be counted. We are attempting to calculate memory for exploratory purposes only in this project. In actual engineering practice, because it is both difficult and meaningless to calculate the memory space usage for some specific objects, developers instead would limit the memory usage for an entire process, or just the number of elements in a container. This approach should never be used in any real-world system.

The C++ STL container classes use *`allocator`* to allocate memory, but the default allocator is stateless, which means we can't keep an variable to count. C++17 introduced stateful *`std::pmr::polymorphic_allocator`*, which is an allocator that exhibits different allocation behavior depending upon the *`std::pmr::memory_resource`* from which it is constructed. By overriding the allocation methods of *`std::pmr::memory_resource`*, we can easily sum up the memory consumption within specific containers.

In addition, we use *`mimalloc`*, a third-party memory pool, to speed up the program. *`mimalloc`* is a free and open-source compact general-purpose memory allocator developed by Microsoft with focus on performance characteristics. The library works as a drop-in replacement for malloc of the C standard library and requires no additional code changes.

4.3 Bits Vector

To save space, when floating point is quantized into bits of a certain width, the bits should be stored consecutively rather than separated by bytes. Although both C++ STL and BOOST provide similar bit array containers, we decided to create our own Bits Vector. Since our index data is arranged in blocks, we designed it so that bits are stored contiguously within a block and separated by bytes between different blocks. A Bits Vector representation diagram is depicted in Figure 2. Figure 3 shows the process of retrieving a number from bits kept across multiple bytes. The process of appending bits to the bits vector is equivalent to Figure 3's reverse process.

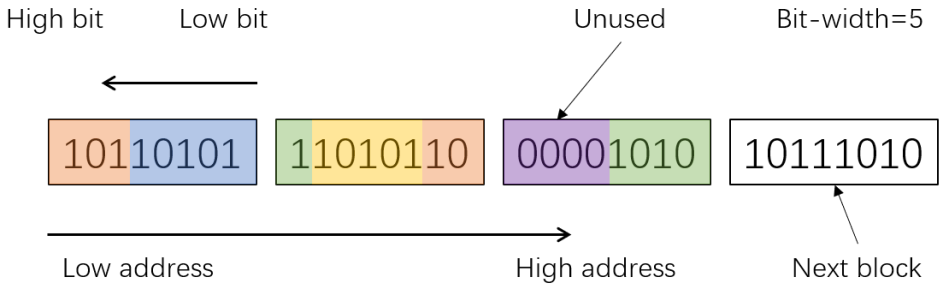

Figure 2 Representation of block-wise Bits Vector

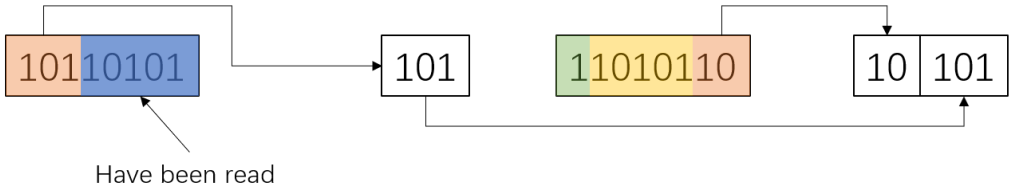

Figure 3 Read fixed length of bits from Bits Vector

4.4 Quantization

When implementing linear quantization, as mentioned in section 3.1, we first map the values to real numbers of the range *[0, 1]* and then to integers of the range *[0, 2^B-1]*. This brings up the question of whether we should multiply the real numbers by 2^B or by $2^B - 1$: The former prompts dealing with integer 2^B that exceeds the bit width, whereas the latter sacrifices some precision. During our experiments, we discovered that we can use a method that falls somewhere between the former and the latter. For example, setting *eps=0.501* and multiplying by $2^B - eps$ before rounding avoids the problem of dealing with overflow values, also it has higher precision than multiplying by $2^B - 1$.

In this case, the equations of linear quantization can be rewritten as:

$$\text{LinearQuant}(x) = \text{Round}\left(\frac{x - \min}{\max - \min} \cdot (2^B - 0.501)\right) \cdots\cdots (5)$$

$$\text{LinearDeQuant}(q) = \min + \frac{q \cdot (max - min)}{(2^B - 0.501)} \cdots\cdots (6)$$

Section 3.2 describes a method from the paper that is only applicable to the quantization of neural network weight parameters. It only accepts exponents in the range *[0, -R]* and cannot deal with negative floating point numbers. Because of that, we did not strictly adhere to the paper's specifications when implementing the logarithmic quantization. As shown in Figure 4, our proposed method is equivalent to linear quantization of exponents, where exponents of negative numbers are mapped to the integers *[1, 2^{m-1}-1]* and exponents of positive numbers are mapped to the integers *[2^{m-1}, 2^m-1]*, with 0 reserved.

For both linear quantization and logarithmic quantization, we find the minimum and maximum values across the entire dataset and save them in the index file.

```
values= [0.7,   -4.2, 24.1,   0.2, 0, -1.2]
exps=   [-0.51, 2.07, 4.59, -2.32, 0, 0.26]
```

$exp_{values<0} \in [0.26, 2.07]$ $exp_{values>0} \in [-2.32, 4.59]$

0	1	2	3	4	5	6	7
0	0.26	1.17	2.07	-2.32	-0.02	2.27	4.59

values<0 values>0

Figure 4 Actual implementation of Logarithmic Quantization

The Adaptive Float implementation differs from the first two because the extra data that should be saved is *exp_bias*, which takes up less than 8 bits. Hence we can run the algorithm in blocks, with each block conveniently keeping its own *exp_bias*. To deal with the mantissa, we first transform the mantissa from the range *[0.5, 1.0)* to *[0.0, 1.0)* and then quantize it in the same way as the first

two approaches.

```
eps = 0.501
# mant: [0.5, 1.0) -> [0.0, 1.0)
mant = 2*mant-1
if sign < 0:
    # q_mant: [1, 2**m-1]
    q_mant = round(mant * (2**m-1-eps)) + 1
elif sign > 0:
    # q_mant: [2**m, 2**(m+1)-1]
    q_mant = round(mant * (2**m-eps)) + 2**m
```

5 EXPERIMENT RESULTS

5.1 Sample Visualization

Prior to benchmarking, we sampled a small portion of impact scores, visualized the original data, and quantized the data with a typical 8-bit width to aid in analysis. Figures 5, 6, and 7 show the distance between the quantized values and the ground truth values. ***Surprisingly, linear quantization produces the best results, whereas adaptive float produces the worst.*** We believe this is because the minimum and maximum values of linear and logarithmic quantization are computed specifically based on the data set, making it more accurate than adaptive float, which only computes a maximum exponent. More importantly, the span of values' range is narrow, so the advantage of adaptive float is compromised.

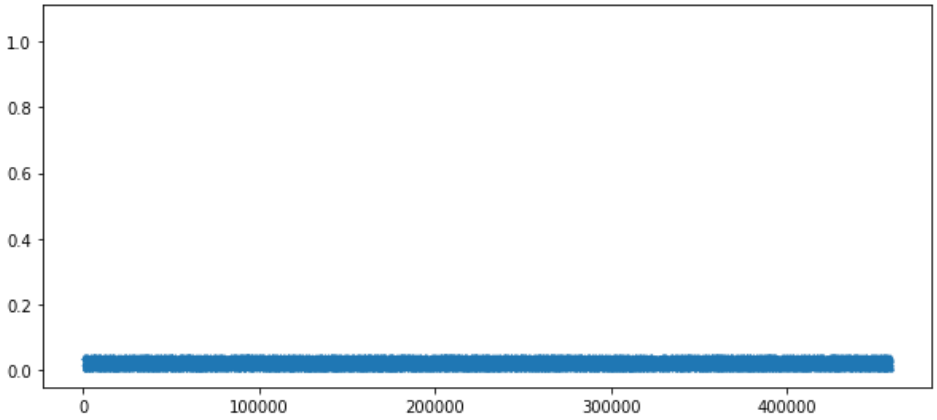

Figure 5 Distance between linear quantization and ground truth

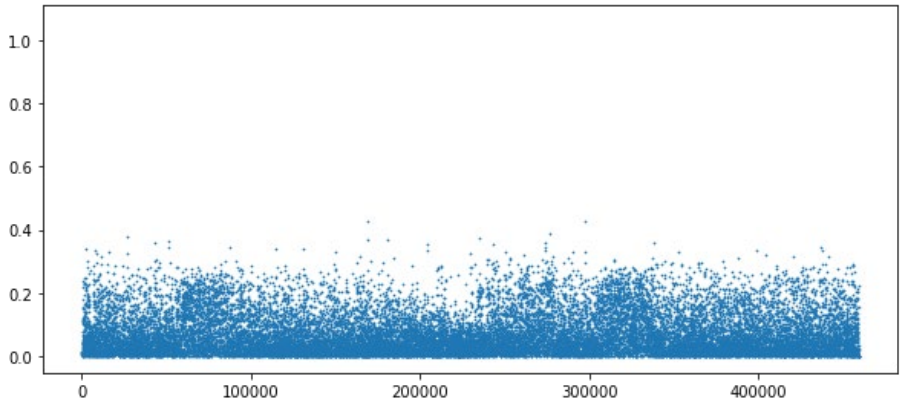

Figure 6 Distance between logarithmic quantization and ground truth

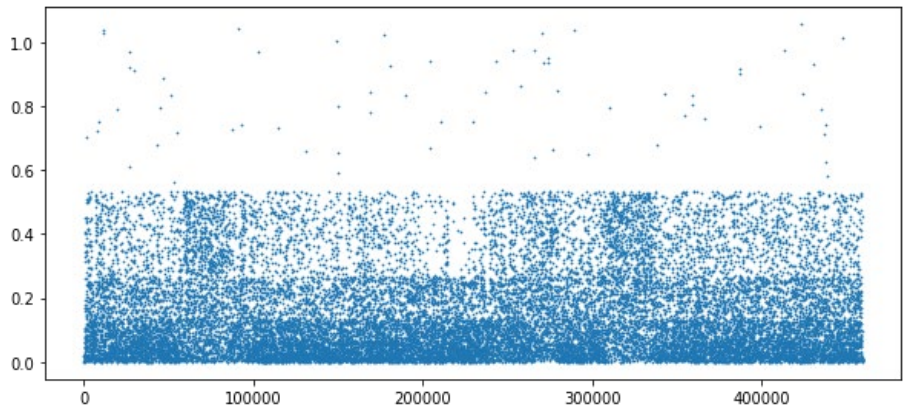

Figure 7 Distance between adaptive float and ground truth

5.2 Benchmark

We denote *Adaptive<B,E>* as *B* bits with *E* bits exponents. Table 1 shows the elapsed time to retrieve all the scores from file, the mean square error (MSE) with ground truth and size of compressed file.

Method	Speed	MSE
No precomputed scores	42.7sec	-
Uncompressed scores	38.9sec	-
Linear<6>	31.2sec	0.0266707
Linear<8>	31.9sec	0.00167407
Linear<10>	35.3sec	0.000104239
Linear<12>	37.3sec	6.50873e-06
Log<6>	36.4sec	0.201164
Log<8>	35.9sec	0.0121823
Log<10>	40.4sec	0.000757132
Log<12>	42.3sec	4.72691e-05
Adaptive<6,3>	34.7sec	0.0725162
Adaptive<8,4>	32.8sec	0.0157352
Adaptive<10,4>	39.9sec	0.000895969
Adaptive<12,4>	43.1sec	5.46161e-05

Bit-width	File Size
Uncompressed (32)	4.65GB
6	966MB
8	1.22GB
10	1.51GB
12	1.79GB

Table 1 Benchmark Result

As a result, we can conclude that linear quantization is the most effective method. Indeed, our impact score dataset accepts values in the range *[-6.59891, 29.5973]*, and such a small range with 8 bits is sufficient to achieve a relatively high accuracy without the need to find exponents as with logarithmic or adaptive float. In terms of scores' retrieval speed, the version without precomputation even outperforms some of the quantized versions. Based on our program profiling, this is due to the large number of bit operations requiring more CPU clock cycles than bytes that can be directly obtained from the address.

6 FUTURE WORKS

There are far too many methods in the field of quantization and compression, and we have only implemented three basic methods in this final project, which has numerous flaws. For example, all

three methods implemented are scalar quantization; none are vector quantization. Many experiment results in the literature show that vector quantization takes a long time and is difficult to converge, but it generally produces better results than scalar quantization. No one appears to have used these vector quantization algorithms for impact scores in recent years, so it is worth a shot. Furthermore, after quantization, additional compression using some of the generic compression methods described above can be considered.

Because of the time constraints, this project did not implement the actual query processing in various methods. However, the most important aspect of a search engine is the quality of query results. All these things should be taken into account in future work.

REFERENCES

[1] Mallia, A., Siedlaczek, M., Mackenzie, J., & Suel, T. (2019). PISA: Performant indexes and search for academia. Proceedings of the Open-Source IR Replicability Challenge.

[2] Masui, K., Amiri, M., Connor, L., Deng, M., Fandino, M., Höfer, C., ... & Vanderlinde, K. (2015). A compression scheme for radio data in high performance computing. Astronomy and Computing, 12, 181-190.

[3] Lindstrom, P. (2014). Fixed-rate compressed floating-point arrays. IEEE transactions on visualization and computer graphics, 20(12), 2674-2683.

[4] Di, S., & Cappello, F. (2016, May). Fast error-bounded lossy HPC data compression with SZ. In 2016 ieee international parallel and distributed processing symposium (ipdps) (pp. 730-739). IEEE.

[5] Tao, D., Di, S., Chen, Z., & Cappello, F. (2017, May). Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 1129-1139). IEEE.

[6] Liang, X., Di, S., Tao, D., Li, S., Li, S., Guo, H., ... & Cappello, F. (2018, December). Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In 2018 IEEE International Conference on Big Data (Big Data) (pp. 438-447). IEEE.

[7] Duwe, K., Lüttgau, J., Mania, G., Squar, J., Fuchs, A., Kuhn, M., ... & Ludwig, T. (2020). State of the Art and Future Trends in Data Reduction for High-Performance Computing. Supercomputing Frontiers and Innovations, 7(1), 4-36.

[8] Huang, C. M., & Harris, R. W. (1993). A comparison of several vector quantization codebook generation approaches. IEEE Transactions on Image Processing, 2(1), 108-112.

[9] Cai, J., Takemoto, M., & Nakajo, H. (2018, December). A deep look into logarithmic quantization of model parameters in neural networks. In Proceedings of the 10th International Conference on Advances in Information Technology (pp. 1-8).

[10] Tambe, T., Yang, E. Y., Wan, Z., Deng, Y., Reddi, V. J., Rush, A., ... & Wei, G. Y. (2019). Adaptivfloat: A floating-point based data type for resilient deep learning inference. arXiv preprint arXiv:1909.13271.