Using Rust to Implement WEASEL+MUSE and RustyDB

Akshar Narain

ABSTRACT

This paper describes my work during my Masters in Computer Science under Professor Stanley Zdonik. Part 1 describes the work regarding multivariate time series classifier WEASEL+MUSE. Part 2 describes the work regarding RustyDB, a datastore that uses multivariate timeseries compression based on an LSM-Tree design.

PART I: WEASEL+MUSE in Rust

1 INTRODUCTION

Multivariate time series data is generated when multiple values are being recorded at the same time over a given time period. One issue that is being addressed is the classification of such time series. Multivarate time series data, however, involves different features across different dimensions, generating a large amount of noise. Thus, the classifier WEASEL+MUSE [4] was developed to address this. In this section, I discuss my implementation of WEASEL+MUSE in the programming language Rust.

Algorithm 1 Build one BOP model using SFA, multiple window lengths, bigrams and the Chi-squared test for feature selection. *l* is the number of Fourier values to keep and *wLen* are the window lengths used for sliding window extraction.

```
1  function WEASEL_MUSE(mts, l, wLen)
2    bag = empty BagOfPattern
3    // extract from each dimension
4    for each dimId in mts:
5      // use multiple window lenghts
6      for each window-length w in wLen:
7        for each (prevWindow, window) in
             SLIDING_WINDOWS(mts[dimId], wLen):
8
9        // BOP computed from unigrams
10       word = SFA(window, l)
11       unigram = concat(dim,w,word)
12       bag[unigram].increaseCounts()
13
14       // BOP computed from bigrams
15       prevWord=SFA(prevWindow, l)
16       bigram =concat(dim,w,prevWord,word)
17       bag[bigram].increaseCounts()
18
19   // feature selection using ChiSquared
20   return CHI_SQUARED_FILTER(bag)
```

Figure 1: Weasel+Muse Algorithm

WEASEL+MUSE works by generating a bag of pattern model. Such a model keeps track of generated words called n-grams, and their occurrences, forming a histogram. Overall, it applies a sliding window of various lengths to each univariate timeseries per dimension of the multivariate time series, extracting multiple features, and uses Symbolic Fourier Approximation (SFA) to each window generated to yield a word from a unique alphabet. A unigram is created by concatenating the dimension of the timeseries with the window length and the generated word. This unigram is stored in the bag of patterns, with its number of counts incremented by 1. Likewise, a bigram is created by concatenating the dimension, window length, previous word and current word together, which is stored in the histogram. This process is detailed in Figure 1. Once the bag of patterns has been filled, it is passed through a chi squared filter, which removes certain patterns that fail to reach a threshold.

1.1 WEASEL+MUSE Implementation

For the Bag of Patterns, I create a struct in rust called Bag that has a single field that is a hash map. This maps an enum BagEntry to an integer respresenting its count. The enum BagEntry consists of two struct, Ugram and Bgram. Ugram has a field representing the dimension of the multivariate time series it represents, the length of the sliding window and the generated word. Bgram has the same fields and an additional one for the previous word. A multivariate time series is represented as a vector of vectors, each representing a univariate time series for a single dimension. To generate the bag of patterns, I iterate through each univerate time series of the multivariate time series. Then, for all window sizes from 4 to the maximum window size, I apply a sliding window function to the univariate time series. I use a dummy sliding window function that sums the elements in the window for testing purposes. Additionally, since I did not implement the symbolic fourier approximation, I simply used the generated window as the word. I create the Ugram and Bgram for each word and store their occurrences in the Bag's histogram. At the end, I return the bag. I then implemented a chi-squared filter algorithm based off the repository at [3].

2 FURTHER WORK

The work presented is a step towards implementing WEASEL+MUSE classifier in Rust. One next step would be to implement the Symbolic Fourier Approximation to generate unigrams and bigrams in the algorithm. Additionally, another area of work is indexing multidimensional time series data. Using WEASEL+MUSE can help to gain insights into time series representations that can help improve ways to index them.

PART II
RustyDB: Multivariate Gorilla Compression for LSM-Tree Based Timeseries Datastore

3 INTRODUCTION

Clickhouse is a column-oriented database management system (DBMS). However, since it is an all-purpose database, there is a chance it may not compress multivariate time series data optimally. Clickhouse compresses data separately for each column. In this part, I introduce RustyDB, a storage engine, with the goal of yielding a higher compression ratio and ingestion speed than those of Clickhouse. This was built along with fellow computer science Masters student Yida Yuan.

We generate a 1.43 GB file using the Time Series Benchmark Suite [6]. We then create compression and decompression algorithms for multivariate timeseries data. Gorilla is a reputable engine for data compression, but it focuses on single-dimension timeseries data (one value per timestamp) [2]. We use an implementation of Gorilla in Rust by Franco Solleza [5] and adapt it to support multivariate data compression. This indirectly tests the effectiveness of extending Gorilla to multivariate-data compression. We design our storage engine using an Log-Structured Merge-tree (LSM tree). Such a data structure is optimized for high write-throughput, so may lead to high data ingestion rate in our storage engine. Section 4 describes the implementation of our Gorilla-based compression algorithm. Section 5 describes our LSM tree. Section 6 describes the experiment we run to test the compression ratio of our LSM-based storage engine to that of Clickhouse. Section 7 presents the conclusions of our experiment. Section 8 discusses further work to be done.

4 MULTIVARIATE TIME SERIES COMPRESSION

Gorilla is an in-memory timeseries database developed by Facebook. It stores data by compressing it using delta of delta and XOR techniques. It only supports univariate timeseries data, that is, data for which there is one value per timestamp. We sought to extend this to support multivariate timeseries data. We used an implementation of Gorilla by Franco Solleza [5] in Rust. For our multivariate data, we must be able to compress the timestamp and values for each datapoint.

4.1 Timestamp Compression

Gorilla works by using a writer to write to a stream of bits, and a reader to extract the old values from the compressed bits. For timestamps, the writer keeps track of the previous written timestamp, and the previous delta, the difference between the last two consecutive timestamps. To compress a timestamp, Gorilla will calculate the delta between the timestamp and the previous timestamp. It will then compute the difference between the delta and the previous delta, known as the delta of deltas. Based on the delta of deltas, the writer will output a certain number of bits to represent this difference. A reader can go through these bits and discern the difference to calculate the original timestamp based on previous timestamp. Since there is still one timestamp for each datapoint in multivariate time series as well as univariate time series, we retain the compression algorithm for time.

4.2 Multidimensional Value Compression

The data under study is multidimensional. For each timestamp, there is more than one value, say n. For univariate time series data, the Gorilla writer keeps track of the previous value of the previous datapoint. It takes the XOR of the current value with the previous value and determines the number of leading and trailing zeroes around what are known as the meaningful bits, those that are not surrounded by zeroes. Then depending on whether the previous value had as many or fewer leading and trailing zeroes stores a bit representation of the value. To adapt this to multivariate time series, we specify a dimension of the data to the writer, n. We keep the previous value, but this time, it is a vector of n dimensions. We apply the univariate Gorilla compression algorithm to each corresponding index in the previous values and the current values, and output each result consecutively to the writer, yielding a stream of compressed values. The reader can use similar logic to decompress the values to their original format.

4.3 API

Our storage engine will store multiple compressed datapoints. Therefore, we created two functions that simplify compressing the data and retrieving the original values. One function takes in a vector of multivariate datapoints and returns a block of compressed values passed through our algorithm. The other function takes in a block of compressed values and decompresses it to their original values.

5 LSM-TREE

The data from the Multivariate Gorilla Compression algorithm is stored in our LSM-Tree based storage engine. The engine supports key-value storage.

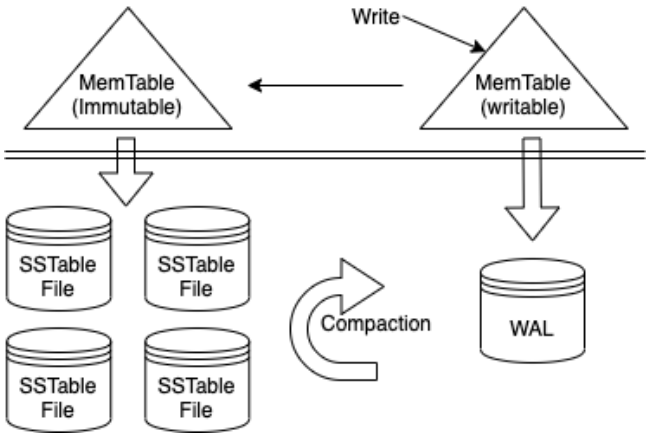

Figure 2: LSM-Tree Architecture

Figure 2 illustrates the components of the LSM tree. An in-memory buffer called *Memtable* stores all insertions. The in-memory buffer takes in a certain threshold of inserstions, and then waits to be flushed to disk as a Sorted String Table (SSTable) file, before accepting any more insertions. We use a Write Ahead Log (WAL) so that the system can recover from crashes. Finally, our storage engine compacts small SSTable files into larger ones as *Memtable* gets flushed to disk.

5.0.1 Keys. A typical time series consists of a set of tags, measurements, and data values. Our goal is to store the incoming multivariate data in batches of a certain size forming a *GorillaBlock*. We conjecture that a combination of tags, measurement, and the initial timestamp of the *GorillaBlock* can be used to uniquely identify a value. In addition, since tags and the measurement are long strings, storing them repetitively for all insertions is certainly inefficient. For efficiency, we pass the long strings of tags and values through a hash function that converts them to 8-byte integer values. We concatenate this hash to the initial timestamp to produce a 16-byte key.

5.0.2 Values. Since each key is 16 bytes, we want values (aka GorillaBlocks) to be significantly larger than keys. We store values as groups of compressed multivariate timeseries datapoints within a certain timeframe. Later, our experiments show the results of modifying the number of datapoints compressed into a GorillaBlock.

5.1 Memtable

In RustyDB, *Memtable* is a structure that contains a buffer holding key-value pairs in Rust's predefined `BTreeMap<String, String>`, which is efficient in performance and caching. It also keeps track of the size of SSTable files that will be generated when the `Memtable` is flushed to disk.

When a key-value pair is inserted into the LSM-Tree, we first check if it exceeds threshold. The default threshold is 4MB. If so, a compaction thread will flush the memtable's contents to disk, and the memtable becomes immutable. A new memtable is created for new insertions. The flushed data is stored in an SSTable file.

5.2 Sorted String Table (SSTable) Files

Figure 3 demonstrates how key-value pairs are stored inside SSTable.

As shown in the Data file format, RustyDB stores each key-value pair in order. Since both key and value are bytes, their lengths are also stored as 4 bytes integers in little endian formats before the real data bytes are stored. Whenever a key-value string pair is fully stored in SSTable, the current location of file cursor is also recorded as the starting location of the next tuple, which will be stored as the offset in the corresponding index file. Note that each offset is represented as a 4-byte integer which means RustyDB supports up to 4GB SSTables.

5.3 Compaction

Since RustyDB is designed to handle time series related workload, we anticipate that most queries will be asking data for the most recent period of time. In addition, the keys are closely related with the timestamps.

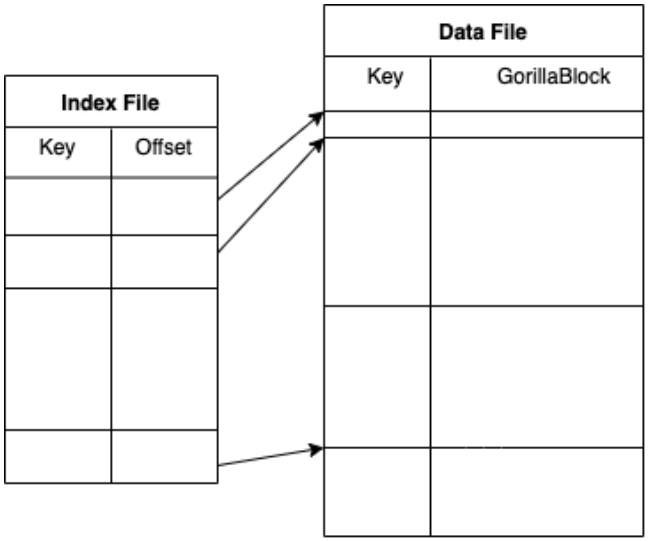

Figure 3: SSTable Index and Data File

RustyDB's SSTable compaction strategy is based on a similar strategy applied in Apache Cassandra [1]. By using this strategy, SSTable files will be more spread out on the time line. Figure 4 demonstrates how the compaction strategy works.

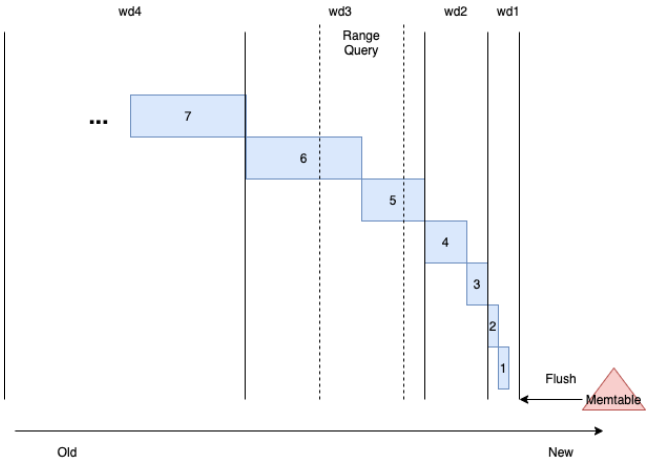

Figure 4: Date-Tier Compaction Strategy

In figure 4, insertions are going to *Memtable* first. Periodically, *Memtable* reaches a threshold and dumps all its data to a small SSTable which bounded by a certain time range.

6 EXPERIMENTS

6.1 Setup

The data was used was generated from the Time Series Benchmark Suite. The data consists of CPU measurements coming from 200 hosts over seven days in 10 second intervals. The file is 1.43 GB. We also use an AWS machine that has sufficient amount of memory where we conduct our experiments via remote access.

6.2 Clickhouse Test

On our remote machine, we install Clickhouse. We import our data file into clickhouse and run the following query:

```
"select name,
formatReadableSize(sum(data_compressed_bytes))
as compressed,
formatReadableSize(sum(data_uncompressed_bytes))
as uncompressed,
sum(data_compressed_bytes)
* 100 / sum(data_uncompressed_bytes)
as compress_ratio
from system.columns where table = 'cpu'
group by name with totals
order by sum(data_compressed_bytes);"
```

This query generates the following results shown in Figure 5.

name	compressed	uncompressed	compress_ratio
additional_tags	52.70 KiB	11.54 MiB	0.44615575396825397
created_date	111.15 KiB	23.07 MiB	0.47046130952380955
tags_id	219.17 KiB	46.14 MiB	0.4638537533068783
usage_softirq	25.75 MiB	103.82 MiB	24.80229828042328
usage_idle	25.76 MiB	103.82 MiB	24.81119470164609
usage_iowait	25.76 MiB	103.82 MiB	24.811398625808348
usage_guest_nice	25.76 MiB	103.82 MiB	24.81158234126984
usage_guest	25.76 MiB	103.82 MiB	24.81214175485009
usage_nice	25.76 MiB	103.82 MiB	24.81336989271017
usage_steal	25.76 MiB	103.82 MiB	24.813840204291594
usage_system	25.77 MiB	103.82 MiB	24.816967960023515
usage_user	25.77 MiB	103.82 MiB	24.820165527630806
usage_irq	25.79 MiB	103.82 MiB	24.836330651087597
created_at	31.87 MiB	46.14 MiB	69.07767237103175
time	57.77 MiB	299.93 MiB	19.262099041005293

Extremes:

name	compressed	uncompressed	compress_ratio
	347.65 MiB	1.43 GiB	23.730010766883307

Figure 5: Clickhouse Compression Results

The resuts show the various compressed and uncompressed data for different properties of inserting the data into Clickhouse. The summarized values at the bottom of the figure show the total compressed and uncompressed size of the data, and the composite compression ratio, which is about 23.7%. The benchmark suite also had an import script, which we could use to determine the import time of Clickhouse, which was 90 seconds.

6.3 RustyDB Test

Next, we test the performance of RustyDB on compressing the data. We test the import time and the compressed size at various different numbers of datapoints compressed together. The lowest number of datapoints we compress together to store as values in our engine is 10, and the highest is 500. The following graph plots the number of datapoints compressed together as blocks in our storage engine against the import time (red) and the compression size (black). Figure 6 summarizes our results.

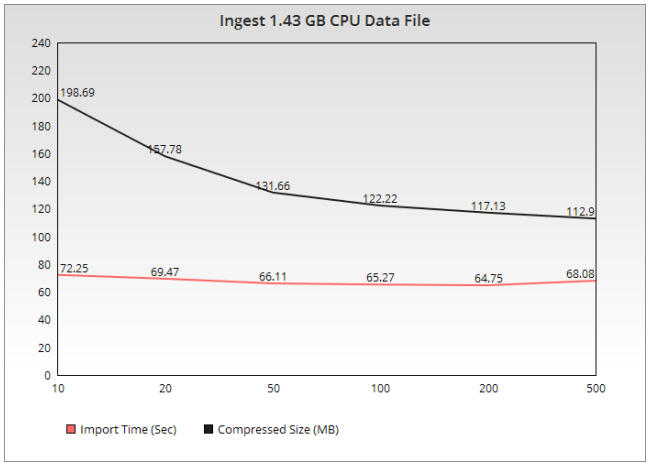

Figure 6: Performance for writing 1.43GB datafile into RustyDB

7 CONCLUSION

From Figure 6, we conclude that RustyDB has a lower compression ratio than Clickhouse for the dataset we generated from the Time Series Benchmark Suite. Additionally, RustyDB has a lower import time than Clickhouse's 90 seconds for the 1.43 GB data file. As we increase the number of datapoints compressed into a block, both the compression size and import time drop.

8 FURTHER WORK

These results suggest that an LSM-Tree based storage engine that uses our adapted Gorilla compression algorithm for multivariate time series could provide efficient storage with its low compression ratio, and be optimized for high write throughput systems with its low import time.

We only tested on one dataset, so one direction to consider would be testing RustyDB's performance on other datasets. Another thing to consider is how it performs compared to other key-value-based systems, like InfluxDB and Prometheus.

We also consider exploring concurrent writing to lower the import time more. This could be by way of fine-grained locking.

Finally, we also consider making SSTable compaction more efficient, by removing superfluous merging.

9 ACKNOWLEDGEMENTS

I am grateful to Stanley Zdonik and Franco Solleza for providing the resources necessary for this work to be possible. I am also grateful to Yida Yuan for his work with RustyDB.

REFERENCES

[1] [n.d.]. Date-Tiered Compaction in Apache Cassandra. https://labs.spotify.com/2014/12/18/date-tiered-compaction. Accessed: 2020-05-03.
[2] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. 2015. *Gorilla: A fast, scalable, in-memory time series database.* 1816–1827.
[3] Patrick Schäfer. 2019. SFA. https://github.com/patrickzib/SFA/blob/58f90bc9b3855488a1741ecf825a26b7a4d94afa/src/main/java/sfa/transformation/MUSE.java.
[4] Patrick Schäfer and Ulf Leser. 2017. *Multivariate Time Series Classification with WEASEL+MUSE.*

[5] Franco Solleza. 2020. Gorilla Rust Implementation. Private Repository (Contact: franco_solleza@brown.edu).
[6] Timescale. 2020. Time Series Benchmark Suite. https://github.com/timescale/tsbs.