Final Presentation

Akshar Narain
Yida Yuan

https://github.com/yyuan21/rustydb-release

Background

- We want to design a storage engine for multivariate timeseries data with tags
- The target database we chose to compare with is Clickhouse, which is a column-oriented DBMS that commonly used for timeseries data.
- Clickhouse keeps time-series data and tags into separate tables and uses joins for each query. In addition, it compress data separately for each column.
- We want to design the engine that has higher compression ratio, along with good ingestion speed and low query latency.

Plan

- Use an example dataset (CPU measurements) generated by Time Series Benchmark Suite for benchmarking

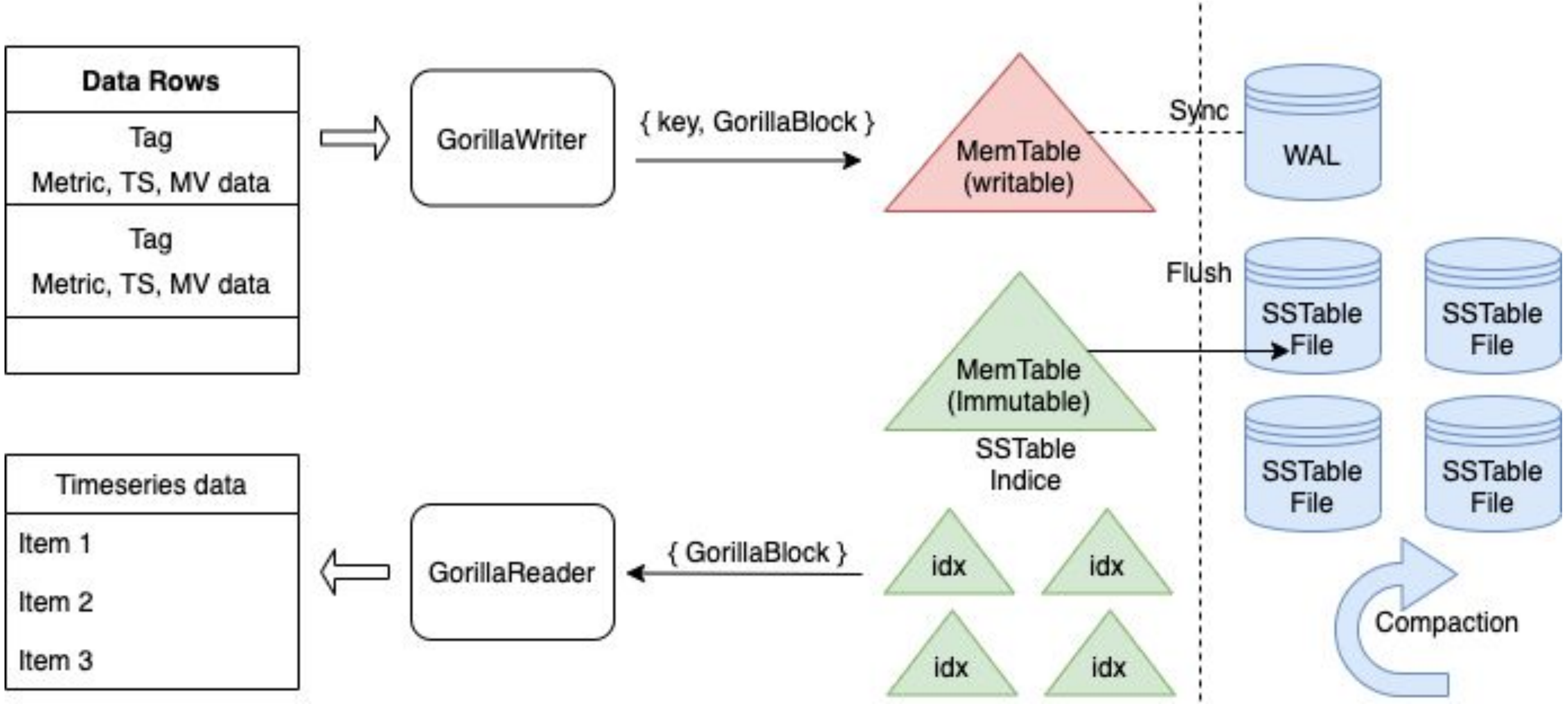
tags,hostname=host_0,region=eu-west-1,datacenter=eu-west-1c,rack=87,os=Ubuntu16.04LTS,arch=x64,team=NYC,
service=18,service_version=1,service_environment=production
Cpu,1577836800000000000,58,2,24,61,22,63,6,44,80,38

- Our main focus is to adapt and extend the compression algorithm introduced in Gorilla paper to multivariate time-series in order to obtain a good compression ratio.
- Our storage engine will be based on a LSM tree model (similar to LevelDB/RocksDB) to aim for high write throughput as well as good read throughput for recent data.
- We will be using Rust to help achieve high code efficiency and ensure data integrity.

Implementation

- Adapted the Gorilla compression algorithm to compress multivariate time-series data
    - Built an API to compress and decompress multiple multivariate datapoints.
- Built an LSM tree to store compressed data points because of its optimization for high write throughput.
- Experiment
    - Start with a 1.6GB file of clickhouse data.
        - Data is cpu measurements coming from 200 hosts over 7 days, in 10 second intervals
    - For each 100 datapoints, compress using multivariate compression algorithm and store in LSM tree, which periodically commits this data to files.
    - We measure the compression ratio by dividing the size of all the files by the size of the original data (1.6 GB).

Workflow


Data Rows
Tag
Metric, TS, MV data
Tag
Metric, TS, MV data
GorillaWriter
{ key, GorillaBlock }
MemTable
(writable)
Sync
WAL
Flush
MemTable
(Immutable)
SSTable
Indice
SSTable
File
SSTable
File
SSTable
File
SSTable
File
idx
idx
idx
idx
Compaction
Timeseries data
Item 1
Item 2
Item 3
GorillaReader
{ GorillaBlock }

Compression algorithm

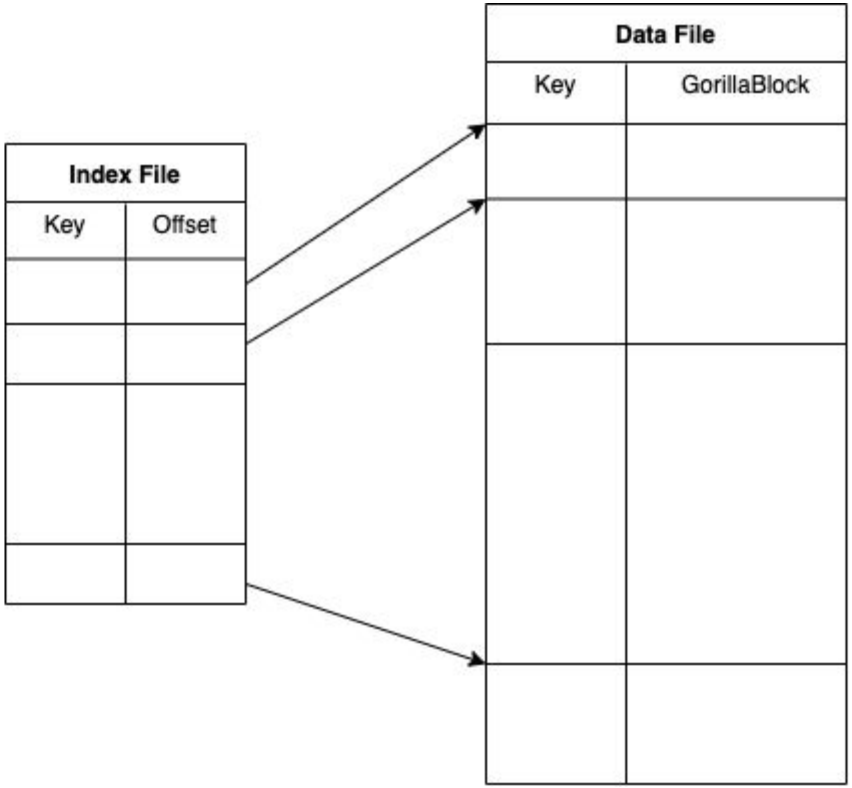
- Original gorilla compresses a timestamp and a single value
- We adapted gorilla such that it compresses multiple values
- Gorilla compresses values by keeping track of the previous values to determine the operations it performs
- We keep a vector of previous values specified by the dimension of our multivariate data.
- A function to compress a n-dimensional data point will run the single-value Gorilla compression algorithm on each index using the corresponding index of the previous data point.
- Time is compressed the same

Compressing/Decompressing Multiple Datapoints

- Our LSM tree will store a block of compressed datapoints.
- We build a functions to compress and decompress multiple values
- Compress_values:
  - Input: Vector of multivariate datapoints, starting time, dimension of data
  - Output: a block of compressed values
  - **We serialize the output as a string to store in the LSM tree
- Retrieve_values:
  - Input: deserialized gorilla block, dimension of data, number of values to decompress
  - Output: Vectore of decompressed datapoints.

Storage (SSTable)

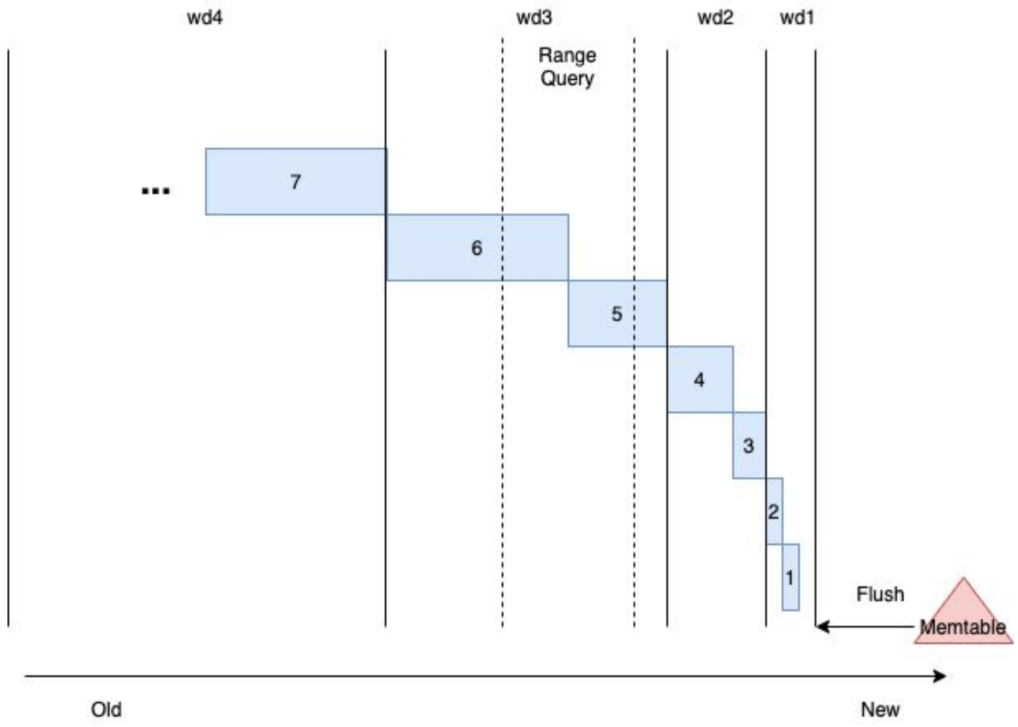
- An SSTable data files contains {key, Gorillablock} pairs
- An SSTable index file contains {key, offset} pairs
- Each SSTable data file will have a corresponding index file.
- Index files with recent time window will be loaded into memory



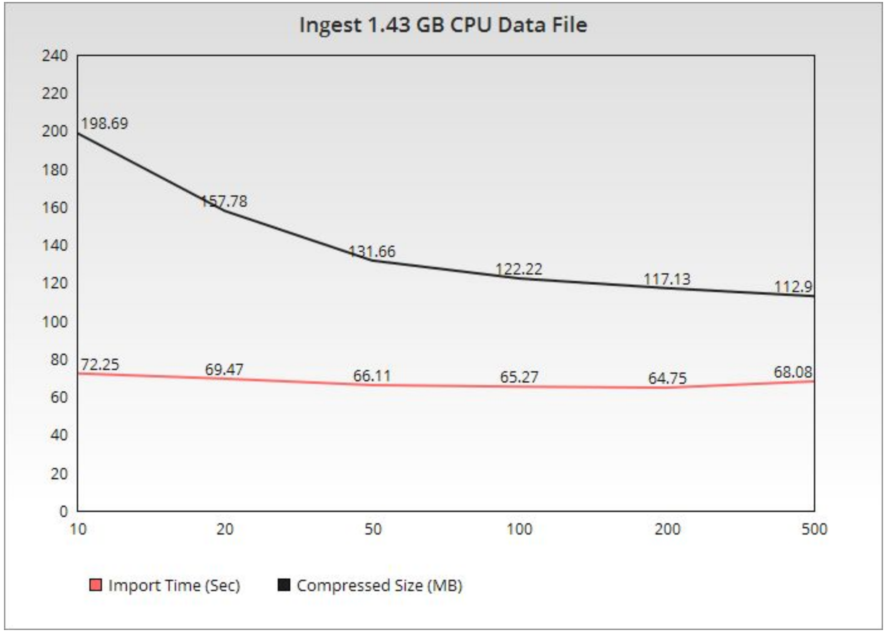
Storage (Compaction)

Date-Tiered Compaction

- Memtable will be flushed to disk if reaches a threshold
- Most recent data on the right, either on memtable or small SST files
- Older data gets compacted less frequently



Experiments



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

Conclusion

- Our experiments was running on AWS EC2 machine with 8GB of memory and 100GB of SSD
- Our multivariate Gorilla Compression algorithm appears to yield a better compression ratio for the 1.43 GB file.
- Our LSM Tree based storage engine also has a relatively high write throughput (~ 65 seconds). ClickHouse takes ~95 seconds to ingest the exact same datafile.

Further Research

- Test performance of range queries
- Design a better way to represent tags
- Compare performance to other TSDB like InfluxDB and Prometheus
- Carefully apply fine grained locking to improve the write throughput further.
- Check if we can make the SSTable compaction process more efficient. For example, the “Dostoevsky” paper presents an interesting idea about removing superfluous merging to obtain better space-time trade-off.