

PROJECT PROPOSAL

ALGORITHM ENGINEERING (CSCI 6105)

Mayursinh Sarvaiya (B00918007). may.ur.sinh@dal.ca

Project Topic

Space and Time efficient key-value storage engine optimized for time series databases.

Project Motivation

Free lunch is over [1]. It was published in way back 2005. In summary, it says that computers won't go any faster than they would have been in previous years. We would need to care about memory and CPU whenever we implement Algorithms and Data Structures.

In-memory key-value databases are everywhere and depending on the use case, there are multiple ways to manage data efficiently. For example, Memcached is a distributed key-value store which handles billions of requests per second on Facebook [2].

There is one particular use case that is crucial for diagnostic complex microservices performance, which is a time series database. Imagine if you spin up your API server and you want to know the memory and CPU consumption for each second. Two data points are what you should care about, one is time, and the other is the value (could be anything varying from memory snapshot, CPU usage, API request performance) at that time. That is the time series database. If you are deploying on a Kubernetes cluster, then you can install Prometheus for observability [3]. The Time series database is the sub-component of Prometheus architecture. Prometheus uses a combination of methodologies used in LevelDB and Gorilla [5] to manage data.

The purpose of this project is to try a different optimized algorithm than Prometheus to implement a basic time-series database from scratch and compare the results.

Hypothesis

If our time series database uses the techniques of SlimDB [7] then as mentioned in the findings, we should have a faster-performing and space-efficient database engine than LevelDB. As a result, it should outperform the TSDB (time series database) used in Prometheus.

Vanilla Stepped-Merge algorithm reduces the write amplification by “r” (See Table 1 in [7]). It also balances the memory cost. However, the negative lookup ratio is almost “r”. Thus, we are sure that with vanilla stepped-merge algorithm implementation, we can get more write throughput, lower memory footprint and if we combine with three-level index, we are half the negative lookup with slight increase of memory usage. SlimDB with stepped-merge algorithm outperform LevelDB on Insertion operations Kops per second and write amplification, on 110GB of data insertion (See Table 7 in [7]).

Tests

Prometheus has a dedicated separate package for TSDB written in Golang [8]. I will be able to use it to benchmark it against our implementation. Basically, there are 2 core functions that power time series database.

1. Append(series, timestamp, value)
2. Find(metric), Find(rangeStart int64, rangeEnd int64, metric string), Find(metric string, series map[string]interface{}), Find(rangeStart int64, rangeEnd int64, metric string, series map[string]interface{})

Generate random data points of 2 metrics (CPU usage and Memory usage). Each metric has the range of data over one day with interval of 5 milliseconds (read Data Source section for more information how to generate). This is more than 3 million data points for 2 metrics.

Setup script that continuously (every 5-10 milliseconds) logs machine resource utilization, more specifically CPU usage, Disk usage and Memory usage [20].

Write 2 identical scripts where one uses package of this project and other one uses from Prometheus upstream [8]. Common in both would be to read and transform input, use Append operation.

Run both programs one after another while also running a script that monitors machine usage.

Write 2 identical scripts to measure Find operation latency. The latency can be measured with the time difference before and after running the operation.

Results

Results after the test would be compared in the format below.

1. Heap Memory – Line graph (X axis – time in seconds interval, Y axis – memory usage in MB)
2. CPU cores usage – Line graph (X axis – time in seconds interval, Y axis – total cores used)
3. Disk write – Line graph (X axis – time in seconds interval, Y axis – disk writes in MB)
4. Disk size – Line graph (X axis – time in seconds interval, Y axis – disk size in MB)
5. Find operation latency – Line graph (X axis – time in seconds interval, Y axis – latency in milliseconds)

Literature Review

Why is building a time-series database a general problem?

In general, medium software companies are generating over 150 million time series data points each day. Some of the constraints are that these data points might not necessarily be queried often as time increases or indexing is different and only depends on the time range. Traditional databases are not optimized for these types of data. Even if they are, it is a multi-month project for specialized backend developers. [6] Thus it makes sense to build an efficient database specialized for time series data. With the level of data ingestion, even a fraction of optimization results in huge changes in the long term.

Potential Algorithms and Implementation

There are lots of parts to consider when building a time series database. But here are the most interesting findings to lead me in the right direction for this project.

One thing that is sure if we want faster read/write is that we must store data in memory! Facebook builds an engine capable of storing 12 million data points per second in memory and a read latency of under 1 millisecond. This requires heavy compression (that is not lossy) and performant in-memory caching. Even if we ignore horizontal scaling, the benefit we get using the mentioned algorithms in this paper is worth the effort. [9]

We certainly cannot keep all the data in memory and at some point, we would need to flush the data in disk. The nature of time series data is that the latest data is useful for the end user

mostly. We can use this fact to flush data to disk after a certain amount of time. This approach is the use case of LSM Trees [10].

We can further optimize LSM-Tree using Stepped Merge Algorithm and its variants of Stepped Merge Algorithm with Multi-level cuckoo filter and three-level index [7].

Primarily, I will use [11] paper as a guide for implementation. However, that is just a simplified version of [7]. Other than that, I will use [9] to cherry-pick algorithm.

Data Source

Time-series data unit will look like this.

<timestamp-in-Unix> - <metric-name> - field1=value1, field2=value2, field3=value3... - value

For example, if I have CPU usage metric, it will roughly look like this

1644424932 – cpu_usage – node=host-1, process-1=60 – 85.5

I can use generate_series() function in PostgreSQL [19]. Here is a simple SQL query that generates random CPU usage data for every 50 milliseconds for 24 hours and store in Json file.

```
psql -U postgres -h 0.0.0.0 -P format=unaligned -P tuples_only=true -c "WITH
time_series AS (
```

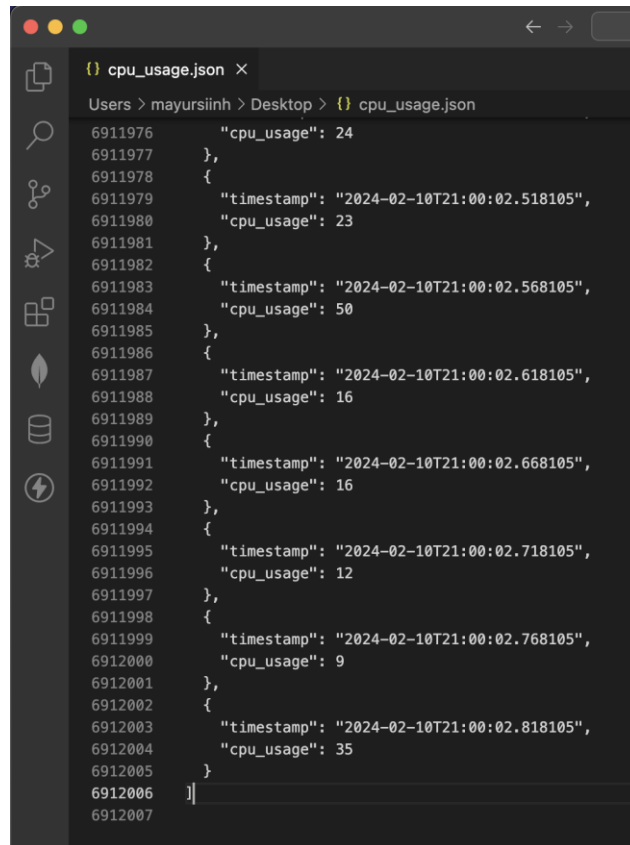
```
    SELECT generate_series(
        CURRENT_TIMESTAMP::timestamp,
        CURRENT_TIMESTAMP::timestamp + interval '1 day',
        interval '50 milliseconds'
    ) AS timestamp
```

```
)
```

```
SELECT
```

```
    json_agg(
        json_build_object(
            'timestamp', timestamp,
            'cpu_usage', round(random() * 100)::numeric(5,2)
        )
    )
```

```
FROM time_series;" | jq > cpu_usage.json
```



```
{
  "cpu_usage": 24
},
{
  "timestamp": "2024-02-10T21:00:02.518105",
  "cpu_usage": 23
},
{
  "timestamp": "2024-02-10T21:00:02.568105",
  "cpu_usage": 50
},
{
  "timestamp": "2024-02-10T21:00:02.618105",
  "cpu_usage": 16
},
{
  "timestamp": "2024-02-10T21:00:02.668105",
  "cpu_usage": 16
},
{
  "timestamp": "2024-02-10T21:00:02.718105",
  "cpu_usage": 12
},
{
  "timestamp": "2024-02-10T21:00:02.768105",
  "cpu_usage": 9
},
{
  "timestamp": "2024-02-10T21:00:02.818105",
  "cpu_usage": 35
}
}
```

Figure 1: sample timeseries input data of 24 hours in range of 50 milliseconds

This was sample data to generate only for CPU usage, I can generate same for RAM usage, Disk usage. I can also tune the query as needed and change the input data size. For example, this data is roughly 125MB. If I want 10x more data (~1GB), I can set intervals for 5 milliseconds or increase the range from 1 day to 10 days. It would just take a few seconds to generate a variety of data but for now it seems 2 metrics (CPU usage, memory usage) are enough to perform experiments.

Rough Plan

Algorithm Engineering Techniques to apply

I plan to apply Computer Models, Implementation and Experiments techniques for this project. Maybe Design of well-suited Algorithm(s) as well.

As this project is practical it needs implementation. It is not new thus experiments with alternatives using computer models. If I want better performance than alternatives, then it needs the Design of proper Algorithm(s).

Tentative Milestones

Define constraints: Database design is a hugely scoped task, to stay focused on the core topic I have to trade some things like query parsing, distributed stores, locking etc.

Design: Cherry-pick the algorithms from 2 core research papers. Write a Pseudocode that is close to implementation in code.

Implement: Choose the programming language (either Golang, C++ or C) and implement basic TSDB using designed Algorithms.

Benchmark (Experiment): Run experiments with huge data ingestion and compare against Prometheus implementation of TSDB.

Report and Presentation: Prepare the report and presentation of findings.

Team

Mayursinh Sarvaiya (B00918007) - may.ur.sinh@dal.ca

References

- [1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", *GotW.ca*. [Online]. Available: <http://www.gotw.ca/publications/concurrency-ddj.htm>. [Accessed 21st Jan. 2024].
- [2] R. Nishtala *et al*, "Scaling Memcache at Facebook", in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, Apr. 2-5, 2013.
- [3] O. Mart, C. Negru, F. Pop and A. Castiglione, "Observability in Kubernetes Cluster: Automatic Anomalies Detection using Prometheus," 2020 IEEE 22nd International Conference on High-Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Yanuca Island, Cuvu, Fiji, 2020, pp. 565-570, doi: 10.1109/HPCC-SmartCity-DSS50907.2020.00071.
- [4] "LevelDB" wikipedia.org. <https://en.wikipedia.org/wiki/LevelDB> (Accessed 21st Jan. 2024).
- [5] F. Reinartz. "Writing a Time Series Database from Scratch", archive.org. <https://web.archive.org/web/20210803115658/https://fabxc.org/tsdb/> (Accessed 21st Jan. 2024).
- [6] P.Dix. "Why Build a Time Series Data Platform?", db-engines.com. https://db-engines.com/en/blog_post/71 (Accessed 21st Jan. 2024).
- [7] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data," *Proceedings of the VLDB Endowment*, vol. 10, pp. 2037–2048, Sept. 2017.
- [8] github.com. <https://github.com/prometheus/prometheus/tree/main/tsdb> (Accessed 21st Jan. 2024).
- [9] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin

- Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, in-Memory Time Series Database. Proc. VLDB Endow. 8, 12 (Aug. 2015), 1816–1827. <https://doi.org/10.14778/2824032.2824078>.
- [10] "Log Structured Merge Tree" scylladb.com. <https://www.scylladb.com/glossary/log-structured-merge-tree/> (Accessed 21st Jan. 2024).
- [11] M. Weise. (Apr. 2020). On the Efficient Design of LSM Stores. [Online]. Available: <https://arxiv.org/pdf/2004.01833.pdf>.
- [12] github.com. <https://github.com/little-angry-clouds/prometheus-data-generator> (Accessed 21st Jan. 2024).
- [13] "Generate Test Prometheus TSDB Data" linuxczar.net. <https://linuxczar.net/blog/2019/03/13/prometheus-test-data> (Accessed 21st Jan. 2024).
- [14] github.com. <https://github.com/Marvin9/licensor?tab=readme-ov-file#demo> (Accessed 10th Feb. 2024).
- [15] github.com. <https://github.com/Marvin9/api-load-test> (Accessed 10th Feb. 2024).
- [16] github.com. <https://github.com/prometheus/prometheus/pulls?q=is%3Apr+author%3AMarvin9> (Accessed 10th Feb. 2024).
- [17] github.com. <https://github.com/backstage/backstage/pulls?q=is%3Apr+author%3AMarvin9> (Accessed 10th Feb. 2024).
- [18] github.com. <https://github.com/argoproj/argo-cd/pulls?q=is%3Apr+author%3AMarvin9> (Accessed 10th Feb. 2024).
- [19] R. Booz. "How to Create (Lots!) of Sample Time-Series Data With PostgreSQL generate_series()", timescale.com. https://www.timescale.com/blog/how-to-create-lots-of-sample-time-series-data-with-postgresql-generate_series/ (Accessed 10th Feb. 2024).
- [20] kloudvm. "Simple Bash Script to Monitor CPU, Memory and Disk Usage on Linux in 10 Lines of Code", medium.com . <https://kloudvm.medium.com/simple-bash-script-to-monitor-cpu-memory-and-disk-usage-on-linux-in-10-lines-of-code-e4819fe38bf1> (Accessed 10th Feb. 2024).