
tstorage reference

`tstorage` is a lightweight local on-disk storage engine for time-series data with a straightforward API. Especially ingestion is massively optimized as it provides goroutine safe capabilities of write into and read from TSDB that partitions data points by time.

Motivation

I'm working on a couple of tools that handle a tremendous amount of time-series data, such as Ali and Gosivy. Especially Ali, I had been facing a problem of increasing heap consumption over time as it's a load testing tool that aims to perform real-time analysis. I little poked around a fast TSDB library that offers simple APIs but eventually nothing works as well as I'd like, that's why I settled on writing this package myself.

To see how much `tstorage` has helped improve Ali's performance, see the release notes [here](#).

Usage

Currently, `tstorage` requires Go version 1.16 or greater

By default, `tstorage.Storage` works as an in-memory database. The below example illustrates how to insert a row into the memory and immediately select it.

```
1 package main
2
3 import (
4     "fmt"
5
6     "github.com/nakabonne/tstorage"
7 )
8
9 func main() {
10     storage, _ := tstorage.NewStorage(
11         tstorage.WithTimestampPrecision(tstorage.Seconds),
12     )
13     defer storage.Close()
14
15     _ = storage.InsertRows([]tstorage.Row{
16         {
17             Metric: "metric1",
18             DataPoint: tstorage.DataPoint{Timestamp: 1600000000, Value:
19                 0.1},
20         })
21 }
```

```
21     points, _ := storage.Select("metric1", nil, 16000000000, 16000000001)
22     for _, p := range points {
23         fmt.Printf("timestamp: %v, value: %v\n", p.Timestamp, p.Value)
24         // => timestamp: 16000000000, value: 0.1
25     }
26 }
```

Using disk

To make time-series data persistent on disk, specify the path to directory that stores time-series data through `WithDataPath` option.

```
1 storage, _ := tstorage.NewStorage(
2     tstorage.WithDataPath("./data"),
3 )
4 defer storage.Close()
```

Labeled metrics

In `tstorage`, you can identify a metric with combination of metric name and optional labels. Here is an example of insertion a labeled metric to the disk.

```
1 metric := "mem_alloc_bytes"
2 labels := []tstorage.Label{
3     {Name: "host", Value: "host-1"},
4 }
5
6 _ = storage.InsertRows([]tstorage.Row{
7     {
8         Metric:    metric,
9         Labels:    labels,
10        DataPoint: tstorage.DataPoint{Timestamp: 16000000000, Value:
11            0.1},
12    },
13 })
14 points, _ := storage.Select(metric, labels, 16000000000, 16000000001)
```

For more examples see the documentation.

Benchmarks

Benchmark tests were made using Intel(R) Core(TM) i7-8559U CPU @ 2.70GHz with 16GB of RAM on macOS 10.15.7

```

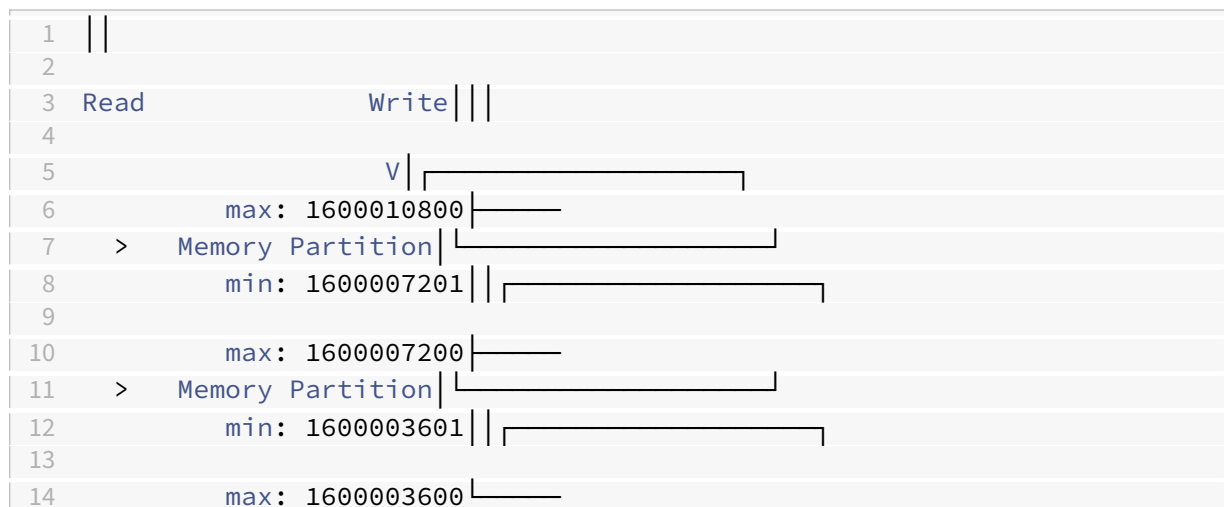
1 $ go version
2 go version go1.16.2 darwin/amd64
3
4 $ go test -benchtime=4s -benchmem -bench=. .
5 goos: darwin
6 goarch: amd64
7 pkg: github.com/nakabonne/tstorage
8 cpu: Intel(R) Core(TM) i7-8559U CPU @ 2.70GHz
9 BenchmarkStorage_InsertRows-8          14135685
   305.9 ns/op          174 B/op          2 allocs/op
10 BenchmarkStorage_SelectAmongThousandPoints-8  20548806
   222.4 ns/op          56 B/op          2 allocs/op
11 BenchmarkStorage_SelectAmongMillionPoints-8  16185709
   292.2 ns/op          56 B/op          1 allocs/op
12 PASS
13 ok      github.com/nakabonne/tstorage    16.501s

```

Internal

Time-series database has specific characteristics in its workload. In terms of write operations, a time-series database has to ingest a tremendous amount of data points ordered by time. Time-series data is immutable, mostly an append-only workload with delete operations performed in batches on less recent data. In terms of read operations, in most cases, we want to retrieve multiple data points by specifying its time range, also, most recent first: query the recent data in real-time. Besides, time-series data is already indexed in time order.

Based on these characteristics, `tstorage` adopts a linear data model structure that partitions data points by time, totally different from the B-trees or LSM trees based storage engines. Each partition acts as a fully independent database containing all data points for its time range.



```
15 > Disk Partition _____
16 min: 16000000000
```

Key benefits: - We can easily ignore all data outside of the partition time range when querying data points. - Most read operations work fast because recent data get cached in heap. - When a partition gets full, we can persist the data from our in-memory database by sequentially writing just a handful of larger files. We avoid any write-amplification and serve SSDs and HDDs equally well.

Memory partition

The memory partition is writable and stores data points in heap. The head partition is always memory partition. Its next one is also memory partition to accept out-of-order data points. It stores data points in an ordered Slice, which offers excellent cache hit ratio compared to linked lists unless it gets updated way too often (like delete, add elements at random locations).

All incoming data is written to a write-ahead log (WAL) right before inserting into a memory partition to prevent data loss.

Disk partition

The old memory partitions get compacted and persisted to the directory prefixed with `p-`, under the directory specified with the `WithDataPath` option. Here is the macro layout of disk partitions:

```
1 $ tree ./data
2 ./data|—
3 p-16000000001-1600003600|
4   |— data|
5   |— meta.json|—
6 p-1600003601-1600007200|
7   |— data|
8   |— meta.json|—
9 p-1600007201-1600010800|—
10  data|—
11  meta.json
```

As you can see each partition holds two files: `meta.json` and `data`. The `data` is compressed, read-only and is memory-mapped with `mmap(2)` that maps a kernel address space to a user address space. Therefore, what it has to store in heap is only partition's metadata. Just looking at `meta.json` gives us a good picture of what it stores:

```
1 $ cat ./data/p-16000000001-1600003600/meta.json
2 {
```

```
3  "minTimestamp": 16000000001,
4  "maxTimestamp": 1600003600,
5  "numDataPoints": 7200,
6  "metrics": {
7    "metric-1": {
8      "name": "metric-1",
9      "offset": 0,
10     "minTimestamp": 16000000001,
11     "maxTimestamp": 1600003600,
12     "numDataPoints": 3600
13   },
14   "metric-2": {
15     "name": "metric-2",
16     "offset": 36014,
17     "minTimestamp": 16000000001,
18     "maxTimestamp": 1600003600,
19     "numDataPoints": 3600
20   }
21 }
22 }
```

Each metric has its own file offset of the beginning. Data point slice for each metric is compressed separately, so all we have to do when reading is to seek, and read the points off.

Out-of-order data points

What data points get out-of-order in real-world applications is not uncommon because of network latency or clock synchronization issues; `tstorage` basically doesn't discard them. If out-of-order data points are within the range of the head memory partition, they get temporarily buffered and merged at flush time. Sometimes we should handle data points that cross a partition boundary. That is the reason why `tstorage` keeps more than one partition writable.

More

Want to know more details on `tstorage` internal? If so see the blog post: Write a time-series database engine from scratch.

Acknowledgements

This package is implemented based on tons of existing ideas. What I especially got inspired by are: - <https://misfra.me/state-of-the-state-part-iii> - <https://fabxc.org/tsdb> - <https://questdb.io/blog/2020/11/26/why-timeseries-data> - <https://akumuli.org/akumuli/2017/04/29/nbplustree> - <https://github.com/VictoriaMetrics/VictoriaMetrics>

A big “thank you!” goes out to all of them.