

Go optimizations in VictoriaMetrics

Aliaksandr Valialkin, CTO at VictoriaMetrics

About me

- <https://github.com/valyala>
- I'm fond of Go and performance optimizations
- Fasthttp author
- VictoriaMetrics core developer

Agenda

- What is VictoriaMetrics?
- What is time series?
- What does time series database do?
- Time series database architecture
- Inverted index implementations, issues and optimizations
- Specialized bitset implementation in Go

What is VictoriaMetrics?

What is VictoriaMetrics?

- Time series database

What is VictoriaMetrics?

- Time series database
- It is based on architecture ideas from ClickHouse
 - MergeTree data structure
 - Parallel computations on all the CPU cores
 - Process data in blocks that fit CPU cache

What is VictoriaMetrics?

- Time series database
- It is based on architecture ideas from ClickHouse
 - MergeTree data structure
 - Parallel computations on all the CPU cores
 - Process data in blocks that fit CPU cache
- Provides the best on-disk data compression

What is VictoriaMetrics?

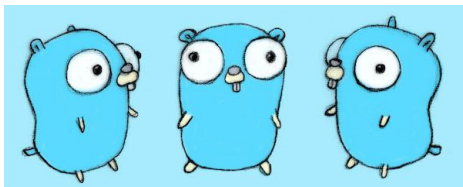
- Time series database
- It is based on architecture ideas from ClickHouse
 - MergeTree data structure
 - Parallel computations on all the CPU cores
 - Process data in blocks that fit CPU cache
- Provides the best on-disk data compression
- Scales vertically and horizontally

What is VictoriaMetrics?

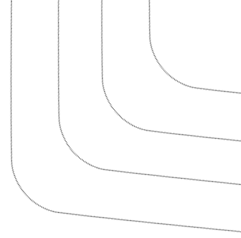
- Time series database
- It is based on architecture ideas from ClickHouse
 - MergeTree data structure
 - Parallel computations on all the CPU cores
 - Process data in blocks that fit CPU cache
- Provides the best on-disk data compression
- Scales vertically and horizontally
- Fast

What is VictoriaMetrics?

- Time series database
- It is based on architecture ideas from ClickHouse
 - MergeTree data structure
 - Parallel computations on all the CPU cores
 - Process data in blocks that fit CPU cache
- Provides the best on-disk data compression
- Scales vertically and horizontally
- Fast
- Written in Go



What is time series?



What is time series?

- A series of (timestamp, value) pairs

What is time series?

- A series of (timestamp, value) pairs
- Pairs are ordered by timestamp

What is time series?

- A series of (timestamp, value) pairs
- Pairs are ordered by timestamp
- Value is float64

What is time series?

- A series of (timestamp, value) pairs
- Pairs are ordered by timestamp
- Value is float64
- Each time series is uniquely identified by a key

What is time series?

- A series of (timestamp, value) pairs
- Pairs are ordered by timestamp
- Value is float64
- Each time series is uniquely identified by a key
- A key contains non-empty set of (label=value) pairs

What is time series?

- A series of (timestamp, value) pairs
- Pairs are ordered by timestamp
- Value is float64
- Each time series is uniquely identified by a key
- A key contains non-empty set of (label=value) pairs
- Example:

```
{__name__="cpu_usage", instance="my-server", datacenter="us-east"}
```

```
(t1, 10), (t2, 20), (t3, 12), .... (tN, 15)
```

Time series applications

Time series applications

- DevOps - CPU, RAM, network, rps, errors count

Time series applications

- DevOps - CPU, RAM, network, rps, errors count
- IoT - temperature, pressure, geo coordinates

Time series applications

- DevOps - CPU, RAM, network, rps, errors count
- IoT - temperature, pressure, geo coordinates
- Finance - stock prices

Time series applications

- DevOps - CPU, RAM, network, rps, errors count
- IoT - temperature, pressure, geo coordinates
- Finance - stock prices
- Industrial monitoring - sensors in wind turbines, factories, robots

Time series applications

- DevOps - CPU, RAM, network, rps, errors count
- IoT - temperature, pressure, geo coordinates
- Finance - stock prices
- Industrial monitoring - sensors in wind turbines, factories, robots
- Cars - engine health, tire pressure, speed, distance

Time series applications

- DevOps - CPU, RAM, network, rps, errors count
- IoT - temperature, pressure, geo coordinates
- Finance - stock prices
- Industrial monitoring - sensors in wind turbines, factories, robots
- Cars - engine health, tire pressure, speed, distance
- Aircraft and aerospace - black box, spaceship telemetry

Time series applications

- DevOps - CPU, RAM, network, rps, errors count
- IoT - temperature, pressure, geo coordinates
- Finance - stock prices
- Industrial monitoring - sensors in wind turbines, factories, robots
- Cars - engine health, tire pressure, speed, distance
- Aircraft and aerospace - black box, spaceship telemetry
- Healthcare - cardiogram, blood pressure

What does time series database do?

What does time series database do?

- Stores (timestamp, value) data points under the given key

What does time series database do?

- Stores (timestamp, value) data points under the given key
- Provides an API for querying time series:

What does time series database do?

- Stores (timestamp, value) data points under the given key
- Provides an API for querying time series:
 - By key

What does time series database do?

- Stores (timestamp, value) data points under the given key
- Provides an API for querying time series:
 - By key
 - By (label=value) pair

What does time series database do?

- Stores (timestamp, value) data points under the given key
- Provides an API for querying time series:
 - By key
 - By (label=value) pair
 - By a set of (label=value) pairs

What does time series database do?

- Stores (timestamp, value) data points under the given key
- Provides an API for querying time series:
 - By key
 - By (label=value) pair
 - By a set of (label=value) pairs
 - By a set of (label=<regexp>) pairs

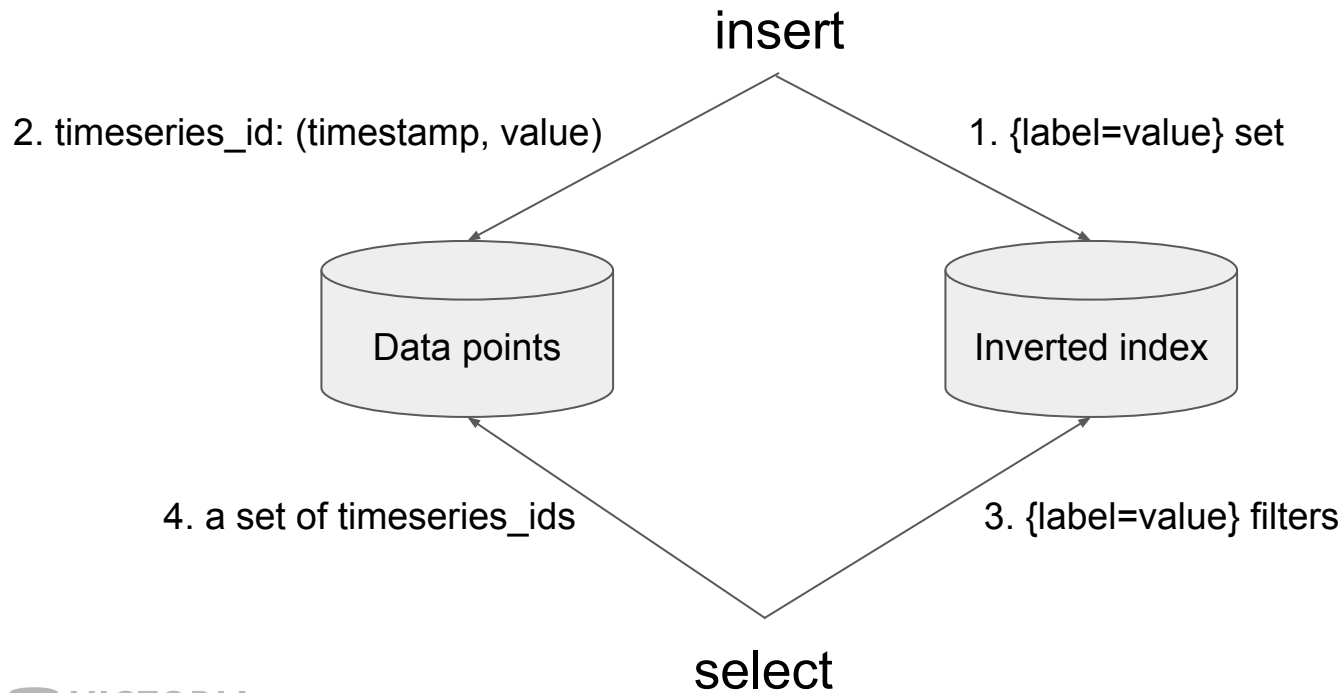
What does time series database do?

- Stores (timestamp, value) data points under the given key
- Provides an API for querying time series:
 - By key
 - By (label=value) pair
 - By a set of (label=value) pairs
 - By a set of (label=<regexp>) pairs
 - Example: `{__name__="cpu_usage", datacenter=~"us-.+"}` would select all the **cpu_usage** time series for all the datacenters in US

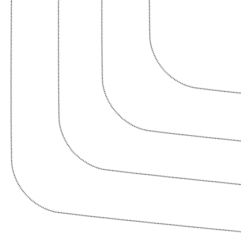
What does time series database do?

- Stores (timestamp, value) data points under the given key
- Provides an API for querying time series:
 - By key
 - By (label=value) pair
 - By a set of (label=value) pairs
 - By a set of (label=<regexp>) pairs
 - Example: `{__name__="cpu_usage", datacenter=~"us-.+"}` would select all the `cpu_usage` time series for all the datacenters in US
- Provides query language for time series data: PromQL, InfluxQL, Flux, Q

Time series database architecture



Query life



Query life

- Select all the `timeseries_ids` from inverted index that match the given set of (label=value or regexp) pairs

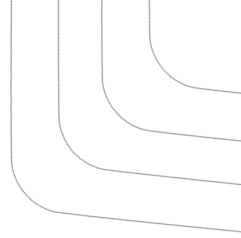
Query life

- Select all the `timeseries_ids` from inverted index that match the given set of (label=value or regexp) pairs
- Select all the data points for the given `timeseries_ids` set in the given time range

Query life

- Select all the `timeseries_ids` from inverted index that match the given set of (label=value or regexp) pairs
- Select all the data points for the given `timeseries_ids` set in the given time range
- Perform additional processing for the selected data points

Inverted index



Inverted index

- A (K: V) map

Inverted index

- A (K: V) map
- K is (label=value) pair

Inverted index

- A (K: V) map
- K is (label=value) pair
- V is a set of timeseries_ids for all the time series with the given (label=value) pair

Inverted index

- A (K: V) map
- K is (label=value) pair
- V is a set of timeseries_ids for all the time series with the given (label=value) pair
- Quickly finds all the timeseries_ids for the given (label=value) pair

Inverted index

- A (K: V) map
- K is (label=value) pair
- V is a set of timeseries_ids for all the time series with the given (label=value) pair
- Quickly finds all the timeseries_ids for the given (label=value) pair
- Quickly finds all the timeseries_ids for the given set of (label=value) pairs

Inverted index implementations

Inverted index: naive implementation

```
var invertedIndex = make(map[string][]int)

func getMetricIDs(labelValues []string) []int {
    metricIDs := invertedIndex[labelValue[0]]
    for _, labelValue := range labelValues[1:] {
        newMetricIDs := invertedIndex[labelValue]
        metricIDs = intersectInts(metricIDs, newMetricIDs)
    }
    return metricIDs
}
```

Inverted index: naive implementation issues

Inverted index: naive implementation issues

- Missing persistence - data is lost on process restart

Inverted index: naive implementation issues

- Missing persistence - data is lost on process restart
- Inverted index must fit RAM - doesn't scale to big number of time series

Inverted index: LevelDB

- Store **{(label=value): timeseries_id}** rows in LevelDB
- Extract all the timeseries_ids from all the rows for the given (label=value) pair

Inverted index: LevelDB

```
var invertedIndex *LevelDB

func getMetricIDs(labelValues []string) []int {
    metricIDs := invertedIndex.GetValues(labelValue[0])
    for _, labelValue := range labelValues[1:] {
        newMetricIDs := invertedIndex.GetValues(labelValue)
        metricIDs = intersectInts(metricIDs, newMetricIDs)
    }
    return metricIDs
}
```

Inverted index: LevelDB issues

Inverted index: LevelDB issues

- Slower than the naive implementation

Inverted index: LevelDB issues

- Slower than the naive implementation
- Cgo overhead for LevelDB and RocksDB

Inverted index: mergeset

Inverted index: mergeset

- Based on MergeTree data structure from ClickHouse

Inverted index: mergeset

- Based on MergeTree data structure from ClickHouse
- Optimized for fast inverted index lookups in VictoriaMetrics

Inverted index: mergeset

- Based on MergeTree data structure from ClickHouse
- Optimized for fast inverted index lookups in VictoriaMetrics
- Written in pure Go

Inverted index: mergeset

- Based on MergeTree data structure from ClickHouse
- Optimized for fast inverted index lookups in VictoriaMetrics
- Written in pure Go
- The API is similar to LevelDB or RocksDB

Inverted index: production issues

- High churn rate

Inverted index: production issues

- High churn rate
- High number of time series matching the given (label=value) pair (hundreds of millions)

Inverted index: production issues

- High churn rate
- High number of time series matching the given (label=value) pair (hundreds of millions)
- Matching (label=<regexp>)

High churn rate

High churn rate

- Certain labels are constantly changed

High churn rate

- Certain labels are constantly changed
- For instance, `{deployment_id="<deployment_id>"}` in Kubernetes

High churn rate

- Certain labels are constantly changed
- For instance, `{deployment_id="<deployment_id>"}` in Kubernetes
- Each deployment starts new set of time series

High churn rate

- Certain labels are constantly changed
- For instance, `{deployment_id="<deployment_id>"}` in Kubernetes
- Each deployment starts new set of time series
- The number of `{datacenter="us-east"}` time series grows over time

High churn rate

- Certain labels are constantly changed
- For instance, `{deployment_id="<deployment_id>"}` in Kubernetes
- Each deployment starts new set of time series
- The number of `{datacenter="us-east"}` time series grows over time
- But the number of `{datacenter="us-east"}` time series for **the last day** remains constant

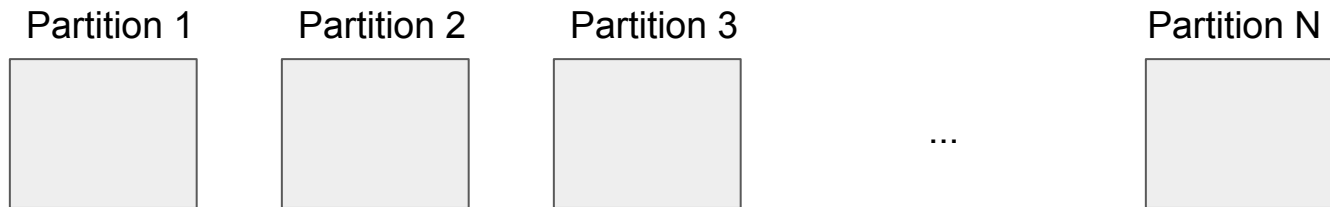
High churn rate

- Certain labels are constantly changed
- For instance, `{deployment_id="<deployment_id>"}` in Kubernetes
- Each deployment starts new set of time series
- The number of `{datacenter="us-east"}` time series grows over time
- But the number of `{datacenter="us-east"}` time series for **the last day** remains constant
- How to provide constant speed for selecting `{datacenter="us-east"}` time series for **the last day**?

High churn rate solutions

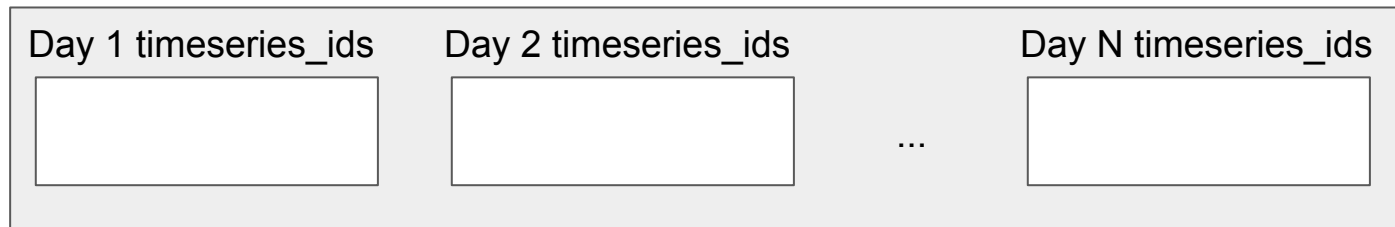
Partition inverted index by time

- Pros:
 - Simple
 - Fast
- Cons:
 - duplicates inverted index data for long-living time series



Per-day timeseries_ids sets for active time series

- Pros:
 - Index for long-living time series is stored only once
- Cons:
 - harder to implement
 - needs to scan bigger timeseries_ids sets



Solutions for high number of time series matching (label=value)

Store multiple timeseries_ids per mergeset row

- Pros:
 - requires less memory (especially for long (label=value) pairs)
 - improves scan speed
- Cons:
 - hard to implement
 - hard to debug

Original rows:

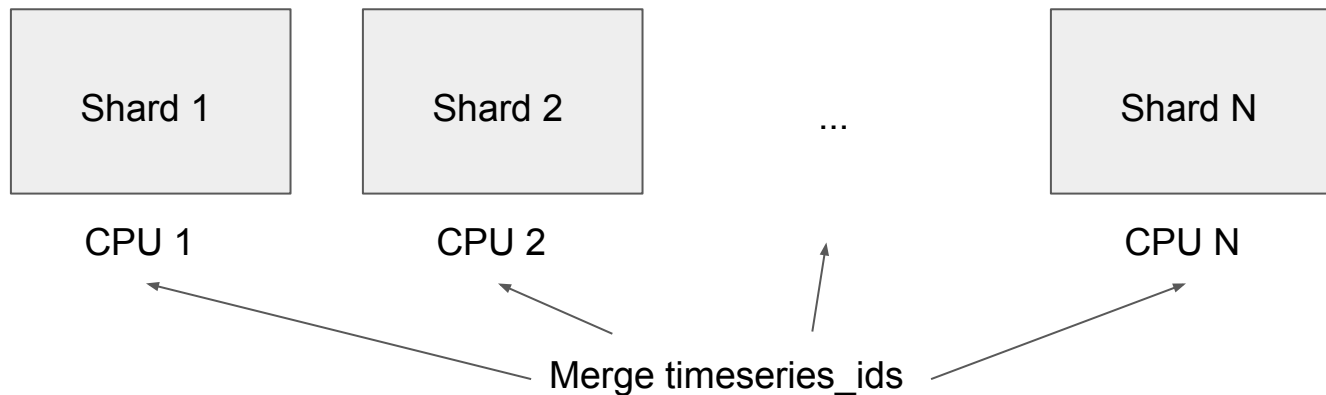
```
(label=value) timeseries_id1  
(label=value) timeseries_id2  
...  
(label=value) timeseries_idN
```

Optimized row:

```
(label=value) timeseries_id1, timeseries_id2, ... timeseries_idN
```

Shard inverted index by time series key (a set of (label=value) pairs)

- Pros:
 - scales to multiple CPUs
- Cons:
 - requires more CPU resources



Optimizations for timeseries_ids sets intersection

- `{datacenter="us-east", job="my-app", instance="my-host"}` requires intersection of three timeseries_ids sets:
 - `(datacenter="us-east")`
 - `(job="my-app")`
 - `(instance="my-host")`
- How to quickly intersect timeseries_ids sets?

Timeseries_ids sets intersection: naive approach

```
// intersectInts returns the intersection of a and b sets
func intereseectInts(a, b []int) []int {
    var result []int
    for _, x := range a {
        for _, y := range b {
            if x == y {
                result = append(result, x)
                break
            }
        }
    }
    return result
}
```

What's wrong with the naive approach?

What's wrong with the naive approach?

- It works slowly with big sets
- It has $O(N^2)$ complexity
- Let $\text{len}(a) == 1\text{M}$, $\text{len}(b) == 1\text{M}$
- Then the max number of iterations equals to **$\text{len}(a)*\text{len}(b)=1\text{M}*1\text{M}=1$ trillion**

Timeseries_ids intersection: map

```
// intersectInts returns the intersection of a and b sets
func intereseectInts(a, b []int) []int {
    m := make(map[int]bool)
    for _, x := range a {
        m[x] = true
    }
    var result []int
    for _, y := range b {
        if m[y] {
            result = append(result, y)
        }
    }
    return result
}
```

Set intersection with map

- Pros:
 - Has $O(N)$ complexity
- Cons:
 - Has high overhead on hashing

Timeseries_ids intersection: customized bitset

```
// intersectInts returns the intersection of a and b sets
func intereseectInts(a, b []uint64) []uint64 {
    s := &uint64set.Set{}
    for _, x := range a {
        s.Add(x)
    }
    var result []uint64
    for _, y := range b {
        if s.Has(y) {
            result = append(result, y)
        }
    }
    return result
}
```

Set intersection with customized bitset

- Pros:
 - Up to 10x better performance comparing to map-based intersection
 - Lower memory usage for big sets (>1M items)
- Cons:
 - Non-trivial implementation
 - Memory usage can explode if improperly used

Customized bitset: implementation details

- Located at lib/uint64set
- Optimized for dense serial timeseries_ids where higher 32 bits are constant
- Doesn't provide data persistence

lib/uint64set API

```
package uint64set // import "github.com/VictoriaMetrics/VictoriaMetrics/lib/uint64set"
```

```
type Set struct {  
    // Has unexported fields.  
}
```

Set is a fast set for uint64.

It should work faster than `map[uint64]struct{}` for semi-sparse uint64 values such as MetricIDs generated by lib/storage.

It is unsafe calling Set methods from concurrent goroutines.

```
func (s *Set) Add(x uint64)  
func (s *Set) AppendTo(dst []uint64) []uint64  
func (s *Set) Clone() *Set  
func (s *Set) Del(x uint64)  
func (s *Set) Has(x uint64) bool  
func (s *Set) Len() int
```

lib/uint64set internals

```
type Set struct {
    itemCount int
    buckets   []*bucket32
}
type bucket32 struct {
    hi      uint32
    b16his  []uint16
    buckets []*bucket16
}
type bucket16 struct {
    bits [wordsPerBucket]uint64
}
const (
    bitsPerBucket = 1 << 16
    wordsPerBucket = bitsPerBucket / 64
)
```

lib/uint64set internals: Set.Add

```
// Add adds x to s.
func (s *Set) Add(x uint64) {
    hi := uint32(x >> 32)
    lo := uint32(x)
    for _, b32 := range s.buckets {
        if b32.hi == hi {
            if b32.add(lo) {
                s.itemsCount++
            }
            return
        }
    }
    s.addAlloc(hi, lo)
}
```


lib/uint64set internals: bucket32.add

```
func (b *bucket32) add(x uint32) bool {
    hi := uint16(x >> 16)
    lo := uint16(x)
    if len(b.buckets) > maxUnsortedBuckets {
        return b.addSlow(hi, lo)
    }
    for i, hi16 := range b.b16his {
        if hi16 == hi {
            return b.buckets[i].add(lo)
        }
    }
    b.addAllocSmall(hi, lo)
    return true
}
```

lib/uint64set internals: bucket16.add

```
func (b *bucket16) add(x uint16) bool {  
    wordNum, bitMask := getWordNumBitMask(x)  
    word := &b.bits[wordNum]  
    ok := *word&bitMask == 0  
    *word |= bitMask  
    return ok  
}
```

More optimizations

More optimizations

- We covered small subset of VictoriaMetrics optimizations

More optimizations

- We covered small subset of VictoriaMetrics optimizations
- There are many more optimizations in the code

More optimizations

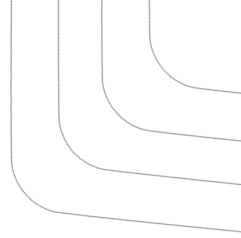
- We covered small subset of VictoriaMetrics optimizations
- There are many more optimizations in the code
- The majority of these optimizations are applied after `go tool pprof` analysis

More optimizations

- We covered small subset of VictoriaMetrics optimizations
- There are many more optimizations in the code
- The majority of these optimizations are applied after `go tool pprof` analysis
- Investigate VictoriaMetrics Go code - it is free and open source:

<https://github.com/VictoriaMetrics/VictoriaMetrics>

Questions?



Aliaksandr Valialkin, CTO at VictoriaMetrics