# 存储位置配置

#storage.local.path

/bin/prometheus -config.file=/etc/prometheus/prometheus.yml -storage.local.path=/prometheus -web.console.libraries=/usr/share/prometheus/console\_libraries -web.console.templates=/usr/share/prometheus/consoles

## 存储参数介绍

**内存使用**

prometheus在内存里保存了最近使用的chunks，具体chunks的最大个数可以通过storage.local.memory-chunks来设定，默认值为1048576，即1048576个chunk，大小为1G。 除了采用的数据，prometheus还需要对数据进行各种运算，因此整体内存开销肯定会比配置的local.memory-chunks大小要来的大，因此官方建议要预留3倍的local.memory-chunks的内存大小。

As a rule of thumb, you should have at least three times more RAM available than needed by the memory chunks alone

可以通过server的metrics去查看prometheus\_local\_storage\_memory\_chunks以及process\_resident\_memory\_byte两个指标值。

prometheus\_local\_storage\_memory\_chunks

The current number of chunks in memory, excluding cloned chunks 目前内存中暴露的chunks的个数

process\_resident\_memory\_byte

Resident memory size in bytes 驻存在内存的数据大小

prometheus\_local\_storage\_persistence\_urgency\_score

介于0-1之间，当该值小于等于0.7时，prometheus离开rushed模式。 当大于0.8的时候，进入rushed模式

prometheus\_local\_storage\_rushed\_mode

1表示进入了rushed mode，0表示没有。进入了rushed模式的话，prometheus会利用storage.local.series-sync-strategy以及storage.local.checkpoint-interval的配置加速chunks的持久化。

**storage.local.memory-chunks**

设定prometheus内存中保留的chunks的最大个数，默认为1048576，即为1G大小

**storage.local.retention**

用来配置采用数据存储的时间，168h0m0s即为24\*7小时，即1周

**storage.local.series-file-shrink-ratio**

用来控制序列文件rewrite的时机，默认是在10%的chunks被移除的时候进行rewrite，如果磁盘空间够大，不想频繁rewrite，可以提升该值，比如0.3，即30%的chunks被移除的时候才触发rewrite。

**storage.local.max-chunks-to-persist**

该参数控制等待写入磁盘的chunks的最大个数，如果超过这个数，Prometheus会限制采样的速率，直到这个数降到指定阈值的95%。建议这个值设定为storage.local.memory-chunks的50%。Prometheus会尽力加速存储速度，以避免限流这种情况的发送。

**storage.local.num-fingerprint-mutexes**

当prometheus server端在进行checkpoint操作或者处理开销较大的查询的时候，采集指标的操作会有短暂的停顿，这是因为prometheus给时间序列分配的mutexes可能不够用，可以通过这个指标来增大预分配的mutexes，有时候可以设置到上万个。

**storage.local.series-sync-strategy**

控制写入数据之后，何时同步到磁盘，有'never', 'always', 'adaptive'. 同步操作可以降低因为操作系统崩溃带来数据丢失，但是会降低写入数据的性能。 默认为adaptive的策略，即不会写完数据就立刻同步磁盘，会利用操作系统的page cache来批量同步。

**storage.local.checkpoint-interval**

进行checkpoint的时间间隔，即对尚未写入到磁盘的内存chunks执行checkpoint操作。

prometheus\_local\_storage\_memory\_series：当前的系列数量在内存中保存。   
prometheus\_local\_storage\_open\_head\_chunks：打开头块的数量。   
prometheus\_local\_storage\_chunks\_to\_persist：仍然需要将其持续到磁盘的内存块数。   
prometheus\_local\_storage\_memory\_chunks：目前在记忆中的块数。如果减去前两个，则可以得到持久化块的数量（如果查询当前没有使用，则它们是可驱动的）。   
prometheus\_local\_storage\_series\_chunks\_persisted：每个批次持续存在块数的直方图。   
prometheus\_local\_storage\_rushed\_mode如果prometheus斯处于“冲动模式”，则为1，否则为0。可用于计算prometheus处于冲动模式的时间百分比。   
prometheus\_local\_storage\_checkpoint\_last\_duration\_seconds：最后一个检查点需要多长时间   
prometheus\_local\_storage\_checkpoint\_last\_size\_bytes：最后一个检查点的大小（以字节为单位）。   
prometheus\_local\_storage\_checkpointing是1，而prometheus是检查点，否则为0。可以用来计算普罗米修斯检查点的时间百分比。   
prometheus\_local\_storage\_inconsistencies\_total：找到存储不一致的计数器。如果大于0，请重新启动服务器进行恢复。   
prometheus\_local\_storage\_persist\_errors\_total：反对持续错误。   
prometheus\_local\_storage\_memory\_dirty\_series：当前脏系列数量。   
process\_resident\_memory\_bytes广义地说，prometheus进程所占据的物理内存。   
go\_memstats\_alloc\_bytes：去堆大小（分配的对象在使用中加分配对象不再使用，但尚未被垃圾回收）。

## 命令行显示参数

usage: prometheus [<args>]

-version false

Print version information.

-config.file "prometheus.yml"

Prometheus configuration file name.

== ALERTMANAGER ==

-alertmanager.notification-queue-capacity 10000

The capacity of the queue for pending alert manager notifications.

-alertmanager.timeout 10s

Alert manager HTTP API timeout.

-alertmanager.url

Comma-separated list of Alertmanager URLs to send notifications to.

== LOG ==

-log.format "\"logger:stderr\""

Set the log target and format. Example:

"logger:syslog?appname=bob&local=7" or "logger:stdout?json=true"

-log.level "\"info\""

Only log messages with the given severity or above. Valid levels:

[debug, info, warn, error, fatal]

== QUERY ==

-query.max-concurrency 20

Maximum number of queries executed concurrently.

-query.staleness-delta 5m0s

Staleness delta allowance during expression evaluations.

-query.timeout 2m0s

Maximum time a query may take before being aborted.

== STORAGE ==

-storage.local.checkpoint-dirty-series-limit 5000

If approx. that many time series are in a state that would require

a recovery operation after a crash, a checkpoint is triggered, even if

the checkpoint interval hasn't passed yet. A recovery operation requires

a disk seek. The default limit intends to keep the recovery time below

1min even on spinning disks. With SSD, recovery is much faster, so you

might want to increase this value in that case to avoid overly frequent

checkpoints. Also note that a checkpoint is never triggered before at

least as much time has passed as the last checkpoint took.

-storage.local.checkpoint-interval 5m0s

The time to wait between checkpoints of in-memory metrics and

chunks not yet persisted to series files. Note that a checkpoint is never

triggered before at least as much time has passed as the last checkpoint

took.

-storage.local.chunk-encoding-version 1

Which chunk encoding version to use for newly created chunks.

Currently supported is 0 (delta encoding), 1 (double-delta encoding), and

2 (double-delta encoding with variable bit-width).

-storage.local.dirty false

If set, the local storage layer will perform crash recovery even if

the last shutdown appears to be clean.

-storage.local.engine "persisted"

Local storage engine. Supported values are: 'persisted' (full local

storage with on-disk persistence) and 'none' (no local storage).

-storage.local.index-cache-size.fingerprint-to-metric 10485760

The size in bytes for the fingerprint to metric index cache.

-storage.local.index-cache-size.fingerprint-to-timerange 5242880

The size in bytes for the metric time range index cache.

-storage.local.index-cache-size.label-name-to-label-values 10485760

The size in bytes for the label name to label values index cache.

-storage.local.index-cache-size.label-pair-to-fingerprints 20971520

The size in bytes for the label pair to fingerprints index cache.

-storage.local.max-chunks-to-persist 0

Deprecated. This flag has no effect anymore.

-storage.local.memory-chunks 0

Deprecated. If set, -storage.local.target-heap-size will be set to

this value times 3072.

-storage.local.num-fingerprint-mutexes 4096

The number of mutexes used for fingerprint locking.

-storage.local.path "data"

Base path for metrics storage.

-storage.local.pedantic-checks false

If set, a crash recovery will perform checks on each series file.

This might take a very long time.

-storage.local.retention 360h0m0s

How long to retain samples in the local storage.

-storage.local.series-file-shrink-ratio 0.1

A series file is only truncated (to delete samples that have

exceeded the retention period) if it shrinks by at least the provided

ratio. This saves I/O operations while causing only a limited storage

space overhead. If 0 or smaller, truncation will be performed even for a

single dropped chunk, while 1 or larger will effectively prevent any

truncation.

-storage.local.series-sync-strategy "adaptive"

When to sync series files after modification. Possible values:

'never', 'always', 'adaptive'. Sync'ing slows down storage performance

but reduces the risk of data loss in case of an OS crash. With the

'adaptive' strategy, series files are sync'd for as long as the storage

is not too much behind on chunk persistence.

-storage.local.target-heap-size 2147483648

The metrics storage attempts to limit its own memory usage such

that the total heap size approaches this value. Note that this is not a

hard limit. Actual heap size might be temporarily or permanently higher

for a variety of reasons. The default value is a relatively safe setting

to not use more than 3 GiB physical memory.

-storage.remote.graphite-address

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

-storage.remote.graphite-prefix

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

-storage.remote.graphite-transport

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

-storage.remote.influxdb-url

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

-storage.remote.influxdb.database

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

-storage.remote.influxdb.retention-policy

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

-storage.remote.influxdb.username

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

-storage.remote.opentsdb-url

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

-storage.remote.timeout

WARNING: THIS FLAG IS UNUSED! Built-in support for InfluxDB,

Graphite, and OpenTSDB has been removed. Use Prometheus's generic remote

write feature for building remote storage integrations. See

https://prometheus.io/docs/operating/configuration/#<remote\_write>

== WEB ==

-web.console.libraries "console\_libraries"

Path to the console library directory.

-web.console.templates "consoles"

Path to the console template directory, available at /consoles.

-web.enable-remote-shutdown false

Enable remote service shutdown.

-web.external-url

The URL under which Prometheus is externally reachable (for

example, if Prometheus is served via a reverse proxy). Used for

generating relative and absolute links back to Prometheus itself. If the

URL has a path portion, it will be used to prefix all HTTP endpoints

served by Prometheus. If omitted, relevant URL components will be derived

automatically.

-web.listen-address ":9090"

Address to listen on for the web interface, API, and telemetry.

-web.max-connections 512

Maximum number of simultaneous connections.

-web.read-timeout 30s

Maximum duration before timing out read of the request, and closing

idle connections.

-web.route-prefix

Prefix for the internal routes of web endpoints. Defaults to path

of -web.external-url.

-web.telemetry-path "/metrics"

Path under which to expose metrics.

-web.user-assets

Path to static asset directory, available at /user.

ERRO[0000] flag: help requested source=main.go:75

# Prometheus 实战于源码分析之storage

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70980587

prometheus不仅支持本地存储还支持远端存储，先从远端存储说起，他是通过一个发送队列queue完成数据发送的。先看一下队列的定义：

func NewQueueManager(cfg QueueManagerConfig) \*QueueManager {

if cfg.QueueCapacity == 0 {

cfg.QueueCapacity = defaultQueueCapacity

}

if cfg.MaxShards == 0 {

cfg.MaxShards = defaultMaxShards

}

if cfg.MaxSamplesPerSend == 0 {

cfg.MaxSamplesPerSend = defaultMaxSamplesPerSend

}

if cfg.BatchSendDeadline == 0 {

cfg.BatchSendDeadline = defaultBatchSendDeadline

}

t := &QueueManager{

cfg: cfg,

queueName: cfg.Client.Name(),

logLimiter: rate.NewLimiter(logRateLimit, logBurst),

numShards: 1,

reshardChan: make(chan int),

quit: make(chan struct{}),

samplesIn: newEWMARate(ewmaWeight, shardUpdateDuration),

samplesOut: newEWMARate(ewmaWeight, shardUpdateDuration),

samplesOutDuration: newEWMARate(ewmaWeight, shardUpdateDuration),

}

t.shards = t.newShards(t.numShards)

numShards.WithLabelValues(t.queueName).Set(float64(t.numShards))

queueCapacity.WithLabelValues(t.queueName).Set(float64(t.cfg.QueueCapacity))

return t

}

这个队列的最大分片是1000，每个分片没秒1000个sample，那么一秒就可以发送1000\*1000个sample。每一种存储，无论是本地存储还有远端存储，写数据都实现Append方法，remote的也一样，在romte的Append就调用了queue的Append方法。

func (t \*QueueManager) Append(s \*model.Sample) error {

var snew model.Sample

snew = \*s

snew.Metric = s.Metric.Clone()

for ln, lv := range t.cfg.ExternalLabels {

if \_, ok := s.Metric[ln]; !ok {

snew.Metric[ln] = lv

}

}

snew.Metric = model.Metric(

relabel.Process(model.LabelSet(snew.Metric), t.cfg.RelabelConfigs...))

if snew.Metric == nil {

return nil

}

t.shardsMtx.Lock()

enqueued := t.shards.enqueue(&snew)

t.shardsMtx.Unlock()

if enqueued {

queueLength.WithLabelValues(t.queueName).Inc()

} else {

droppedSamplesTotal.WithLabelValues(t.queueName).Inc()

if t.logLimiter.Allow() {

log.Warn("Remote storage queue full, discarding sample. Multiple subsequent messages of this kind may be suppressed.")

}

}

return nil

}

通过enqueued := t.shards.enqueue(&snew)发到队列里面，

func (s \*shards) enqueue(sample \*model.Sample) bool {

s.qm.samplesIn.incr(1)

fp := sample.Metric.FastFingerprint()

shard := uint64(fp) % uint64(len(s.queues))

select {

case s.queues[shard] <- sample:

return true

default:

return false

}

}

这个里面是简单的求余数分组的方法，如果这里使用一致hash会不会更好点呢！把数据发动到分片的队列中。QueueManager启动的时候就启动了队列发送任务

func (s \*shards) start() {

for i := 0; i < len(s.queues); i++ {

go s.runShard(i)

}

}

继续看runShard

func (s \*shards) runShard(i int) {

defer s.wg.Done()

queue := s.queues[i]

// Send batches of at most MaxSamplesPerSend samples to the remote storage.

// If we have fewer samples than that, flush them out after a deadline

// anyways.

pendingSamples := model.Samples{}

for {

select {

case sample, ok := <-queue:

if !ok {

if len(pendingSamples) > 0 {

log.Debugf("Flushing %d samples to remote storage...", len(pendingSamples))

s.sendSamples(pendingSamples)

log.Debugf("Done flushing.")

}

return

}

queueLength.WithLabelValues(s.qm.queueName).Dec()

pendingSamples = append(pendingSamples, sample)

for len(pendingSamples) >= s.qm.cfg.MaxSamplesPerSend {

s.sendSamples(pendingSamples[:s.qm.cfg.MaxSamplesPerSend])

pendingSamples = pendingSamples[s.qm.cfg.MaxSamplesPerSend:]

}

case <-time.After(s.qm.cfg.BatchSendDeadline):

if len(pendingSamples) > 0 {

s.sendSamples(pendingSamples)

pendingSamples = pendingSamples[:0]

}

}

}

}

具体发送样本的方法还要看里面的sendSamples

func (s \*shards) sendSamples(samples model.Samples) {

// Samples are sent to the remote storage on a best-effort basis. If a

// sample isn't sent correctly the first time, it's simply dropped on the

// floor.

begin := time.Now()

err := s.qm.cfg.Client.Store(samples)

duration := time.Since(begin)

if err != nil {

log.Warnf("error sending %d samples to remote storage: %s", len(samples), err)

failedSamplesTotal.WithLabelValues(s.qm.queueName).Add(float64(len(samples)))

} else {

sentSamplesTotal.WithLabelValues(s.qm.queueName).Add(float64(len(samples)))

}

sentBatchDuration.WithLabelValues(s.qm.queueName).Observe(duration.Seconds())

s.qm.samplesOut.incr(int64(len(samples)))

s.qm.samplesOutDuration.incr(int64(duration))

}

最终通过Store方法发送数据

func (c \*Client) Store(samples model.Samples) error {

req := &WriteRequest{

Timeseries: make([]\*TimeSeries, 0, len(samples)),

}

for \_, s := range samples {

ts := &TimeSeries{

Labels: make([]\*LabelPair, 0, len(s.Metric)),

}

for k, v := range s.Metric {

ts.Labels = append(ts.Labels,

&LabelPair{

Name: string(k),

Value: string(v),

})

}

ts.Samples = []\*Sample{

{

Value: float64(s.Value),

TimestampMs: int64(s.Timestamp),

},

}

req.Timeseries = append(req.Timeseries, ts)

}

data, err := proto.Marshal(req)

if err != nil {

return err

}

buf := bytes.Buffer{}

if \_, err := snappy.NewWriter(&buf).Write(data); err != nil {

return err

}

httpReq, err := http.NewRequest("POST", c.url.String(), &buf)

if err != nil {

return err

}

httpReq.Header.Add("Content-Encoding", "snappy")

ctx, \_ := context.WithTimeout(context.Background(), c.timeout)

httpResp, err := ctxhttp.Do(ctx, c.client, httpReq)

if err != nil {

return err

}

defer httpResp.Body.Close()

if httpResp.StatusCode/100 != 2 {

return fmt.Errorf("server returned HTTP status %s", httpResp.Status)

}

return nil

}

Store里面就是通过POST方式发送数据。说完了远端存储，再解释一下本地存储，这个设计的挺复杂，它是先放到内存中，并会批量将内存数据导入到磁盘中保存，具体看内存存储管理

type MemorySeriesStorage struct {

// archiveHighWatermark and numChunksToPersist have to be aligned for atomic operations.

archiveHighWatermark model.Time // No archived series has samples after this time.

numChunksToPersist int64 // The number of chunks waiting for persistence.

maxChunksToPersist int // If numChunksToPersist reaches this threshold, ingestion will be throttled.

rushed bool // Whether the storage is in rushed mode.

rushedMtx sync.Mutex // Protects entering and exiting rushed mode.

throttled chan struct{} // This chan is sent to whenever NeedsThrottling() returns true (for logging).

fpLocker \*fingerprintLocker

fpToSeries \*seriesMap

options \*MemorySeriesStorageOptions

loopStopping, loopStopped chan struct{}

logThrottlingStopped chan struct{}

maxMemoryChunks int

dropAfter time.Duration

checkpointInterval time.Duration

checkpointDirtySeriesLimit int

persistence \*persistence

mapper \*fpMapper

evictList \*list.List

evictRequests chan chunk.EvictRequest

evictStopping, evictStopped chan struct{}

quarantineRequests chan quarantineRequest

quarantineStopping, quarantineStopped chan struct{}

persistErrors prometheus.Counter

queuedChunksToPersist prometheus.Counter

numSeries prometheus.Gauge

numHeadChunks prometheus.Gauge

dirtySeries prometheus.Gauge

seriesOps \*prometheus.CounterVec

ingestedSamplesCount prometheus.Counter

discardedSamplesCount \*prometheus.CounterVec

nonExistentSeriesMatchesCount prometheus.Counter

maintainSeriesDuration \*prometheus.SummaryVec

persistenceUrgencyScore prometheus.Gauge

rushedMode prometheus.Gauge

}

他是一个内存存储管理器。和remote一样，他也是实现了Append方法去保存sample。

func (s \*MemorySeriesStorage) Append(sample \*model.Sample) error {

for ln, lv := range sample.Metric {

if len(lv) == 0 {

delete(sample.Metric, ln)

}

}

rawFP := sample.Metric.FastFingerprint()

s.fpLocker.Lock(rawFP)

fp := s.mapper.mapFP(rawFP, sample.Metric)

defer func() {

s.fpLocker.Unlock(fp)

}() // Func wrapper because fp might change below.

if fp != rawFP {

// Switch locks.

s.fpLocker.Unlock(rawFP)

s.fpLocker.Lock(fp)

}

series, err := s.getOrCreateSeries(fp, sample.Metric)

if err != nil {

return err // getOrCreateSeries took care of quarantining already.

}

if sample.Timestamp == series.lastTime {

// Don't report "no-op appends", i.e. where timestamp and sample

// value are the same as for the last append, as they are a

// common occurrence when using client-side timestamps

// (e.g. Pushgateway or federation).

if sample.Timestamp == series.lastTime &&

series.lastSampleValueSet &&

sample.Value.Equal(series.lastSampleValue) {

return nil

}

s.discardedSamplesCount.WithLabelValues(duplicateSample).Inc()

return ErrDuplicateSampleForTimestamp // Caused by the caller.

}

if sample.Timestamp < series.lastTime {

s.discardedSamplesCount.WithLabelValues(outOfOrderTimestamp).Inc()

return ErrOutOfOrderSample // Caused by the caller.

}

completedChunksCount, err := series.add(model.SamplePair{

Value: sample.Value,

Timestamp: sample.Timestamp,

})

if err != nil {

s.quarantineSeries(fp, sample.Metric, err)

return err

}

s.ingestedSamplesCount.Inc()

s.incNumChunksToPersist(completedChunksCount)

return nil

}

这个里面先通过getOrCreateSeries获取series，series你可以理解为，相同类型的监控数据放到一起，这样便于压缩查找，通过series.add保存。但这只是保存到内存中，怎么持久化呢？   
在MemorySeriesStorage启动的时候

p, err = newPersistence(

s.options.PersistenceStoragePath,

s.options.Dirty, s.options.PedanticChecks,

syncStrategy,

s.options.MinShrinkRatio,

)

if err != nil {

return err

}

s.persistence = p

// Persistence must start running before loadSeriesMapAndHeads() is called.

go s.persistence.run()

...

go s.loop()

这个persistence负责把内存中的数据写到磁盘中，loop中

for {

select {

case <-s.loopStopping:

break loop

case fp := <-memoryFingerprints:

if s.maintainMemorySeries(fp, model.Now().Add(-s.dropAfter)) {

dirty := atomic.AddInt64(&dirtySeriesCount, 1)

s.dirtySeries.Set(float64(dirty))

// Check if we have enough "dirty" series so that we need an early checkpoint.

// However, if we are already behind persisting chunks, creating a checkpoint

// would be counterproductive, as it would slow down chunk persisting even more,

// while in a situation like that, where we are clearly lacking speed of disk

// maintenance, the best we can do for crash recovery is to persist chunks as

// quickly as possible. So only checkpoint if the urgency score is < 1.

if dirty >= int64(s.checkpointDirtySeriesLimit) &&

s.calculatePersistenceUrgencyScore() < 1 {

checkpointTimer.Reset(0)

}

}

case fp := <-archivedFingerprints:

s.maintainArchivedSeries(fp, model.Now().Add(-s.dropAfter))

}

}

maintainMemorySeries保存series，

func (s \*MemorySeriesStorage) maintainMemorySeries(

fp model.Fingerprint, beforeTime model.Time,

) (becameDirty bool) {

defer func(begin time.Time) {

s.maintainSeriesDuration.WithLabelValues(maintainInMemory).Observe(

time.Since(begin).Seconds(),

)

}(time.Now())

s.fpLocker.Lock(fp)

defer s.fpLocker.Unlock(fp)

series, ok := s.fpToSeries.get(fp)

if !ok {

// Series is actually not in memory, perhaps archived or dropped in the meantime.

return false

}

defer s.seriesOps.WithLabelValues(memoryMaintenance).Inc()

closed, err := series.maybeCloseHeadChunk()

if err != nil {

s.quarantineSeries(fp, series.metric, err)

s.persistErrors.Inc()

}

if closed {

s.incNumChunksToPersist(1)

s.numHeadChunks.Dec()

}

seriesWasDirty := series.dirty

if s.writeMemorySeries(fp, series, beforeTime) {

// Series is gone now, we are done.

return false

}

iOldestNotEvicted := -1

for i, cd := range series.chunkDescs {

if !cd.IsEvicted() {

iOldestNotEvicted = i

break

}

}

// Archive if all chunks are evicted. Also make sure the last sample has

// an age of at least headChunkTimeout (which is very likely anyway).

if iOldestNotEvicted == -1 && model.Now().Sub(series.lastTime) > headChunkTimeout {

s.fpToSeries.del(fp)

s.numSeries.Dec()

s.persistence.archiveMetric(fp, series.metric, series.firstTime(), series.lastTime)

s.seriesOps.WithLabelValues(archive).Inc()

oldWatermark := atomic.LoadInt64((\*int64)(&s.archiveHighWatermark))

if oldWatermark < int64(series.lastTime) {

if !atomic.CompareAndSwapInt64(

(\*int64)(&s.archiveHighWatermark),

oldWatermark, int64(series.lastTime),

) {

panic("s.archiveHighWatermark modified outside of maintainMemorySeries")

}

}

return

}

// If we are here, the series is not archived, so check for Chunk.Desc

// eviction next.

series.evictChunkDescs(iOldestNotEvicted)

return series.dirty && !seriesWasDirty

}

writeMemorySeries把数据写到磁盘，里面再调用persistChunks

func (p \*persistence) persistChunks(fp model.Fingerprint, chunks []chunk.Chunk) (index int, err error) {

f, err := p.openChunkFileForWriting(fp)

if err != nil {

return -1, err

}

defer p.closeChunkFile(f)

if err := p.writeChunks(f, chunks); err != nil {

return -1, err

}

// Determine index within the file.

offset, err := f.Seek(0, os.SEEK\_CUR)

if err != nil {

return -1, err

}

index, err = chunkIndexForOffset(offset)

if err != nil {

return -1, err

}

return index - len(chunks), err

}

那这些series怎么查询呢？它有个index列表，通过著名的leveldb保存index，这样就可以通过index去查询了。他是一个keyvalue[**数据库**](http://lib.csdn.net/base/mysql)，接口定义storage/local/index/interface.[**Go**](http://lib.csdn.net/base/go)

type KeyValueStore interface {

Put(key, value encoding.BinaryMarshaler) error

// Get unmarshals the result into value. It returns false if no entry

// could be found for key. If value is nil, Get behaves like Has.

Get(key encoding.BinaryMarshaler, value encoding.BinaryUnmarshaler) (bool, error)

Has(key encoding.BinaryMarshaler) (bool, error)

// Delete returns (false, nil) if key does not exist.

Delete(key encoding.BinaryMarshaler) (bool, error)

NewBatch() Batch

Commit(b Batch) error

// ForEach iterates through the complete KeyValueStore and calls the

// supplied function for each mapping.

ForEach(func(kv KeyValueAccessor) error) error

Close() error

}

它的实现在storage/local/index/leveldb.go里面，代码比较多，我就不粘出来了。

# 2@Prometheus 实战于源码分析之API与联邦

- 柳清风的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/u010278923/article/details/70891379

在进行源码讲解关于prometheus还有一些配置和使用，需要解释一下。首先是API的使用，prometheus提供了一套HTTP的接口

curl http://localhost:9090/api/v1/query?query=go\_goroutines|python -m json.tool

{

"data": {

"result": [

{

"metric": {

"\_\_name\_\_": "go\_goroutines",

"instance": "localhost:9090",

"job": "prometheus"

},

"value": [

1493347106.901,

"119"

]

},

{

"metric": {

"\_\_name\_\_": "go\_goroutines",

"instance": "10.39.0.45:9100",

"job": "node"

},

"value": [

1493347106.901,

"13"

]

},

{

"metric": {

"\_\_name\_\_": "go\_goroutines",

"instance": "10.39.0.53:9100",

"job": "node"

},

"value": [

1493347106.901,

"11"

]

}

],

"resultType": "vector"

},

"status": "success"

}

上面演示一个查询go\_goroutines这一个监控指标的数据。让然也可以基于开始时间和截止时间查询，但更强大的功能应该是支持OR查询

[root@slave3 ~]# curl -g 'http://localhost:9090/api/v1/series?match[]=up&match[]=process\_start\_time\_seconds{job="prometheus"}'|python -m json.tool

{

"data": [

{

"\_\_name\_\_": "up",

"instance": "10.39.0.53:9100",

"job": "node"

},

{

"\_\_name\_\_": "up",

"instance": "localhost:9090",

"job": "prometheus"

},

{

"\_\_name\_\_": "up",

"instance": "10.39.0.45:9100",

"job": "node"

},

{

"\_\_name\_\_": "process\_start\_time\_seconds",

"instance": "localhost:9090",

"job": "prometheus"

}

],

"status": "success"

}

查询一个系列的数据，当然还可以通过DELETE去删除系列。还记得上一篇说的设置job和targets了吗？也可以通过API查询

curl http://localhost:9090/api/v1/label/job/values

{"status":"success","data":["node","prometheus"]}

当然有哪些监控对象也可以查询

curl http://localhost:9090/api/v1/targets|python -m json.tool

{

"data": {

"activeTargets": [

{

"discoveredLabels": {

"\_\_address\_\_": "10.39.0.53:9100",

"\_\_metrics\_path\_\_": "/metrics",

"\_\_scheme\_\_": "http",

"job": "node"

},

"health": "up",

"labels": {

"instance": "10.39.0.53:9100",

"job": "node"

},

"lastError": "",

"lastScrape": "2017-04-28T02:47:40.871586825Z",

"scrapeUrl": "http://10.39.0.53:9100/metrics"

},

{

"discoveredLabels": {

"\_\_address\_\_": "10.39.0.45:9100",

"\_\_metrics\_path\_\_": "/metrics",

"\_\_scheme\_\_": "http",

"job": "node"

},

"health": "up",

"labels": {

"instance": "10.39.0.45:9100",

"job": "node"

},

"lastError": "",

"lastScrape": "2017-04-28T02:47:45.144032466Z",

"scrapeUrl": "http://10.39.0.45:9100/metrics"

},

{

"discoveredLabels": {

"\_\_address\_\_": "localhost:9090",

"\_\_metrics\_path\_\_": "/metrics",

"\_\_scheme\_\_": "http",

"job": "prometheus"

},

"health": "up",

"labels": {

"instance": "localhost:9090",

"job": "prometheus"

},

"lastError": "",

"lastScrape": "2017-04-28T02:47:44.079111193Z",

"scrapeUrl": "http://localhost:9090/metrics"

}

]

},

"status": "success"

}

查询这些target。alertmanagers也是通过/api/v1/alertmanagers可以查询的。

## 本地存储配置

对应prometheus的本地存储还有一些关键的配置需要注意：   
prometheus\_local\_storage\_memory\_series：当前的系列数量在内存中保存。   
prometheus\_local\_storage\_open\_head\_chunks：打开头块的数量。   
prometheus\_local\_storage\_chunks\_to\_persist：仍然需要将其持续到磁盘的内存块数。   
prometheus\_local\_storage\_memory\_chunks：目前在记忆中的块数。如果减去前两个，则可以得到持久化块的数量（如果查询当前没有使用，则它们是可驱动的）。   
prometheus\_local\_storage\_series\_chunks\_persisted：每个批次持续存在块数的直方图。   
prometheus\_local\_storage\_rushed\_mode如果prometheus斯处于“冲动模式”，则为1，否则为0。可用于计算prometheus处于冲动模式的时间百分比。   
prometheus\_local\_storage\_checkpoint\_last\_duration\_seconds：最后一个检查点需要多长时间   
prometheus\_local\_storage\_checkpoint\_last\_size\_bytes：最后一个检查点的大小（以字节为单位）。   
prometheus\_local\_storage\_checkpointing是1，而prometheus是检查点，否则为0。可以用来计算普罗米修斯检查点的时间百分比。   
prometheus\_local\_storage\_inconsistencies\_total：找到存储不一致的计数器。如果大于0，请重新启动服务器进行恢复。   
prometheus\_local\_storage\_persist\_errors\_total：反对持续错误。   
prometheus\_local\_storage\_memory\_dirty\_series：当前脏系列数量。   
process\_resident\_memory\_bytes广义地说，prometheus进程所占据的物理内存。   
go\_memstats\_alloc\_bytes：去堆大小（分配的对象在使用中加分配对象不再使用，但尚未被垃圾回收）。

## 集群联邦

prometheus还另一个高级应用就是集群联邦，通过定义slave，这样就可以在每个数据中心部署一个，然后通过联邦汇聚。

- scrape\_config:

- job\_name: dc\_prometheus

honor\_labels: true

metrics\_path: /federate

params:

match[]:

- '{\_\_name\_\_=~"^job:.\*"}' # Request all job-level time series

static\_configs:

- targets:

- dc1-prometheus:9090

- dc2-prometheus:9090

当然如果存储量不够还可以通过分片去采集，

global:

external\_labels:

slave: 1 # This is the 2nd slave. This prevents clashes between slaves.

scrape\_configs:

- job\_name: some\_job

# Add usual service discovery here, such as static\_configs

relabel\_configs:

- source\_labels: [\_\_address\_\_]

modulus: 4 # 4 slaves

target\_label: \_\_tmp\_hash

action: hashmod

- source\_labels: [\_\_tmp\_hash]

regex: ^1$ # This is the 2nd slave

action: keep

上面定义hash的方式去决定每个prometheus负责的target。

- scrape\_config:

- job\_name: slaves

honor\_labels: true

metrics\_path: /federate

params:

match[]:

- '{\_\_name\_\_=~"^slave:.\*"}' # Request all slave-level time series

static\_configs:

- targets:

- slave0:9090

- slave1:9090

- slave3:9090

- slave4:9090

下面定义了多个slave。这样数据就可以分片存储了。

# 3@[ 翻译 ] 从头编写一款时间序列数据库

http://devopstarter.info/translate-writing-a-time-series-database-from-scratch/

## 正文

我从事监控方面的工作。尤其是专注在[Prometheus](https://prometheus.io/)，一款内置了自己定制的时间序列数据库的监控系统，以及它和[Kubernetes](https://kubernetes.io/)的集成工作。

从很多方面来说，Kubernetes表现出了一切Prometheus专门设计的东西。它使得持续部署，自动扩缩，以及高度动态环境的其他功能更易于实现。它的查询语言和操作模型，还有许多其他概念方面的决策使得Prometheus尤其适合这样的环境。然而，如果被监控的工作负载变得更加显著动态的话，这也会给监控系统本身带来新的压力。基于这一点的考虑，与其再次回顾Prometheus已经很好解决的问题，还不如专注于在这样一个高度动态或短生命周期服务的环境里提高它的性能。

Prometheus的存储层在历史上有着惊人的性能表现，一个单台服务器每秒可以摄取多达100万个采样，数百万个时间序列，同时仅占用令人惊叹的少量磁盘空间。尽管当前的存储已经给我们提供了不错的服务，笔者构思了一个新设计的存储子系统用来纠正现有解决方案的一些短板，并且可以用来配备支撑下一代的集群规模。

注意：笔者并没有数据库方面的背景。我在这里所说的话可能是错误的或是带有误导性的。你可以在Freenode上的#prometheus频道里将你的批评指正反馈到我（fabxc）。

### 问题，难题，问题域

首先，快速概括一下我们试图完成的任务以及这里面暴露出的关键问题。针对每一点，我们会先看一看Prometheus目前的做法，它在哪些地方做的出色，以及我们旨在通过新的设计想解决哪些问题。

### 时间序列数据

我们有一个根据时间采集数据点的系统。

**identifier -> (t0, v0), (t1, v1), (t2, v2), (t3, v3), ....**

每个数据点都是一个由时间戳和值组成的元组。为了达成监控的目的，时间戳是一个整数，值则可以是任意数字。经验来看，一个64位的浮点数往往能够很好地展现计数器（counter）和测量（gauge）的值，因此我们也不例外。一个时间序列是一组时间上严格单调递增的数据点序列，它可以通过一个标识符来寻址。我们的标识符便是一个度量（metric）名带上一个多维标签的字典。多维标签会将单个度量的测量空间分区。每个度量名加上一串唯一的标签便组成了它自己的时间序列（time series），它会有一个与之关联的值序列流。下面是一组典型的序列标识符，它是度量请求计数的一部分：

**requests\_total{path="/status", method="GET", instance=”10.0.0.1:80”}**

**requests\_total{path="/status", method="POST", instance=”10.0.0.3:80”}**

**requests\_total{path="/", method="GET", instance=”10.0.0.2:80”}**

让我们快速简化一下这个表达形式：我们不妨将一个度量名视为另一种标签维度 - 在我们的场景里便是**\_\_name\_\_**。在查询级别上，它可能会被特殊对待，但是它并不会关注我们采用何种方式来存放它，这一点我们将在后面看到。

**{\_\_name\_\_="requests\_total", path="/status", method="GET", instance=”10.0.0.1:80”}**

**{\_\_name\_\_="requests\_total", path="/status", method="POST", instance=”10.0.0.3:80”}**

**{\_\_name\_\_="requests\_total", path="/", method="GET", instance=”10.0.0.2:80”}**

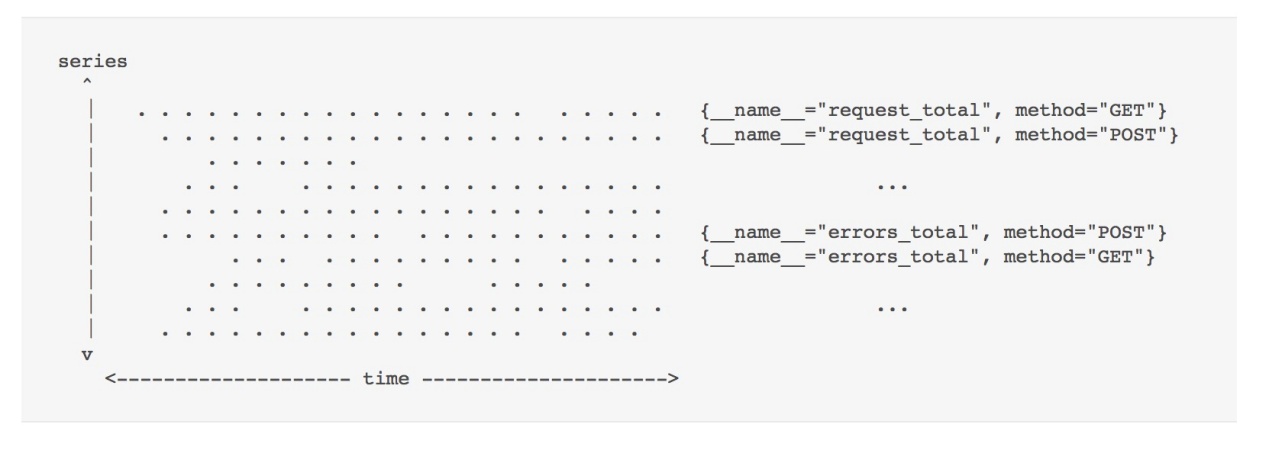
当查询时间序列数据时，我们想通过指定标签来选择。最简单的例子莫过于**{\_\_name\_\_="requests\_total"}**会选出所有属于**requests\_total**度量的序列。针对所有被选中的序列来说，我们会在一个指定的时间窗口里检索出对应的数据点。

而在更复杂的查询里，我们可能希望同时选择满足多个标签选择器的序列，并且就表达形式来说也会存在比等于更复杂的条件。比如，取反（**method!="GET"**）或者正则表达式匹配（**method=~"PUT|POST"**）。

这大体上决定了所需存储的数据以及它们该如何被调用。

### 横轴和纵轴

在一个简化的视图中，所有数据点都可以在一个二维平面上分布。横轴代表时间，而序列标识符的空间遍及整个纵轴。



Prometheus通过定期抓取一组时间序列的当前值来检索得到数据点。这样一个我们检索批次的实体被称作一个目标（target）。由于每个目标的样本数据都是单独抓取的，因此写入模式是完全垂直并且高度并发的。这里提供一些衡量尺度：一个单个的Prometheus实例会从成千上万的目标采集数据点，每个目标可以暴露出数百上千个不同的时间序列。

就每秒采集数百万个数据点的规模而言，批量写入是一个不可调和的性能需求。分散地写入单个数据点到磁盘的话又会是一个非常缓慢的过程。因此，我们想要实现的是按顺序写入更大的数据块。对于机械的旋转磁盘而言这样做并不出奇，因为它们的头会一直物理地移动到不同的区块。虽然SSD以快速地随机写入性能而闻名，但是实际上它们却不能修改单个字节，而只能写入4KiB或更大的的页。这意味着写一个16字节的样本同写一个完整的4KiB页没什么两样。这种行为即是所谓的[写入放大](https://en.wikipedia.org/wiki/Write_amplification)的一部分，作为一个“额外红利”，它会耗损你的SSD —— 因此它不仅仅只是会变慢而已，还会在几天或者几周内完全毁掉你的硬件。关于这个问题的更详细信息，系列博客["针对SSD编程"系列](http://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents/)会是一个不错的资源。我们不妨考虑一下这里面的主要关键点：顺序和批量写入是旋转磁盘和SSD的理想写入模式。 这是一个应该遵循的简单规则。

时间序列数据库的查询模型跟写模型相比，更是有明显不同的区别。我们可以对一个单个序列查询一个单个的数据点，在10000个序列里查询一个单个的数据点，在一个单个序列里查询几周的数据点，甚至在10000个序列里查询几周的数据点，等等。因此在我们的二维平面上，查询既不是完全垂直的，也不是水平的，而是二者的矩形组合。[记录规则](https://prometheus.io/docs/practices/rules/)可以减轻已知的一些查询方面的问题，但是仍然不是临时查询（ad-hoc queries）的一个通用解决方案，这些查询也必须能很好的进行下去。

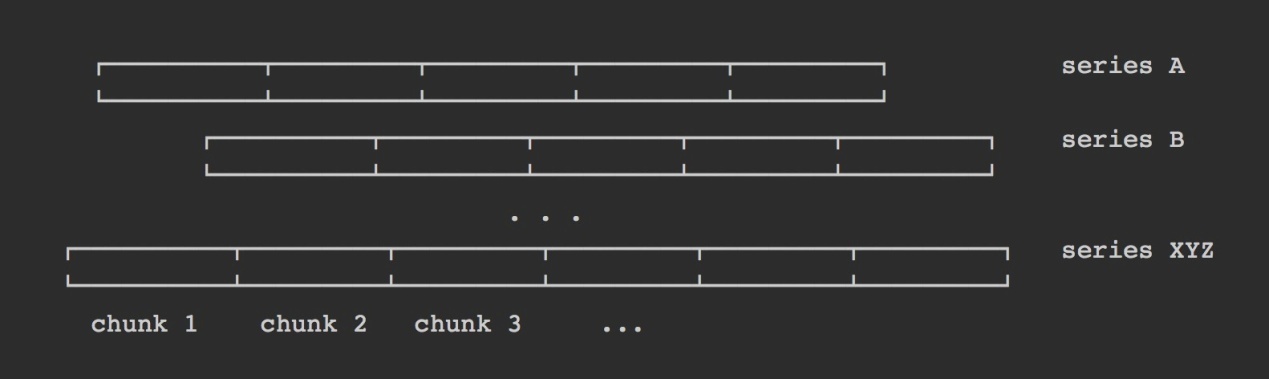
须知我们想要的是批量写入，但是我们得到的批次只是序列之间一个纵向的数据点集合。当在一个时间窗口上针对某个序列查询数据点时，不仅难以确定各个数据点可以被找到的位置，我们还不得不从磁盘上大量的随机位置进行读取。每次查询操作可能涉及到数以百万的样例数据，即使在最快的SSD上这样的操作也会变慢。读操作还将从磁盘上检索更多的数据，而不仅仅只是所请求的16字节大小的样本。 SSD将加载一整页，HDD将至少读取整个扇区。 无论哪种方式，我们都会浪费宝贵的读吞吐量。

因此，在理想情况下，相同序列的样本数据将会被顺序存储，这样一来我们便可以用尽可能少的读来扫描得到它们。 在上层，我们只需要知道这个序列可以访问的所有数据点的开始位置。

在将收集的数据写入磁盘的理想模式和为服务的查询操作提供更显著有效的存储格式之间显然存在着强烈的冲突。这是我们的时间序列数据库要解决的根本问题。

### 当前的解决方案

是时候来看看Prometheus当前的存储是如何实现的，我们不妨叫它“V2”，它致力于解决这个问题。我们会为每个时间序列创建一个文件，它会按照时间顺序包含所有的样本数据。由于每隔几秒就把单个的样本数据添加到所有这些文件的成本不小，我们针对每个序列在内存里批量存放了1KiB的数据块，一旦它们填满了再把这些块添加到一个个的文件里。这一方案解决了很大一部分问题。写操作如今是分批次的，样本数据也是顺序存储的。它还能为我们提供一个令人难以置信的高效压缩格式，这是基于一个给定的样本相对于相同序列里前面的那些样本数据只有非常少量的变化这一属性而设计。Facebook在它们的Gorilla TSDB的论文里描述了一种类似的基于块（Chunk）的存储方法，并且[引入了一个压缩格式](http://www.vldb.org/pvldb/vol8/p1816-teller.pdf)，将16个字节的样本减少到平均1.37字节。V2存储使用了各种压缩格式，包括Gorilla的一个变种。



尽管基于块的实现方案很棒，如何为每个序列维护一个单独的文件却也是V2存储引擎困扰的地方，这里面有几个原因：

* 我们实际上需要维护的文件数量多于我们正在收集数据的时间序列数量。在“序列分流”一节会详解介绍到这点。由于产生了几百万个文件，不久的将来或者迟早有一天，我们的文件系统会出现[inode](https://en.wikipedia.org/wiki/Inode)耗尽的情况。在这种情况下我们只能通过重新格式化磁盘来恢复，这样做可能带有侵入性和破坏性。通常我们都希望避免格式化磁盘，特别是需要适配某个单个应用时更是如此。
* 即便做了分块，每秒也会产生数以千计的数据块并且准备好被持久化。这仍然需要每秒完成几千次单独的磁盘写操作。尽管这一点可以通过为每个序列填满的数据块做分批处理来缓解压力，这反过来又会增加等待被持久化的数据总的内存占用。
* 保持打开所有文件来读取和写入是不可行的。特别是因为在24小时后超过99%的数据便不再会被查询。如果它还是被查询到的话，我们就不得不打开数千个文件，查找和读取相关的数据点到内存，然后再重新关闭它们。而这样做会导致很高的查询延迟，数据块被相对积极地缓存的话又会导致一些问题，这一点会在“耗用资源”一节里进一步概述。
* 最终，旧数据必须得被清理掉，而且数据需要从数百万的文件前面被抹除。这意味着删除实际上是写密集型操作。此外，循环地在这数百万的文件里穿梭然后分析它们会让这个过程常常耗费数个小时。在完成时有可能还需要重新开始。呵呵，删除旧文件将会给你的SSD带来进一步的写入放大！
* 当前堆积的数据块只能放在内存里。如果应用崩溃的话，数据将会丢失。为了避免这种情况，它会定期地保存内存状态的检查点（Checkpoint）到磁盘，这可能比我们愿意接受的数据丢失窗口要长得多。从检查点恢复估计也会花上几分钟，造成痛苦而漫长的重启周期。

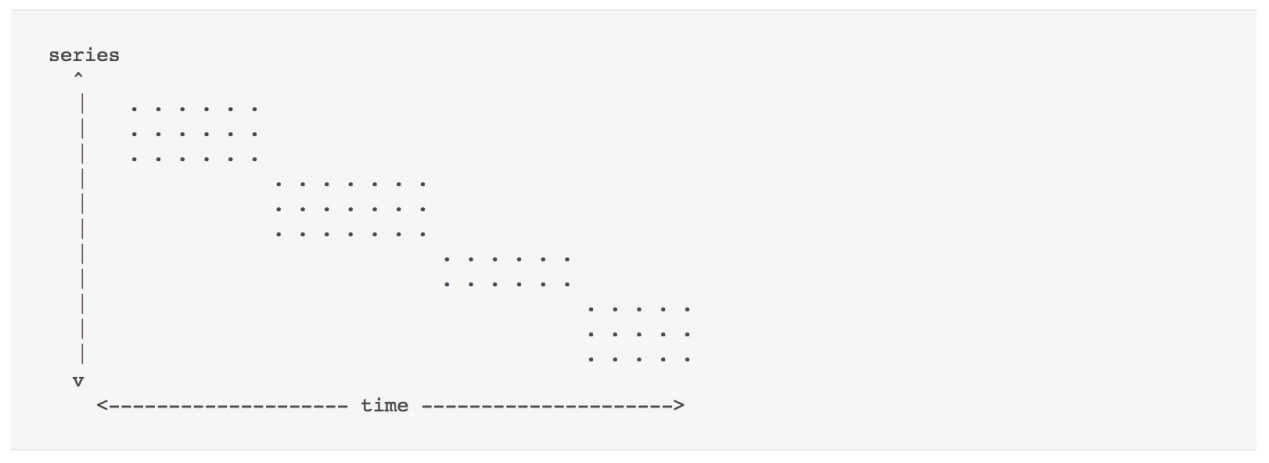
从现有的设计中脱颖而出的关键在于块的概念，我们当然希望保留这一设计。大多数最近的块被保留在内存里一般来说也是一个不错的做法。毕竟，最大幅度被查询数据里大部分便是这些最近的点。

一个时间序列对应一个文件这一概念是我们想要替换的。

### 序列分流(Series Churn)

在Prometheus的上下文里，我们使用术语“序列分流”来描述一组时间序列变得不活跃，即不再接收数据点，取而代之的是有一组新的活跃的序列出现。

举个例子，由一个给定的微服务实例产出的所有序列各自都有一个标识它起源的“instance”标签。如果我们对该微服务完成了一次滚动更新然后将每个实例切换到了一个更新的版本的话，序列分流就产生了。在一个更加动态的环境里，这些事件可能会以小时的频率出现。像Kubernetes这样的集群编排系统允许应用程序不断地自动伸缩和频繁的滚动更新，它可能会创建出数万个新的应用程序实例，并且每天都会使用全新的时间序列。



因此，即便整个基础设施大体上保持不变，随着时间的推移，我们数据库里的时间序列数据量也会呈线性增长。 尽管Prometheus服务器很愿意去采集1000万个时间序列的数据，但是如果不得不在十亿个序列中查找数据的话，很明显查询性能会受到影响。

### 当前解决方案

Prometheus当前V2版本的存储针对当前被存放的所有序列都有一个基于LevelDB的索引。它允许包含一个指定的标签对来查询序列，但是缺乏一个可扩展的方式以组合来自不同标签选择的结果。举个例子，用户可以有效地选出带有标签**\_\_name \_\_ =“requests\_total”**的所有序列，但是选择所有满足**instance =“A” AND \_\_name \_\_ =“requests\_total”**的序列则都有可扩展性的问题。我们稍后会重新审视为什么会造成这样的结果，要改善查询延迟的话要做哪些必要的调整。

实际上这一问题正是触发要实现一个更好的存储系统的最初动力。Prometheus需要一个改进的索引方法从数亿个时间序列里进行快速搜索。

### 耗用资源

耗用资源是试图扩展Prometheus（或者任何东西，真的）时不变的话题之一。但是实际上烦恼用户的问题并不是绝对的资源匮乏。实际上，由于给定需求的驱动，Prometheus管理着令人难以置信的吞吐量。问题更在于是面对变化的相对未知性和不稳定性。由于V2存储本身的架构设计，它会缓慢地构建出大量的样本数据块，而这会导致内存消耗随着时间的推移不断增加。随着数据块被填满，它们会被写入到磁盘，随即便能够从内存中被清理出去。最终，Prometheus的内存使用量会达到一个稳定的状态。直到受监控的环境发生变化 - 每次我们扩展应用程序或进行滚动更新时，序列分流 会造成内存，CPU和磁盘IO占用方面的增长。

如果变更是正在进行的话，那么最终它将再次达到一个稳定的状态，但是比起一个更加静态的环境而言，它所消耗的资源将会显著提高。过渡期的时长一般长达几个小时，而且很难说最大资源使用量会是多少。

每个时间序列对应一个单个文件的方式使得单个查询很容易就击垮Prometheus的进程。而当所要查询的数据没有缓存到内存时，被查询序列的文件会被打开，然后包含相关数据点的数据块会被读取到内存里。倘若数据量超过了可用内存，Prometheus会因为OOM被杀死而退出。待查询完成后，加载的数据可以再次释放，但通常会缓存更长时间，以便在相同数据上更快地提供后续查询。后者显然是一件好事。

最后，我们看下SSD上下文里的写入放大，以及Prometheus是如何通过批量写入来解决这个问题。然而，这里仍然有几处会造成写入放大，因为存在太多小的批次而且没有精确地对准页面边界。针对更大规模的Prometheus服务器，现实世界已经有发现硬件寿命缩短的情况。可能对于具有高写入吞吐量的数据库应用程序来说，这仍属正常，但是我们应该关注是否可以缓解这一情况。

### 从头开始

如今，我们对我们的问题域有了一个清晰的了解，V2存储是如何解决它的，以及它在设计上存在哪些问题。我们也看到一些很棒的概念设计，这些也是我们想要或多或少无缝适配的。相当数量的V2版本存在的问题均可以通过一些改进和部分的重新设计来解决，但为了让事情变得更好玩些（当然，我这个决定是经过深思熟虑的），我决定从头开始编写一款全新的时间序列数据库 —— 从零开始，即，将字节数据写到文件系统。

性能和资源使用这样的关键问题会直接引领我们做出存储格式方面的选择。我们必须为我们的数据找到一个正确的算法和磁盘布局以实现一个性能优良的存储层。

这便是我直接迈向成功时走的捷径 —— 忽略之前经历过的头疼，无数失败的想法，数不尽的草图，眼泪，还有绝望。

### V3 - 宏观设计

我们新版存储引擎的宏观设计是怎样的？简略来讲，只要到我们的data目录下运行**tree**命令，一切便都一目了然。不妨看下这幅美妙的画面它能带给我们怎样的一个惊喜。

**$ tree ./data**

**./data**

**├── b-000001**

**│ ├── chunks**

**│ │ ├── 000001**

**│ │ ├── 000002**

**│ │ └── 000003**

**│ ├── index**

**│ └── meta.json**

**├── b-000004**

**│ ├── chunks**

**│ │ └── 000001**

**│ ├── index**

**│ └── meta.json**

**├── b-000005**

**│ ├── chunks**

**│ │ └── 000001**

**│ ├── index**

**│ └── meta.json**

**└── b-000006**

**├── meta.json**

**└── wal**

**├── 000001**

**├── 000002**

**└── 000003**

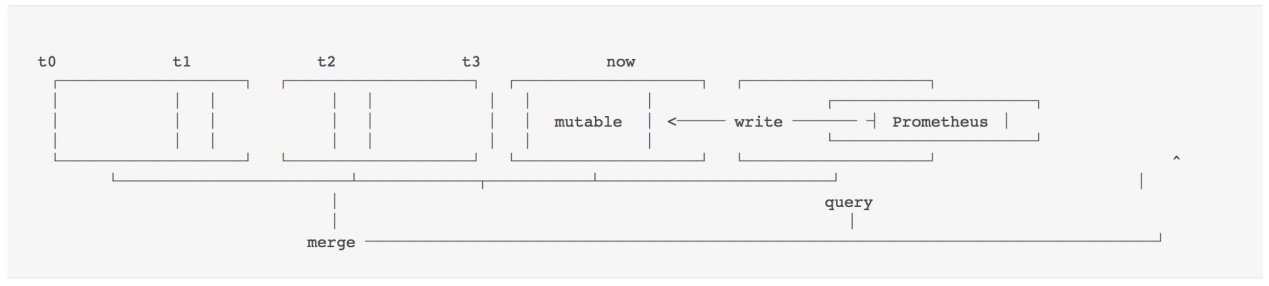
在最上面一层，我们有一组带编号的块，它们均有一个前缀**b-**。 每个块显然都维护一个包含索引的文件以及一个包含更多编号文件的"chunk"目录。"chunks"目录没别的，就多个序列的一些数据点的原始块。跟V2的做法一样，这样可以用非常低的成本来读取一个时间窗口里的序列数据，并且允许我们采用相同的有效压缩算法。这个概念已经被证实是行之有效的，我们自然就沿用这一点。很显然，这里不再是每个序列对应一个单个文件，取而代之的是，几个文件包含许多序列的数据块。

"index"文件的存在是预料之中的事情。我们不妨假定它包含了大量的黑魔法，允许我们找出标签，它们可能的值，整个时间序列，以及存放数据点的数据块。

但是，为什么有几个目录是一个索引和一些块文件这样的布局？为什么最后一个目录里取而代之的是有一个“wal”目录？搞清楚这两个问题的话可以解决我们90％的问题。

### 众多的小型数据库

我们将我们的水平维度，即时间空间分割成非重叠的块。 每个块当成一个完全独立的数据库，包含其时间窗口的所有时间序列数据。因此，它有自己的索引和一组块文件。



每个块的数据均是无法更改的。当然，在我们采集到新数据时我们必须能够将新序列和样本数据添加到最近的数据块里。对于这个数据块，所有新数据都将写入到内存数据库里，跟我们持久化的数据块一样，它也会提供相同的查找属性。内存里的数据结构也可以被有效地更新。为了防止数据丢失，所有传入的数据还会被写入预写日志（write ahead log），即我们的“wal”目录中的一组文件，我们可以在重新启动时基于这些文件将之前内存里的数据重新填充到内存数据库。

所有这些文件都带有自己的序列化格式，它附带了许多标志，偏移量，变体和CRC32校验和。比起无聊地读着介绍，读者朋友自己去发现它们也许会更有乐趣些。

这种布局允许我们查出所有和被查询的时间范围相关的数据块。每个块的部分结果被合并到一起形成最终的完整结果。

这种水平分区解锁了一些很棒的功能：

* 当查询一个时间范围时，我们可以轻松地忽略该范围外的所有数据块。 通过减少一系列开始时需要检查的数据，它可以初步解决序列分流的问题。
* 当完成一个数据块的填充时，我们可以通过顺序写入数据到一些较大的文件来保存内存数据库中的数据。 这样就避免了任何写入放大的问题，并且同样适用于SSD和HDD。
* 我们继承了V2优秀的地方，最近最多被查询的数据块总是作为热点保存在内存里。
* 棒棒哒，我们再也不需要通过固定的1KiB块大小设定来更好地对齐磁盘上的数据。 我们可以选择任何对于个别数据点和选定的压缩格式最有意义的大小。
* 删除旧数据变得非常低成本和及时。我们只需要删除一个目录。 请记住，在旧存储中，我们不得不分析并重新编写高达数亿个文件，这一操作可能需要几个小时才能收敛。

每个块还包含一个**meta.json**文件。 它简单地保存该数据块的人类可读信息，便于用户轻松了解数据块的存储状态及其包含的数据。

### mmap

从数以百万的小文件改成几个更大的文件使得我们能够以很小的成本保持所有文件的打开句柄。这也解锁了使用**mmap(2)**的玩法，它是一个系统调用，允许我们通过文件内容透明地回传到一个虚拟内存区域。为了简单起见，你可以联想它类似于交换(swap)空间，只是我们所有的数据已经在磁盘上，并且在将数据交换出内存后不会发生写入。这意味着我们可以将数据库里的所有内容均视为内存而不占用任何物理RAM。只有我们访问我们的数据库文件中的某些字节范围时，操作系统才会从磁盘惰性地加载页面。这就把和我们持久化数据相关的所有内存管理都交给了操作系统负责。 一般来说，操作系统更有资格做出这样的决定，因为它对整个机器及其所有过程有更全面的看法。查询数据可以相当积极地被缓存在内存里，而一旦面临内存压力，页面便会被逐出(evicted)。如果机器有未使用的内存，Prometheus将会很高兴去缓存整个数据库，而一旦另一个应用程序需要它，它将立即返回。

这样一来，比起受到RAM的大小限制，即便查询更多的持久化数据，查询操作也不会再轻易造成进程的OOM。内存的缓存大小变得完全自适应，只有在查询实际需要的数据时才会加载数据。

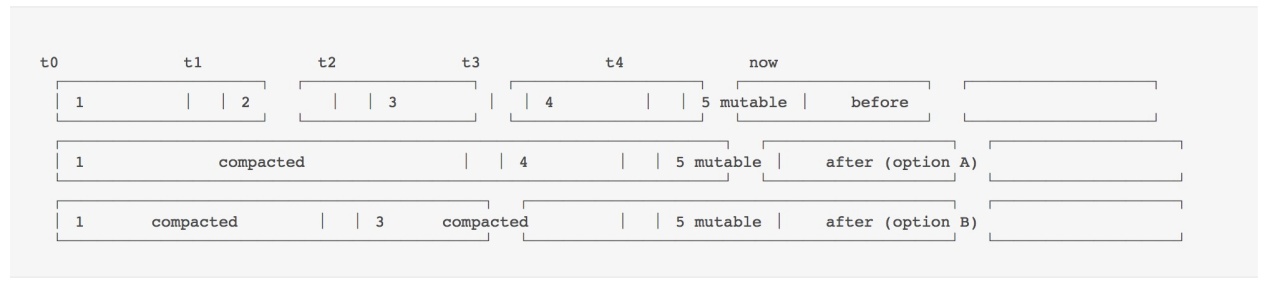
就我个人的理解，这是今天的很多数据库的工作方式，如果磁盘格式允许的话，这是一个理想的方法 - 除非你有信心在进程里做的工作能够超越操作系统。我们自己做了很少一部分工作而确实从外部系统收获了大量功能。

### 压缩（compaction）

存储引擎必须定期地“切出”一个新的块，并将之前完成的块写入到磁盘。只有块被成功持久化后，用于恢复内存块的预写日志文件（wal）才会被删除。

我们有兴趣将每个块的保存时间设置的相对短一些（一般设置大约两个小时），以避免在内存中堆积太多的数据。当查询多个块时，我们必须将其结果合并为一个完整结果。 这个合并过程显然会有一个成本，一个一周长的查询不应该需要合并超过80个的部分结果。

为了实现两者共同的需求，我们引入数据压缩（compaction）。它描述了采集一个或多个数据块并将其写入一个可能会更大的块的过程。它还可以沿途修改现有的数据，例如，清理已删除的数据，或重组我们的样本数据块以提高查询性能。



在这个例子里，我们有一组顺序的块**[1, 2, 3, 4]**。数据块1，2，和3可以被一起压缩，然后形成的新结构便是**[1, 4]**。或者，将它们成对地压缩成“[1，3]”。 所有的时间序列数据仍然存在，但是现在总体的数据块更少。 这显著降低了查询时的合并成本，因为现在需要被合并的部分查询结果会更少。

### 保留（Retention）

我们看到，删除旧数据在V2存储引擎里是一个缓慢的过程，而且会消耗CPU，内存和磁盘。那么，我们该如何在基于块的设计中删除旧数据呢？简单来讲，只需删除该目录下在我们配置的保留窗口里没有数据的块。 在下面的示例中，块1可以安全地被删除，而2必须保留到完全落在边界之后才行。



获取越旧的数据，数据块可能就变得越大，这是因为我们会不断地压缩以前压缩的块。 因此必须得有一个压缩的上限，这样一来块就不会扩展到跨越整个数据库从而影响到我们设计的最初优势。

另一个方便之处在于，这样也可以限制部分在保留窗口里部分在外面的数据块的总磁盘开销，即上面示例中的块2.当用户将最大块的大小设置为总保留窗口的10％时，保留块2的总开销也有10％的上限。

总而言之，保留删除的实现从非常高的成本变成了几乎零成本。

如果看到这里，而且读者朋友本人有一些数据库的背景的话，你可能会问一件事：这是一个新玩法吗？ —— 其实不是；而且大概还可以做得更好

在内存里批量处理数据，在预写日志（wal）里跟踪，并定期刷新到磁盘，这种模式在今天是被广泛采纳的。

无论数据特指的问题域是什么，我们所看到的好处几乎都是普遍适用的。 遵循这一方法的突出开源案例是LevelDB，Cassandra，InfluxDB或HBase。而这里面的关键是要避免重复发明劣质轮子，研究经过生产验证的方法，并采取正确的姿势应用它们。

这里仍然留有余地可以添加用户自己的黑科技。

### 索引(index)

调研存储的改进方案的源动力便是为了解决序列分流引发的问题。基于块的布局设计减少了为查询提供服务所涉及的时间序列的总数。因此，假设我们索引查找的时间复杂度是O（n ^ 2），我们设法减少n个相等的数量，那么现在就有一个改进的复杂度O（n ^ 2） - uhm，等一下... 哇靠。

这时候脑海里迅速回忆起“算法101”提醒我们的事情，理论上讲，这并没有给我们带来任何改善。 如果以前做的不好，那现在也差不多。理论有时候真的挺让人沮丧的。

通过实践，我们大部分的查询明显会被更快地应答。然而，跨越全部时间范围的查询仍然很慢，即便他们只需要找到少量的系列。在所有这些工作开始之前，我最初的想法都是想要一个切实解决这个问题的方案：我们需要一个更强大的[倒排索引](https://en.wikipedia.org/wiki/Inverted_index)。

倒排索引基于它们内容的子集提供对数据项的快速查找。简单来讲，用户可以找出所有带有标签**“app =”nginx“**的序列，而无需遍历每一个序列然后再检查它是否包含该标签。

为此，每个序列被分配一个唯一的ID，通过它可以在恒定的时间内检索，即O（1）。在这种情况下，ID就是我们的正向索引。

示例：如果ID为10,29和9的序列包含标签**“app =”nginx“**，标签”nginx“的倒排索引便是一个简单的列表**[10,29,9]**，它可以用来快速检索包含该标签的所有序列。即便还有200亿个序列，这也不会影响该次查找的速度。

简而言之，如果n是我们的序列总数，m是给定查询的结果大小，那么使用索引的查询复杂度便是O（m）。查询操作扩展到根据其检索的数据量（m）而不是正在搜索的数据体（n）是一个很棒的特性，因为一般来说m明显会更小些。

为了简单起见，我们假定可以在恒定的时间内完成倒排索引列表本身的检索。

实际上，这也几乎就是V2版本所拥有的倒排索引的类型，也是为数百万序列提供高性能查询的最低要求。敏锐的观察者会注意到，在最坏的情况下，所有的系列都存在一个标签，因此，m又是O（n）。 这是预料中的事情，而且也完全合理。 如果用户要查询所有的数据，自然就需要更长的时间。 一旦涉及到更复杂的查询这里可能就有问题了。

### 组合标签(Combining Labels)

标签被关联到数百万序列是很常见的。 假设有一个拥有数百个实例的横向可扩缩的“foo”微服务，每个实例有数千个系列。 每个系列都会有**“app =”foo“**的标签。当然，用户一般不会去查询所有的系列，而是通过进一步的过滤标签来限制查询，例如，我想知道我的服务实例收到多少个请求，那查询语句便是**\_\_name \_\_ =“requests\_total” AND app =“foo”**。

为了找出满足两个标签选择器的所有系列，我们取每个标签选择器的倒排索引列表然后取交集。 所得到的集合通常比每个输入列表小一个数量级。由于每个输入列表具有最差情况的复杂度是O（n），所以在两个列表上嵌套迭代的暴力解都具有O（n ^ 2）的运行时间。 其他集合操作也是相同的成本，例如union（**app =“foo”OR app =“bar”**）。当用户向查询添加进一步的标签选择器时，指数会增加到O（n ^ 3），O（n ^ 4），O（n ^ 5），... O（n ^ k）。 通过更改执行顺序，可以玩很多技巧来最大限度地有效减少运行时间。越复杂，就越需要了解数据样式和标签之间的关系。这引入了更多复杂度，但是并没有减少我们算法的最坏运行时间。

以上基本便是V2存储里采取的方式，幸运的是，看似微不足道的修改足以获得显著的提升。如果我们说我们的倒排索引中ID是排序好的话会发生什么？

假设我们初始查询的列表示例如下：

**\_\_name\_\_="requests\_total" -> [ 9999, 1000, 1001, 2000000, 2000001, 2000002, 2000003 ]**

**app="foo" -> [ 1, 3, 10, 11, 12, 100, 311, 320, 1000, 1001, 10002 ]**

**intersection => [ 1000, 1001 ]**

它们的交集相当小。我们可以通过在每个列表的开始处设置一个光标，并且始终从较小的数字那端依次推进。 当两个数字相等时，我们将数字添加到我们的结果中并推进两个游标。总的来说，我们以这种之字形模式（zig-zag pattern）扫描这两个列表，这样一来我们总的成本会是O（2n）= O（n），因为我们只是在任意一个列表中向前移动。

两个以上列表的不同集合操作的过程也是类似的效果。因此，k个集合操作的数量仅仅只会将时间复杂度修改为（O（k \* n））而不是我们最坏情况的查找运行时的指数级（O（n ^ k））。真是一个大进步。

我在这里描述的内容几乎就是任意一款[全文搜索引擎](https://en.wikipedia.org/wiki/Search_engine_indexing#Inverted_indices) 所使用的规范搜索索引的简化版本。每个序列的描述符被视为一个简短的“文档”，每个标签（名称+固定值）作为其中的“单词”。我们可以忽略通常在搜索引擎索引中遇到的大量附加数据，例如字位置和出现频率等数据。

业内似乎都在无休止的研究探索改进实际运行时的方法，他们也常常对输入数据做出一些假设。不出所料的是，许多可以压缩倒排索引的技术均是有利有弊的。而由于我们的“文档”很小，“文字”在所有序列里都是非常重复的，所以压缩变得几乎无关紧要。 例如，一个约440万系列的现实世界数据集，每个标签约有12个，拥有少于5,000个唯一的标签。在我们最开始的存储版本里，我们坚持使用基本方法而不进行压缩，只添加了一些简单的调整来跳过大范围的非相交ID。

维持排序好的ID听上去可能很简单，但是实际坚持下来却是不太容易办到的。比如，V2存储引擎将一个哈希值作为ID赋给新的序列，我们无法有效地基于此建立一个排序好的倒排索引。另一个艰巨的任务是在数据被删除或更新时修改磁盘上的索引。通常，最简单的方法是简单地重新计算和重写它们，但是得在保证数据库可查询和一致性的同时执行这一操作。V3版本的存储引擎通过在每个块中分配一个单独的不可变索引来彻底解决这一问题，只能通过压缩时的重写来进行修改。而且，只有整个保存在内存里的可变块的索引才需要被更新。

### 基准测试（Benchmark）

我发起了一个最初开发版本V3存储的基准测试，它是基于从现实世界数据集中提取的大约440万个序列描述符，并生成合成的数据点到对应的序列。这种遍历测试了单独的存储模块，而且对于快速识别性能瓶颈和触发仅在高并发负载下才会遇到的死锁尤为重要。

在完成概念性的实施之后，基准测试可以在我的Macbook Pro上保持每秒2000万个数据点的写吞吐量 —— 而所有的Chrome Tab和Slack都在持续运行。所以尽管这听上去很棒，但也表明推动这一基准测试没有进一步的价值（或者在这个问题里的随机环境下运行是这样的）。毕竟，这是合成的，这就决定了第一印象不会太好。对比最初的设计目标放大到近20倍的数据量，那么是时候将它嵌入到真正的Prometheus服务器里了，我们可以在上面添加所有只会在更贴近现实的环境里才会遇到的一切实际开销和情景。

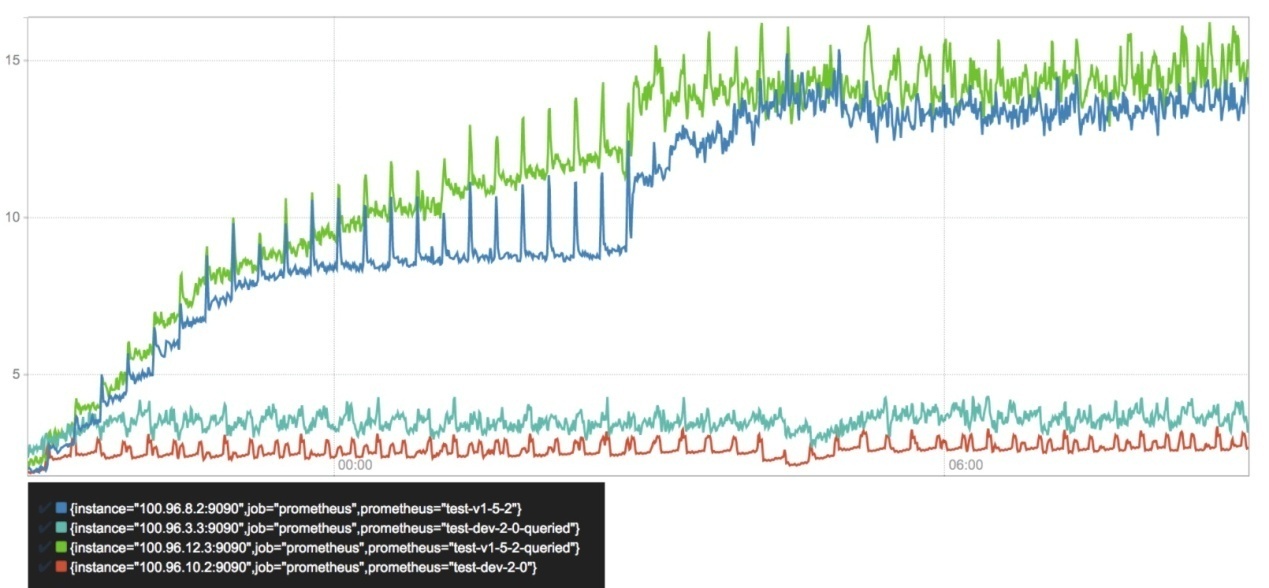
实际上，我们没有可重复的Prometheus基准测试配置，特别是没有允许不同版本的A / B测试。 亡羊补牢为时不晚，[现在我们有一个](https://github.com/prometheus/prombench)了！

我们的工具允许我们声明式地定义一个基准测试场景，然后将其部署到AWS上的Kubernetes集群。 虽然这不是全面的基准测试的最佳环境，但它肯定能反映出我们的用户基本上会比64内核和128GB内存的专用裸机服务器跑的更好。我们部署了两台Prometheus 1.5.2的服务器（V2存储引擎）以及两台基于2.0开发分支（V3存储引擎）部署的两台Prometheus服务器。每台Prometheus服务都是运行在一台配备有一块SSD的专用服务器上。我们将一个横向可扩展的应用程序部署到了工作节点上并让它对外暴露典型的微服务度量。此外，Kubernetes集群和节点本身也正在被监控。全部配置均由另一个Meta-Prometheus监督，它会监控每台Prometheus服务器的健康性和性能。为了模拟序列分流，微服务会定期地向上扩容和向下缩容，以去除旧的pod，并产生新的pod，从而生成新的序列。 查询负载以“典型”地选择查询来模拟，对每个Prometheus版本的一台服务器执行操作。

总体而言，缩放和查询负载以及采样频率显著超过了今天Prometheus的生产部署。 例如，我们每15分钟换掉60％的微服务实例以产生序列分流。在现代化的基础设施中这应该每天只会发生1-5次。 这样就能确保我们的V3设计能够处理未来几年的工作负载。 因此，比起一个更为温和的环境，在现在这样的情况下，Prometheus 1.5.2和2.0之间的性能差异更大。我们每秒总共从850个同一时间暴露50万个序列的目标里收集大约11万个样本。

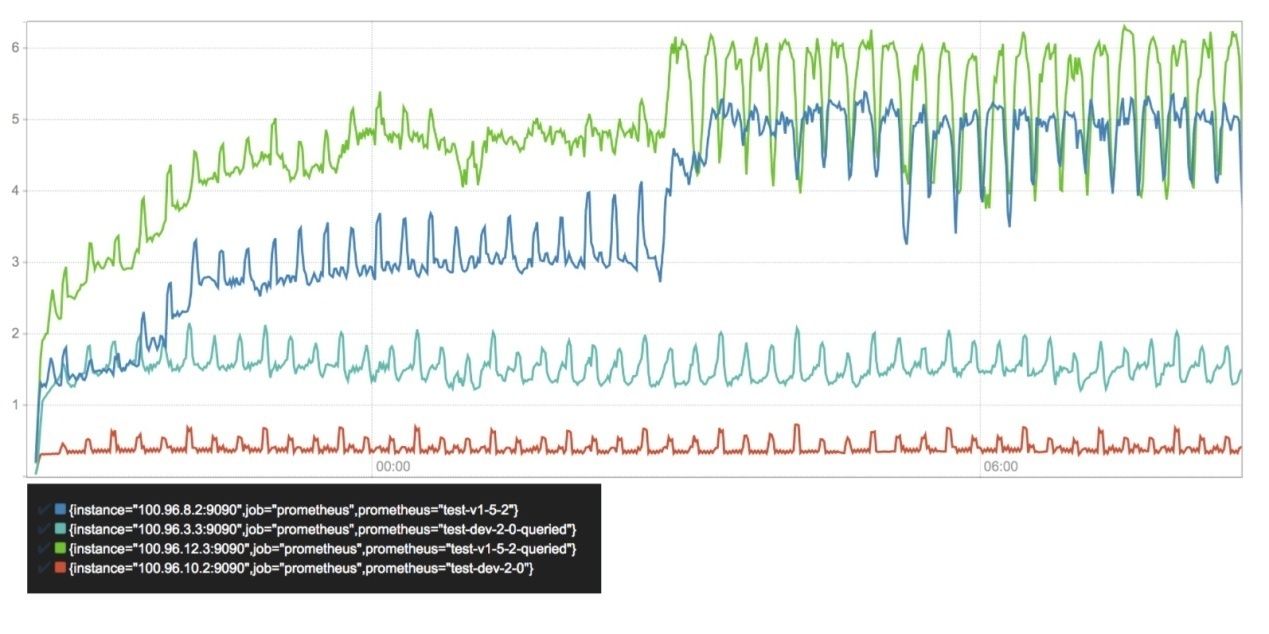
在放任这一配置运行一段时间后，我们可以来看些数字。我们评估一下前12个小时内两个版本均达到稳定状态的几个指标。

请注意在Prometheus图形界面上的屏幕截图中略微截断的Y轴。



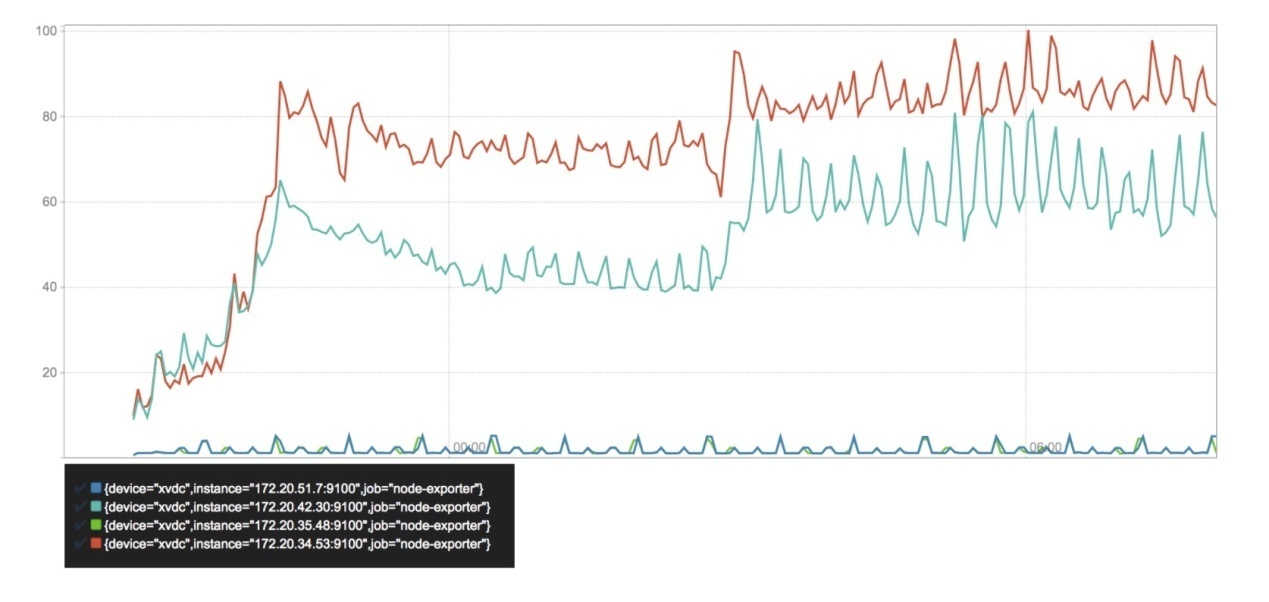
堆内存使用（GB）

内存使用是当今用户最为困扰的资源问题，因为它是相对无法预测的，而且可能会导致进程崩溃。显然，被查询的服务器正在消耗更多的内存，这主要得归咎于查询引擎的开销，而这一点在未来将有望得到优化。总的来说，Prometheus 2.0的内存消耗减少了3-4倍。 大约六个小时后，Prometheus 1.5版本就有一个明显的尖峰，与六个小时的保留边界一致。 由于删除操作成本很高，资源消耗也随之增加。这将在下面的各种其他图表中体现。



CPU使用率, 核心/秒

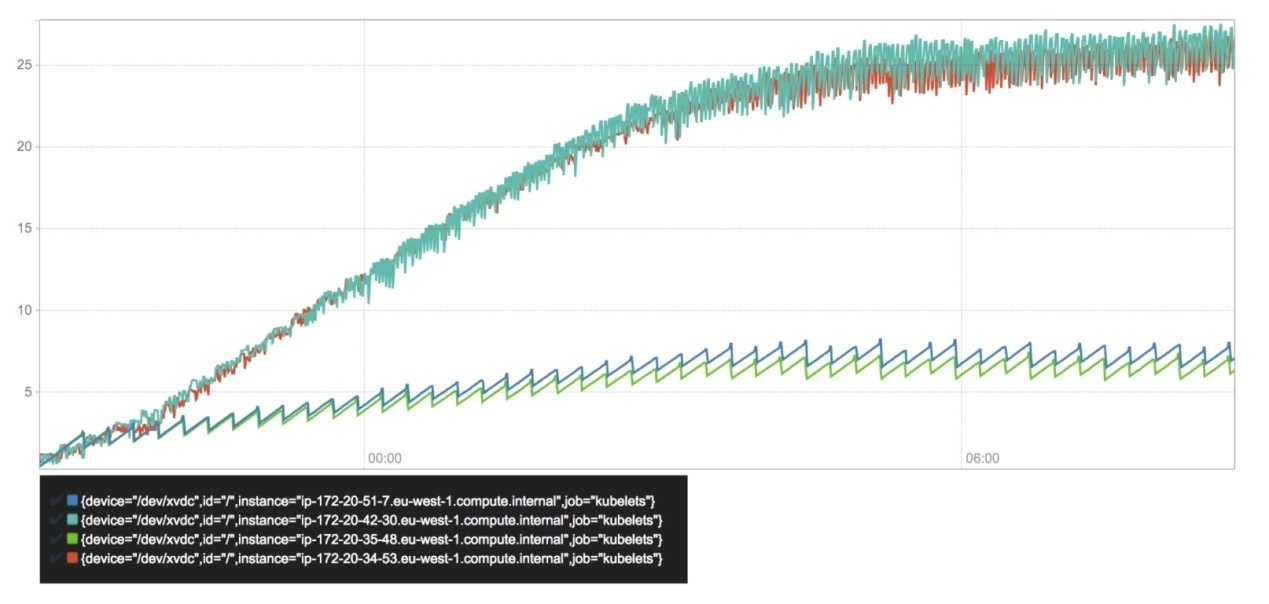
CPU使用率的展示也是类似的模式，但是这里面查询服务器与非查询服务器之间的增量差异更为明显。以约0.5个核心/秒的平均值摄取大约110,000个样本/秒，与查询计算所花费的时间周期相比，我们的新存储消耗成本几乎可以忽略不计。 总的来说，新存储需要的CPU资源减少了3-10倍。



磁盘写入MB/秒

我们磁盘的写入利用率方面展示出了最突出和意想不到的改进。 这清晰地表明了为什么Prometheus 1.5容易造成SSD的耗损。 一旦第一个块被持久化到序列文件里，我们就能看到最开始会有一个飙升的过程，一旦删除然后开始重写，就会出现第二次飙升。令人诧异的是，被查询和非查询的服务器显示出完全不同的资源消耗。

另一方面，Prometheus 2.0只是以大约每秒一兆字节的写入速度写入到wal文件。 当块被压缩到磁盘时，写入周期性地出现一个尖峰。 这总体上节省了：惊人的97-99％。

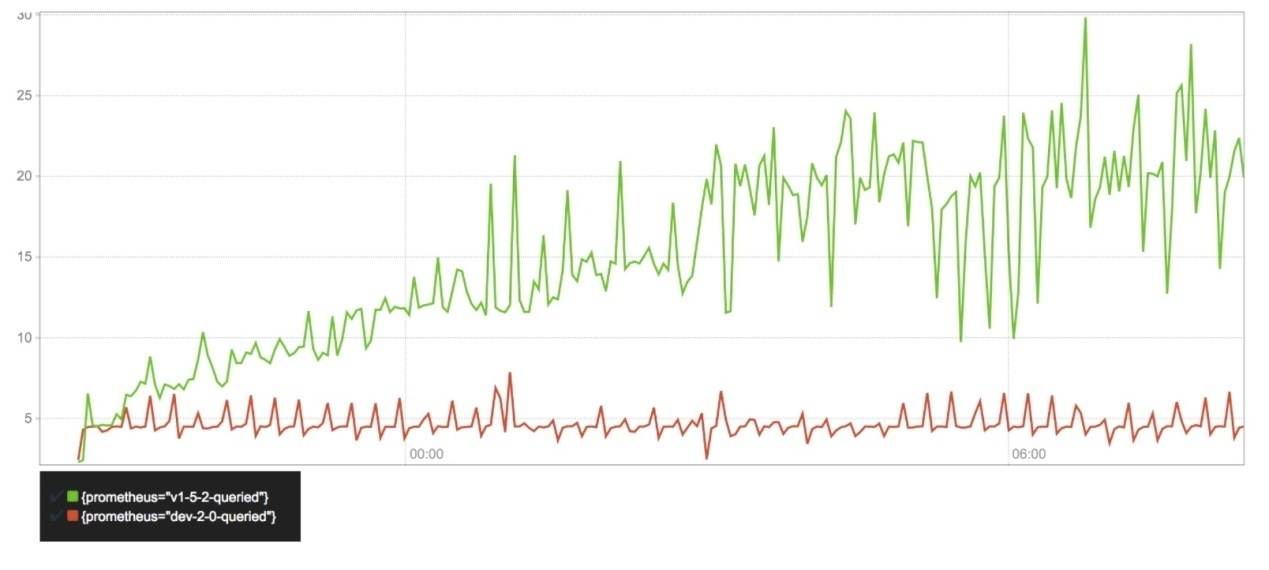


磁盘大小（GB）

与磁盘写入量密切相关的是磁盘空间的总占用量。 由于我们对样本，即我们数据中的大部分组成，使用几乎相同的压缩算法，因此它们也应该是大致相同的。 在一个更稳定的环境中，这样做在很大程度上是合理的，但是因为我们要处理的是高度的序列分流，我们还得考虑每个序列的开销。

可以看到，Prometheus 1.5在两个版本都抵达稳定状态之前，消耗的存储空间因为保留策略的执行而迅速飙升。而Prometheus 2.0似乎在每个序列的开销都有一个明显的降幅。我们可以很高兴地看到磁盘空间是由预写日志文件线性填充的，并随着其压缩会瞬间下降。 事实上，Prometheus 2.0服务器不完全匹配线性增长的情况也是需要进一步调查的。

一切看上去都是充满希望的。 剩下的重要部分便是查询延迟。 新的索引应该提高了我们的查找复杂度。 没有实质改变的是这些数据的处理，例如 **rate（）**函数或聚合。 这些是查询引擎的一部分。

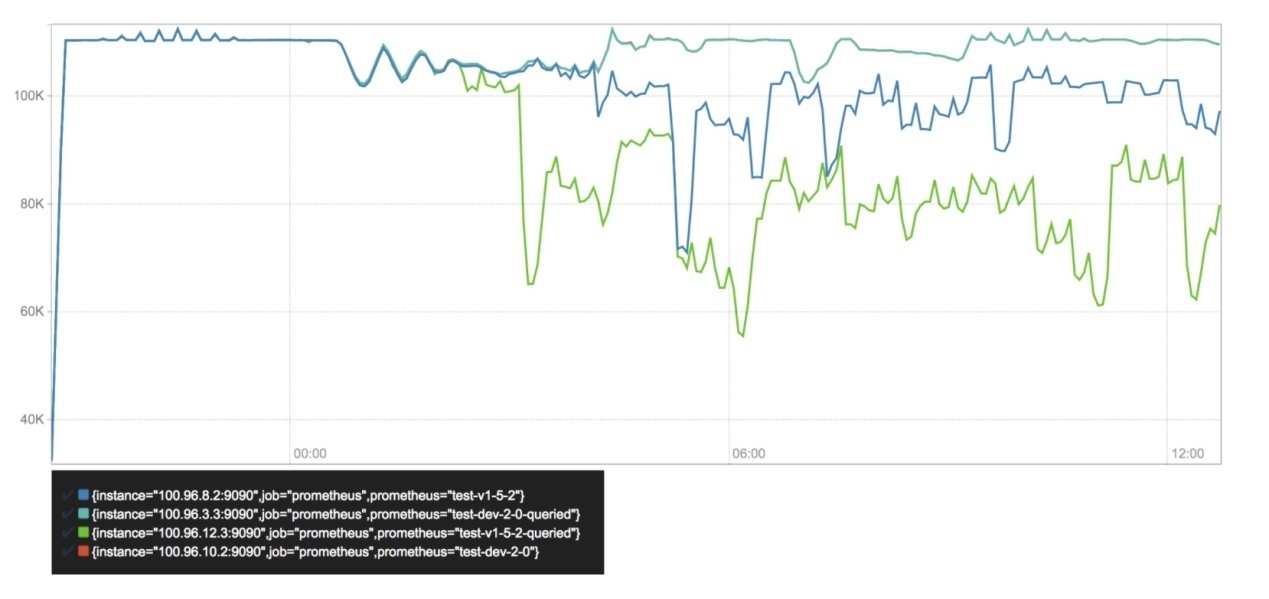


99百分位数查询延迟（以秒为单位）

数据完全符合预期。 在Prometheus 1.5中，随着存储更多的序列，查询延迟会随时间而增加。 一旦保留策略开始执行，旧的系列被删除，它才会平息。 相比之下，Prometheus 2.0从一开始就停留在合理的位置。

这个数据怎样被收集则需要用户花些心思。对服务器发出的查询请求取决于一个时间范围值和即时查询估计的最佳搭档，压缩或轻或重，以及涉及的序列或多或少等等。它不一定代表查询的真实分布。它也不能代表冷数据的查询性能，我们可以假设所有样本数据实际上总是存储在内存中的热点数据。

尽管如此，我们仍然可以非常有信心地说，新版存储引擎在序列分流方面整体查询的性能变得非常有弹性，并且在我们高压的基准场景中存储的性能提高了4倍。在一个更加静态的环境里，我们可以假定查询时间主要用于查询引擎本身，而且延迟明显可以被改进到更低值。



采样/秒

最后，快速过一下我们对不同Prometheus服务器的采样率。 我们可以看到，配备V3存储的两台服务器是相同的采样率。几个小时后，它变得不稳定，这是由于基准集群的各个节点高负载造成的失去响应而跟Prometheus实例本身无关。 （这两行2.0的数据完全匹配的事实希望能让人信服）

即便还有更多可用的CPU和内存资源，Prometheus 1.5.2服务器的采样速率也在大大降低。 序列分流的高压导致它无法收集更大量的数据。

那么，现在每秒可以抓取的绝对最大样本数是多少？ 我不知道 - 而且也故意不关注这一点。

影响Prometheus数据流量的因素众多，而这里面没有哪个单个数字能够衡量捕获质量。最大采样率历来是导致基准偏倚的一个指标，它忽略了更重要的方面，如查询性能以及对序列分流的抵御能力。 一些基本测试证实了资源使用线性增长的粗略假设。而这很容易推断出存在什么可能的结果。

我们的基准测试设置模拟了一个高度动态的环境，它给Prometheus施加的压力比今天大多数现实世界的设定要更大。 结果表明，我们在最优设计目标的基础上运行，而在不是最棒的云服务器上跑着。当然，最终衡量是否成功还是得取决于用户的反馈而不是基准数字。

注意：在撰写本文时，Prometheus 1.6正在开发中，它将允许更可靠地配置最大内存使用量，并且可能会显著降低总体的消耗，略微有利于提高CPU利用率。我并没有进行重复的测试，因为整体的结果仍然变化不大，特别是当面对高度的序列分流时更是如此。

### 结论

Prometheus开始准备应对独立样本的高基数序列及吞吐的处理。 这仍然是一项很具有挑战的任务，但是新的存储引擎似乎使得我们对于超大规模，超收敛的[GIFEE](https://github.com/GIFEE/GIFEE)基础设施的未来感到满意。恩，它似乎跑的不错。

配备新版V3存储引擎的[第一个Alpha版本的Prometheus 2.0](https://prometheus.io/blog/2017/04/10/promehteus-20-sneak-peak/)已经可用于测试。在这个早期阶段，预计会发生崩溃，死锁和其他错误。

存储引擎本身的代码可以在[单独的项目](https://github.com/prometheus/tsdb) 中找到。对于Prometheus本身而言，这是非常不可知论的，而且它也可以广泛用于一大波正在苦苦寻觅一个有效的本地时间序列数据库存储的应用。

这里得感谢很多人对这项工作的贡献。以下名单不分前后：

Bojoern Rabenstein和Julius Volz在V2存储引擎上的打磨工作以及他们对于V3的反馈为这新一代设计里所能看到的一切事物奠定了基础。

Wilhelm Bierbaum持续不断地意见和见解为新一代的设计做出了重大贡献。Brian Brazil源源不断的反馈也确保我们最终采用语义上合理的方案。与Peter Bourgon的精辟讨论验证了新的设计，并且造就了这篇文章。

当然也别忘了我所在的CoreOS整个团队和公司本身对这项工作的支持和赞助。感谢那些能够耐心听我一次又一次地扯着SSD，浮点数和序列化格式的每一位同学。

原文链接：[writing-a-time-series-database-from-scratch](https://fabxc.org/blog/2017-04-10-writing-a-tsdb/) （翻译：Colstuwjx）

About Author



**colstuwjx**

互联网运维工程师，IT屌丝一枚，好技术。

## Writing a Time Series Database from Scratch

https://fabxc.org/blog/2017-04-10-writing-a-tsdb/

### fabxc (Fabian Reinartz)

https://github.com/fabxc

I work on monitoring. In particular on [Prometheus](https://prometheus.io/), a monitoring system that includes a custom time series database, and its integration with [Kubernetes](https://kubernetes.io/).

In many ways Kubernetes represents all the things Prometheus was designed for. It makes continuous deployments, auto scaling, and other features of highly dynamic environments easily accessible. The query language and operational model, among many other conceptual decisions make Prometheus particularly well-suited for such environments. Yet, if monitored workloads become significantly more dynamic, this also puts new strains on monitoring system itself. With this in mind, rather than doubling back on problems Prometheus already solves well, we specifically aim to increase its performance in environments with highly dynamic, or transient services.

Prometheus's storage layer has historically shown outstanding performance, where a single server is able to ingest up to one million samples per second as several million time series, all while occupying a surprisingly small amount of disk space. While the current storage has served us well, I propose a newly designed storage subsystem that corrects for shortcomings of the existing solution and is equipped to handle the next order of scale.

**Note:** I've no background in databases. What I say might be wrong and mislead. You can channel your criticism towards me (fabxc) in #prometheus on Freenode.

**Problems, Problems, Problem Space**

First, a quick outline of what we are trying to accomplish and what key problems it raises. For each, we take a look at Prometheus' current approach, what it does well, and which problems we aim to address with the new design.

**Time series data**

We have a system that collects data points over time.

identifier -> (t0, v0), (t1, v1), (t2, v2), (t3, v3), ....

Each data point is a tuple of a timestamp and a value. For the purpose of monitoring, the timestamp is an integer and the value any number. A 64 bit float turns out to be a good representation for counter as well as gauge values, so we go with that. A sequence of data points with strictly monotonically increasing timestamps is a series, which is addressed by an identifier. Our identifier is a metric name with a dictionary of label dimensions. Label dimensions partition the measurement space of a single metric. Each metric name plus a unique set of labels is its own time series that has a value stream associated with it.

This is a typical set of series identifiers that are part of metric counting requests:

requests\_total{path="/status", method="GET", instance=”10.0.0.1:80”}

requests\_total{path="/status", method="POST", instance=”10.0.0.3:80”}

requests\_total{path="/", method="GET", instance=”10.0.0.2:80”}

Let's simplify this representation right away: A metric name can be treated as just another label dimension —\_\_name\_\_ in our case. At the query level, it might be be treated specially but that doesn't concern our way of storing it, as we will see later.

{\_\_name\_\_="requests\_total", path="/status", method="GET", instance=”10.0.0.1:80”}

{\_\_name\_\_="requests\_total", path="/status", method="POST", instance=”10.0.0.3:80”}

{\_\_name\_\_="requests\_total", path="/", method="GET", instance=”10.0.0.2:80”}

When querying time series data, we want to do so by selecting series by their labels. In the simplest case{\_\_name\_\_="requests\_total"} selects all series belonging to the requests\_total metric. For all selected series, we retrieve data points within a specified time window.  
In more complex queries, we may wish to select series satisfying several label selectors at once and also represent more complex conditions than equality. For example, negative (method!="GET") or regular expression matching (method=~"PUT|POST").

This largely defines the stored data and how it is recalled.

**Vertical and Horizontal**

In a simplified view, all data points can be laid out on a two-dimensional plane. The horizontal dimension represents the time and the series identifier space spreads across the vertical dimension.

series

^

│ . . . . . . . . . . . . . . . . . . . . . . {\_\_name\_\_="request\_total", method="GET"}

│ . . . . . . . . . . . . . . . . . . . . . . {\_\_name\_\_="request\_total", method="POST"}

│ . . . . . . .

│ . . . . . . . . . . . . . . . . . . . ...

│ . . . . . . . . . . . . . . . . . . . . .

│ . . . . . . . . . . . . . . . . . . . . . {\_\_name\_\_="errors\_total", method="POST"}

│ . . . . . . . . . . . . . . . . . {\_\_name\_\_="errors\_total", method="GET"}

│ . . . . . . . . . . . . . .

│ . . . . . . . . . . . . . . . . . . . ...

│ . . . . . . . . . . . . . . . . . . . .

v

<-------------------- time --------------------->

Prometheus retrieves data points by periodically scraping the current values for a set of time series. The entity from which we retrieve such a batch is called a target. Thereby, the write pattern is completely vertical and highly concurrent as samples from each target are ingested independently.  
To provide some measurement of scale: A single Prometheus instance collects data points from tens of thousands of targets, which expose hundreds to thousands of different time series each.

At the scale of collecting millions of data points per second, batching writes is a non-negotiable performance requirement. Writing single data points scattered across our disk would be painfully slow. Thus, we want to write larger chunks of data in sequence.  
This is an unsurprising fact for spinning disks, as their head would have to physically move to different sections all the time. While SSDs are known for fast random writes, they actually can't modify individual bytes but only write inpages of 4KiB or more. This means writing a 16 byte sample is equivalent to writing a full 4KiB page. This behavior is part of what is known as [write amplification](https://en.wikipedia.org/wiki/Write_amplification), which as a bonus causes your SSD to wear out – so it wouldn't just be slow, but literally destroy your hardware within a few days or weeks.  
For more in-depth information on the problem, the blog series ["Coding for SSDs" series](http://codecapsule.com/2014/02/12/coding-for-ssds-part-1-introduction-and-table-of-contents/) is a an excellent resource. Let's just consider the main take away: sequential and batched writes are the ideal write pattern for spinning disks and SSDs alike. A simple rule to stick to.

The querying pattern is significantly more differentiated than the write the pattern. We can query a single datapoint for a single series, a single datapoint for 10000 series, weeks of data points for a single series, weeks of data points for 10000 series, etc. So on our two-dimensional plane, queries are neither fully vertical or horizontal, but a rectangular combination of the two.  
[Recording rules](https://prometheus.io/docs/practices/rules/) mitigate the problem for known queries but are not a general solution for ad-hoc queries, which still have to perform reasonably well.

We know that we want to write in batches, but the only batches we get are vertical sets of data points across series. When querying data points for a series over a time window, not only would it be hard to figure out where the individual points can be found, we'd also have to read from a lot of random places on disk. With possibly millions of touched samples per query, this is slow even on the fastest SSDs. Reads will also retrieve more data from our disk than the requested 16 byte sample. SSDs will load a full page, HDDs will at least read an entire sector. Either way, we are wasting precious read throughput.  
So ideally, samples for the same series would be stored sequentially so we can just scan through them with as few reads as possible. On top, we only need to know where this sequence starts to access all data points.

There's obviously a strong tension between the ideal pattern for writing collected data to disk and the layout that would be significantly more efficient for serving queries. It is the fundamental problem our TSDB has to solve.

**Current solution**

Time to take a look at how Prometheus's current storage, let's call it "V2", addresses this problem.  
We create one file per time series that contains all of its samples in sequential order. As appending single samples to all those files every few seconds is expensive, we batch up 1KiB chunks of samples for a series in memory and append those chunks to the individual files, once they are full. This approach solves a large part of the problem. Writes are now batched, samples are stored sequentially. It also enables incredibly efficient compression formats, based on the property that a given sample changes only very little with respect to the previous sample in the same series. Facebook's paper on their Gorilla TSDB describes a similar chunk-based approach and [introduces a compression format](http://www.vldb.org/pvldb/vol8/p1816-teller.pdf) that reduces 16 byte samples to an average of 1.37 bytes. The V2 storage uses various compression formats including a variation of Gorilla’s.

┌──────────┬─────────┬─────────┬─────────┬─────────┐ series A

└──────────┴─────────┴─────────┴─────────┴─────────┘

┌──────────┬─────────┬─────────┬─────────┬─────────┐ series B

└──────────┴─────────┴─────────┴─────────┴─────────┘

. . .

┌──────────┬─────────┬─────────┬─────────┬─────────┬─────────┐ series XYZ

└──────────┴─────────┴─────────┴─────────┴─────────┴─────────┘

chunk 1 chunk 2 chunk 3 ...

While the chunk-based approach is great, keeping a separate file for each series is troubling the V2 storage for various reasons:

* We actually need a lot more files than the number of time series we are currently collecting data for. More on that in the section on "Series Churn". With several million files, sooner or later way may run out of[inodes](https://en.wikipedia.org/wiki/Inode) on our filesystem. This is a condition we can only recover from by reformatting our disks, which is as invasive and disruptive as it could be. We generally want to avoid formatting disks specifically to fit a single application.
* Even when chunked, several thousands of chunks per second are completed and ready to be persisted. This still requires thousands of individual disk writes every second. While it is alleviated by also batching up several completed chunks for a series, this in return increases the total memory footprint of data which is waiting to be persisted.
* It's infeasible to keep all files open for reads and writes. In particular because ~99% of data is never queried again after 24 hours. If it is queried though though, we have to open up to thousands of files, find and read relevant data points into memory, and close them again. As this would result in high query latencies, data chunks are cached rather aggressively leading to problems outlined further in the section on "Resource Consumption".
* Eventually, old data has to be deleted and data needs to be removed from the front of millions of files. This means that deletions are actually write intensive operations. Additionally, cycling through millions of files and analyzing them makes this a process that often takes hours. By the time it completes, it might have to start over again. Oh yea, and deleting the old files will cause further write amplification for your SSD!
* Chunks that are currently accumulating are only held in memory. If the application crashes, data will be lost. To avoid this, the memory state is periodically checkpointed to disk, which may take significantly longer than the window of data loss we are willing to accept. Restoring the checkpoint may also take several minutes, causing painfully long restart cycles.

The key take away from the existing design is the concept of chunks, which we most certainly want to keep. The most recent chunks always being held in memory is also generally good. After all, the most recent data is queried the most by a large margin.  
Having one file per time series is a concept we would like to find an alternative to.

**Series Churn**

In the Prometheus context, we use the term series churn to describe that a set of time series becomes inactive, i.e. receives no more data points, and a new set of active series appears instead.  
For example, all series exposed by a given microservice instance have a respective “instance” label attached that identifies its origin. If we perform a rolling update of our microservice and swap out every instance with a newer version, series churn occurs. In more dynamic environments those events may happen on an hourly basis. Cluster orchestration systems like Kubernetes allow continuous auto-scaling and frequent rolling updates of applications, potentially creating tens of thousands of new application instances, and with them completely new sets of time series, every day.

series

^

│ . . . . . .

│ . . . . . .

│ . . . . . .

│ . . . . . . .

│ . . . . . . .

│ . . . . . . .

│ . . . . . .

│ . . . . . .

│ . . . . .

│ . . . . .

│ . . . . .

v

<-------------------- time --------------------->

So even if the entire infrastructure roughly remains constant in size, over time there's a linear growth of time series in our database. While a Prometheus server will happily collect data for 10 million time series, query performance is significantly impacted if data has to be found among a billion series.

**Current solution**

The current V2 storage of Prometheus has an index based on LevelDB for all series that are currently stored. It allows querying series containing a given label pair, but lacks a scalable way to combine results from different label selections.  
For example, selecting all series with label \_\_name\_\_="requests\_total" works efficiently, but selecting all series withinstance="A" AND \_\_name\_\_="requests\_total" has scalability problems. We will later revisit what causes this and which tweaks are necessary to improve lookup latencies.

This problem is in fact what spawned the initial hunt for a better storage system. Prometheus needed an improved indexing approach for quickly searching hundreds of millions of time series.

**Resource consumption**

Resource consumption is one of the consistent topics when trying to scale Prometheus (or anything, really). But it's not actually the absolute resource hunger that is troubling users. In fact, Prometheus manages an incredible throughput given its requirements. The problem is rather its relative unpredictability and instability in face of changes. By its architecture the V2 storage slowly builds up chunks of sample data, which causes the memory consumption to ramp up over time. As chunks get completed, they are written to disk and can be evicted from memory. Eventually, Prometheus's memory usage reaches a steady state. That is until the monitored environment changes — series churn increases the usage of memory, CPU, and disk IO every time we scale an application or do a rolling update.  
If the change is ongoing, it will yet again reach a steady state eventually but it will be significantly higher than in a more static environment. Transition periods are often multiple hours long and it is hard to determine what the maximum resource usage will be.

The approach of having a single file per time series also makes it way too easy for a single query to knock out the Prometheus process. When querying data that is not cached in memory, the files for queried series are opened and the chunks containing relevant data points are read into memory. If the amount of data exceeds the memory available, Prometheus quits rather ungracefully by getting OOM-killed.  
After the query is completed the loaded data can be released again but it is generally cached much longer to serve subsequent queries on the same data faster. The latter is a good thing obviously.

Lastly, we looked at write amplification in the context of SSDs and how Prometheus addresses it by batching up writes to mitigate it. Nonetheless, in several places it still causes write amplification by having too small batches and not aligning data precisely on page boundaries. For larger Prometheus servers, a reduced hardware lifetime was observed in the real world. Chances are that this is still rather normal for database applications with high write throughput, but we should keep an eye on whether we can mitigate it.

**Starting Over**

By now we have a good idea of our problem domain, how the V2 storage solves it, and where its design has issues. We also saw some great concepts that we want to adapt more or less seamlessly. A fair amount of V2's problems can be addressed with improvements and partial redesigns, but to keep things fun (and after carefully evaluating my options, of course), I decided to take a stab at writing an entire time series database — from scratch, i.e. writing bytes to the file system.

The critical concerns of performance and resource usage are a direct consequence of the chosen storage format. We have to find the right set of algorithms and disk layout for our data to implement a well-performing storage layer.

This is where I take the shortcut and drive straight to the solution — skip the headache, failed ideas, endless sketching, tears, and despair.

**V3 — Macro Design**

What's the macro layout of our storage? In short, everything that is revealed when running tree on our data directory. Just looking at that gives us a surprisingly good picture of what is going on.

$ tree ./data

./data

├── b-000001

│ ├── chunks

│ │ ├── 000001

│ │ ├── 000002

│ │ └── 000003

│ ├── index

│ └── meta.json

├── b-000004

│ ├── chunks

│ │ └── 000001

│ ├── index

│ └── meta.json

├── b-000005

│ ├── chunks

│ │ └── 000001

│ ├── index

│ └── meta.json

└── b-000006

├── meta.json

└── wal

├── 000001

├── 000002

└── 000003

At the top level, we have a sequence of numbered blocks, prefixed with b-. Each block obviously holds a file containing an index and a "chunk" directory holding more numbered files. The “chunks” directory contains nothing but raw chunks of data points for various series. Just as for V2, this makes reading series data over a time windows very cheap and allows us to apply the same efficient compression algorithms. The concept has proven to work well and we stick with it. Obviously, there is no longer a single file per series but instead a handful of files holds chunks for many of them.  
The existence of an “index” file should not be surprising. Let's just assume it contains a lot of black magic allowing us to find labels, their possible values, entire time series and the chunks holding their data points.

But why are there several directories containing the layout of index and chunk files? And why does the last one contain a "wal" directory instead? Understanding those two questions, solves about 90% of our problems.

**Many Little Databases**

We partition our horizontal dimension, i.e. the time space, into non-overlapping blocks. Each block acts as a fully independent database containing all time series data for its time window. Hence, it has its own index and set of chunk files.

t0 t1 t2 t3 now

┌───────────┐ ┌───────────┐ ┌───────────┐ ┌───────────┐

│ │ │ │ │ │ │ │ ┌────────────┐

│ │ │ │ │ │ │ mutable │ <─── write ──── ┤ Prometheus │

│ │ │ │ │ │ │ │ └────────────┘

└───────────┘ └───────────┘ └───────────┘ └───────────┘ ^

└──────────────┴───────┬──────┴──────────────┘ │

│ query

│ │

merge ─────────────────────────────────────────────────┘

Every block of data is immutable. Of course, we must be able to add new series and samples to the most recent block as we collect new data. For this block, all new data is written to an in-memory database that provides the same lookup properties as our persistent blocks. The in-memory data structures can be updated efficiently. To prevent data loss, all incoming data is also written to a temporary write ahead log, which is the set of files in our “wal” directory, from which we can re-populate the in-memory database on restart.  
All these files come with their own serialization format, which comes with all the things one would expect: lots of flags, offsets, varints, and CRC32 checksums. Good fun to come up with, rather boring to read about.

This layout allows us to fan out queries to all blocks relevant to the queried time range. The partial results from each block are merged back together to form the overall result.

This horizontal partitioning adds a few great capabilities:

* When querying a time range, we can easily ignore all data blocks outside of this range. It trivially addresses the problem of series churn by reducing the set of inspected data to begin with.
* When completing a block, 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.
* We keep the good property of V2 that recent chunks, which are queried most, are always hot in memory.
* Nicely enough, we are also no longer bound to the fixed 1KiB chunk size to better align data on disk. We can pick any size that makes the most sense for the individual data points and chosen compression format.
* Deleting old data becomes extremely cheap and instantaneous. We merely have to delete a single directory. Remember, in the old storage we had to analyze and re-write up to hundreds of millions of files, which could take hours to converge.

Each block also contains a meta.json file. It simply holds human-readable information about the block to easily understand the state of our storage and the data it contains.

**mmap**

Moving from millions of small files to a handful of larger allows us to keep all files open with little overhead. This unblocks the usage of [mmap(2)](https://en.wikipedia.org/wiki/Mmap), a system call that allows us to transparently back a virtual memory region by file contents. For simplicity, you might want to think of it like swap space, just that all our data is on disk already and no writes occur when swapping data out of memory.

This means we can treat all contents of our database as if they were in memory without occupying any physical RAM. Only if we access certain byte ranges in our database files, the operating system lazily loads pages from disk. This puts the operating system in charge of all memory management related to our persisted data. Generally, it is more qualified to make such decisions, as it has the full view on the entire machine and all its processes. Queried data can be rather aggressively cached in memory, yet under memory pressure the pages will be evicted. If the machine has unused memory, Prometheus will now happily cache the entire database, yet will immediately return it once another application needs it.  
Therefore, queries can longer easily OOM our process by querying more persisted data than fits into RAM. The memory cache size becomes fully adaptive and data is only loaded once the query actually needs it.

From my understanding, this is how a lot of databases work today and an ideal way to do it if the disk format allows — unless one is confident to outsmart the OS from within the process. We certainly get a lot of capabilities with little work from our side.

**Compaction**

The storage has to periodically "cut" a new block and write the previous one, which is now completed, onto disk. Only after the block was successfully persisted, the write ahead log files, which are used to restore in-memory blocks, are deleted.  
We are interested in keeping each block reasonably short (about two hours for a typical setup) to avoid accumulating too much data in memory. When querying multiple blocks, we have to merge their results into an overall result. This merge procedure obviously comes with a cost and a week-long query should not have to merge 80+ partial results.

To achieve both, we introduce compaction. Compaction describes the process of taking one or more blocks of data and writing them into a, potentially larger, block. It can also modify existing data along the way, e.g. dropping deleted data, or restructuring our sample chunks for improved query performance.

t0 t1 t2 t3 t4 now

┌────────────┐ ┌──────────┐ ┌───────────┐ ┌───────────┐ ┌───────────┐

│ 1 │ │ 2 │ │ 3 │ │ 4 │ │ 5 mutable │ before

└────────────┘ └──────────┘ └───────────┘ └───────────┘ └───────────┘

┌─────────────────────────────────────────┐ ┌───────────┐ ┌───────────┐

│ 1 compacted │ │ 4 │ │ 5 mutable │ after (option A)

└─────────────────────────────────────────┘ └───────────┘ └───────────┘

┌──────────────────────────┐ ┌──────────────────────────┐ ┌───────────┐

│ 1 compacted │ │ 3 compacted │ │ 5 mutable │ after (option B)

└──────────────────────────┘ └──────────────────────────┘ └───────────┘

In this example we have the sequential blocks [1, 2, 3, 4]. Blocks 1, 2, and 3 can be compacted together and the new layout is [1, 4]. Alternatively, compact them in pairs of two into [1, 3]. All time series data still exist but now in fewer blocks overall. This significantly reduces the merging cost at query time as fewer partial query results have to be merged.

**Retention**

We saw that deleting old data was a slow process in the V2 storage and put a toll on CPU, memory, and disk alike. How can we drop old data in our block based design? Quite simply, by just deleting the directory of a block that has no data within our configured retention window. In the example below, block 1 can safely be deleted, whereas 2 has to stick around until it falls fully behind the boundary.

|

┌────────────┐ ┌────┼─────┐ ┌───────────┐ ┌───────────┐ ┌───────────┐

│ 1 │ │ 2 | │ │ 3 │ │ 4 │ │ 5 │ . . .

└────────────┘ └────┼─────┘ └───────────┘ └───────────┘ └───────────┘

|

|

retention boundary

The older data gets, the larger the blocks may become as we keep compacting previously compacted blocks. An upper limit has to be applied so blocks don’t grow to span the entire database and thus diminish the original benefits of our design.  
Conveniently, this also limits the total disk overhead of blocks that are partially inside and partially outside of the retention window, i.e. block 2 in the example above. When setting the maximum block size at 10% of the total retention window, our total overhead of keeping block 2 around is also bound by 10%.

Summed up, retention deletion goes from very expensive, to practically free.

If you've come this far and have some background in databases, you might be asking one thing by now: Is any of this new? — Not really; and probably for the better.

The pattern of batching data up in memory, tracked in a write ahead log, and periodically flushed to disk is ubiquitous today.  
The benefits we have seen apply almost universally regardless of the data's domain specifics. Prominent open source examples following this approach are LevelDB, Cassandra, InfluxDB, or HBase. The key takeaway is to avoid reinventing an inferior wheel, researching proven methods, and applying them with the right twist.  
Running out of places to add your own magic dust later is an unlikely scenario.

**The Index**

The initial motivation to investigate storage improvements were the problems brought by series churn. The block-based layout reduces the total number of series that have to be considered for serving a query. So assuming our index lookup was of complexity O(n^2), we managed to reduce the n a fair amount and now have an improved complexity of O(n^2) — uhm, wait... damnit.  
A quick flashback to "Algorithms 101" reminds us that this, in theory, did not buy us anything. If things were bad before, they are just as bad now. Theory can be depressing.

In practice, most of our queries will already be answered significantly faster. Yet, queries spanning the full time range remain slow even if they just need to find a handful of series. My original idea, dating back way before all this work was started, was a solution to exactly this problem: we need a more capable [inverted index](https://en.wikipedia.org/wiki/Inverted_index).  
An inverted index provides a fast lookup of data items based on a subset of their contents. Simply put, I can look up all series that have a label app=”nginx" without having to walk through every single series and check whether it contains that label.

For that, each series is assigned a unique ID by which it can be retrieved in constant time, i.e. O(1). In this case the ID is our forward index.

**Example:** If the series with IDs 10, 29, and 9 contain the label app="nginx", the inverted index for the label "nginx" is the simple list [10, 29, 9], which can be used to quickly retrieve all series containing the label. Even if there were 20 billion further series, it would not affect the speed of this lookup.

In short, if n is our total number of series, and m is the result size for a given query, the complexity of our query using the index is now O(m). Queries scaling along the amount of data they retrieve (m) instead of the data body being searched (n) is a great property as m is generally significantly smaller.  
For brevity, let’s assume we can retrieve the inverted index list itself in constant time.

Actually, this is almost exactly the kind of inverted index V2 has and a minimum requirement to serve performant queries across millions of series. The keen observer will have noticed, that in the worst case, a label exists in all series and thus m is, again, in O(n). This is expected and perfectly fine. If you query all data, it naturally takes longer. Things become problematic once we get involved with more complex queries.

**Combining Labels**

Labels associated with millions of series are common. Suppose a horizontally scaling “foo” microservice with hundreds of instances with thousands of series each. Every single series will have the label app="foo". Of course, one generally won't query all series but restrict the query by further labels, e.g. I want to know how many requests my service instances received and query \_\_name\_\_="requests\_total" AND app="foo".

To find all series satisfying both label selectors, we take the inverted index list for each and intersect them. The resulting set will typically be orders of magnitude smaller than each input list individually. As each input list has the worst case size O(n), the brute force solution of nested iteration over both lists, has a runtime of O(n^2). The same cost applies for other set operations, such as the union (app="foo" OR app="bar"). When adding further label selectors to the query, the exponent increases for each to O(n^3), O(n^4), O(n^5), ... O(n^k). A lot of tricks can be played to minimize the effective runtime by changing the execution order. The more sophisticated, the more knowledge about the shape of the data and the relationships between labels is needed. This introduces a lot of complexity, yet does not decrease our algorithmic worst case runtime.

This is essentially the approach in the V2 storage and luckily a seemingly slight modification is enough gain significant improvements. What happens if we assume that the IDs in our inverted indices are sorted?

Suppose this example of lists for our initial query:

\_\_name\_\_="requests\_total" -> [ 9999, 1000, 1001, 2000000, 2000001, 2000002, 2000003 ]

app="foo" -> [ 1, 3, 10, 11, 12, 100, 311, 320, 1000, 1001, 10002 ]

intersection => [ 1000, 1001 ]

The intersection is fairly small. We can find it by setting a cursor at the beginning of each list and always advancing the one at the smaller number. When both numbers are equal, we add the number to our result and advance both cursors. Overall, we scan both lists in this zig-zag pattern and thus have a total cost of O(2n) = O(n) as we only ever move forward in either list.

The procedure for more than two lists of different set operations works similarly. So the number of k set operations merely modifies the factor (O(k\*n)) instead of the exponent (O(n^k)) of our worst-case lookup runtime. A great improvement.  
What I described here is a simplified version of the canonical search index used by practically any [full text search engine](https://en.wikipedia.org/wiki/Search_engine_indexing#Inverted_indices) out there. Every series descriptor is treated as a short "document", and every label (name + fixed value) as a "word" inside of it. We can ignore a lot of additional data typically encountered in search engine indices, such as word position and frequency data.  
Seemingly endless research exists on approaches improving the practical runtime, often making some assumptions about the input data. Unsurprisingly, there are also plenty of techniques to compress inverted indices that come with their own benefits and drawbacks. As our "documents" are tiny and the “words” are hugely repetitive across all series, compression becomes almost irrelevant. For example, a real-world dataset of ~4.4 million series with about 12 labels each has less than 5,000 unique labels. For our initial storage version, we stick to the basic approach without compression, and just a few simple tweaks added to skip over large ranges of non-intersecting IDs.

While keeping the IDs sorted may sound simple, it is not always a trivial invariant to keep up. For instance, the V2 storage assigns hashes as IDs to new series and we cannot efficiently build up sorted inverted indices.  
Another daunting task is modifying the indices on disk as data gets deleted or updated. Typically, the easiest approach is to simply recompute and rewrite them but doing so while keeping the database queryable and consistent. The V3 storage does exactly this by having a separate immutable index per block that is only modified via rewrite on compaction. Only the indices for the mutable blocks, which are held entirely in memory, need to be updated.

**Benchmarking**

I started initial development of the storage with a benchmark based on ~4.4 million series descriptors extracted from a real world data set and generated synthetic data points to feed into those series. This iteration just tested the stand-alone storage and was crucial to quickly identify performance bottlenecks and trigger deadlocks only experienced under highly concurrent load.

After the conceptual implementation was done, the benchmark could sustain a write throughput of 20 million data points per second on my Macbook Pro — all while a dozen Chrome tabs and Slack were running. So while this sounded all great it also indicated that there's no further point in pushing this benchmark (or running it in a less random environment for that matter). After all, it is synthetic and thus not worth much beyond a good first impression. Starting out about 20x above the initial design target, it was time to embed this into an actual Prometheus server, adding all the practical overhead and flakes only experienced in more realistic environments.

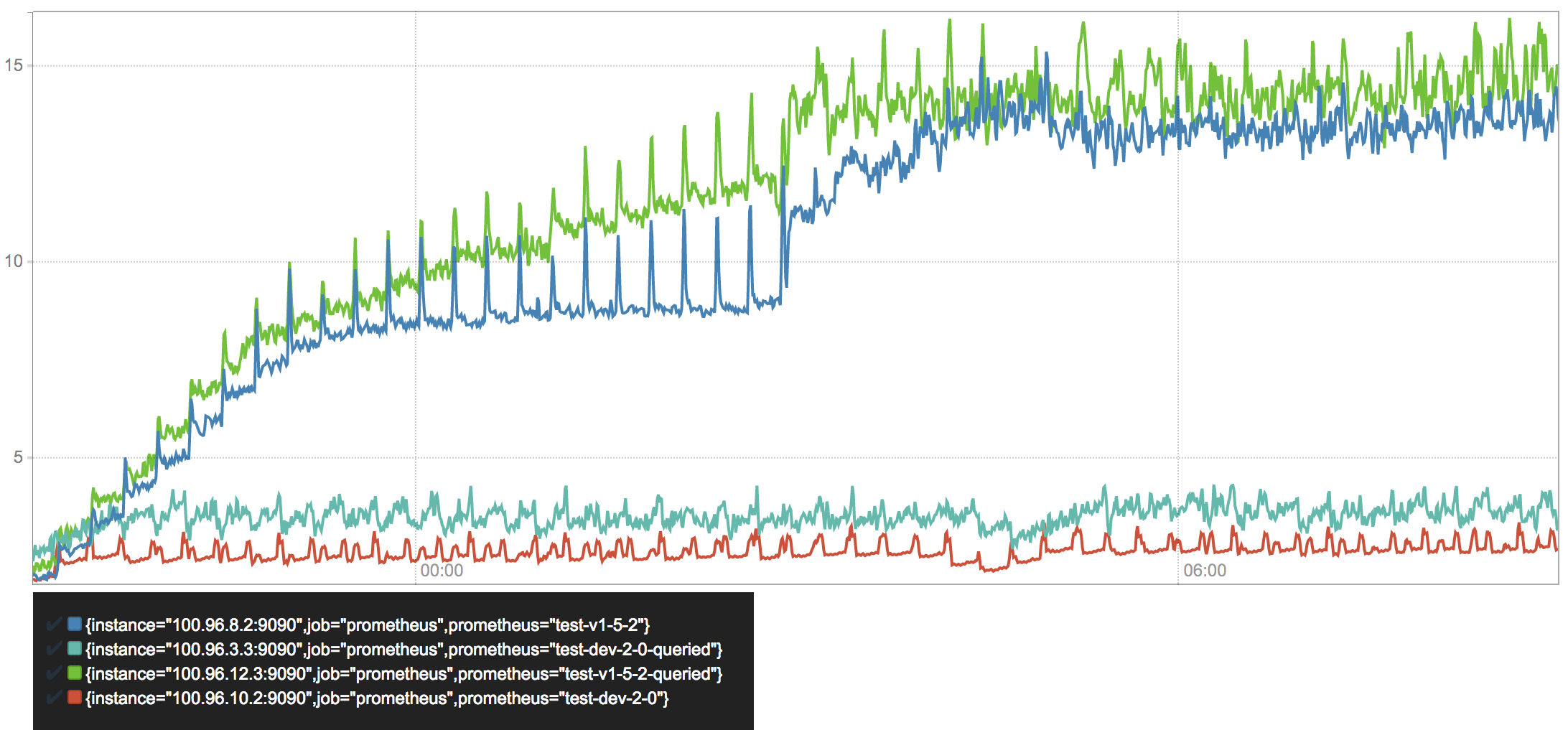
We actually had no reproducible benchmarking setup for Prometheus, in particular none that allowed A/B testing of different versions. Concerning in hindsight, but [now we have one](https://github.com/prometheus/prombench)!

Our tool allows us to declaratively define a benchmarking scenario, which is then deployed to a Kubernetes cluster on AWS. While this is not the best environment for all-out benchmarking, it certainly reflects our user base better than dedicated bare metal servers with 64 cores and 128GB of memory.  
We deploy two Prometheus 1.5.2 servers (V2 storage) and two Prometheus servers from the 2.0 development branch (V3 storage). Each Prometheus server runs on a dedicated machine with an SSD. A horizontally scaled application exposing typical microservice metrics is deployed to worker nodes. Additionally, the Kubernetes cluster itself and the nodes are being monitored. The whole setup is supervised by yet another Meta-Prometheus, monitoring each Prometheus server for health and performance.  
To simulate series churn, the microservice is periodically scaled up and down to remove old pods and spawn new pods, exposing new series. Query load is simulated by a selection of "typical" queries, run against one server of each Prometheus version.

Overall the scaling and querying load as well as the sampling frequency significantly exceed today's production deployments of Prometheus. For instance, we swap out 60% of our microservice instances every 15 minutes to produce series churn. This would likely only happen 1-5 times a day in a modern infrastructure. This ensures that our V3 design is capable of handling the workloads of the years ahead. As a result, the performance differences between Prometheus 1.5.2 and 2.0 are larger than in a more moderate environment.  
In total, we are collecting about 110,000 samples per second from 850 targets exposing half a million series at a time.

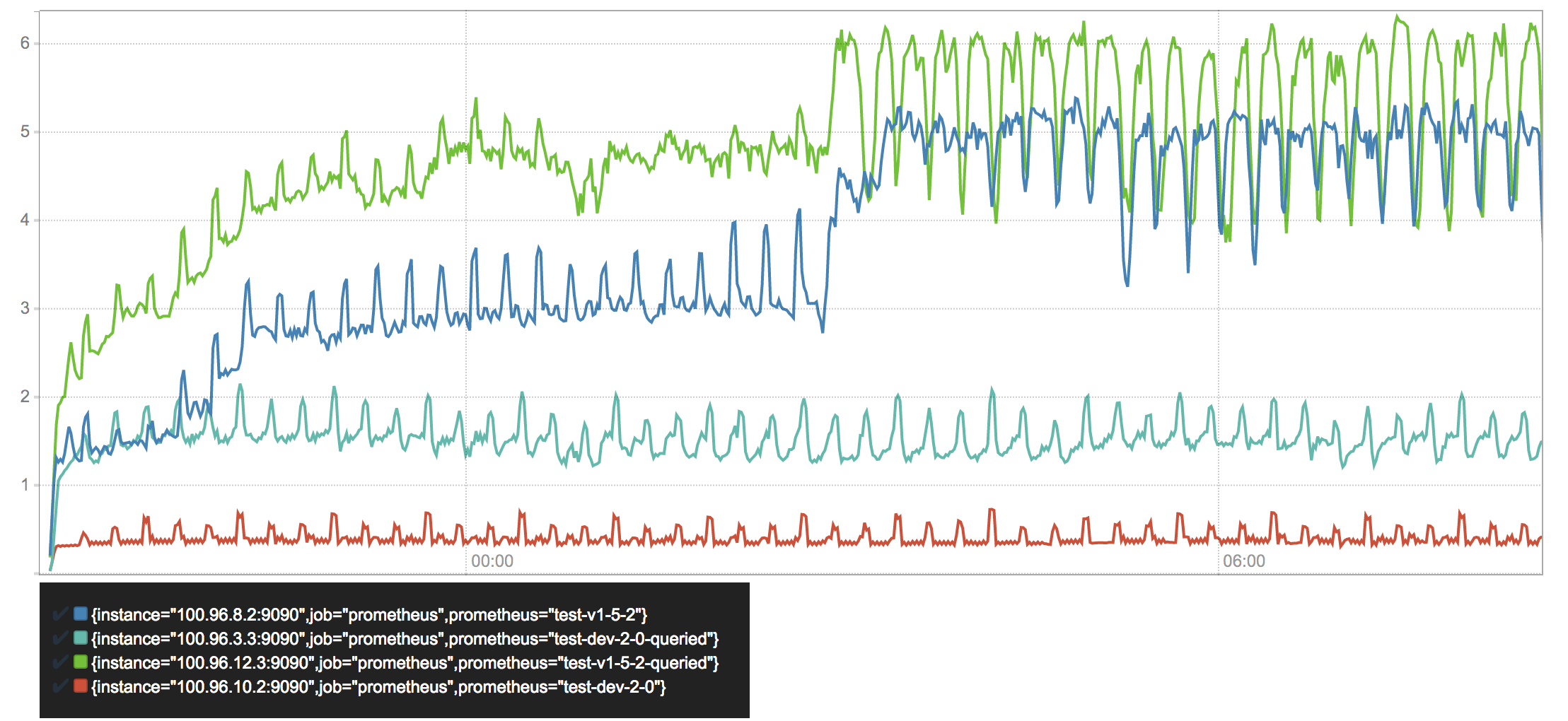
After leaving this setup running for a while, we can take a look at the numbers. We evaluate several metrics over the first 12 hours within both versiones reached a steady state.

Be aware of the slightly truncated Y axis in screen shots from the Prometheus graph UI.



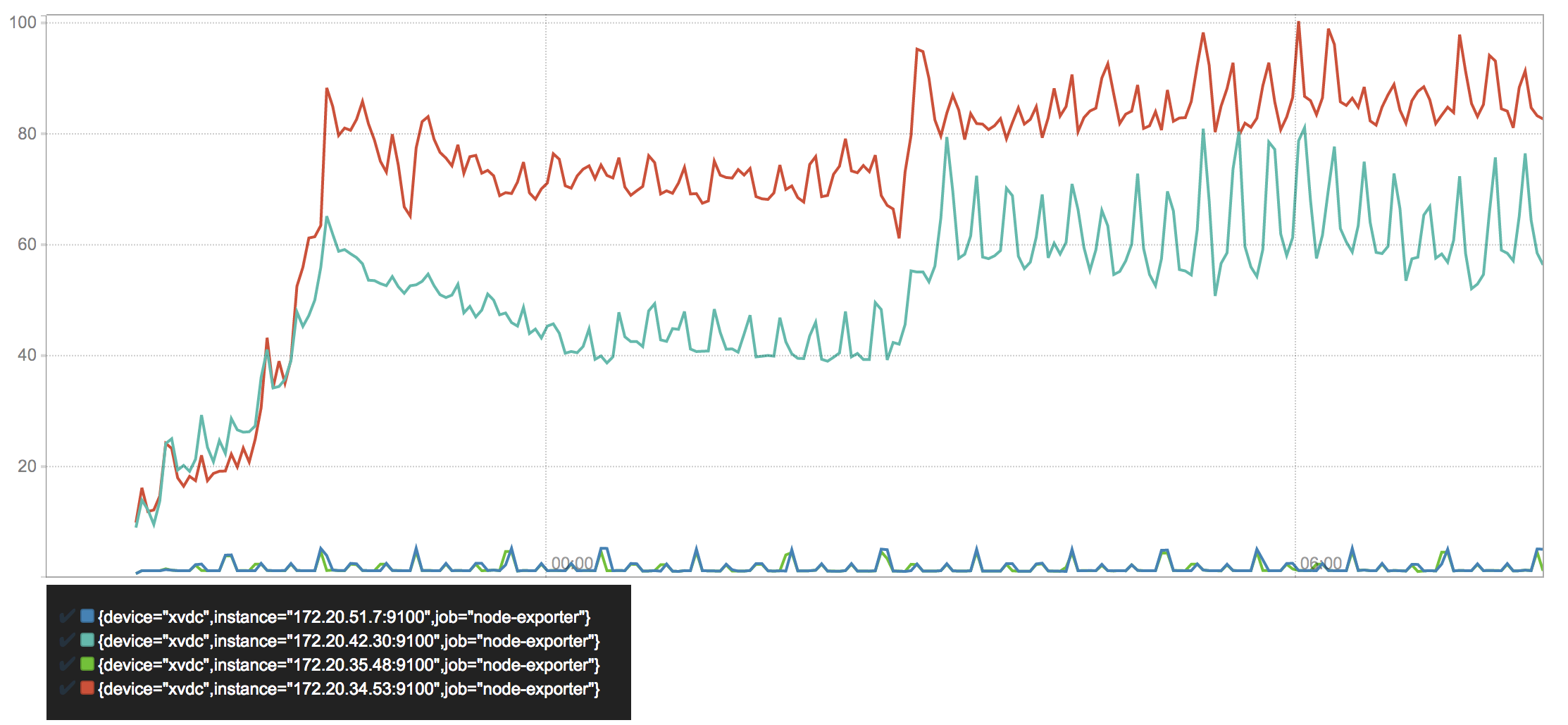
Heap memory usage in GB

Memory usage is the most troubling resource for users today as it is relatively unpredictable and it may cause the process to crash.  
Obviously, the queried servers are consuming more memory, which can largely be attributed to overhead of the query engine, which will be subject to future optimizations. Overall, Prometheus 2.0's memory consumption is reduced by 3-4x. After about six hours, there is a clear spike in Prometheus 1.5, which aligns with the our retention boundary at six hours. As deletions are quite costly, resource consumption ramps up. This will become visible throughout various other graphs below.



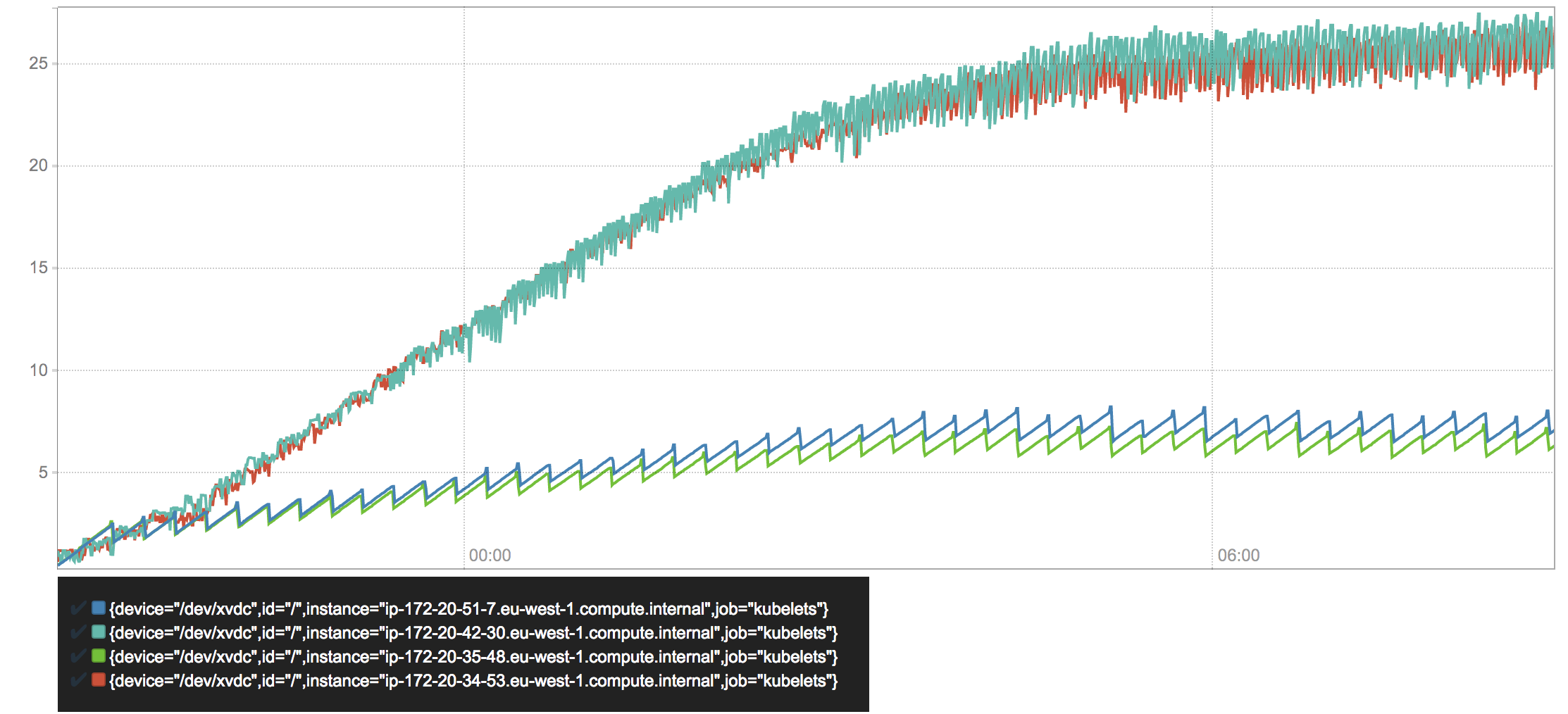
CPU usage in cores/second

A similar pattern shows for CPU usage, but the delta between queried and non-queried servers is more significant. Averaging at about 0.5 cores/sec while ingesting about 110,000 samples/second, our new storage becomes almost negligible compared to the cycles spent on query evaluation. In total the new storage needs 3-10 times fewer CPU resources.



Disk writes in MB/second

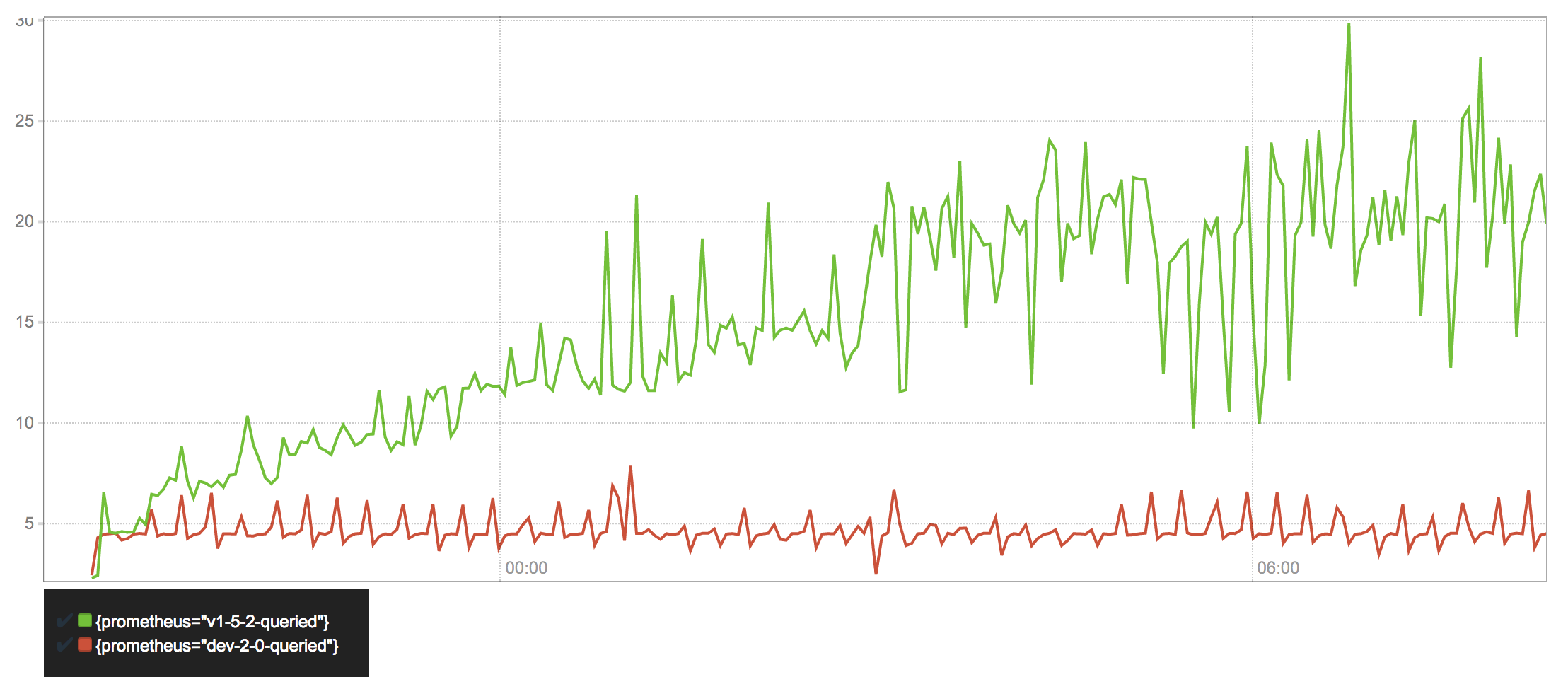
The by far most dramatic and unexpected improvement shows in write utilization of our disk. It clearly shows why Prometheus 1.5 is prone to wear out SSDs. We see an initial ramp-up as soon as the first chunks are persisted into the series files and a second ramp-up once deletion starts rewriting them. Surprisingly, the queried and non-queried server show a very different utilization.  
Prometheus 2.0 on the other hand, merely writes about a single Megabyte per second to its write ahead log. Writes periodically spike when blocks are compacted to disk. Overall savings: staggering 97-99%.



Disk size in GB

Closely related to disk writes is the total amount of occupied disk space. As we are using almost the same compression algorithm for samples, which is the bulk of our data, they should be about the same. In a more stable setup that would largely be true, but as we are dealing with high series churn, there's also the per-series overhead to consider.  
As we can see, Prometheus 1.5 ramps up storage space a lot faster before both versions reach a steady state as the retention kicks in. Prometheus 2.0 seems to have a significantly lower overhead per individual series. We can nicely see how space is linearly filled up by the write ahead log and instantaneously drops as its gets compacted. The fact that the lines for both Prometheus 2.0 servers do not exactly match is a fact that needs further investigation.

This all looks quite promising. The important piece left is query latency. The new index should have improved our lookup complexity. What has not substantially changed is processing of this data, e.g. in rate() functions or aggregations. Those aspects are part of the query engine.



99th percentile query latency in seconds

Expectations are completely met by the data. In Prometheus 1.5 the query latency increases over time as more series are stored. It only levels off once retention starts and old series are deleted. In contrast, Prometheus 2.0 stays in place right from the beginning.  
Some caution must be taken on how this data was collected. The queries fired against the servers were chosen by estimating a good mix of range and instant queries, doing heavier and more lightweight computations, and touching few or many series. It does not necessarily represent a real-world distribution of queries. It is also not representative for queries hitting cold data and we can assume that all sample data is practically always hot in memory in either storage.  
Nonetheless, we can say with good confidence, that the overall query performance became very resilient to series churn and improved by up to 4x in our straining benchmarking scenario. In a more static environment, we can assume query time to be mostly spent in the query engine itself and the improvement to be notably lower.



Ingested samples/second

Lastly, a quick look into our ingestion rates of the different Prometheus servers. We can see that both servers with the V3 storage have the same ingestion rate. After a few hours it becomes unstable, which is caused by various nodes of the benchmarking cluster becoming unresponsive due to high load rather than the Prometheus instances. (The fact that both 2.0 lines exactly match is hopefully convincing enough.)  
Both Prometheus 1.5.2 servers start suffering from significant drops in ingestion rate even though more CPU and memory resources are available. The high stress of series churn causes a larger amount of data to not be collected.

But what's the absolute maximum number of samples per second you could ingest now?

I don't know — and deliberately don't care.

There are a lot of factors that shape the data flowing into Prometheus and there is no single number capable of capturing quality. Maximum ingestion rate has historically been a metric leading to skewed benchmarks and neglect of more important aspects such as query performance and resilience to series churn. The rough assumption that resource usage increases linearly was confirmed by some basic testing. It is easy to extrapolate what could be possible.

Our benchmarking setup simulates a highly dynamic environment stressing Prometheus more than most real-world setups today. The results show we went way above our initial design goal, while running on non-optimal cloud servers. Ultimately, success will be determined by user feedback rather than benchmarking numbers.

**Note:** At time of writing this, Prometheus 1.6 is in development, which will allow configuring the maximum memory usage more reliably and may notably reduce overall consumption in favor of slightly increased CPU utilization. I did not repeat the tests against this as the overall results still hold, especially when facing high series churn.

**Conclusion**

Prometheus sets out to handle high cardinality of series and throughput of individual samples. It remains a challenging task, but the new storage seems to position us well for the hyper-scale, hyper-convergent, GIFEE infrastructure of the futu... well, it seems to work pretty well.

A [first alpha release of Prometheus 2.0](https://prometheus.io/blog/2017/04/10/promehteus-20-sneak-peak/) with the new V3 storage is available for testing. Expect crashes, deadlocks, and other bugs at this early stage.

The code for the storage itself can be found [in a separate project](https://github.com/prometheus/tsdb). It's surprisingly agnostic to Prometheus itself and could be widely useful for a wider range of applications looking for an efficient local storage time series database.

There's a long list of people to thank for their contributions to this work. Here they go in no particular order:

The groundlaying work by Bjoern Rabenstein and Julius Volz on the V2 storage engine and their feedback on V3 was fundamental to everything seen in this new generation.

Wilhelm Bierbaum's ongoing advice and insight contributed significantly to the new design. Brian Brazil's continous feedback ensured that we ended up with a semantically sound approach. Insightful discussions with Peter Bourgon validated the design and shaped this write-up.

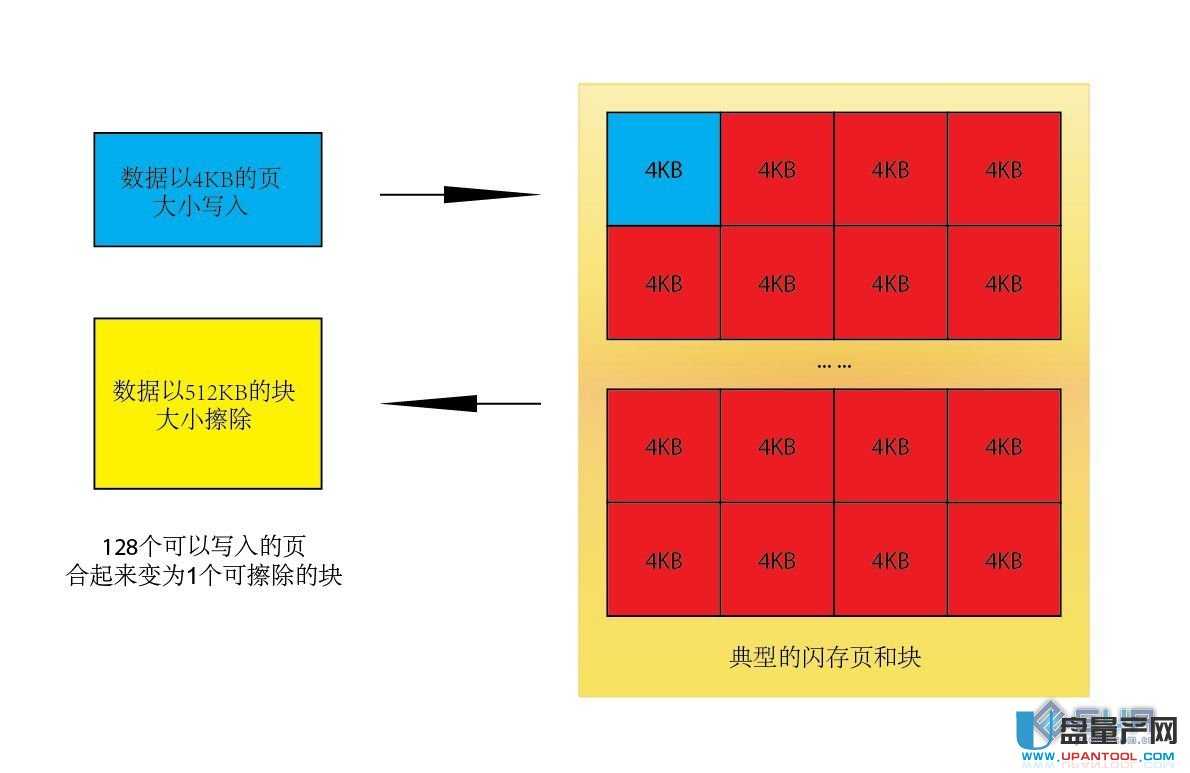
Not to forget my entire team at CoreOS and the company itself for supporting and sponsoring this work. Thanks to everyone who listened to my ramblings about SSDs, floats, and serialization formats again and again.

## 2@SSD的写入放大技术是什么-Write amplification

-固态硬盘教程-U盘量产网

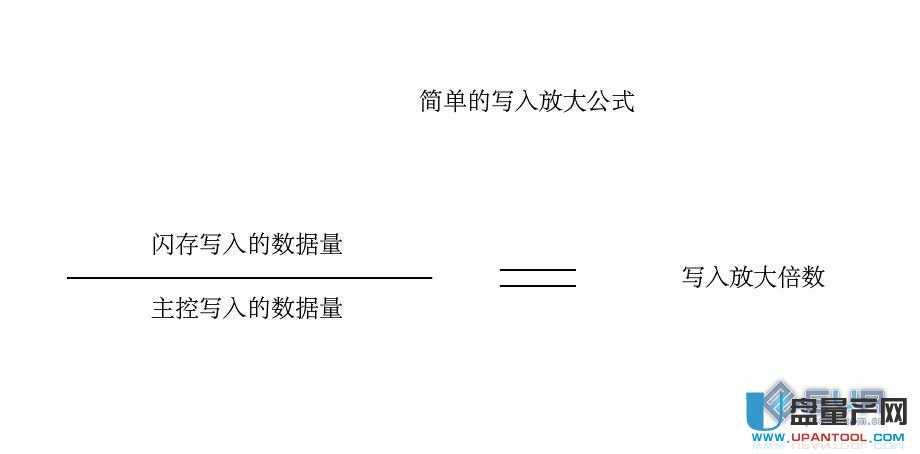
http://www.upantool.com/jiaocheng/ssd/2012/1499.html

**基本SSD操作方式**

写入放大（WA）是闪存和固态硬盘之间相关联的一个属性，因为闪存必须先删除才能改写（我们也叫“编程“），在执行这些操作的时候，移动（或重写）用户数 据和元数据(metadata)不止一次。这些多次的操作，不但增加了写入数据量，减少了SSD的使用寿命，而且还吃光了闪存的带宽（间接地影响了随机写 入性能）。许多因素会影响到SSD的写入放大，下面我就来稍微详细的解释一下。  
  
早在2008年，Intel公司和SiliconSystems公司（2009 年被西部数字收购）第一次提出了写入放大并在公开稿件里用到这个术语。他们当时的说法是，写入算法不可能低于1，但是这种说法在2009年被 SandForce打破，SandForce说他们的写入放大是0.5。  
  
  
由于闪存的运作特性，数据不能像在普通机械硬盘里那样被直接覆盖。当数据第一次写入SSD的时候，由于SSD内所有的颗粒都为已擦除状态，所以数据能够以 页为最小单位直接写入进去（一般是4K，参考颗粒资料），SSD上的主控制器，使用了逻辑和物理的映射系统来管理着闪存。（逻辑我们一般指的是LBA,而 物理指的是FTL)。当有新的数据写入时需要替换旧的数据时，SSD主控制器将把新的数据写入到另外的空白的闪存空间上（已擦除状态）然后更新逻辑LBA 地址来指向到新的物理FTL地址。而旧的地址内容就变成了无效的数据，但是要在上面再次写入的话，就需要首先擦除掉这个无效数据。（闪存运作特性，写入最 小单位是页，而擦除最小单位是块，一般为128~256个页）

那么问题就来了，闪存有编程和擦除的次数限制，这样每次的编程/擦除就叫做1个P/E(program/erase cycles)周期，大家应该都知道MLC一般是5000~10000次，而SLC是10万次左右（查闪存资料）。也就是说写入放大越低，P/E周期就越 少，闪存寿命就越久。

**写入放大的计算**

2008年，Intel公司和SiliconSystems公司（2009 年被西部数字收购）第一次提出了写入放大并在公开稿件里用到这个术语。所有的SSD都有一个写入放大值，这个数值是非固定的，取决于这个SSD写入的数据 是随机的还是持续的？写入量是多少？主控做了那些操作，等等。  
  
计算写入放大的公式大致是这样:  
  
对于单次操作，最简单的例子，比如我要写入一个4KB的数据，最坏的情况就是，一个块里已经没有干净空间了，但是有无效数据可以擦除，所以主控就把所有的 数据读出来，擦除块，再加上这个4KB新数据写回去，这个操作带来的写入放大就是: 我实际写4K的数据，造成了整个块（512KB）的写入操作，那就是128倍放大。同时带来了原本只需要简单的写4KB的操作变成读取(512KB)，擦 (512KB)，改写(512KB)，造成了延迟大大增加，速度慢是自然了。

**影响写入放大的因素**

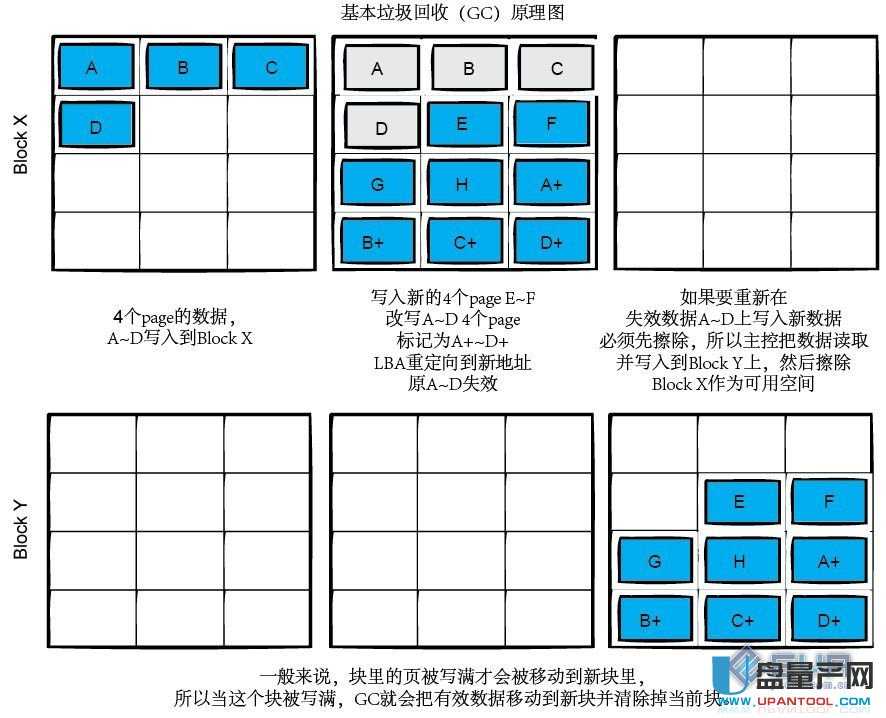
许多因素影响SSD的写入放大。下面我列出了主要因素，以及它们如何影响写放大。

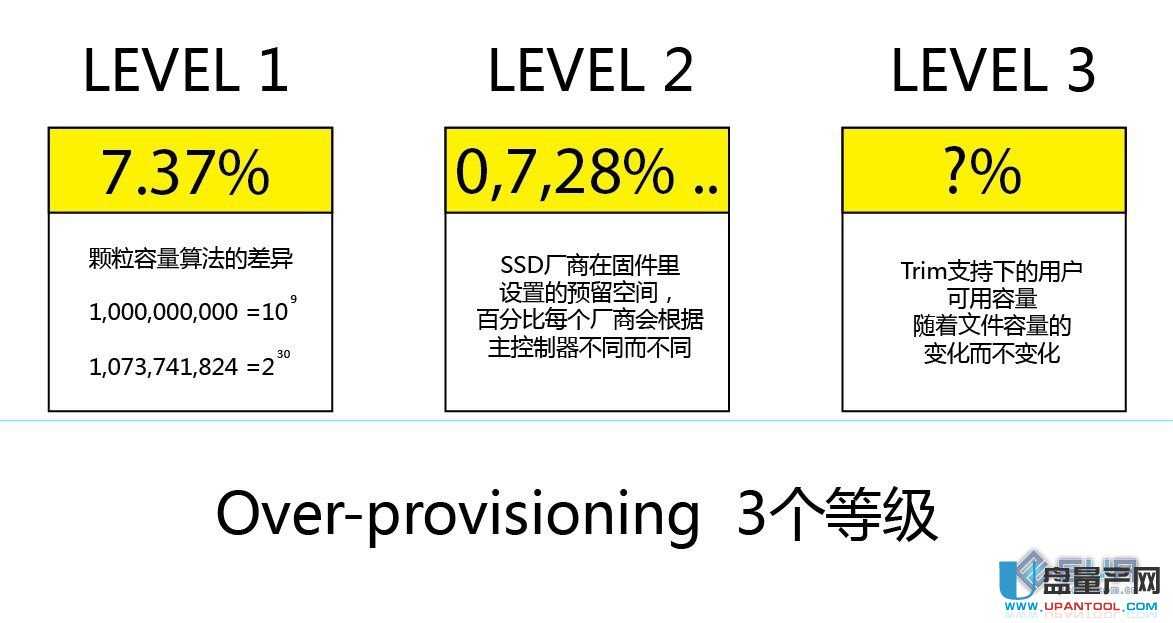
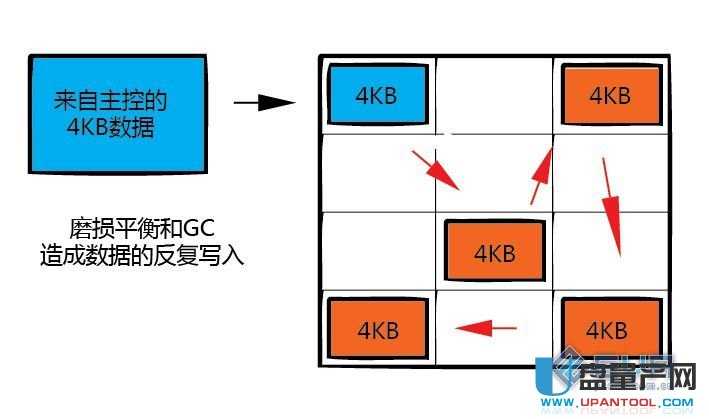
1. 垃圾回收(GC) Garbage collection --- 虽然增加了写入放大，但是速度有提升。

这个比较特殊的算法用来整理，移动，合并，删除闪存块来提升效率。

2. 预留空间（OP) Over-provisioning --- 减少写入放大，好。(预留空间越大，写入放大越低）   
  
在SSD的闪存上划出一部分空间留给主控做优化，用户不能操作的空间。  
  
3. TRIM 开启后可以减少写入放大，好。  
  
一个ATA指令，由操作系统发送给SSD主控，告诉主控哪些数据是无效的并且可以不用做垃圾回收操作。  
  
4. 可用容量 减少写入放大，好。(可用空间越大，写入放大越低）  
  
用户使用中没有用到的空间，需要有Trim支持，不然不会影响写入放大。  
  
5. 安全擦除 Secure Erase 减少写入放大，好  
  
清除所有用户数据和相关元数据，让SSD重置到初始性能。  
  
6. 静动数据分离 Separating Static and Dynamic Data 减少写入放大，好  
  
分组常改写和不常改写的数据。  
  
7. 持续写入 Sequential writes 减少写入放大，好  
  
理论上来说，持续写入的写入放大为1，但是某些因素还是会影响这个数值。  
  
8. 随机写入 Random writes 提高写入放大，不好  
  
随机写入会写入很多非连续的LBA,将会大大提升写入放大。  
  
9. 磨损平衡（WL） Wear Leveling 直接提高写入放大，不好  
  
确保闪存的每个块被写入的次数相等的一种机制。

**详细解释**

垃圾回收 Garbage collection  
  
 [](http://www.upantool.com/uploads/allimg/120701/2320163T3-2.jpg)  
一旦SSD的所有块都已经写入了一次，SSD主控制器将会初始化那些包含无效数据的块。（陈旧数据，这些块里的数据已经被更新的数据替换，已经无效了，没 了LBA地址），现在他们正在等待被删除，以便新的数据可以写入其中,如何优化并整理这些个等待被删除的无效数据，这个算法被称为垃圾收集（GC）。我们 可以看出这个操作是要有前提的，就是SSD必须要支持Trim技术，不然GC就显不出他的优势了（这也是为啥目前只有支持Trim的SSD才会有GC功 能），而GC的本质区别是它们何时处理？效率多少？

数据的最小写入单位是页，然而擦除的最小单位是块（大小取决于闪存，自己查资料，一般128~256页）。如果在块上的某些页中的数据不再需要，与在该块 内好的数据的其他所有页必须全部读取并重新写入到新的已擦除的块内。这个操作叫做Copy Block，每个主控都会带。（包括U盘主控，这也是为了磨损平衡考虑）然后主控制器再删除掉这个块，用来给下一次写入数据用。这种操作一切指令来自主控 而非用户的叫做GC，将会影响写入放大。请记住，GC有点像整理硬盘，所以要保证有一定的可用容量，可用容量越大，GC效率越高  
  
GC分为后台GC和主动GC  
  
我们知道垃圾的收集过程，包括读取和重写数据到闪存。意味着这样操作会大大降低主控的性能，因为占用了主控的能力和带宽。所以，一些SSD控制器采用所谓 的后台垃圾收集算法（也称为闲置垃圾收集），该控制器会使用空闲的时间来做垃圾收集，让主控在使用时一直保持高性能。试想一下如果垃圾回收把所有的空间都 整理合并过了，那样在性能提升的同时，也增加了写入放大，所以像barefoot主控的SSD（闲置GC）一般只垃圾回收一小部分的空白空间来限制过多的 写入操作。另一种方案就是主动GC，这需要有相当性能的主控制器，来保证在操作数据的同时进行GC操作，这类GC适合在服务器里用到，因为个人用户可以把 电脑闲置了做GC，但是服务器可不行，所以要保证性能的话必须在运行的同时做GC，这对主控制器的性能提出了很高的要求，SandForce的主控就是这 类。。  
  
手动GC  
  
记得以前我曾讨论过手动GC的操作  
  
乘着现在正好讨论到写入放大，我也再次强调下为何写入GC能成功的道理：  
  
不管你的SSD是不是支持Trim或者在RAID阵列状态下，手动GC或多或少都会有点作用（特殊的除外），为何？  
  
SSD的NAND颗粒有2个状态，物理上来说就是颗粒充电表示1，颗粒放电表示0，擦除数据意味着全盘写1（充电）。颗粒必须以块为最小单位一下子充电， 能以页为单位一个个放电。前面我们得知，如果SSD不支持Trim的话（RAID阵列目前都不支持），在全部SSD写满后，主控并不知道颗粒的块中哪些数 据是无效的，所以它认为他们都有效，操作系统的LBA却知道（因为新数据有LBA,无效数据的LBA已经被重定向了），所以不支持Trim的SSD就意味 着全盘颗粒写满后（指的是颗粒写满，其中包括有效和无效数据，不是我们通常看到的系统里可用容量满）再写入数据要等待写前的擦除操作，系统就会在没用到的 LBA地址下做写入操作。说白了就是系统知道哪里可以写而SSD主控不知道。手动GC正是利用了这一点，做了全盘填FF操作，FF在逻辑上就是1，那么就 是全盘颗粒里的无效数据处填1，1在NAND颗粒上代表充电，代表擦除。所以随着用户的可用容量越来越少，然后再一下子删除这个生成出来的大文件，代表了 全部可用容量区域都为逻辑1（擦除状态），这个状态就是GC操作后删除无效数据区块后的状态，所以叫做手动GC，自然这些区块在之后的操作中可以直接写入 而不需要再擦除了。  
  
预留空间 Over-provisioning  
  
预留空间一般是指用户不可操作的容量，为实际物理闪存容量减去用户可用容量。这块取用一般被用来做优化，包括磨损均衡，GC,Trim和坏块映射。  
  
预留空间分为3层：  
[](http://www.upantool.com/uploads/allimg/120701/23201C012-3.jpg)  
第一层为固定的7.37%，这个数字是如何得出的哪？我们知道机械硬盘和SSD的厂商容量是这样算的，1GB是1,000,000,000字节（10的9 次方），但是闪存的实际容量是每GB=1,073,741,824，(2的30次方) ，2者相差7.37%。所以说假设1块128GB的SSD，用户得到的容量是128,000,000,000字节，多出来的那个7.37%就被主控固件用 做OP了。  
  
第二层来自制造商的设置，通常为0%，7%和28%等，打个比方，对于128G颗粒的SandForce主控SSD,市场上会有120G和100G两种型号卖，这个取决于厂商的固件设置，这个容量不包括之前的第一层7.37%。  
  
第三层是用户在日常使用中可以分配的预留空间，像Fusion-IO公司还给用户工具自己调节大小来满足不同的耐用度和性能，而用户也可以自己在分区的时候，不分到完全的SSD容量来达到同样的目的。（要有Trim支持）  
  
预留空间虽然让SSD的可用容量小了，但是带来了减少写入放大，提高耐久，提高性能的效果。  
  
TRIM  
  
Trim是一种SATA命令，他能让操作系统在删除某个文件或者格式化后告诉SSD主控这个数据块不再需要了。  
  
一般情况下，当LBA被操作系统更新后，只有随着之后的每次数据写入（其实等于覆盖），SSD主控制器才知道这个地址原来早已经失效了。（之前认为每个数 据都是有效的）在Win7里，由于Trim的引入解决了这个问题，当某些文件被删除或者格式化了整个分区，操作系统把Trim指令和在操作中更新的LBA 一起发给SSD主控制器（其中包含了无效数据地址），这样在之后的GC操作中，无效数据就能被清空了，减少了写入放大同时也提升了性能。  
  
可以看这个帖子： [Trim解析](http://www.upantool.com/jiaocheng/ssd/2012/1500.html)  
  
Trim的依赖性和局限性  
  
1. Trim命令需要SSD的支持，某些老型号的SSD可以靠刷新固件得到Trim支持（G2,barefoot,YK40），或者用一些独特的工具（barefoot wiper)提取出系统里所有无效的LBA告诉SSD主控并清除。  
  
2. Trim命令之后，速度并不一定是立马就能提升的，因为Trim后的干净空间可能随机的包含在每个块里，只有等着多次的copy block操作和主控的GC操作才能明显感觉到速度的提升。  
  
3. 就算操作系统，驱动，SSD主控固件都满足Trim命令了，也不代表在某些特定环境下能工作，比如RAID阵列和数据库（至少到目前为止）。  
  
可用空间  
  
SSD控制器会使用所有的可用空间做垃圾回收和磨损均衡。保证一定的可用空间可以提升SSD效率，减少写入放大。（前提是支持Trim)  
  
安全擦除 Secure erase  
  
ATA安全擦除命令用来清除在磁盘上的所有用户数据，这个指令会让SSD回到出厂性能（最优性能，最少写入放大），但是随着之后的使用，GC，写入放大又会慢慢增加回来。  
  
许多软件使用ATA安全擦除指令来重置磁盘，最著名的为HDDErase。对SSD来说，重置就是全盘加电（逻辑1），瞬间即可完成清除所有数据让SSD回到初始状态。  
  
静动数据分离 Separating Static and Dynamic Data  
  
高端SSD主控制器支持静态和动态数据的分离处理，此操作要求SSD主控制器对LBA里经常写入（动态数据，热数据）和不经常写入（静态数据，冷数据）的 数据块进行归类，因为如果块里包含了静态和动态数据，在做GC操作的时候会为了改写其实没必要改写的静态数据而增加了写入放大，所以把包含静态数据的块归 类后，因为不常改写，减少了写入放大。但是迟早SSD主控会把这块静态的数据转移到别的地方来弥补平衡磨损。（因为静态数据占着的数据块一直不改写，编程 次数低于平均值的话，会造成颗粒磨损不平衡，违背了WL，真够矛盾的。）  
  
持续写入  
  
当SSD持续的写入数据时，写入放大一般为1，原因是随着数据写入，整个块都是持续的填充着同一个文件，如果系统确认这个文件需要改写或者删除，整个块都 可以被标记为无效（需要Trim支持），自然就不需要之后的GC操作了。（读取整个块并写入新的块）这个块只需要擦除，比读，改，写更快速有效。  
  
随机写入  
  
一个SSD主控的随机写入峰值速度一般发生在安全擦除后，完全GC，全盘Trim，或新安装的状态下。  
而随机写入的能力取决于主控制器的1.通道数。2.固件效率。3.闪存颗粒写入页面的性能。  
然后就是写入放大的多少，越接近于1越好，小于1更那好。  
当全盘颗粒都被写过后，GC功能就将被启用，速度就会受到影响，之后的写入放大就会达到SSD主控制器的最大倍数。大量的随机小文件的写入是“闪存杀手”。  
  
磨损平衡（WL） Wear Leveling  
[](http://www.upantool.com/uploads/allimg/120701/23201A035-4.jpg)  
假设一个特定的块被持续的编程写入而不编程写入到别的块，那么这个块将很快被消耗掉编程寿命，造成整个SSD的报废。处于这个原因，SSD主控制器要平均 分配每个块的编程次数，这个技术叫做磨损平衡。在最乐观的情况下，这个技术会让全盘的颗粒磨损程度接近并同时报废。可惜的是这个技术要牺牲写入放大，假设 对于冷数据，必须经常的移动到别的块，再把热数据移过来，保证2边的块都是一样的磨损度，无谓的增加了写入次数。关键就是要找一个最优化的算法来尽可能的 同时最佳化这2个矛盾的条件。  
  
总结：  
  
写入放大是个很关键的SSD指标，我们知道闪存的写入速度比读取速度慢的多，所以现在的SSD主控制器用多通道来提升写入速度（相对写入速度，读取速度其实早已经不是啥大问题了），但是如果写入放大太高，意味着一样的操作需要写入更多的数据，这样速度自然就快不起来了。  
  
废话那么多，就是想说好主控和坏主控之间的性能差别主要就是体现在放大上，你想想，假设大家都一样的通道数，颗粒也相同，可是差别为何那么大？说白了就是一个写的少而另个写的多了嘛。一切算法技术的根本目标是要尽可能的减少写入放大。

# 联邦例子

## Prometheus Time Series Collection and Processing Server

https://prometheus.wmflabs.org/status

**Runtime Information**

|  |  |
| --- | --- |
| **Uptime** | 2016-11-01 19:01:48.414119863 +0000 UTC |

**Build Information**

|  |  |
| --- | --- |
| **branch** | debian/sid |
| **buildDate** | 20160125-11:05:02 |
| **buildUser** | pkg-go-maintainers@lists.alioth.debian.org |
| **goVersion** | go1.5.3 |
| **revision** | 0.16.2+ds-1 |
| **version** | 0.16.2+ds |

**Configuration**

global:

scrape\_interval: 60s

evaluation\_interval: 15s

external\_labels:

monitor: test-prometheus

zone: labs

scrape\_configs:

- job\_name: prometheus

target\_groups:

- targets:

- localhost:9090

- job\_name: node

target\_groups:

- targets:

- localhost:9100

- job\_name: mysqld\_exporter

target\_groups:

- targets:

- labsdb1001.eqiad.wmnet:9501

- labsdb1002.eqiad.wmnet:9502

- labsdb1003.eqiad.wmnet:9503

- labsdb1004.eqiad.wmnet:9504

- labsdb1005.eqiad.wmnet:9505

- labsdb1006.eqiad.wmnet:9506

- labsdb1007.eqiad.wmnet:9507

relabel\_configs:

- source\_labels: [\_\_address\_\_]

regex: (.\*):(.\*)

target\_label: instance

replacement: ${1}:9104 # fake mysqld exporter port for now

- source\_labels: [\_\_address\_\_]

regex: (.\*):(.\*)

target\_label: \_\_address\_\_

replacement: test-prometheus2:${2} # mysqld exporter.

# federation example: collect all metrics starting with 'project:' from

# per-project prometheus servers

- job\_name: 'federate'

scrape\_interval: 60s

honor\_labels: true

metrics\_path: '/federate'

params:

'match[]':

- '{\_\_name\_\_=~"^project:"}'

target\_groups:

- labels:

project: swift

targets:

- 'swift-prometheus:9090'

- labels:

project: graphite

targets:

- 'graphite-prometheus:9090'

- labels:

project: monitoring

targets:

- 'monitoring-prometheus2:9090'

- labels:

project: ganglia

targets:

- 'ganglia-prometheus:9090'

- job\_name: 'blackbox-http'

scrape\_interval: 60s

metrics\_path: /probe

params:

module: [http\_2xx] # Look for a HTTP 200 response.

file\_sd\_configs:

- names:

- /etc/prometheus/targets/blackbox.yml

relabel\_configs:

- source\_labels: [\_\_address\_\_]

regex: (.\*)(:80)

target\_label: \_\_param\_target

replacement: http://${1}

- source\_labels: [\_\_address\_\_]

regex: (.\*)(:443)

target\_label: \_\_param\_target

replacement: https://${1}

#- source\_labels: [\_\_address\_\_]

# regex: (payments.wikimedia.org)(:443)

# target\_label: \_\_param\_target

# replacement: https://${1}/index.php?title=Special:GlobalCollectGateway

- source\_labels: [\_\_address\_\_]

regex: (en.(m.)?wikipedia.org)(:443)

target\_label: \_\_param\_target

replacement: https://${1}/wiki/Special:BlankPage

- source\_labels: [\_\_address\_\_]

regex: (ticket.wikimedia.org)(:443)

target\_label: \_\_param\_target

replacement: https://${1}/otrs/index.pl

- source\_labels: [\_\_address\_\_]

regex: (stream.wikimedia.org)(:443)

target\_label: \_\_param\_target

replacement: https://${1}/rc

- source\_labels: [\_\_address\_\_]

regex: (graphite.wikimedia.org)(:443)

target\_label: \_\_param\_target

replacement: https://${1}/render

- source\_labels: [\_\_address\_\_]

regex: (rest.wikimedia.org)(:443)

target\_label: \_\_param\_target

replacement: https://${1}/en.wikipedia.org/v1/?spec

- source\_labels: [\_\_param\_target]

regex: (.\*)

target\_label: instance

replacement: ${1}

- source\_labels: []

regex: .\*

target\_label: \_\_address\_\_

replacement: test-prometheus2:9115 # Blackbox exporter.

- job\_name: 'blackbox-tcp'

scrape\_interval: 60s

metrics\_path: /probe

params:

module: [tcp\_connect]

file\_sd\_configs:

- names:

- /etc/prometheus/targets/blackbox.yml

relabel\_configs:

- source\_labels: [\_\_address\_\_]

regex: (.\*)

target\_label: \_\_param\_target

replacement: ${1}

- source\_labels: [\_\_param\_target]

regex: (.\*)

target\_label: instance

replacement: ${1}

- source\_labels: []

regex: .\*

target\_label: \_\_address\_\_

replacement: test-prometheus2:9115 # Blackbox exporter.

rule\_files:

- /etc/prometheus/rules/hostdown.conf

- /etc/prometheus/rules/ssl.conf

- /etc/prometheus/rules/tcp.conf

- /etc/prometheus/rules/prometheus.conf

**Rules**

ALERT [InstanceDown](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22ALERTS%7Balertname%3D%5C%22InstanceDown%5C%22%7D%22%2C%22tab%22%3A0%7D%5D)

IF [up == 0](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22up%20%3D%3D%200%22%2C%22tab%22%3A0%7D%5D)

FOR 1m

WITH {severity="page"}

SUMMARY "Instance {{$labels.instance}} down"

DESCRIPTION "{{$labels.instance}} of job {{$labels.job}} has been down for more than 1 minute."

RUNBOOK ""  
ALERT [SslCertExpiry60d](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22ALERTS%7Balertname%3D%5C%22SslCertExpiry60d%5C%22%7D%22%2C%22tab%22%3A0%7D%5D)

IF [probe\_ssl\_earliest\_cert\_expiry{job="blackbox-http"} - time() < 5184000](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22probe_ssl_earliest_cert_expiry%7Bjob%3D%5C%22blackbox-http%5C%22%7D%20-%20time%28%29%20%3C%205184000%22%2C%22tab%22%3A0%7D%5D)

FOR 1h

WITH {severity="warn"}

SUMMARY "ssl-cert {{$labels.instance}} about to expire in {{$value | humanizeDuration}}"

DESCRIPTION "XXX"

RUNBOOK ""  
ALERT [SslCertExpiry30d](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22ALERTS%7Balertname%3D%5C%22SslCertExpiry30d%5C%22%7D%22%2C%22tab%22%3A0%7D%5D)

IF [probe\_ssl\_earliest\_cert\_expiry{job="blackbox-http"} - time() < 2592000](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22probe_ssl_earliest_cert_expiry%7Bjob%3D%5C%22blackbox-http%5C%22%7D%20-%20time%28%29%20%3C%202592000%22%2C%22tab%22%3A0%7D%5D)

FOR 1h

WITH {severity="page"}

SUMMARY "ssl-cert {{$labels.instance}} about to expire in {{$value | humanizeDuration}}"

DESCRIPTION "XXX"

RUNBOOK ""  
ALERT [SslHighLatency](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22ALERTS%7Balertname%3D%5C%22SslHighLatency%5C%22%7D%22%2C%22tab%22%3A0%7D%5D)

IF [probe\_duration\_seconds{job="blackbox-http"} > 1](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22probe_duration_seconds%7Bjob%3D%5C%22blackbox-http%5C%22%7D%20%3E%201%22%2C%22tab%22%3A0%7D%5D)

FOR 1m

WITH {severity="page"}

SUMMARY ">1s latency on {{$labels.instance}}: {{$value | humanizeDuration}}"

DESCRIPTION "XXX"

RUNBOOK ""  
ALERT [TcpHighLatency](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22ALERTS%7Balertname%3D%5C%22TcpHighLatency%5C%22%7D%22%2C%22tab%22%3A0%7D%5D)

IF [probe\_duration\_seconds{job="blackbox-tcp"} > 0.5](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22probe_duration_seconds%7Bjob%3D%5C%22blackbox-tcp%5C%22%7D%20%3E%200.5%22%2C%22tab%22%3A0%7D%5D)

FOR 1m

WITH {severity="page"}

SUMMARY ">0.5s latency on {{$labels.instance}}: {{$value | humanizeDuration}}"

DESCRIPTION "XXX"

RUNBOOK ""  
ALERT [PrometheusChunkPersistence](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22ALERTS%7Balertname%3D%5C%22PrometheusChunkPersistence%5C%22%7D%22%2C%22tab%22%3A0%7D%5D)

IF [prometheus\_local\_storage\_chunks\_to\_persist / prometheus\_local\_storage\_max\_chunks\_to\_persist > 0.8](https://prometheus.wmflabs.org/graph#%5B%7B%22expr%22%3A%22prometheus_local_storage_chunks_to_persist%20%2F%20prometheus_local_storage_max_chunks_to_persist%20%3E%200.8%22%2C%22tab%22%3A0%7D%5D)

FOR 30m

WITH {severity="page"}

SUMMARY "prometheus too many chunks to persist, {{$value}}% of maximum"

DESCRIPTION "XXX"

RUNBOOK ""

**Targets**

| **blackbox-http** | | | | |
| --- | --- | --- | --- | --- |
| **Endpoint** | **State** | **Base Labels** | **Last Scrape** | **Error** |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FSpecial%3ABlankPage) **module="http\_2xx"target="https://en.wikipedia.org/wiki/Special:BlankPage"** | UP | **net="wmf"** | 47.580757713s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fupload.wikimedia.org) **module="http\_2xx"** **target="https://upload.wikimedia.org"** | UP | **net="wmf"** | 42.79820084s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fen.m.wikipedia.org%2Fwiki%2FSpecial%3ABlankPage) **module="http\_2xx"target="https://en.m.wikipedia.org/wiki/Special:BlankPage"** | UP | **net="wmf"** | 10.844036832s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fphabricator.wikimedia.org) **module="http\_2xx"target="https://phabricator.wikimedia.org"** | UP | **net="wmf"** | 49.790387ms ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Farchiva.wikimedia.org) **module="http\_2xx"** **target="https://archiva.wikimedia.org"** | UP | **net="wmf"** | 49.294440309s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fblog.wikimedia.org) **module="http\_2xx"** **target="https://blog.wikimedia.org"** | UP | **net="wmf"** | 11.670004536s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fdumps.wikimedia.org) **module="http\_2xx"** **target="https://dumps.wikimedia.org"** | UP | **net="wmf"** | 46.681146001s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fen.planet.wikimedia.org) **module="http\_2xx"target="https://en.planet.wikimedia.org"** | UP | **net="wmf"** | 58.652906381s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fganglia.wikimedia.org) **module="http\_2xx"** **target="https://ganglia.wikimedia.org"** | UP | **net="wmf"** | 5.892918686s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fgerrit.wikimedia.org) **module="http\_2xx"** **target="https://gerrit.wikimedia.org"** | UP | **net="wmf"** | 23.641764148s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fgrafana.wikimedia.org) **module="http\_2xx"** **target="https://grafana.wikimedia.org"** | UP | **net="wmf"** | 50.015171641s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fgraphite.wikimedia.org%2Frender) **module="http\_2xx"target="https://graphite.wikimedia.org/render"** | UP | **net="wmf"** | 45.815034448s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Ficinga.wikimedia.org) **module="http\_2xx"** **target="https://icinga.wikimedia.org"** | UP | **net="wmf"** | 8.529329513s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Flibrenms.wikimedia.org) **module="http\_2xx"target="https://librenms.wikimedia.org"** | UP | **net="wmf"** | 16.443633446s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Flists.wikimedia.org) **module="http\_2xx"** **target="https://lists.wikimedia.org"** | UP | **net="wmf"** | 20.692328839s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fphab.wmfusercontent.org) **module="http\_2xx"target="https://phab.wmfusercontent.org"** | UP | **net="wmf"** | 53.531651856s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fpolicy.wikimedia.org) **module="http\_2xx"** **target="https://policy.wikimedia.org"** | UP | **net="wmf"** | 10.689747084s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Frest.wikimedia.org%2Fen.wikipedia.org%2Fv1%2F%3Fspec) **module="http\_2xx"target="https://rest.wikimedia.org/en.wikipedia.org/v1/?spec"** | UP | **net="wmf"** | 5.773553353s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Frt.wikimedia.org) **module="http\_2xx"** **target="https://rt.wikimedia.org"** | UP | **net="wmf"** | 51.168107197s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fsecure.wikimedia.org) **module="http\_2xx"** **target="https://secure.wikimedia.org"** | UP | **net="wmf"** | 19.993169032s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fstream.wikimedia.org%2Frc) **module="http\_2xx"target="https://stream.wikimedia.org/rc"** | UP | **net="wmf"** | 38.802236685s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Ftendril.wikimedia.org) **module="http\_2xx"** **target="https://tendril.wikimedia.org"** | UP | **net="wmf"** | 23.549555346s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fticket.wikimedia.org%2Fotrs%2Findex.pl) **module="http\_2xx"target="https://ticket.wikimedia.org/otrs/index.pl"** | UP | **net="wmf"** | 9.175760358s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Ftools.wmflabs.org) **module="http\_2xx"** **target="https://tools.wmflabs.org"** | UP | **net="wmf"** | 8.890768794s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fwikitech.wikimedia.org) **module="http\_2xx"target="https://wikitech.wikimedia.org"** | UP | **net="wmf"** | 34.92165344s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fwww.toolserver.org) **module="http\_2xx"** **target="https://www.toolserver.org"** | UP | **net="wmf"** | 59.052972429s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Feventdonations.wikimedia.org) **module="http\_2xx"target="https://eventdonations.wikimedia.org"** | UP | **net="ext"** | 46.922366898s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=http_2xx&target=https%3A%2F%2Fbenefactorevents.wikimedia.org) **module="http\_2xx"target="https://benefactorevents.wikimedia.org"** | UP | **net="ext"** | 23.759948029s ago |  |
| **blackbox-tcp** | | | | |
| **Endpoint** | **State** | **Base Labels** | **Last Scrape** | **Error** |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=en.wikipedia.org%3A443) **module="tcp\_connect"** **target="en.wikipedia.org:443"** | UP | **net="wmf"** | 52.320779351s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=upload.wikimedia.org%3A443) **module="tcp\_connect"target="upload.wikimedia.org:443"** | UP | **net="wmf"** | 42.858579915s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=en.m.wikipedia.org%3A443) **module="tcp\_connect"** **target="en.m.wikipedia.org:443"** | UP | **net="wmf"** | 30.271688855s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=phabricator.wikimedia.org%3A443) **module="tcp\_connect"target="phabricator.wikimedia.org:443"** | UP | **net="wmf"** | 12.761769342s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=archiva.wikimedia.org%3A443) **module="tcp\_connect"target="archiva.wikimedia.org:443"** | UP | **net="wmf"** | 46.923370866s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=blog.wikimedia.org%3A443) **module="tcp\_connect"** **target="blog.wikimedia.org:443"** | UP | **net="wmf"** | 46.6876503s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=dumps.wikimedia.org%3A443) **module="tcp\_connect"target="dumps.wikimedia.org:443"** | UP | **net="wmf"** | 19.358093888s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=en.planet.wikimedia.org%3A443) **module="tcp\_connect"target="en.planet.wikimedia.org:443"** | UP | **net="wmf"** | 51.396108309s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=ganglia.wikimedia.org%3A443) **module="tcp\_connect"target="ganglia.wikimedia.org:443"** | UP | **net="wmf"** | 12.593030473s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=gerrit.wikimedia.org%3A443) **module="tcp\_connect"** **target="gerrit.wikimedia.org:443"** | UP | **net="wmf"** | 22.709513922s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=grafana.wikimedia.org%3A443) **module="tcp\_connect"target="grafana.wikimedia.org:443"** | UP | **net="wmf"** | 33.084279961s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=graphite.wikimedia.org%3A443) **module="tcp\_connect"target="graphite.wikimedia.org:443"** | UP | **net="wmf"** | 2.812335284s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=icinga.wikimedia.org%3A443) **module="tcp\_connect"** **target="icinga.wikimedia.org:443"** | UP | **net="wmf"** | 55.015505565s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=librenms.wikimedia.org%3A443) **module="tcp\_connect"target="librenms.wikimedia.org:443"** | UP | **net="wmf"** | 28.075070735s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=lists.wikimedia.org%3A443) **module="tcp\_connect"** **target="lists.wikimedia.org:443"** | UP | **net="wmf"** | 5.999599142s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=phab.wmfusercontent.org%3A443) **module="tcp\_connect"target="phab.wmfusercontent.org:443"** | UP | **net="wmf"** | 59.742495669s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=policy.wikimedia.org%3A443) **module="tcp\_connect"** **target="policy.wikimedia.org:443"** | UP | **net="wmf"** | 52.121984624s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=rest.wikimedia.org%3A443) **module="tcp\_connect"** **target="rest.wikimedia.org:443"** | UP | **net="wmf"** | 968.778726ms ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=rt.wikimedia.org%3A443) **module="tcp\_connect"** **target="rt.wikimedia.org:443"** | UP | **net="wmf"** | 22.995978967s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=secure.wikimedia.org%3A443) **module="tcp\_connect"target="secure.wikimedia.org:443"** | UP | **net="wmf"** | 46.422315422s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=stream.wikimedia.org%3A443) **module="tcp\_connect"target="stream.wikimedia.org:443"** | UP | **net="wmf"** | 54.115604508s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=tendril.wikimedia.org%3A443) **module="tcp\_connect"** **target="tendril.wikimedia.org:443"** | UP | **net="wmf"** | 32.051817426s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=ticket.wikimedia.org%3A443) **module="tcp\_connect"** **target="ticket.wikimedia.org:443"** | UP | **net="wmf"** | 28.829501297s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=tools.wmflabs.org%3A443) **module="tcp\_connect"** **target="tools.wmflabs.org:443"** | UP | **net="wmf"** | 47.807624556s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=wikitech.wikimedia.org%3A443) **module="tcp\_connect"target="wikitech.wikimedia.org:443"** | UP | **net="wmf"** | 39.128565781s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=www.toolserver.org%3A443) **module="tcp\_connect"** **target="www.toolserver.org:443"** | UP | **net="wmf"** | 31.869072267s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=eventdonations.wikimedia.org%3A443) **module="tcp\_connect"target="eventdonations.wikimedia.org:443"** | UP | **net="ext"** | 36.384377034s ago |  |
| [http://test-prometheus2:9115/probe](http://test-prometheus2:9115/probe?module=tcp_connect&target=benefactorevents.wikimedia.org%3A443) **module="tcp\_connect"target="benefactorevents.wikimedia.org:443"** | UP | **net="ext"** | 47.537601202s ago |  |
| **federate** | | | | |
| **Endpoint** | **State** | **Base Labels** | **Last Scrape** | **Error** |
| [http://monitoring-prometheus2:9090/federate](http://monitoring-prometheus2:9090/federate?match%5B%5D=%7B__name__%3D~%22%5Eproject%3A%22%7D) **match[]="{\_\_name\_\_=~"^project:"}"** | DOWN | **project="monitoring"** | 17.201984223s ago | Get http://monitoring-prometheus2:9090/federate?match%5B%5D=%7B\_\_name\_\_%3D~%22%5Eproject%3A%22%7D: dial tcp: lookup monitoring-prometheus2 on 208.80.155.118:53: no such host |
| [http://ganglia-prometheus:9090/federate](http://ganglia-prometheus:9090/federate?match%5B%5D=%7B__name__%3D~%22%5Eproject%3A%22%7D) **match[]="{\_\_name\_\_=~"^project:"}"** | DOWN | **project="ganglia"** | 7.220927087s ago | Get http://ganglia-prometheus:9090/federate?match%5B%5D=%7B\_\_name\_\_%3D~%22%5Eproject%3A%22%7D: dial tcp: lookup ganglia-prometheus on 208.80.155.118:53: no such host |
| [http://graphite-prometheus:9090/federate](http://graphite-prometheus:9090/federate?match%5B%5D=%7B__name__%3D~%22%5Eproject%3A%22%7D) **match[]="{\_\_name\_\_=~"^project:"}"** | DOWN | **project="graphite"** | 32.683902669s ago | Get http://graphite-prometheus:9090/federate?match%5B%5D=%7B\_\_name\_\_%3D~%22%5Eproject%3A%22%7D: dial tcp 10.68.16.134:9090: getsockopt: connection refused |
| [http://swift-prometheus:9090/federate](http://swift-prometheus:9090/federate?match%5B%5D=%7B__name__%3D~%22%5Eproject%3A%22%7D) **match[]="{\_\_name\_\_=~"^project:"}"** | DOWN | **project="swift"** | 45.430029317s ago | Get http://swift-prometheus:9090/federate?match%5B%5D=%7B\_\_name\_\_%3D~%22%5Eproject%3A%22%7D: dial tcp 10.68.16.67:9090: getsockopt: connection refused |
| **mysqld\_exporter** | | | | |
| **Endpoint** | **State** | **Base Labels** | **Last Scrape** | **Error** |
| <http://test-prometheus2:9501/metrics> | UP | **none** | 582.429638ms ago |  |
| <http://test-prometheus2:9502/metrics> | UP | **none** | 55.125427537s ago |  |
| <http://test-prometheus2:9503/metrics> | UP | **none** | 46.459990883s ago |  |
| <http://test-prometheus2:9504/metrics> | UP | **none** | 33.60717974s ago |  |
| <http://test-prometheus2:9505/metrics> | UP | **none** | 58.701601722s ago |  |
| <http://test-prometheus2:9506/metrics> | DOWN | **none** | 51.657405878s ago | Get http://test-prometheus2:9506/metrics: read tcp 10.68.18.221:51652->10.68.23.199:9506: i/o timeout |
| <http://test-prometheus2:9507/metrics> | DOWN | **none** | 15.702944515s ago | Get http://test-prometheus2:9507/metrics: read tcp 10.68.18.221:33938->10.68.23.199:9507: i/o timeout |
| **node** | | | | |
| **Endpoint** | **State** | **Base Labels** | **Last Scrape** | **Error** |
| [http://localhost:9100/metrics](http://test-prometheus3:9100/metrics) | UP | **none** | 23.312294376s ago |  |
| **prometheus** | | | | |
| **Endpoint** | **State** | **Base Labels** | **Last Scrape** | **Error** |
| [http://localhost:9090/metrics](http://test-prometheus3:9090/metrics) | UP | **none** | 41.679393879s ago |  |

**Startup Flags**

|  |  |
| --- | --- |
| **alertmanager.http-deadline** | 10s |
| **alertmanager.notification-queue-capacity** | 100 |
| **alertmanager.url** |  |
| **config.file** | /etc/prometheus/prometheus.yml |
| **log.format** |  |
| **log.level** | info |
| **query.max-concurrency** | 20 |
| **query.staleness-delta** | 5m0s |
| **query.timeout** | 2m0s |
| **storage.local.checkpoint-dirty-series-limit** | 5000 |
| **storage.local.checkpoint-interval** | 5m0s |
| **storage.local.chunk-encoding-version** | 1 |
| **storage.local.dirty** | false |
| **storage.local.index-cache-size.fingerprint-to-metric** | 10485760 |
| **storage.local.index-cache-size.fingerprint-to-timerange** | 5242880 |
| **storage.local.index-cache-size.label-name-to-label-values** | 10485760 |
| **storage.local.index-cache-size.label-pair-to-fingerprints** | 20971520 |
| **storage.local.max-chunks-to-persist** | 1048576 |
| **storage.local.memory-chunks** | 1048576 |
| **storage.local.path** | /srv/prometheus/metrics |
| **storage.local.pedantic-checks** | false |
| **storage.local.retention** | 4320h0m0s |
| **storage.local.series-file-shrink-ratio** | 0.1 |
| **storage.local.series-sync-strategy** | adaptive |
| **storage.remote.graphite-address** |  |
| **storage.remote.graphite-prefix** |  |
| **storage.remote.graphite-transport** | tcp |
| **storage.remote.influxdb-url** |  |
| **storage.remote.influxdb.database** | prometheus |
| **storage.remote.influxdb.retention-policy** | default |
| **storage.remote.influxdb.username** |  |
| **storage.remote.opentsdb-url** |  |
| **storage.remote.timeout** | 30s |
| **version** | false |
| **web.console.libraries** | /etc/prometheus/console\_libraries |
| **web.console.templates** | /etc/prometheus/consoles |
| **web.enable-remote-shutdown** | false |
| **web.external-url** | http://test-prometheus3:9090/ |
| **web.listen-address** | :9090 |
| **web.local-assets** | /usr/share/prometheus/web/ |
| **web.telemetry-path** | /metrics |
| **web.user-assets** |  |

# 2@Scaling and Federating Prometheus | Robust Perception

https://www.robustperception.io/scaling-and-federating-prometheus/

**A single** [**Prometheus**](https://prometheus.io/) **server can easily handle millions of time series. That’s enough for a thousand servers with a thousand time series each scraped every 10 seconds. As your systems scale beyond that, Prometheus can scale too.**

## Initial Deployment

**When starting out it’s best to keep things simple. A single Prometheus server per datacenter or similar failure domain (e.g. EC2 region) can typically handle a thousand servers, so should last you for a good while. Running one per datacenter avoids having the internet or WAN links on the critical path of your monitoring.**

**If you’ve more than one datacenter, you may wish to have global aggregates of some time series. This is done with a “global Prometheus” server, which federates from the datacenter Prometheus servers.**

**- scrape\_config:**

**- job\_name: dc\_prometheus**

**honor\_labels: true**

**metrics\_path: /federate**

**params:**

**match[]:**

**- '{\_\_name\_\_=~"^job:.\*"}' # Request all job-level time series**

**static\_configs:**

**- targets:**

**- dc1-prometheus:9090**

**- dc2-prometheus:9090**

**It’s suggested to run two global** [**Prometheis**](http://prometheus.io/docs/introduction/faq/#what-is-the-plural-of-prometheus) **in different datacenters. This keeps your global monitoring working even if one datacenter has an outage.**

## Splitting By Use

**As you grow you’ll come to a point where a single Prometheus isn’t quite enough. The next step is to run multiple Prometheus servers per datacenter. Each one will own monitoring for some team or slice of the stack. A first pass may result in fronted, backend and machines (node exporter) for example.**

**As you continue to grow, this process can be repeated. MySQL and Cassandra monitoring may end up with their own Prometheis, or each Cassandra cluster may have a Prometheus server dedicated to it.**

**You may also wish to start splitting by use before there are performance issues, as teams may not want to share Prometheis or to improve isolation.**

## Horizontal Sharding

**When you can’t subdivide Prometheus servers any longer, the final step in scaling is to scale out. This usually requires that a single job has thousands of instances, a scale that most users never reach. This is more complex setup and is much more involved to manage than a normal Prometheus deployment, so should be avoided for as long as you can.**

**The architecture is to have multiple slave Prometheis, each scraping a subset of the targets and aggregating them up within the slave. A master federates the aggregates produced by the slaves, and then the master aggregates them up to the job level.**

**On the slaves you can use a hash of the address to select only some targets to scrape:**

**global:**

**external\_labels:**

**slave: 1 # This is the 2nd slave. This prevents clashes between slaves.**

**scrape\_configs:**

**- job\_name: some\_job**

**# Add usual service discovery here, such as static\_configs**

**relabel\_configs:**

**- source\_labels: [\_\_address\_\_]**

**modulus: 4 # 4 slaves**

**target\_label: \_\_tmp\_hash**

**action: hashmod**

**- source\_labels: [\_\_tmp\_hash]**

**regex: ^1$ # This is the 2nd slave**

**action: keep**

**And the master federates from the slaves:**

**- scrape\_config:**

**- job\_name: slaves**

**honor\_labels: true**

**metrics\_path: /federate**

**params:**

**match[]:**

**- '{\_\_name\_\_=~"^slave:.\*"}' # Request all slave-level time series**

**static\_configs:**

**- targets:**

**- slave0:9090**

**- slave1:9090**

**- slave3:9090**

**- slave4:9090**

**Information for dashboards is usually taken from the master. If you wanted to drill down to a particular target, you’d do so via its slave.**

**Have questions about scaling Prometheus?**[***Contact us***](mailto:prometheus@robustperception.io)**.**

[**federation**](https://www.robustperception.io/tag/federation/)**,** [**prometheus**](https://www.robustperception.io/tag/prometheus/)**,** [**scaling**](https://www.robustperception.io/tag/scaling/)

# Federation, what is it good for? | Robust Perception

https://www.robustperception.io/federation-what-is-it-good-for/

**There’s various ways Prometheus** [**federation**](https://prometheus.io/docs/operating/federation/) **can be used. To ensure your monitoring is scalable and reliable, let’s look at how to best use it.**

**There are two general cases for which federation is well suited. We’ll look at each in turn.**

**Prometheus Hierarchy**

**As** [**previously**](https://www.robustperception.io/monitoring-without-consensus/)[**discussed**](https://www.robustperception.io/scaling-and-federating-prometheus/)**, Prometheus is intended to have at least one instance per datacenter usually also with a global Prometheus for global graphing or alerting. Federation allows for pulling aggregates up the hierarchy. Usually there’s only two levels to the hierarchy, but three or four aren’t completely unheard of.**

**Let’s for example say you wanted to aggregate up the total amount of memory on all your machines at a global level.**

**First there’s part of the config file for each datacenter Prometheus:**

**global:**

**external\_labels:**

**datacenter: eu-west-1**

**rule\_files:**

**- node.rules**

**scrape\_configs:**

**etc.**

**Note the external labels, every Prometheus in an organization should have unique external labels.**

**Then in the node.rules rules file we aggregate up to the dataenter level:**

**job:node\_memory\_MemTotal:sum = sum without(instance)(node\_memory\_MemTotal{job="node"})**

**As only the job label will be left on the time series it gets a job:prefix, and we’re summing so it’s a :sum suffix.**

**In the global Prometheus config we pull in this timeseries:**

**global:**

**external\_labels:**

**datacenter: global # In a HA setup, this would be global1 or global2**

**scrape\_configs:**

**- job\_name: datacenter\_federation**

**honor\_labels: true**

**metrics\_path: /federate**

**params:**

**match[]:**

**- '{\_\_name\_\_=~"^job:.\*"}'**

**static\_configs:**

**- targets:**

**- eu-west-1-prometheus:9090**

**etc.**

**The match[] here requests all job-level time series, so by following this naming convention you don’t have to adjust the config every time there’s a new aggregating rule.**

**Now you can use the expression sum without(datacenter)(job:node\_memory\_MemTotal:sum) to get the memory available across your entire global fleet!**

**Cross-Service Aggregates**

**The second general use case for federation is when you want to pull a handful of time series from a sibling Prometheus. For example as a sanity check, you may wish to check that your HAProxy Prometheus and your frontend Prometheus both are seeing about the same number of requests getting through.**

**The config would look similar to the above, though you’d ask for each time series you want explicitly by name. If the Prometheus is run by another team, don’t forget to ask them first as reaching into someone else’s Prometheus can be like reaching into the undocumented internals of a library in terms of maintenance and stability.**

**Using federation like this is usually only done for alerting, as Grafana supports multiple data sources (e.g. Prometheus servers) being used in one graph.**

**Where Federation Doesn’t Fit**

**In both the above cases, federation is being used to pull in a limited and aggregated set of time series from another Prometheus. That Prometheus is otherwise continuing to do it’s duty, firing alerts and serving graph queries.**

**Where federation isn’t suitable is if you use it to pull large swatches of time series – or even all time series – from another Prometheus, and then do alerting and graphing only from there. There’s three broad reasons for this.**

**The first is performance and scaling. As the limiting factor of Prometheus performance is how much can be handled by a single machine, routing all your data to one global Prometheus limits your monitoring to what one machine can handle. By instead pulling only aggregated time series you’re only limited to what one datacenter Prometheus can handle, allowing you to add new add datacenters without having to worry about scaling the global Prometheus. The federation request itself can also be heavy to serve for the receiving Prometheus.**

**The second is reliability. If the data you need to do alerting is moved from one Prometheus to another then you’ve added an additional point of failure. This is particularly risky when WAN links such as the internet are involved. As far as is possible, you should try and push alerting as deep down the federation hierarchy as possible. For example an alert about a target being down should be setup on the Prometheus scraping that target, not a global Prometheus which could be several steps removed.**

**The third is correctness. Due to how it works, federation will pull in data some time after it was scraped and may also miss some data due to races. While the artifacts and oddness this causes in a global Prometheus are generally tolerable in a setup where your datacenter Prometheus servers are the primary workhorses for graphing and alerting, this is much more likely to cause issues when you’re federating everything.**

**These issues don’t just apply between datacenter and global Prometheus servers. Some have attempted to use a Prometheus as a type of proxy server, using federation to pull all data out of the scraping Prometheus into another Prometheus where all the real work is done. There are issues with this, such as the above mentioned correctness problems. If you find yourself in this situation either make the scraping Prometheus handle alerting and graphing, or do the scrapes via an actual proxy server using proxy\_url.**

**Unsure how to architect your Prometheus federation setup?**[***Contact us***](mailto:prometheus@robustperception.io)**.**

[**best practices**](https://www.robustperception.io/tag/best-practices/)**,** [**federation**](https://www.robustperception.io/tag/federation/)**,** [**prometheus**](https://www.robustperception.io/tag/prometheus/)**,** [**promql**](https://www.robustperception.io/tag/promql/)**,** [**scaling**](https://www.robustperception.io/tag/scaling/)

# Recommended settings for running a machine with 2GB ram

· Issue #1836 · prometheus/prometheus

https://github.com/prometheus/prometheus/issues/1836

Hi,

We are using a machine with 2CPUs/2GB RAM (ubuntu 14.04) where prometheus and alertmanager are running as Docker containers.

Currently the prometheus application runs OOM at least once day. The Prometheus server instance scrapes < 30 targets (every 15s) to collect the node machine metrics exported by node-exporter on the targets.

Which settings would you recommend to run this kind of setup? Can the settings be tuned to get a stable Prometheus or is the machine simply to small to handle this scenario?

Thx,  
Sven

Current cmd line flags for docker container

-config.file=/etc/prometheus/prometheus.yml

-storage.local.path=/prometheus

-web.console.libraries=/etc/prometheus/console\_libraries

-web.console.templates=/etc/prometheus/consoles

-web.external-url=https://xyz

-alertmanager.url=http://alertmanager:9093

Stackstrace

fatal error: runtime: out of memory

runtime stack:

runtime.throw(0xfca050, 0x16)

/usr/local/go/src/runtime/panic.go:547 +0x90

runtime.sysMap(0xc8864e0000, 0x100000, 0x0, 0x1528e98)

/usr/local/go/src/runtime/mem\_linux.go:206 +0x9b

runtime.(\*mheap).sysAlloc(0x150e380, 0x100000, 0x1061ea670)

/usr/local/go/src/runtime/malloc.go:429 +0x191

runtime.(\*mheap).grow(0x150e380, 0x28, 0x0)

/usr/local/go/src/runtime/mheap.go:651 +0x63

runtime.(\*mheap).allocSpanLocked(0x150e380, 0x28, 0xc81ccd9400)

/usr/local/go/src/runtime/mheap.go:553 +0x4f6

runtime.(\*mheap).alloc\_m(0x150e380, 0x28, 0x100000000, 0x150f8f0)

/usr/local/go/src/runtime/mheap.go:437 +0x119

runtime.(\*mheap).alloc.func1()

Runtime Information

Uptime: 2016-07-20 10:44:09.210337115 +0000 UTC

Build Information

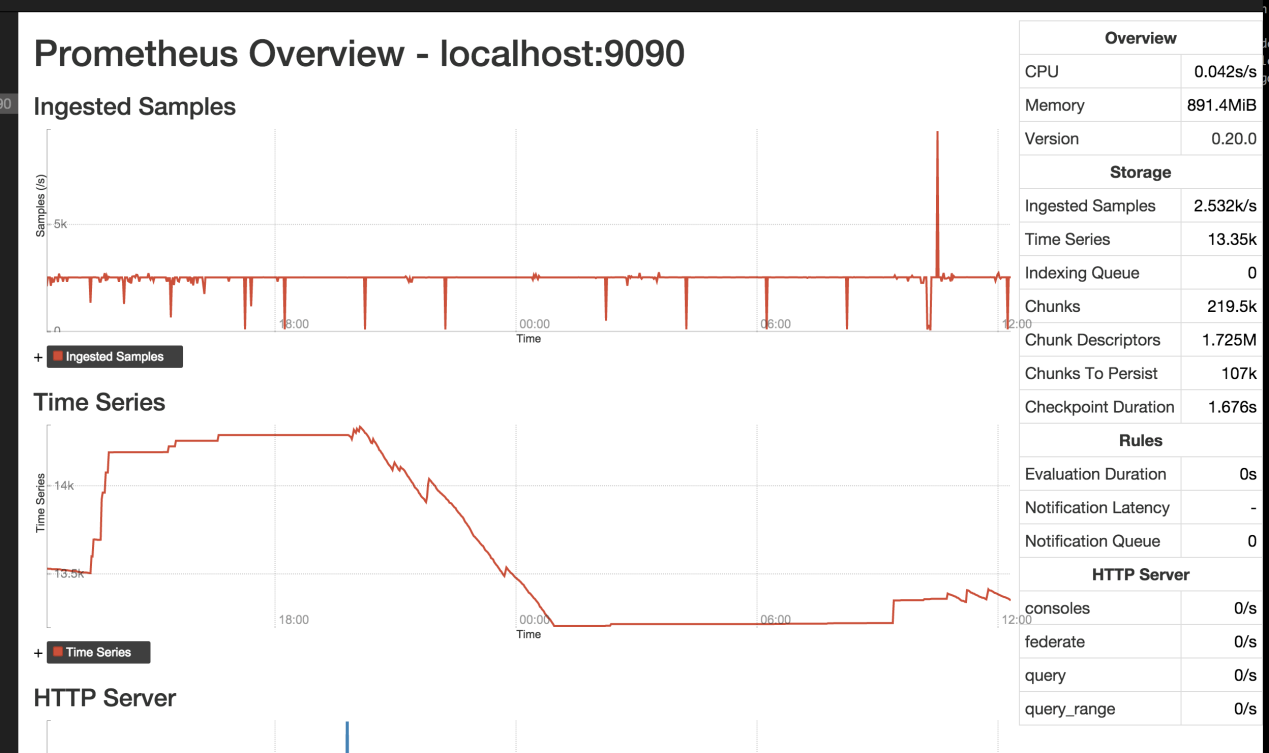
Version: 0.20.0  
Revision: [aeab25c](https://github.com/prometheus/prometheus/commit/aeab25c35daeb06db9d7e7eaa5882b72a1721c48)  
Branch: master  
BuildUser: root@77050118f904  
BuildDate: 20160616-08:38:14  
GoVersion: go1.6.2

See <https://prometheus.io/docs/operating/storage/>

I'd start with -storage.local.memory-chunks=500000. As long as you don't have more than ~200k time series, you should be fine with that (check metric prometheus\_local\_storage\_memory\_series)

Thx for the input!

Here is s screenshot from the prometheus overview console;

[](https://cloud.githubusercontent.com/assets/131689/16986103/2b6fe7da-4e85-11e6-9c9d-bfcf2df3e600.png)

graph for prometheus\_local\_storage\_memory\_series (last 2h), current value: 13341

[](https://cloud.githubusercontent.com/assets/131689/16986227/99080552-4e85-11e6-9033-fbd7b138deda.png)

Yeah, light load. You are only running out of memory because Prometheus tries to utilize ~4GiB by default. With the setting above, you should be all good.

With the settings, I expect Prometheus to fully utilize your memory.  
In general, you want to use all your memory, but not more. ;)

If you still run OOM, you can tweak the flag even lower. More context and tweaks are described at<https://prometheus.io/docs/operating/storage/>

Ideally, Prometheus would auto-tune memory usage, but that's a non-trivial problem, see [#455](https://github.com/prometheus/prometheus/issues/455) .

In the meantime i was trying storage.local.memory-chunks with value "400000" but still i'm seeing Prometheus running into OOMs

Try lower values until you don't run into OOMs anymore.

You can also perform heap profiling to find out where the memory is gone, start with go tool pprof http://localhost:6060/debug/pprof/heap - details as <https://blog.golang.org/profiling-go-programs> ,<https://golang.org/pkg/net/http/pprof/>

# 3@Prometheus的架构及持久化 - davygeek - 博客园

http://www.cnblogs.com/davygeek/p/6668706.html

Prometheus是什么

Prometheus是一个开源的系统监控和报警工具，特点是

* 多维数据模型（时序列数据由metric名和一组key/value组成）
* 在多维度上灵活的查询语言(PromQl)
* 不依赖分布式存储，单主节点工作.
* 通过基于HTTP的pull方式采集时序数据
* 可以通过push gateway进行时序列数据推送(pushing)
* 可以通过服务发现或者静态配置去获取要采集的目标服务器
* 多种可视化图表及仪表盘支持

**pull方式**

Prometheus采集数据是用的pull也就是拉模型,通过HTTP协议去采集指标，只要应用系统能够提供HTTP接口就可以接入监控系统，相比于私有协议或二进制协议来说开发、简单。

**push方式**

对于定时任务这种短周期的指标采集，如果采用pull模式，可能造成任务结束了，Prometheus还没有来得及采集，这个时候可以使用加一个中转层，客户端推数据到Push Gateway缓存一下，由Prometheus从push gateway pull指标过来。(需要额外搭建Push Gateway，同时需要新增job去从gateway采数据)

**组成及架构**

* Prometheus server 主要负责数据采集和存储，提供PromQL查询语言的支持
* 客户端sdk 官方提供的客户端类库有go、java、scala、python、ruby，其他还有很多第三方开发的类库，支持nodejs、php、erlang等
* Push Gateway 支持临时性Job主动推送指标的中间网关
* PromDash 使用rails开发的dashboard，用于可视化指标数据
* exporters 支持其他数据源的指标导入到Prometheus，支持数据库、硬件、消息中间件、存储系统、http服务器、jmx等
* alertmanager 实验性组件、用来进行报警
* prometheus\_cli 命令行工具
* 其他辅助性工具

架构图如下：

**默认配置**

docker exec -it a9bd827a1d18 less /etc/prometheus/prometheus.yml

得到

# my global config

global:

scrape\_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.

evaluation\_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.

# scrape\_timeout is set to the global default (10s).

# Attach these labels to any time series or alerts when communicating with

# external systems (federation, remote storage, Alertmanager).

external\_labels:

monitor: 'codelab-monitor'

# Load rules once and periodically evaluate them according to the global 'evaluation\_interval'.

rule\_files:

# - "first.rules"

# - "second.rules"

# A scrape configuration containing exactly one endpoint to scrape:

# Here it's Prometheus itself.

scrape\_configs:

# The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.

- job\_name: 'prometheus'

# metrics\_path defaults to '/metrics'

# scheme defaults to 'http'.

static\_configs:

- targets: ['localhost:9090']

* scrape\_interval 这里是指每隔15秒钟去抓取数据(这里)
* evaluation\_interval 指的是计算rule的间隔

**Push Gateway**

pushgateway有单独的镜像

docker pull prom/pushgateway

对于喜欢用push模式的应用来说，可以专门搭建一个push gateway，来适配一下。

**storage**

prometheus使用了G家的LevelDB来做索引(PromSQL重度依赖LevelDB)，对于大量的采样数据有自己的存储层，Prometheus为每个时序数据创建一个本地文件，以1024byte大小的chunk来组织。

**磁盘文件**

Prometheus在storage.local.path指定的路径存储文件，默认为./data。关于chunk编码有三种

* type 0

第一代的编码格式，simple delta encoding

* type 1

目前默认的编码格式，double-delta encoding

* type 2

variable bit-width encoding，facebook的时间序列数据库Beringei采用的编码方式

**内存使用**

prometheus在内存里保存了最近使用的chunks，具体chunks的最大个数可以通过storage.local.memory-chunks来设定，默认值为1048576，即1048576个chunk，大小为1G。 除了采用的数据，prometheus还需要对数据进行各种运算，因此整体内存开销肯定会比配置的local.memory-chunks大小要来的大，因此官方建议要预留3倍的local.memory-chunks的内存大小。

As a rule of thumb, you should have at least three times more RAM available than needed by the memory chunks alone

可以通过server的metrics去查看prometheus\_local\_storage\_memory\_chunks以及process\_resident\_memory\_byte两个指标值。

prometheus\_local\_storage\_memory\_chunks

The current number of chunks in memory, excluding cloned chunks 目前内存中暴露的chunks的个数

process\_resident\_memory\_byte

Resident memory size in bytes 驻存在内存的数据大小

prometheus\_local\_storage\_persistence\_urgency\_score

介于0-1之间，当该值小于等于0.7时，prometheus离开rushed模式。 当大于0.8的时候，进入rushed模式

prometheus\_local\_storage\_rushed\_mode

1表示进入了rushed mode，0表示没有。进入了rushed模式的话，prometheus会利用storage.local.series-sync-strategy以及storage.local.checkpoint-interval的配置加速chunks的持久化。

**storage参数**

docker run -p 9090:9090 \

-v /tmp/prometheus-data:/prometheus-data \

prom/prometheus \

-storage.local.retention 168h0m0s \

-storage.local.max-chunks-to-persist 3024288 \

-storage.local.memory-chunks=50502740 \

-storage.local.num-fingerprint-mutexes=300960

**storage.local.memory-chunks**

设定prometheus内存中保留的chunks的最大个数，默认为1048576，即为1G大小

**storage.local.retention**

用来配置采用数据存储的时间，168h0m0s即为24\*7小时，即1周

**storage.local.series-file-shrink-ratio**

用来控制序列文件rewrite的时机，默认是在10%的chunks被移除的时候进行rewrite，如果磁盘空间够大，不想频繁rewrite，可以提升该值，比如0.3，即30%的chunks被移除的时候才触发rewrite。

**storage.local.max-chunks-to-persist**

该参数控制等待写入磁盘的chunks的最大个数，如果超过这个数，Prometheus会限制采样的速率，直到这个数降到指定阈值的95%。建议这个值设定为storage.local.memory-chunks的50%。Prometheus会尽力加速存储速度，以避免限流这种情况的发送。

**storage.local.num-fingerprint-mutexes**

当prometheus server端在进行checkpoint操作或者处理开销较大的查询的时候，采集指标的操作会有短暂的停顿，这是因为prometheus给时间序列分配的mutexes可能不够用，可以通过这个指标来增大预分配的mutexes，有时候可以设置到上万个。

**storage.local.series-sync-strategy**

控制写入数据之后，何时同步到磁盘，有'never', 'always', 'adaptive'. 同步操作可以降低因为操作系统崩溃带来数据丢失，但是会降低写入数据的性能。 默认为adaptive的策略，即不会写完数据就立刻同步磁盘，会利用操作系统的page cache来批量同步。

**storage.local.checkpoint-interval**

进行checkpoint的时间间隔，即对尚未写入到磁盘的内存chunks执行checkpoint操作。

**doc**

* [prometheus-configuration](https://prometheus.io/docs/operating/configuration/)
* [prometheus-storage](https://prometheus.io/docs/operating/storage/)
* [promdash](https://prometheus.io/docs/visualization/promdash/)
* [config.go](https://github.com/prometheus/prometheus/blob/master/cmd/prometheus/config.go)

# Semaphore 和 Mutex - carprog - 博客园

http://www.cnblogs.com/shangdawei/p/3520102.html

mutex和semaphore有什么区别呢？

**mutex是用作互斥的，而semaphore是用作同步的。**

也就是说，mutex的初始化一定是为1，而semaphore可以是任意的数，

所以如果使用mutex，那第一个进入临界区的进程一定可以执行，而其他的进程必须等待。

而semaphore则不一定，如果一开始初始化为0，则所有进程都必须等待。

同时mutex和semaphore还有一个区别是，获得mutex的进程必须亲自释放它，而semaphore则可以一个进程获得，另一个进程释放。

<http://www.cppblog.com/martin/archive/2009/03/18/hello.html>

**mutex与semaphore的区别**

＂互斥(mutext)和旗语(semaphore)之间有什么不同？＂这样的问题简短而有力，但要回答却相当困难．  
即使有经验的实时操作系统(RTOS)用户在区别如何正确使用mutex和semaphore时也存在着困难．

但这一点很不幸而且很危险，因为无任这两种原生RTOS中的哪一种被错误使用，都会导致嵌入式系统出现意想不到的错误，特别是这些系统为有关生命安全的产品时.  
有关mutex和semaphore的荒诞说法是它们是相似的，甚至是可以互换的．

正确的事实是尽管mutex和semaphore在它们的执行上有相似之处，但是我们还是应该在使用它们时加以区别对待．

最普遍（但也是不正确）的答案是：mutex和semphore非常相似，它们只有一个区别，那就是semaphores的计数可以超过1.

差不多所有的工程师都能正确的理解：mutex是一个二进制标志，可以通过它来确保执行流在代码关键区(critical section of code)互相排斥,从而对共享资源加一保护．

但当他们被要求进一步回答如何使用＂计算方法semaphore"的方式时，大部分工程师的回答就如同教科书书一般的刻板---semaphore用于保护多重同类资源．

通过类比办法，我们很容易解释为什么"多重资源＂场景是有缺陷的.如果你认为一个mutex是由操作系统拥有的关键值的话，我们可以很容易地将个别的mutex比喻是城市咖啡店中一间浴室的钥匙．

如果你想使用浴室，却找不到钥匙，你就必须在一个队列中等候．同样地，mutex则协串行化多项任务，以取得全域资源的共享，并且为等待队列中的任务分配一个静候其循序渐进的位置．

但这种简单的资源保护协议并不使用于两间相同浴室的情况．如果把一个semaphore概括为一个mutex，使其能保护两个或更多相同的资源，

那么在我们的比喻中，它就象是放着两把相同钥匙的蓝子，你可以用任何一把打开任何一扇浴室的门．

因此，semaphore本身并不能解决多个相同资源的问题．咖啡店中的客人可能只知道有一把钥匙，但并不知道哪间浴室可用．

如果你试图以此方式使用semaphore，你将会发现需要更多的状态信息---它们通常是由不同的mutex所保护的共享资源．

正确使用semaphore是为了使信号从一项任务传至另一项任务．

mutex意味着取得与释放，使用受保护共享资源的每一次任务都是以这样的顺序进行．

相比之下，使用semaphore的任务通常不是发送信号，就是进入等待状态，不可能同时发生．

例如，任务1可能包含程序代码，当按下＂电源＂(power)按钮时，即可提出(如发送信号或增量)一个特别的semaphore;

任务2则依据相同的semaphore而用于唤醒显示器. 在这种情况下，其中一项任务是信号的生产者，另一项任务是信号的消费者．  
  
用一个例子来做总结，首先展示如何使用mutex：

**/\* Task 1 \*/  
mutexWait(mutex\_mens\_room);  
// Safely use shared resource  
mutexRelease(mutex\_mens\_room);  
  
/\* Task 2 \*/  
mutexWait(mutex\_mens\_room);  
// Safely use shared resource  
mutexRelease(mutex\_mens\_room);  
  
相应地，你总是采用下列方法使用semaphore:  
/\* Task 1 - Producer \*/  
semPost(sem\_power\_btn); // Send the signal  
  
/\* Task 2 - Consumer \*/  
semPend(sem\_power\_btn); // Wait for signal**重要的是，semaphores可以被interrupt service routine(ISR)中断服务程序用来向task发送信号．

发送一个semaphore是一个非阻塞的RTOS行为，并且ISR安全．

因为这种技术排除了在task级别的为了是中断不使能而引起的错误的可能性，

从ISR中发出信号是一种使嵌入式软件更加可靠的设计方式.

<http://ousysrobin.blog.hexun.com/57137773_d.html>

理解Semaphore和Mutex

Mutex是一把钥匙，一个人拿了就可进入一个房间，出来的时候把钥匙交给队列的第一个。

一般的用法是用于串行化对critical section代码的访问，保证这段代码不会被并行的运行。

Semaphore是一件可以容纳N人的房间，如果人不满就可以进去，如果人满了，就要等待有人出来。

对于N=1的情况，称为binary semaphore。一般的用法是，用于限制对于某一资源的同时访问。

**Binary semaphore与Mutex的差异：**

在有的系统中Binary semaphore与Mutex是没有差异的。在有的系统上，主要的差异是mutex一定要由获得锁的进程来释放。  
而semaphore可以由其它进程释 放（这时的semaphore实际就是个原子的变量，大家可以加或减），因此semaphore可以用于进程间同步。  
Semaphore的同步功能是所有 系统都支持的，而Mutex能否由其他进程释放则未定，  
因此建议mutex只用于保护critical section。而semaphore则用于同步或者保护某变量。

关于semaphore和mutex的区别，网上有著名的厕所理论（<http://koti.mbnet.fi/niclasw/MutexSemaphore.html>）：

**The Toilet Example (c) Copyright 2005, Niclas Winquist ;)**

<http://www.cnblogs.com/xiangshancuizhu/p/3305609.html>

**Mutex:**

Is a key to a toilet. One person can have the key - occupy the toilet - at the time.   
When finished, the person gives (frees) the key to the next person in the queue.

mutex是厕所钥匙，一次只能一人那着这把钥匙去厕所。结束了，这个人把钥匙给队列中的下一个人。

Officially: “Mutexes are typically used to serialise access to a section of re-entrant code   
that cannot be executed concurrently by more than one thread. A mutex object only   
allows one thread into a controlled section, forcing other threads which attempt to   
gain access to that section to wait until the first thread has exited from that section.”  
Ref: Symbian Developer Library

**Semaphore:**

Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys.   
The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free),   
then the count value is decremented as people are coming in.   
If all toilets are full, ie. there are no free keys left, the semaphore count is 0.   
Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key),   
and given to the next person in the queue.

信号量是一个自由的官方厕所钥匙，我们有四个厕所，他们的锁和钥匙是一样的。  
信号量开始设置为4，表示4个厕所是自由滴，如果一个人进去了，数量就-1.  
如果厕所满了，钥匙数目就为0，信号量数目这时也是0.如果一个人离开厕所，信号量+1，队列中的下一个人可以用啦！

Officially: “A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number.   
Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished   
using the resource (incrementing the semaphore).” Ref: Symbian Developer Library

所以，mutex就是一个binary semaphore （值就是0或者1）。但是他们的区别又在哪里呢？主要有两个方面：

\* 初始状态不一样：mutex的初始值是1（表示锁available），而semaphore的初始值是0（表示unsignaled的状态）。  
随后的操 作基本一样。mutex\_lock和sem\_post都把值从0变成1，mutex\_unlock和sem\_wait都把值从1变成0（如果值是零就等 待）。  
初始值决定了：虽然mutex\_lock和sem\_wait都是执行V操作，但是sem\_wait将立刻将当前线程block住，直到有其他线程 post；  
mutex\_lock在初始状态下是可以进入的。

\* 用法不一样（对称 vs. 非对称）：这里说的是“用法”。Semaphore实现了signal，但是mutex也有signal  
（当一个线程lock后另外一个线程 unlock，lock住的线程将收到这个signal继续运行）。  
在mutex的使用中，模型是对称的。unlock的线程也要先lock。  
而 semaphore则是非对称的模型，对于一个semaphore，只有一方post，另外一方只wait。  
就拿上面的厕所理论来说，mutex是一个钥 匙不断重复的使用，传递在各个线程之间，  
而semaphore择是一方不断的制造钥匙，而供另外一方使用（另外一方不用归还）。

(A mutex is really a semaphore with value 1.) ????

<http://www.freertos.org/Real-time-embedded-RTOS-mutexes.html>

**Mutexes are binary semaphores that include a priority inheritance mechanism.**

Whereas binary semaphores are the better choice for implementing synchronisation   
(between tasks or between tasks and an interrupt),

mutexes are the better choice for implementing simple mutual exclusion (hence 'MUT'ual 'EX'clusion).

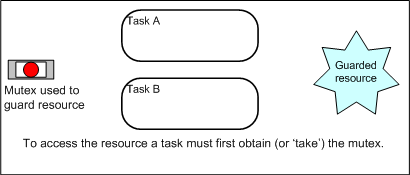
When used for mutual exclusion the mutex acts like a token that is used to guard a resource.   
When a task wishes to access the resource it must first obtain ('take') the token.   
When it has finished with the resource it must 'give' the token back -   
allowing other tasks the opportunity to access the same resource.

Mutexes use the same semaphore access API functions so also permit a block time to be specified.   
The block time indicates the maximum number of 'ticks' that a task should enter the Blocked state   
when attempting to 'take' a mutex if the mutex is not immediately available.

Unlike binary semaphores however - mutexes employ priority inheritance.   
This means that if a high priority task blocks while attempting to obtain a mutex (token)   
that is currently held by a lower priority task, then the priority of the task holding the token   
is temporarily raised to that of the blocking task.

This mechanism is designed to ensure the higher priority task is kept in the blocked state   
for the shortest time possible, and in so doing minimise the 'priority inversion' that has already occurred.

Priority inheritance does not cure priority inversion! It just minimises its effect in some situations.   
Hard real time applications should be designed such that priority inversion does not happen in the first place.



<http://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html>

**Binary semaphores are used for both mutual exclusion and synchronisation purposes.**

Binary semaphores and mutexes are very similar but have some subtle differences:

Mutexes include a priority inheritance mechanism, binary semaphores do not.

This makes binary semaphores the better choice for implementing synchronisation   
(between tasks or between tasks and an interrupt), and   
mutexes the better choice for implementing simple mutual exclusion.

The description of how a mutex can be used as a mutual exclusion mechanism holds equally for binary semaphores.   
This sub section will only describe using binary semaphores for synchronisation.

Semaphore API functions permit a block time to be specified.   
The block time indicates the maximum number of 'ticks' that   
a task should enter the Blocked state when attempting to 'take' a semaphore,   
should the semaphore not be immediately available.

If more than one task blocks on the same semaphore then the task with   
the highest priority will be the task that is unblocked the next time the semaphore becomes available.

Think of a binary semaphore as a queue that can only hold one item.   
The queue can therefore only be empty or full (hence binary).   
Tasks and interrupts using the queue don't care what the queue holds -   
they only want to know if the queue is empty or full.

This mechanism can be exploited to synchronise (for example) a task with an interrupt.

Consider the case where a task is used to service a peripheral.

Polling the peripheral would be wasteful of CPU resources, and prevent other tasks from executing.   
It is therefore preferable that the task spends most of its time in the Blocked state (allowing other tasks to execute)   
and only execute itself when there is actually something for it to do.   
This is achieved using a binary semaphore by having the task Block while attempting to 'take' the semaphore.   
An interrupt routine is then written for the peripheral that just 'gives' the semaphore when the peripheral requires servicing.   
The task always 'takes' the semaphore (reads from the queue to make the queue empty), but never 'gives' it.   
The interrupt always 'gives' the semaphore (writes to the queue to make it full) but never takes it.   
The source code provided on the xSemaphoreGiveFromISR() documentation page should make this clearer.

Task prioritisation can be used to ensure peripherals get services in a timely manner   
- effectively generating a 'differed interrupt' scheme.

An alternative approach is to use a queue in place of the semaphore.   
When this is done the interrupt routine can capture the data associated   
with the peripheral event and send it on a queue to the task.   
The task unblocks when data becomes available on the queue,   
retrieves the data from the queue, then performs any data processing that is required.   
This second scheme permits interrupts to remain as short as possible, with all post processing instead occurring within a task.

See the Semaphores/Mutexes section of the user documentation for a list of semaphore related API functions.   
Searching the files in the FreeRTOS/Demo/Common/Minimal directory will reveal multiple examples of their usage.   
Note that interrupts must NOT use API functions that do not end in "FromISR" .

Using a semaphore to synchronise a task with an interrupt.   
The interrupt only ever 'gives' the semaphore,   
while the task only ever 'takes' the semaphore.



<http://stackoverflow.com/questions/62814/difference-between-binary-semaphore-and-mutex>

**Their synchronization semantics are very different:**

* mutexes allow serialization of access to a given resource i.e. multiple threads wait for a lock,   
  one at a time and as previously said, the thread *owns* the lock until it is done:   
  **only** this particular thread can unlock it.
* a binary semaphore is a counter with value 0 and 1: a task blocking on it until **any** task does a sem\_post.   
  The semaphore advertises that a resource is available, and it provides the mechanism   
  to wait until it is signaled as being available.

As such one can see a mutex as a token passed from task to tasks   
and a semaphore as traffic red-light (it **signals** someone that it can proceed).

On Windows, there are two differences between mutexes and binary semaphores:

1. A mutex can only be released by the thread which has ownership,   
   i.e. the thread which previously called the Wait function, (or which took ownership when creating it).   
   A semaphore can be released by any thread.
2. A thread can call a wait function repeatedly on a mutex without blocking.   
   However, if you call a wait function twice on a binary semaphore   
   without releasing the semaphore in between, the thread will block.

In windows the difference is as below.   
**MUTEX:** process which successfully executes **wait** has to execute a**signal** and vice versa.   
**BINARY SEMAPHORES:** Different processes can execute **wait** or **signal** operation on a semaphore.

A *Mutex* controls access to a single shared resource.   
It provides operations to *acquire()* access to that resource and *release()* it when done.

A *Semaphore* controls access to a shared pool of resources.   
It provides operations to *Wait()* until one of the resources in the pool becomes available,   
and *Signal()* when it is given back to the pool.

When number of resources a Semaphore protects is greater than 1, it is called a *Counting Semaphore*.   
When it controls one resource, it is called a *Boolean Semaphore*.   
  
A boolean semaphore is equivalent to a mutex.

Thus a Semaphore is a higher level abstraction than Mutex.   
A Mutex can be implemented using a Semaphore but not the other way around.

[**http://www.geeksforgeeks.org/mutex-vs-semaphore/**](http://www.geeksforgeeks.org/mutex-vs-semaphore/)

Strictly speaking, **a mutex is locking mechanism** used to synchronize access to a resource.   
Only one task (can be a thread or process based on OS abstraction) can acquire the mutex.   
It means there will be ownership associated with mutex, and only the owner can release the lock (mutex).

**Semaphore is signaling mechanism** (“I am done, you can carry on” kind of signal).   
For example, if you are listening songs (assume it as one task) on your mobile   
and at the same time your friend called you, an interrupt will be triggered upon   
which an interrupt service routine (ISR) will signal the call processing task to wakeup.

<http://www.geeksforgeeks.org/mutex-vs-semaphore/>

**Mutex vs Semaphore**

What are the differences between Mutex vs Semaphore?

When to use mutex and when to use semaphore?

Concrete understanding of Operating System concepts is required to design/develop smart applications.   
Our objective is to educate the reader on these concepts and learn from other expert geeks.

As per operating system terminology, the mutex and semaphore are kernel resources that provide synchronization services   
(also called as *synchronization primitives*).

Why do we need such synchronization primitives? Won’t be only one sufficient?   
To answer these questions, we need to understand few keywords.   
Please read the posts on [atomicity](http://geeksforgeeks.org/?p=9179) and [critical section](http://geeksforgeeks.org/?p=9180).   
We will illustrate with examples to understand these concepts well, rather than following usual OS textual description.

**The** [**producer-consumer**](http://en.wikipedia.org/wiki/Producer-consumer_problem) **problem:**

Note that the content is generalized explanation. Practical details will vary from implementation.

Consider the standard producer-consumer problem.   
Assume, we have a buffer of 4096 byte length.   
A producer thread will collect the data and writes it to the buffer.   
A consumer thread will process the collected data from the buffer.   
Objective is, both the threads should not run at the same time.

**Using Mutex:**

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work.   
As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

**Using Semaphore:**

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources).   
A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

**Misconception:**

There is an ambiguity between *binary semaphore* and *mutex*.

We might have come across that a mutex is binary semaphore.   
But they are not! The purpose of mutex and semaphore are different.   
May be, due to similarity in their implementation a mutex would be referred as binary semaphore.

Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource.   
Only one task (can be a thread or process based on OS abstraction) can acquire the mutex.   
It means there will be ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal).   
For example, if you are listening songs (assume it as one task) on your mobile and   
at the same time your friend called you, an interrupt will be triggered upon   
which an interrupt service routine (ISR) will signal the call processing task to wakeup.

**General Questions:**

1. Can a thread acquire more than one lock (Mutex)?

Yes, it is possible that a thread will be in need of more than one resource, hence the locks.   
If any lock is not available the thread will wait (block) on the lock.

2. Can a mutex be locked more than once?

A mutex is a lock. Only one state (locked/unlocked) is associated with it.   
However, a recursive mutex can be locked more than once (POSIX complaint systems),   
in which a count is associated with it, yet retains only one state (locked/unlocked).   
The programmer must unlock the mutex as many number times as it was locked.

3. What will happen if a non-recursive mutex is locked more than once.

Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again,   
it will enter into the waiting list of that mutex, which results in deadlock.   
It is because no other thread can unlock the mutex.   
An operating system implementer can exercise care in identifying the owner of mutex   
and return if it is already locked by same thread to prevent deadlocks.

4. Are binary semaphore and mutex same?

No. We will suggest to treat them separately, as it was explained signalling vs locking mechanisms.   
But a binary semaphore may experience the same critical issues (e.g. priority inversion) associated with mutex.   
We will cover these later article.

A programmer can prefer mutex rather than creating a semaphore with count 1.

5. What is a mutex and critical section?

Some operating systems use the same word critical section in the API.   
Usually a mutex is costly operation due to protection protocols associated with it.   
At last, the objective of mutex is atomic access.   
There are other ways to achieve atomic access like disabling interrupts   
which can be much faster but ruins responsiveness.   
The alternate API makes use of disabling interrupts.

6. What are events?

The semantics of mutex, semaphore, event, critical section, etc… are same.   
All are synchronization primitives. Based on their cost in using them they are different.   
We should consult the OS documentation for exact details.

7. Can we acquire mutex/semaphore in an Interrupt Service Routine?

An ISR will run asynchronously in the context of current running thread.   
It is **not recommended** to query (blocking call) the availability of synchronization primitives in an ISR.   
The ISR are meant be short, the call to mutex/semaphore may block the current running thread.   
**However, an ISR can signal a semaphore or unlock a mutex.**

8. What we mean by “thread blocking on mutex/semaphore” when they are not available?

Every synchronization primitive will have waiting list associated with it.   
When the resource is not available, the requesting thread will be moved from the running list of processor   
to the waiting list of the synchronization primitive.

When the resource is available, the higher priority thread on the waiting list   
will get resource (more precisely, it depends on the scheduling policies).

9. Is it necessary that a thread must block always when resource is not available?

Not necessarily. If the design is sure ‘what has to be done when resource is not available‘,   
the thread can take up that work (a different code branch).   
To support application requirements the OS provides non-blocking API.

For example POSIX pthread\_mutex\_trylock() API.   
When the mutex is not available the function will return immediately   
where as the API pthread\_mutex\_lock() will block the thread till resource is available.

**References:**

[**http://www.netrino.com/node/202**](http://www.netrino.com/node/202)

[**http://doc.trolltech.com/4.7/qsemaphore.html**](http://doc.trolltech.com/4.7/qsemaphore.html)

Also compare mutex/semaphores with Peterson’s algorithm and Dekker’s algorithm.   
A good reference is the Art of Concurrency book. Also explore reader locks and writer locks in Qt documentation.

**Exercise:**

Implement a program that prints a message “An instance is running” when executed more than once in the same session.   
For example, if we observe word application or Adobe reader in Windows, we can see only one instance in the task manager.

How to implement it?

Article compiled by **Venki**.

Please write comments if you find anything incorrect, or

you want to share more information about the topic discussed above.