
Performance Monitoring with Prometheus and Grafana

Release 1.4.0

Sep 21, 2020

1	Overview	3
1.1	The Ecosystem	3
1.2	Architecture	4
1.3	Further Clarifications	4
2	Prometheus Server Deployment	7
2.1	Deployment	7
2.2	Resources	7
3	Exporter Deployment	9
3.1	Operating System Exporters	9
3.1.1	Linux node_exporter	9
3.1.2	Windows wmi_exporter	10
3.1.3	Configure Prometheus	10
3.2	SNMP Exporter for Network Equipement	10
3.2.1	SNMP Introduction	11
3.2.2	snmp_exporter	15
3.2.3	Configure Prometheus	24
4	Grafana Dashboard	27
4.1	Grafana Deployment	27
4.2	Understand Prometheus Metrics	27
4.2.1	Exporter Metrics	27
4.2.2	Prometheus Metrics	28
4.3	Add Data Source	29
4.4	Create Dashboard	30
4.4.1	Variables	30
4.4.2	label_values	30
4.4.3	Define Variables	31
4.4.4	Add Panel	37
4.5	Grouping and Group Repeating	43
4.6	Save Dashboard Settings	47
4.7	Reference	47
5	Alerting	49
5.1	Alertmanager Deployment	49
5.2	Alerting Rules	50

6	Tips	53
6.1	Install Prometheus as a Systemd Service	53
6.2	Import Community Defined Grafana Dashboard	54
6.3	Add New Labels	55
6.4	Select Legends to Display on Grafana Panel	56
6.5	Graph Top N in Grafana	56
6.6	Use Telegraf as Exporters	57
6.7	Collect Metrics with Arbitrary Scripts	58
6.8	Use Alerta to Manage Alerts	58
6.9	Show Diagrams on Grafana Panel	58
6.10	The Built-in “up” Metric	59
6.11	Scrape Interval Pitfall	59
6.12	Reload Prometheus through HTTP POST	60
6.13	Add a Label with PromQL	60

This document will guide readers on how to get started performance monitoring with Prometheus and Grafana dashboard. The source of this document can be found [here on github](#).

Performance monitoring (and alerts management) is one key focus of IT operations. There exist several good solutions, such as [Zabbix](#), [Nagios](#), etc.

Prometheus is the most popular solution recent years for the same target since it supports Kubernetes monitoring which makes it the best choice for monitoring cloud native solutions when cloud becomes the trend of IT. However, its popularity is not simply because of the rapid adoption of Kubernetes. Prometheus is designed with smart architecture, great API for extension, wonderful features for scaling, etc. All these advantages make Prometheus suitable for almost all kinds of monitoring requirements in data centers.

1.1 The Ecosystem

When we talk about Prometheus, actually we are talking about an ecosystem for performance monitoring around Prometheus but not just itself. The ecosystem consists of several major components:

- Prometheus Server: scrapes and stores time series data;
- Exporters: collect metrics from data sources, like Linux, Windows, network equipment, databases, etc.;
- Pushgateway: allows ephemeral and batch jobs to expose their metrics to Prometheus;
- Grafana: displays metrics through a powerful web based GUI;
- Alertmanager: handles alerts.

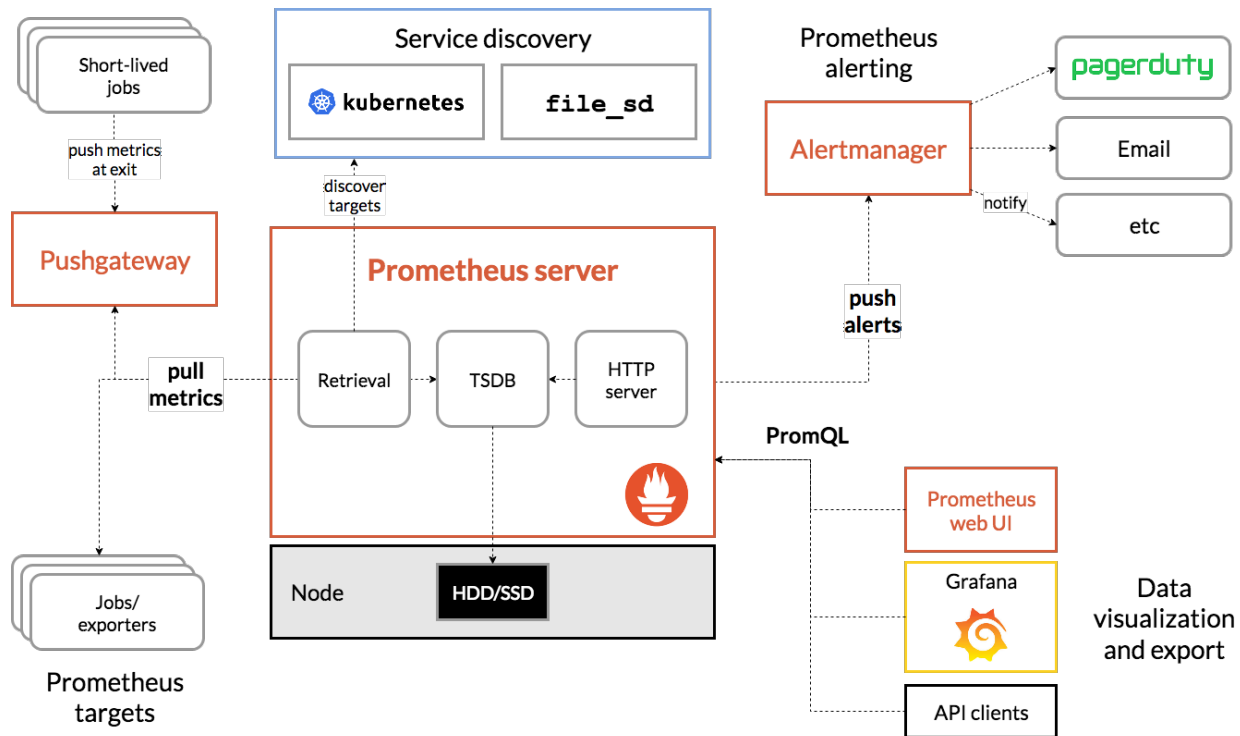
All these components work closely to achieve the monitoring goals with a **data flow** as below:

1. Prometheus exporters are responsible for collecting performance metrics from targets (monitored devices). They work as standalone apps running close to or even on the targets. The Prometheus company or users can implement their own exporters based on their requirements. There exist quite a few exporters for use directly, refer [here](#);
2. When the monitoring targets are ephemeral jobs, it is not worthwhile running an exporter for ever. Under such a condition, metrics can be collected through whatever implementations (shell, ruby, python, java, etc. as long as the output follows the required format) and pushed to Pushgateway;

3. Prometheus itself does not collect metrics from targets directly, instead, it collects (called **scrape** by Prometheus) metrics from enabled exporters and Pushgateway and save metric data locally into the **/data** directly or remotely to a database (such as InfluxDB);
4. Grafana leverages Prometheus as a data source through PromQL (Prometheus query language) and showcases metrics based on user defined dashboards;
5. Alertmanager works by defining rules related with specified metrics on Prometheus server and get alerted when the metrics exceed defined thresholds.

1.2 Architecture

To understand the solution better, let's diagram the full idea:



1.3 Further Clarifications

Prometheus is a matured monitoring solution suitable for most IT cases, for more information about it, please refer to the [official overview](#).

Before moving forward to the next chapter, there are some more clarifications about the relationships on exporter, Prometheus, and Grafana you should always be clear about:

- Exporters only collect metrics from targets, they won't save metrics;
- Prometheus scrapes metrics from exporters and save the metrics centrally. Advanced queries against the metrics are supported by Prometheus through PromQL;
- Grafana leverages PromQL and its builtin query functions to filter metrics and display them on dashboards;

- Grafana uses Prometheus as a kind of data source, and also support quite a lot other data sources such as ELK, InfluxDB, etc.

Prometheus Server Deployment

Prometheus server is the center of the monitoring solution. It is responsible for:

- Scraping metrics from exporters and Pushgateway;
- Supporting queries through PromQL;
- Triggering alerts to Alertmanager;
- ...

2.1 Deployment

Let's deploy a Prometheus server:

1. Download the latest tarball (we use Linux) from [the download page](#);
2. Decompress the package and change into the directory;
3. The configuration file is **prometheus.yml**, there is no need to change it for now since no export has been deployed;
4. Kick started Prometheus **./prometheus**;
5. We should be able to access it through **http://<Prometheus Server IP>:9090**.

After getting a Prometheus server running, we can move forward deploying exporters and scraping metrics from them.

2.2 Resources

Prometheus is easy and straightforward, but it stills need some reading work if you want to use it flexible. Here are some documents you should go through when there is enough bandwidth:

- [Prometheus Getting Started](#)
- [Prometheus Configuration Reference](#)

- [Querying Prometheus](#)

Exporter Deployment

Based on our previous introduction on Prometheus, we know that exporters are the actual workers collect metrics from data sources like OSs, network devices, etc.

In this chapter, we will go through how to deploy `node_exporter` for Linux, how to deploy `wmi_exporter` for Windows, and how to deploy `snmp_exporter` for network equipment (switch, routers, firewall, etc.).

Actually, all exporters follow the similar deployment method and involve similar Prometheus configuration changes. After reading this chapter, you should have a common idea on how to deploy and enable any exporter.

3.1 Operating System Exporters

Linux and Windows are major OSs used in data centers today. As a popular performance monitoring solution, Prometheus of course supports collecting metrics from them. This chapter will cover how to install exporters on Linux and Windows. The metrics collected by these exporters will be scraped by Prometheus server, and then leveraged by Grafana to create dashboards.

3.1.1 Linux `node_exporter`

Since Linux is the most popular OS used in data centres, the company of Prometheus develops an official exporter for collecting Linux metrics, which is called `node_exporter`.

The installation of `node_exporter` is as easy as decompressing a Linux tarball:

1. Download `node_exporter` from [here](#);
2. On the target Linux, decompress it and change directory into the decompressed folder;
3. Kick started it as `./node_exporter`;
4. That is it, `node_exporter` is up and running. All collected metrics can be seen by accessing **`http://<Linux IP>:9100/metrics`**

3.1.2 Windows wmi_exporter

Windows is also adopted widely in data centers. Unfortunately, the company of Prometheus does not develop an official export for Windows. Luckily enough, Prometheus has a really active community and the contributors develop an exporter for Windows, A.K.A wmi_exporter.

The installation of wmi_exporter won't take more effort than installing node_export on Linux:

1. Download the latest msi package from its [github repo release page](#);
2. Double click the package to start the installation and follow the wizard to complete the process;
3. That is it. All collected metrics can be seen by accessing **http://<Windows IP>:9182/metrics**

3.1.3 Configure Prometheus

Once an exporter (node_exporter, wmi_exporter and all others) is up and running, it starts to collect metrics from the monitored device(s). However, Prometheus has no idea about an exporter before it is enabled in the Prometheus configuration file (prometheus.yml).

To make Prometheus scrape metrics from node_exporter and wmi_exporter:

1. Modify the Prometheus configuration file prometheus.yml and add below job definitions:

```
# For node_exporter
- job_name: 'node_exporter'
  static_configs:
    - targets:
      - '<Linux IP>:9100'

# For wmi_exporter
- job_name: 'wmi_exporter'
  static_configs:
    - targets:
      - '<Windows IP>:9182'
```

2. Restart Prometheus: since each Prometheus server will scrape multiple targets (exporters and pushgateway), it is not recommended to restart the whole Prometheus server process directly since it impacts all targets, instead, it is recommended to reload the configuration file only:

```
# Find the Prometheus server process ID
ps -ef | grep prometheus
# Reload configuration file by sending SIGHUP
kill -s SIGHUP <prometheus process ID>
```

3. If everything is fine, the newly added node_exporter and wmi_exporter should appear as a target under **http://<prometheus server IP>:9090/targets**;
4. You should be able to see all metrics by clicking the corresponding targets.

3.2 SNMP Exporter for Network Equipment

Prometheus provides an official SNMP exporter, A.K.A snmp_exporter, which can be used for monitoring devices which support SNMP, such as switches, routers, firewall, etc.

This chapter will cover all ideas on how to enable snmp_exporter for monitoring switches only. For other devices which support SNMP, the enablement process is common.

3.2.1 SNMP Introduction

SNMP, the acronym for **Simple Network Management Protocol**, is an application–layer protocol defined by the Internet Architecture Board (IAB) in RFC1157 for exchanging management information between network devices. It is a part of TCP/IP protocol suite.

Its main usage focus on collecting and organizing information about managed devices on IP networks and for modifying that information to change device behavior. Devices that typically support SNMP include cable modems, routers, switches, servers, workstations, printers, and more.

Notes: This document only shares the most basic knowledge on SNMP, for detailed in-depth information, please leverage google or read books.

Main Components

SNMP consist of:

- SNMP Manager
- SNMP Agent
- MIB

SNMP Manager

A SNMP manager or management system is responsible to communicate with the SNMP agent implemented network devices. This is typically a computer that is used to run one or more network management systems (NMS). The key functions are as below:

- Query agents
- Set variables in agents
- Get notifications from agents

SNMP Agent

A SNMP agent is a program that is packaged within the managed devices. Enabling agents allow agents collect the management information databases from the managed devices locally and make them available to the SNMP manager during query. These agents could be standard (e.g. Net-SNMP) or specific to a vendor (e.g. HP insight agent).

The key functions of a SNMP agent is as below:

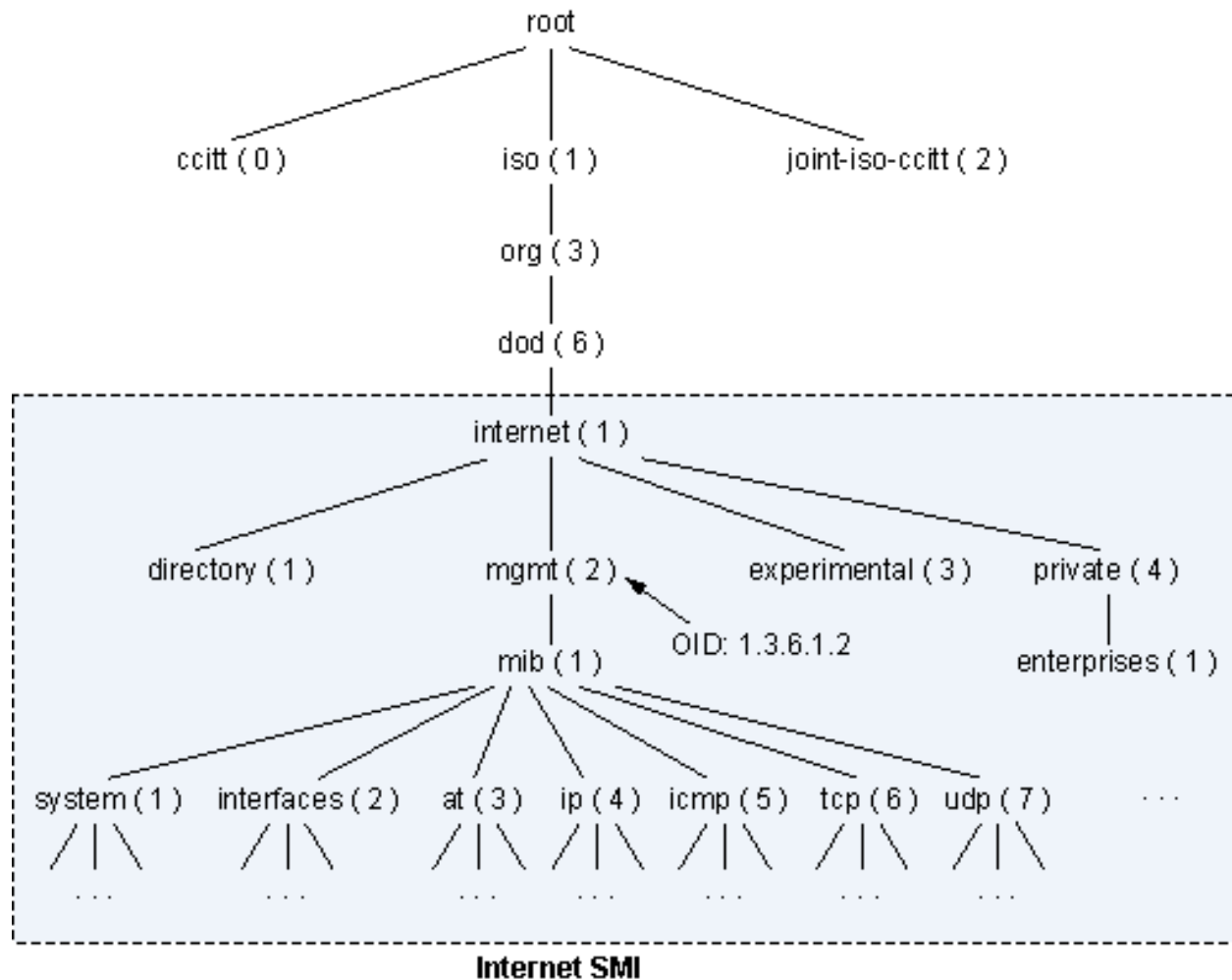
- Collect management information locally on managed devices
- Stores and retrieves management information as defined in the MIB
- Signals an event to the manager

MIB

MIB is short for Management Information Base, which describes the managed device parameters, such as port status, throughput, etc. The SNMP manager uses this database to request the agent for specific information and further translates the information as needed for the Network Management System (NMS).

Typically these MIB contains standard set of statistical and control values defined for hardware nodes on a network. SNMP also allows the extension of these standard values with values specific to a particular agent through the use of **private MIBs**.

MIB is organized as a well defined tree structure with each leaf stands for a specific object of the managed device, which is referred as **Object Identifier (OID)**. Each OID is unique and can be located from the root of the MIB tree with an address like .1.1.2.3.x.x.x. For common frequently used OIDs, human friendly names will be assigned/mapped to the number dot address, e.g., “sysDescr” is the same as “.1.3.6.1.2.1.1.1”, which is defined by RFC1213 .



OID Details

We need to have a good understanding on what each OID stands for when SNMP is used for monitoring because some calculation work may be needed for collected stats - we need understand the unit, the meaning, etc.. This task can be done with the help of MIB browser and <http://oidref.com/>.

Let's take a look at the below example:

1. OID **IfHCInOctets** is collected, we want to understand what it stands for;
2. MIB browser will show a short description on the OID once clicked:

Name	ifHCInOctets
OID	.1.3.6.1.2.1.31.1.1.1.6
MIB	IF-MIB
Syntax	COUNTER64
Access	read-only
Status	current
DefVal	
Augments	ifEntry
Descr	<p>The total number of octets received on the interface, including framing characters. This object is a 64-bit version of ifInOctets.</p> <p>Discontinuities in the value of this counter can occur at re-initialization of the management system, and at other times as indicated by the value of ifCounterDiscontinuityTime.</p>

3. More detailed information can be gotten with oidref:

← → ↻ ⓘ Not secure **oidref.com/1.3.6.1.2.1.31.1.1.1.6**

[Main page](#) [Organizations list](#) [Contacts](#)

Reference record for OID 1.3.6.1.2.1.31.1.1.6

1 iso > 3 identified-organization, org, iso-identified-organization > 6 dod > 1 internet > 2 mgmt > 1 mib-2, mib > 31 ifMIB > 1 ifMIBObjects > 1 ifXTable > 1 ifXEntry > 6 ifHCInOctets

parent
1.3.6.1.2.1.31.1.1 (ifXEntry)
node code
6
node name
ifHCInOctets
dot oid
1.3.6.1.2.1.31.1.1.6
type
OBJECT-TYPE
asnl oid
<ul style="list-style-type: none"> • {iso(1) identified-organization(3) dod(6) internet(1) mgmt(2) mib-2(1) ifMIB(31) ifMIBObjects(1) ifXTable(1) ifXEntry(1) ifHCInOctets(6)} • {iso(1) identified-organization(3) dod(6) internet(1) mgmt(2) mib(1) ifMIB(31) ifMIBObjects(1) ifXTable(1) ifXEntry(1) ifHCInOctets(6)} • {iso(1) org(3) dod(6) internet(1) mgmt(2) mib-2(1) ifMIB(31) ifMIBObjects(1) ifXTable(1) ifXEntry(1) ifHCInOctets(6)} • {iso(1) org(3) dod(6) internet(1) mgmt(2) mib(1) ifMIB(31) ifMIBObjects(1) ifXTable(1) ifXEntry(1) ifHCInOctets(6)} • {iso(1) iso-identified-organization(3) dod(6) internet(1) mgmt(2) mib-2(1) ifMIB(31) ifMIBObjects(1) ifXTable(1) ifXEntry(1) ifHCInOctets(6)} • {iso(1) iso-identified-organization(3) dod(6) internet(1) mgmt(2) mib(1) ifMIB(31) ifMIBObjects(1) ifXTable(1) ifXEntry(1) ifHCInOctets(6)}

After understanding the OID thoroughly, we can do calculations. E.g., IfHCInOctets unit is byte, then we can translate it into KB, MB, etc.

SNMP Versions

There are 3 versions of SNMP protocol:

- SNMPv1
- SNMPv2 (SNMPv2c, SNMPv2u)
- SNMPv3

The main differences focus on security. Simply speaking, SNMPv1 is not secure enough, SNMPv3 is too strict, hence SNMPv2 are the most popular adopted deployment.

Community String

A SNMP manager needs to talk to a SNMP agent to work, so a mechanism to protect the connection is required. Community based or user based authentication can be used for the purpose.

Community string is the most straightforward method for authentication if SNMPv2c is used. Its implementation is quite simple: a string is defined as a kind of password on SNMP agents, and SNMP manger queries agents by providing the correct string for authentication.

SNMP can be used to query agents, and also can be leveraged to set variables to change something (e.g., online/offline a port). The community string provides READ and WRITE capability accordingly:

- **READ ONLY:** also referred as “public community string”, and the default value is “public” for most managed devices once SNMP agents are enabled. It can only be used to query MIB information;
- **READ WRITE/WRITE:** also referred as “private community string”, and this is not enabled/set by default. It can be used to change object status, such as reboot, port online/offline, etc.

Poll and Trap

SNMP supports 2 ways to get information from MIB:

- **Poll:** Poll is triggered from SNMP managers, which send queries to SNMP agents on managed devices, which listen at UDP port 161. Each poll is a synchronous operation, BTW.
- **Trap:** Instead of performing queries from SNMP managers, trap is a mechanism to let SNMP agents send asynchronous events to SNMP managers directly. With this scenario, SNMP managers listen at UDP port 162 for agent connections, and may take actions following the events (ack, etc.).

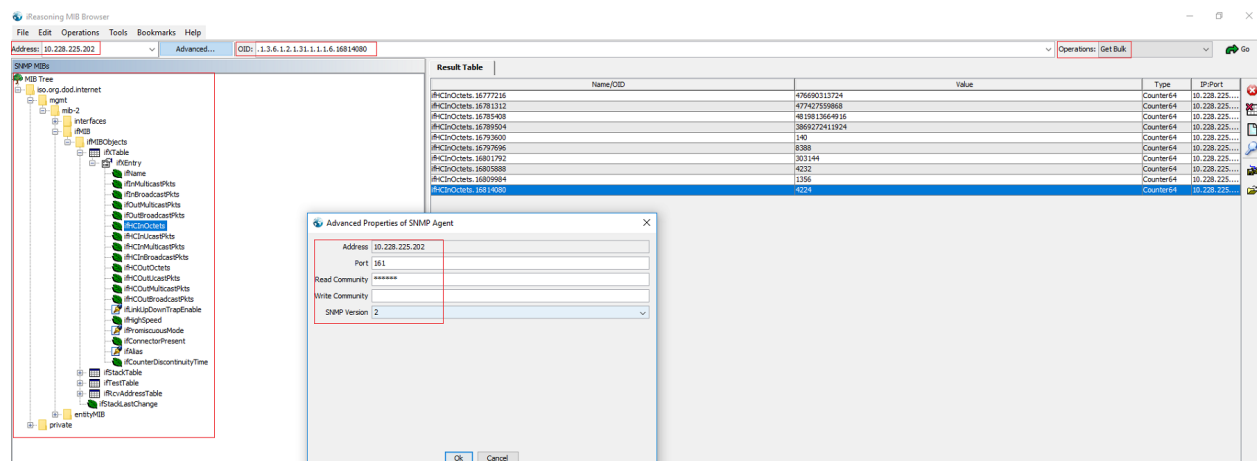
Poll Commands

SNMP ships very simple commands to support queries to MIB. The most frequently used commands are as below:

- **GET:** retrieve information on one specified OID
- **GET NEXT:** retrieve information on the next OID
- **GET BULK:** retrieve information for a group of OIDs which share similar features
- **WALK:** actually WALK is not a SNMP command, but just a wrapper of GET NEXT. It is used to get information from a tree of OIDs.

MIB Browser

Beside network management system (SNMP Manager), a lightweight tool called **MIB Browser** can be leveraged to explore SNMP MIB information. Below is an overview of a GUI based MIB browser from iReasoning (free to use).



3.2.2 snmp_exporter

Prometheus provides official SNMP support through `snmp_exporter`, which consists of:

- **exporter:** collect metrics from managed devices through SNMP, acts as a NMS;

- generator: create configurations for exporter by mapping SNMP OIDs to counters, gauges which can be understood by Prometheus;

This document will cover both topics.

generator

Simply speaking, generator is the tool parsing SNMP MIBs and creating a configuration file containing specified OIDs which are mapped to indicators of Prometheus. Then exporter queries SNMP agents for those specified OIDs and map the results as counters/gauges based on the configuration file waiting for Prometheus scrapes.

It is not easy to understand the story without an example, so let's do it. By the way, generator can be gotten from [here](#)

Consolidated MIBs

The public/standard MIBs(defined by RFC) contain only the basic information (OIDs) for managed devices, which are far more less than expected most of times. Each vendor, such as Cisco, will provide their extended/private MIBs to support more features (OIDs). Such MIBs can be downloaded from vendors' support site. Thanks to open source network manage system (NMS), we do not need to search and download each MIB directly, we can leverage already consolidated MIBs directly from open source NMS.

LibreNMS is such a open source NMS, it consolidates MIBs from all major vendors covering switches, servers, storage, etc. For more information, check [here](#)

The first step of this example is getting a copy of these consolidated MIBs, this is easy since it is on github - just clone it. After the download, we can have a check of those MIBs under directory **librenms/mib**: there exists hundreds of MIBs, wonderful!

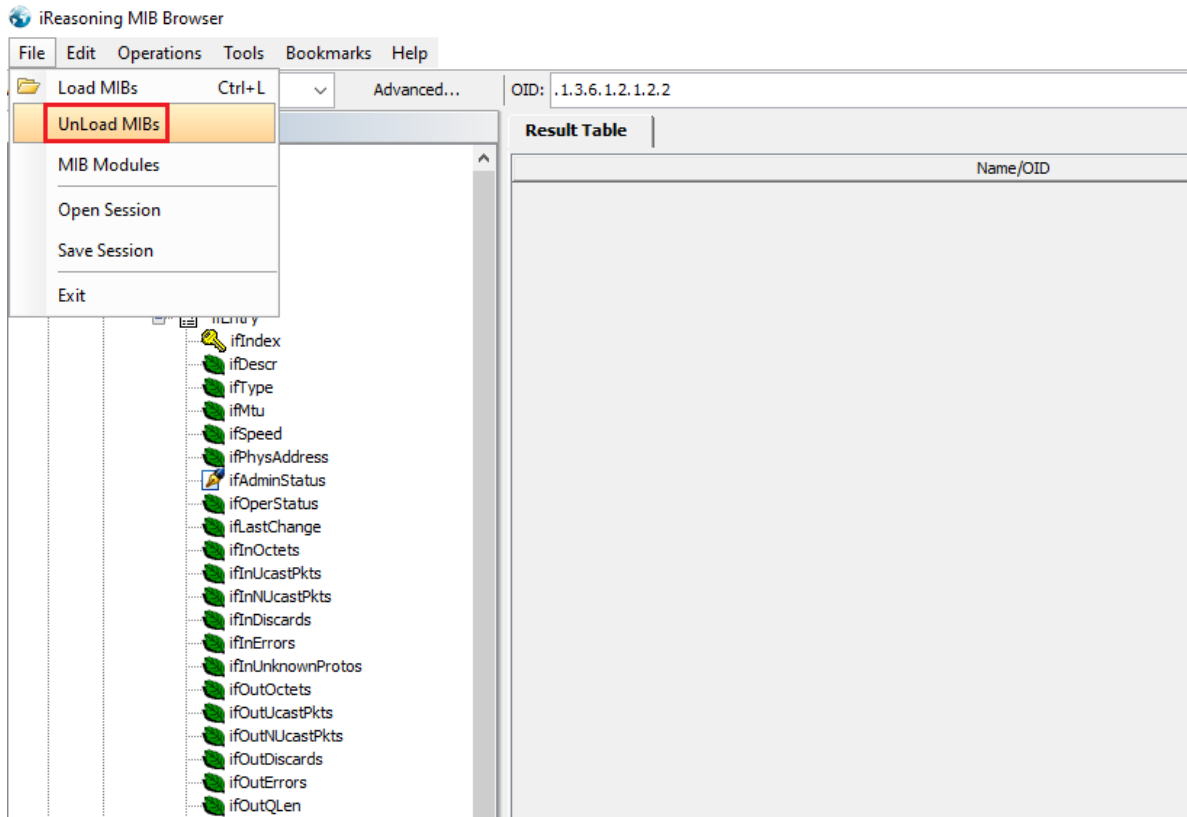
Identify OIDs

The goal of using Prometheus is collecting information we care. For switch, the goal becomes collecting information for OIDs we are interested in. Before creating the configuration file with generator, we should locate these wanted OIDs.

In this example, we want to monitor Cisco switch interface throughput and overall processor and memory usage.

To locate related MIB OIDs, MIB browser is an important tool. In our example, we use the free **iReasoning MIB Browser**. After opening it, some public frequently used MIBs are already loaded automatically.

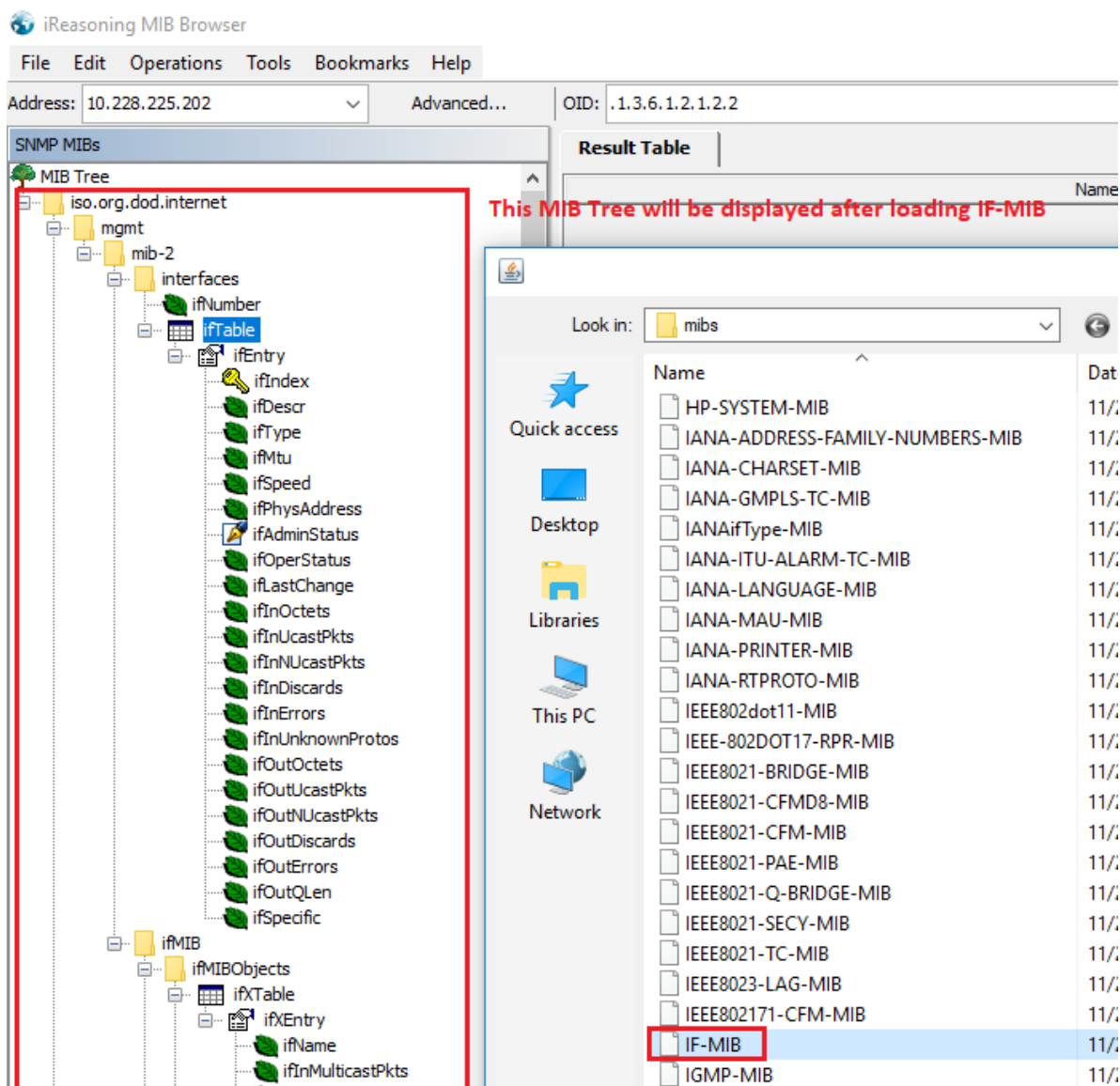
1. Let's unload all existing MIBs and start from scratch to demonstrate how to perform the task



2. Let's find the MIBs for switch interface stats

- Go to <http://www.net-snmp.org/docs/mibs/>
- Search **interface**
- **IF-MIB** pops up

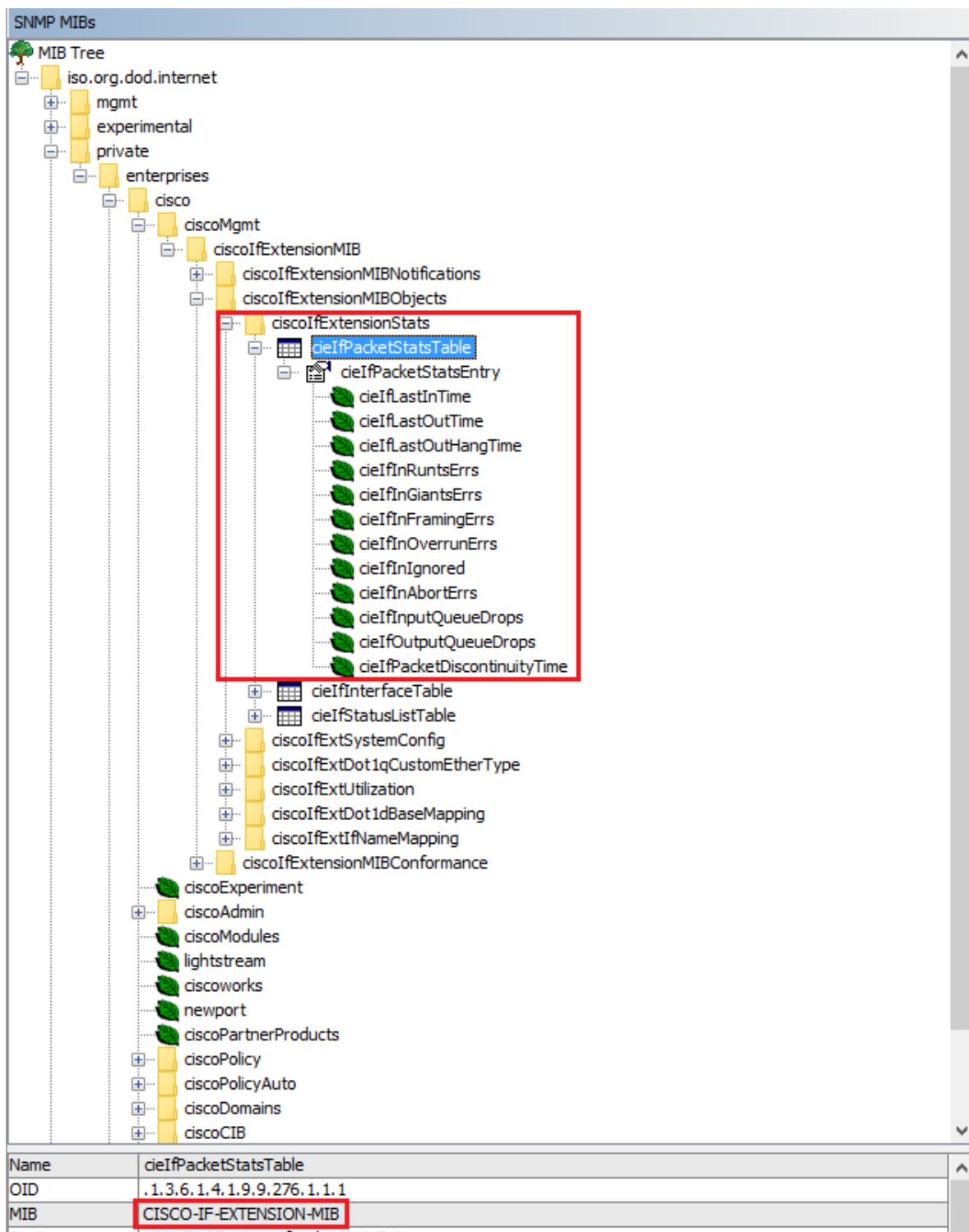
3. Load IF-MIB in MIB browser:



4. After loading IF-MIB, we can see OIDs related with interfaces. But since we want to monitor Cisco switch, if Cisco provides extend/enhanced MIB for IF-MIB, it will be better since much more information can be gotten.
5. Let's google, and **CISCO-IF-EXTENSION-MIB** can be found:



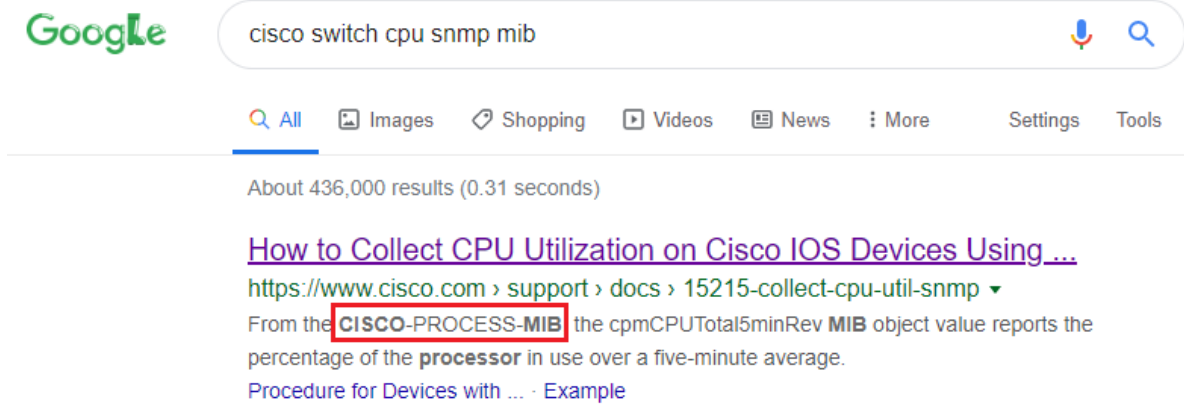
6. Let's load the CISCO-IF-EXTENSION-MIB which is available within librenms/mib/cisco. After loading the MIB, more information about switch interfaces can be seen as below:



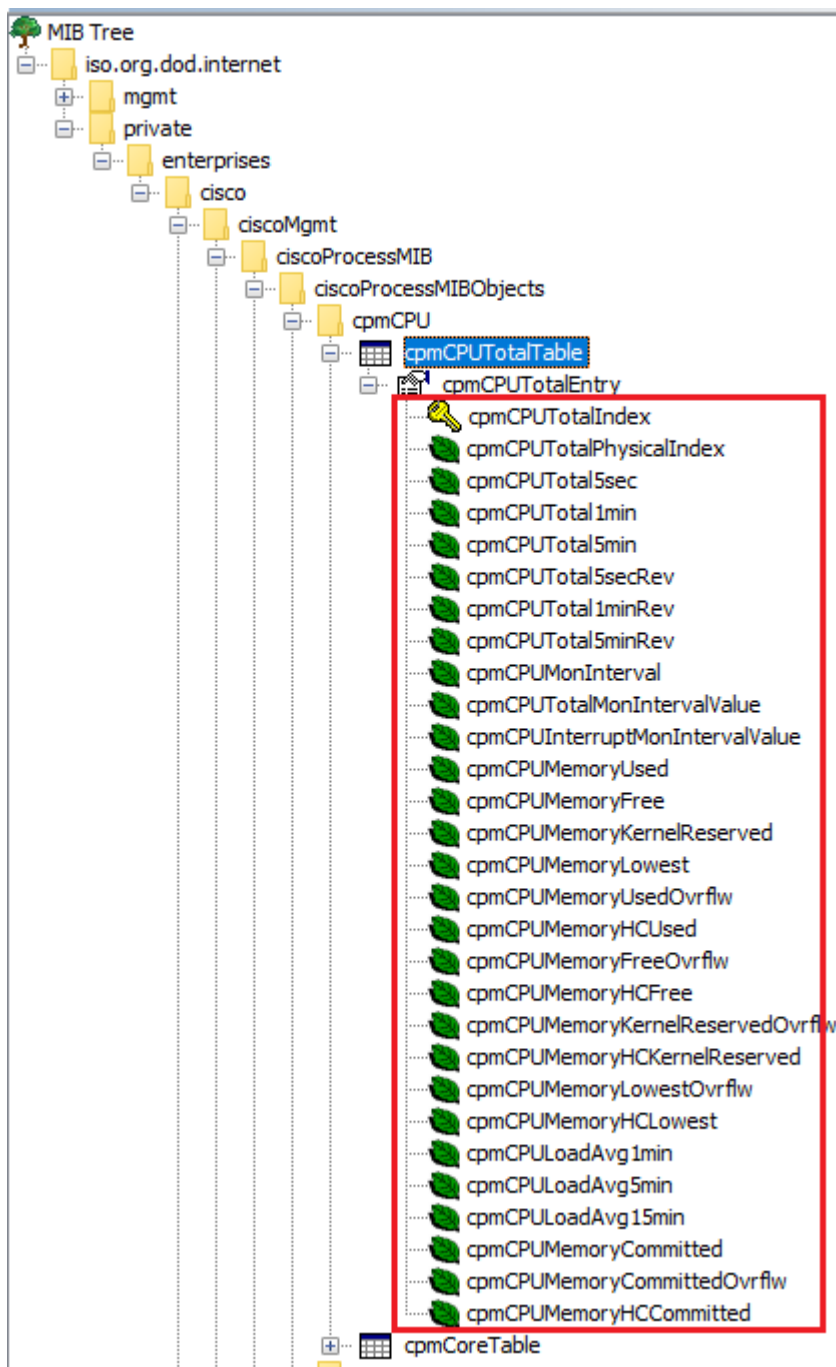
The screenshot shows the SNMP MIBs browser interface. The MIB Tree is expanded to show the 'ciscoIfExtensionStats' MIB, which contains the 'cieIfPacketStatsTable'. This table is highlighted with a red box. Below the tree, a table shows the details for 'cieIfPacketStatsTable'.

Name	Value
Name	cieIfPacketStatsTable
OID	.1.3.6.1.4.1.9.9.276.1.1.1
MIB	CISCO-IF-EXTENSION-MIB

7. It is time to find MIBs for CPU and memory stats
8. Again, search CPU and memory with <http://www.net-snmp.org/docs/mibs>, but this time, no result can be found
9. Let's google "Cisco switch cpu snmp mib" to locate the CPU usage information at first



10. Let's load the MIB **CISCO-PROCESS-MIB** from directory librenms/mib - great, both CPU and memory information are supported from this MIB:



11. MIBs are ready, let's identify OIDs with the help of MIB browser:

- Interface related stats:
 - ifEntry: .1.3.6.1.2.1.2.2.1
 - ifXTable: .1.3.6.1.2.1.31.1.1
- Cisco private MIBs related with interface stats:
 - cieIfPacketStatsEntry
- CPU and memory related stats:

– cpmCPUTotalTable: .1.3.6.1.4.1.9.9.109.1.1.1

Create generator configuration file

After getting OIDs, it is required to create a configuration file for generator to define how to generate the configuration file for exporter.

```
git clone https://github.com/prometheus/snmp_exporter.git
cd snmp_exporter/generator
vim generator.yml
```

Make changes based on OIDs collected in the above section, the original generator.yml sample can be referred as the blueprint. Below is the one we are going to use:

```
modules:
  cisco_mib:
    auth:
      community: public
    walk:
      - sysDescr
      - sysUpTime
      - sysName
      - interfaces
      - ifXTable
      - cieIfPacketStatsEntry
      - 1.3.6.1.4.1.9.9.109.1.1 # Defined within Cisco private mib CISCO-PROCESS-MIB
    lookups:
      - source_indexes: [ifIndex]
        lookup: ifAlias
      - source_indexes: [ifIndex]
        lookup: ifDescr
      - source_indexes: [ifIndex]
        lookup: 1.3.6.1.2.1.31.1.1.1.1 # ifName
    overrides:
      ifAlias:
        ignore: true # Lookup metric
      ifDescr:
        ignore: true # Lookup metric
      ifName:
        ignore: true # Lookup metric
      ifType:
        type: EnumAsInfo
```

Notes: the community string needs to be in line with what exactly is used on target switches.

Create exporter configuration file

Once the generator configuration file is ready, it is time to generate the configuration file for exporter:

```
cd snmp_exporter/generator
go build
export MIBDIRS=../../librenms/mibs:../../librenms/mibs/cisco
./generator generate
copy snmp.yml /tmp
```

After running above commands, the exporter configuration file **snmp.yml** is generated. It is time to run the exporter.

exporter

The exporter is responsible for collecting OIDs information and map them to Prometheus understandable metrics based on the configuration file (snmp.yml).

Let's install and run it:

1. Download the latest tarball from its [github repo release page](#) based on your OS (we use the same Linux server where the generator code is on);
2. Decompress the tarball and change directory into the decompressed folder;
3. Copy **snmp.yml** generated above by the generator to the directory **cp /tmp/snmp.yml .**;
4. Kick started snmp_exporter as **./snmp_exporter**;
5. That is it, snmp_exporter is up and running. It can be accessed through **http://<Exporter Server IP>:9116**;
6. Input a switch IP and the corresponding mib name (such as cisco_mib), you should be able to see the metrics as configured in snmp.yml.

3.2.3 Configure Prometheus

To make Prometheus scrape metrics from the snmp_exporter:

1. Modify the Prometheus configuration file prometheus.yml and add below job definition:

```
- job_name: 'snmp'
  static_configs:
    - targets:
      - 10.226.70.248
      - 10.226.70.249
  metrics_path: /snmp
  params:
    module: [dell_mib]
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: <Exporter IP>:9116 # The SNMP exporter's real hostname:port.
```

2. Notes:

- job_name: define a job name, “snmp” is used in this example;
 - targets: define the switches to scrape metrics from with the snmp_exporter. Here, 2 x switches are defined;
 - replacement: define the address and port where the snmp_exporter itself is listening. Do not use “local-host:<port>” even when it works, since this will make it difficult to distinguish endpoints on the Prometheus target display page.
3. Restart Prometheus: since each Prometheus server will scrape multiple targets (exporters and pushgateway), it is not recommended to restart the whole Prometheus server process directly since it impacts all targets, instead, it is recommended to reload the configuration file only:

```
# Find the Prometheus server process ID
ps -ef | grep prometheus
# Reload configuration file by sending SIGHUP
kill -s SIGHUP <prometheus process ID>
```

4. If everything is fine, the newly added snmp_exporter should appear as a target under **http://<prometheus server IP>:9090/targets**;
5. You should be able to see all metrics for each switch by clicking the corresponding targets under the snmp group.

This chapter will cover the knowledge on how to create Grafana dashboards to showcase metrics scraped by Prometheus.

4.1 Grafana Deployment

First thing first, we need to have a running Grafana server before creating any dashboard.

The Grafana deployment is pretty easy, and the official document is wonderful. E.g., to deploy Grafana on CentOS, the official [Install on RPM-based Linux](#) is good enough to kick started the installation. For other OSs, similar documents can be found from the same page. We won't provide any words on the installation any more.

4.2 Understand Prometheus Metrics

Before moving forward, we need to have basic understanding on how metrics look like from the perspective of exporters and Prometheus. We are going to show this idea by using the `node_exporter` we previously deployed in a previous chapter.

4.2.1 Exporter Metrics

Metrics collected by an exporter (`node_exporter` in this chapter) are as below:

```
# HELP node_disk_io_time_seconds_total Total seconds spent doing I/Os.
# TYPE node_disk_io_time_seconds_total counter
node_disk_io_time_seconds_total{device="dm-0"} 74.134
node_disk_io_time_seconds_total{device="dm-1"} 0.546
node_disk_io_time_seconds_total{device="dm-10"} 4279.887
node_disk_io_time_seconds_total{device="dm-11"} 4280.253
node_disk_io_time_seconds_total{device="dm-12"} 4280.296
node_disk_io_time_seconds_total{device="dm-13"} 0.3
node_disk_io_time_seconds_total{device="dm-2"} 0.316
node_disk_io_time_seconds_total{device="dm-3"} 0.20600000000000002
node_disk_io_time_seconds_total{device="dm-4"} 4279.92
node_disk_io_time_seconds_total{device="dm-5"} 4280.101
node_disk_io_time_seconds_total{device="dm-6"} 4280.561
node_disk_io_time_seconds_total{device="dm-7"} 4276.1630000000005
node_disk_io_time_seconds_total{device="dm-8"} 0.368
node_disk_io_time_seconds_total{device="dm-9"} 4280.003
node_disk_io_time_seconds_total{device="sda"} 74.854
node_disk_io_time_seconds_total{device="sdaa"} 1591.775
node_disk_io_time_seconds_total{device="sdab"} 0.2
node_disk_io_time_seconds_total{device="sdac"} 4098.705
node_disk_io_time_seconds_total{device="sdad"} 4098.8330000000005
```

Let's explain the metrics (counter, gauge, etc., refer to [Metrics Types](#)) with `node_disk_io_time_seconds_total` as an example:

- Each metric has a name, in this example, its name is `node_disk_io_time_seconds_total`;
- An metric may have some labels associated with it to distinguish its instances. In this example, `node_disk_io_time_seconds_total` has only one label "device". Based on the label values, instances can be differentiated easily - this is important for data filter;
- Metrics will be collected from exporters but won't be saved on exporters.

4.2.2 Prometheus Metrics

Metrics scraped by Prometheus from an exporter (node_exporter in this chapter) are as below:

Prometheus Alerts **Graph** Status ▾ Help

☐ Enable query history

`node_disk_io_time_seconds_total` Enter PromQL here

Execute - insert metric at cursor - ▾

Graph Console

◀ Moment ▶

Element

<code>node_disk_io_time_seconds_total{device="dm-0",instance="10.226.68.144:9100",job="node_exporter"}</code>
<code>node_disk_io_time_seconds_total{device="dm-1",instance="10.226.68.144:9100",job="node_exporter"}</code>
<code>node_disk_io_time_seconds_total{device="dm-10",instance="10.226.68.144:9100",job="node_exporter"}</code>
<code>node_disk_io_time_seconds_total{device="dm-11",instance="10.226.68.144:9100",job="node_exporter"}</code>
<code>node_disk_io_time_seconds_total{device="dm-12",instance="10.226.68.144:9100",job="node_exporter"}</code>
<code>node_disk_io_time_seconds_total{device="dm-13",instance="10.226.68.144:9100",job="node_exporter"}</code>
<code>node_disk_io_time_seconds_total{device="dm-2",instance="10.226.68.144:9100",job="node_exporter"}</code>

Let's explain the differences with the same metric **node_disk_io_time_seconds_total**:

- Query/Filter can be executed for all metrics supported by exporters. In this example, `node_disk_io_time_seconds_total` is a metric scraped from a `node_exporter`, hence we can query it from Prometheus directly;
- Beside the labels provided by an exporter (as above), Prometheus will add several more labels. In this example, 2 x labels are added: `instance`, `job`:
 - `instance`: this label is added to all exporters. It is the same as the **targets** configured for a scrape job;
 - `job`: this label is added to all exporters. It is the same as the job name as defined in `prometheus.yml`;
 - Additional labels can be added. Refer to [static_config](#) and [relabel_config](#)
- Advanced queries/filters can be achieved through the use of [PromQL](#).

4.3 Add Data Source

Grafana is only responsible for displaying time series metrics as graphs(within panels), it does not store metrics but retrieve metrics from data sources. Before using Grafana, the first step is adding at least a data source.

Grafana can use quite a lot systems as data sources, including Prometheus (our focus), Graphite, InfluxDB, etc. It is

easy to add a data source: **Configuration->Data Sources->Add Data Srouce->Prometheus->Input Inforamtion->Save & Test->Done**

4.4 Create Dashabord

Grafana organizes panels(each panel containing a graph) as dashboards. In other words, a dashboard is the container for holding graphs(within panels) - hence a dashabord need to be created before adding any graph. The creation of a dashboard is straightffoward: **Create->Dashboard**

Notes: Remember to save changes by clicking **Save dashboard** on the up right corner. Otherwise, your customization effort will be lost.

4.4.1 Varaiables

Dashboards have some special settings. The most important one is **Variables**. By defining variables, we can control the behavior of graphs within a dashboard flexibly but not hard coded.

Well defined variables should focus on extracting label values from metrics' labels, and graphs (panels) can leverage these to distinguish jobs, instances, metrics, etc. The most important builtin functions for this is **label_values**. We will cover the most common usage in this section, for knowledge not covered here, please refer to [Query variable](#).

Once varaibles are defined, they can be shown as choices (single or multiple selection) and graphs will change dynamically based on your chocies.

4.4.2 label_values

label_values is the function used to grab the value(s) of label(s) and turn the result into Grafana varaibles. Let's expain it with examples:

1. We have a metric from Prometheus as below:

```
node_disk_io_time_seconds_total{device="dm-0", instance="10.226.68.144:9100", job=
↪ "node_exporter"}
node_disk_io_time_seconds_total{device="dm-1", instance="10.226.68.144:9100", job=
↪ "node_exporter"}
node_disk_io_time_seconds_total{device="sda", instance="10.226.68.144:9100", job=
↪ "node_exporter"}
node_disk_io_time_seconds_total{device="sdb", instance="10.226.68.144:9100", job=
↪ "node_exporter"}
.....
```

2. If we want to extract the job name from the job label, we can call label_values as below:

```
label_values (node_disk_io_time_seconds_total, job)
```

The result will be simply "node_exporter"

3. If we want to extract the device name from the "device" label:

```
label_values (node_disk_io_time_seconds_total, device)
```

The result will be a list ["dm-0", "dm-1", ..., "sda", "sdb", ...]

4.4.3 Define Variables

To define variables: Select the dashboard->Dashboard settings->Variables->Add variable. The configuration page as below will be shown:

General

Name	job <small>The variable name</small>	Type	Query
Label	Job <small>The name will be shown on the dashboard for your selection</small>	Hide	

Query Options

Data source	prometheus69163 <small>data source</small>	Refresh	On Dashboard Load <small>how this variable refresh</small>
Query	label_values(node_disk_io_time_seconds_total, job) <small>the query function</small>		
Regex	/.*-(.*)-.*/		
Sort	Disabled		

Selection Options

Multi-value	<input type="checkbox"/>
Include All option	<input type="checkbox"/>

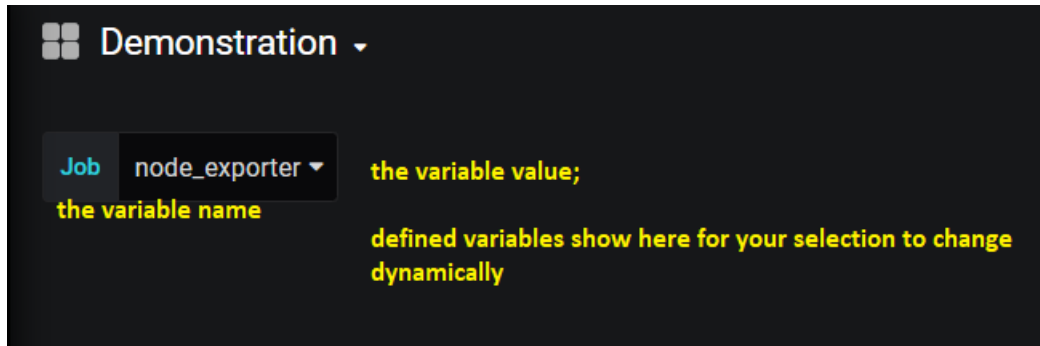
Value groups/tags (Experimental feature)

Enabled	<input type="checkbox"/>
---------	--------------------------

Preview of values

node_exporter	<small>the result: the variable values</small>
---------------	--

Once such a variable is defined and saved, a selection on the dashboard will be shown as below:



Variable Reference

We can refer to existing variables when we define new variables. E.g., we have defined a variable named “job”, then we can refer to it while we define new variable “disk” as below:

Name	disk	Type	Query
Label	Disk	Hide	

Query Options

Data source	prometheus69163	Refresh	On Dashboard Load
Query	label_values(node_disk_io_time_seconds_total{job="\$job"}, device)		
Regex	/.*-(.*)-*/ Refer to existing variables		
Sort	Disabled		

Selection Options

Multi-value	<input type="checkbox"/>
Include All option	<input type="checkbox"/>

Value groups/tags (Experimental feature)

Enabled ☐

Preview of values

dm-0	dm-1	dm-10	dm-11	dm-12	dm-13	dm-2	dm-3	dm-4	dm-5	dm-6	dm-7	dm-8	dm-9	sda	sdaa	sdab	sdac	sdad	sdae
------	------	-------	-------	-------	-------	------	------	------	------	------	------	------	------	-----	------	------	------	------	------

Show more

Please make sure referred to variables should be defined before variables who refer to them.

Multi-value and All

There are options as below while defining variables:

- Multi-value
- Include All option

They are mainly used for repeat operation. The behavior for them is as below:

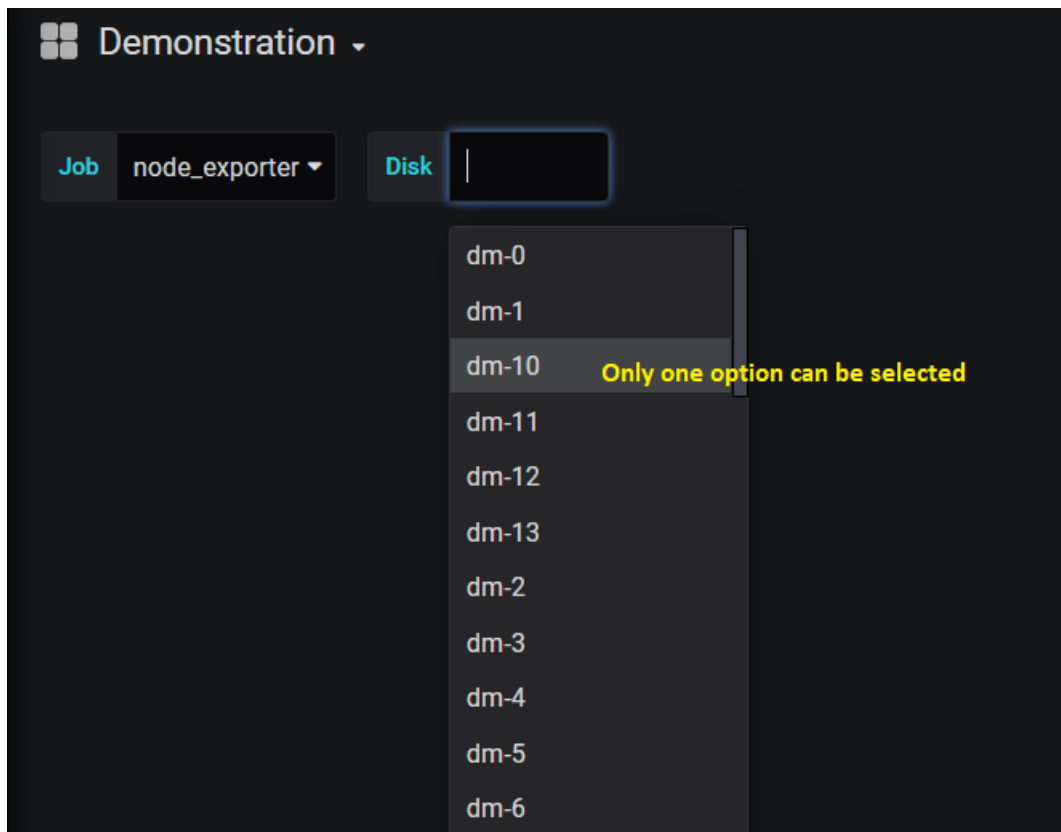
1. Let's say we have below metrics:

```
node_disk_io_time_seconds_total{device="dm-0", instance="10.226.68.144:9100", job=
↪ "node_exporter"}
node_disk_io_time_seconds_total{device="dm-1", instance="10.226.68.144:9100", job=
↪ "node_exporter"}
node_disk_io_time_seconds_total{device="sda", instance="10.226.68.144:9100", job=
↪ "node_exporter"}
node_disk_io_time_seconds_total{device="sdb", instance="10.226.68.144:9100", job=
↪ "node_exporter"}
.....
```

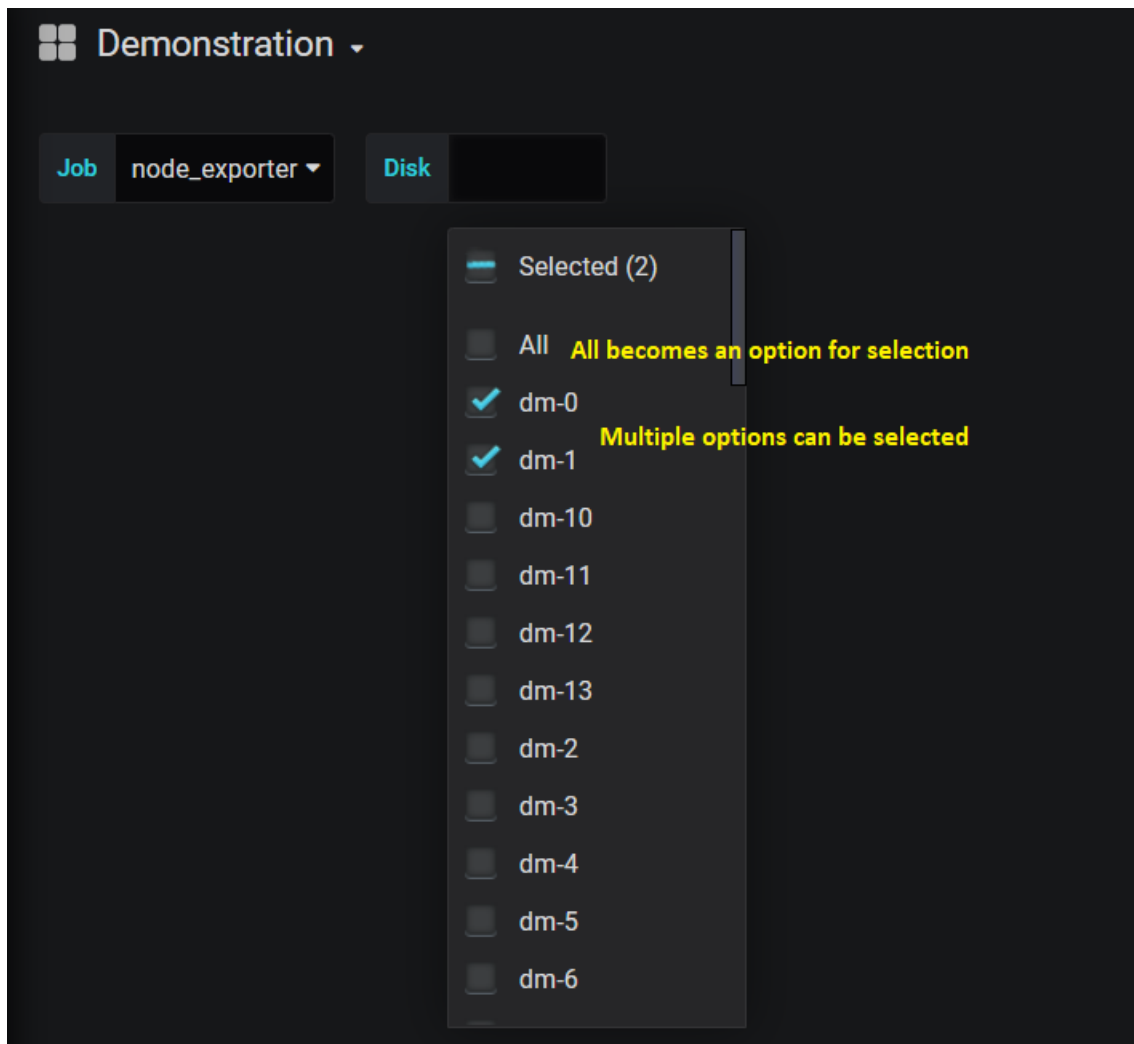
2. We have defined variable “job” and “disk” as below with label_values:

```
label_values(node_disk_io_time_seconds_total, job)
label_values(node_disk_io_time_seconds_total{job="$job"}, device)
```

3. Variable disk returns result as [“dm-0”, “dm-1”, ..., “sda”, “sdb”, ...]. By default, only one of them can be selected on the dashboard:



4. But when we turn on “Multi-value” and “Include All option”, we can select multiple options on the dashboard or select the all option which represent all results:



Extract values based on regular expression

Most of times, `label_values` works well extracting what we want. But sometimes, it is not possible to use it. Under such condition, we can use raw PromQL as below:

Query Options

Data source	prometheus69163	Refresh	On Dashboard Load
Query	node_disk_io_time_seconds_total{job="\$job"} Raw PromQL		
Regex	/*-(.*)-*/		
Sort	Disabled		

Selection Options

Multi-value	<input type="checkbox"/>
Include All option	<input type="checkbox"/>

Value groups/tags (Experimental feature)

Enabled	<input type="checkbox"/>
---------	--------------------------

Preview of values

Raw results which need further processing

```
node_disk_io_time_seconds_total{device="dm-0",instance="10.226.68.144:9100",job="node_exporter"}
node_disk_io_time_seconds_total{device="dm-1",instance="10.226.68.144:9100",job="node_exporter"}
node_disk_io_time_seconds_total{device="dm-10",instance="10.226.68.144:9100",job="node_exporter"}
```

The tricky thing here is the result is a list of raw PromQL results. If we want to extract what we want, we need to use regular expression to grab them out:

Query Options

Data source

prometheus69163

Refresh

On Dashboard Load

Query

node_disk_io_time_seconds_total{job="\$job"}

Regex

/device="(dm-.+?)/

Use RE to extract what we want with parenthesis

Sort

Disabled

Selection Options

Multi-value

Include All option

Value groups/tags (Experimental feature)

Enabled

Preview of values

The results after processing

dm-0

dm-1

dm-10

dm-11

dm-12

dm-13

dm-2

dm-3

dm-4

dm-5

dm-6

dm-7

dm-8

dm-9

Actually, regular expression also works smoothly with label_values:

Query Options

Data source	prometheus69163	Refresh	On Dashboard Load
Query	label_values(node_disk_io_time_seconds_total{job="\$job"}, device)		
Regex	/(dm-.+?)/ RE can be used to process results of label_values too		
Sort	Disabled		

Selection Options

Multi-value	<input checked="" type="checkbox"/>
Include All option	<input checked="" type="checkbox"/>
Custom all value	blank = auto

Value groups/tags (Experimental feature)

Enabled ☐

Preview of values

All dm-0 dm-1 dm-2 dm-3 dm-4 dm-5 dm-6 dm-7 dm-8 dm-9

4.4.4 Add Panel

Once variables have been defined, we can go ahead defining graphs by adding panels. Assume we have define below variables:

Variables		New	
Variable	Definition		
\$job	label_values(node_disk_io_time_seconds_total, job)	↓ Duplicate	×
\$disk	label_values(node_disk_io_time_seconds_total{job="\$job"}, device)	↑ ↓ Duplicate	×
\$cpu	label_values(node_cpu_seconds_total{job="\$job"}, cpu)	↑ ↓ Duplicate	×
\$cpu_mode	label_values(node_cpu_seconds_total{job="\$job"}, mode)	↑ Duplicate	×

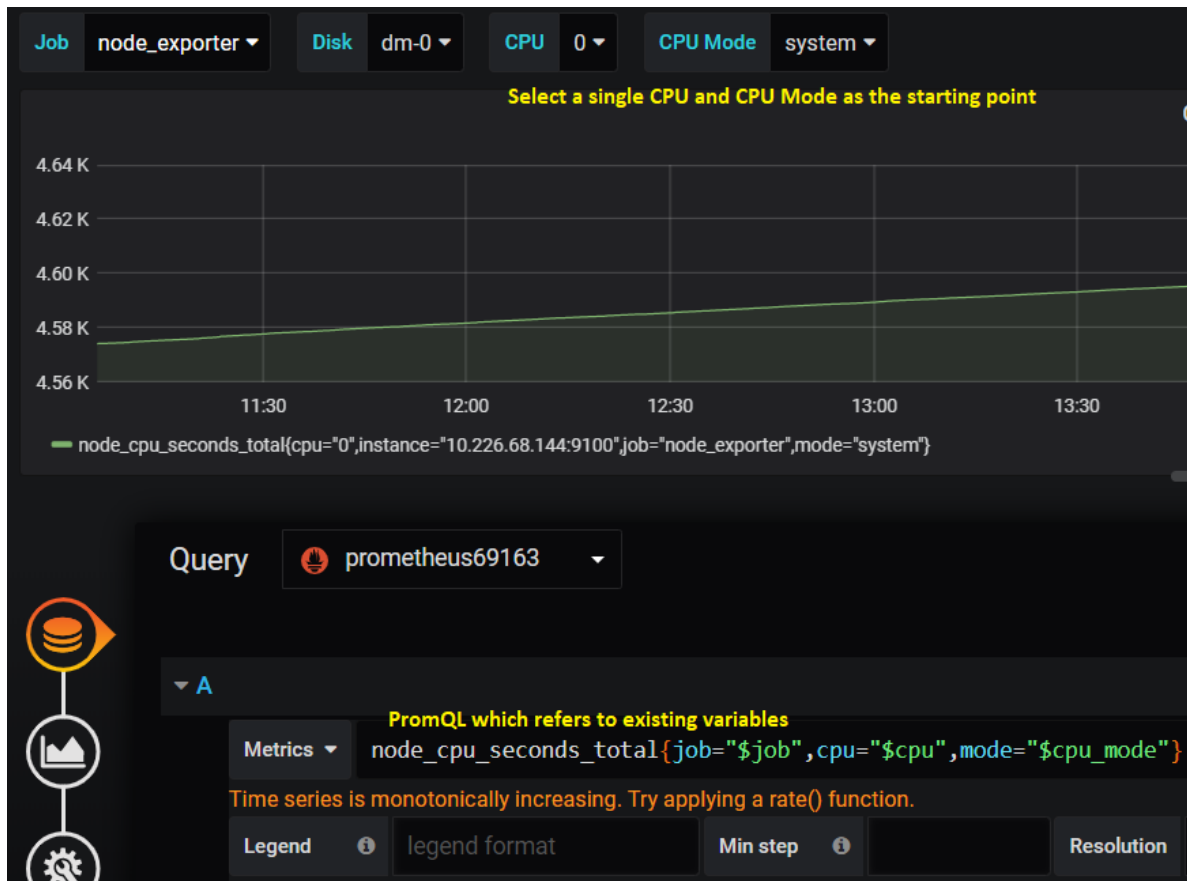
Singlestat

The most basic graph supported by Grafana is “Singlestat”. It is used mainly for simple metric like uptime, cpu usage, etc. Let’s explain this with an example.

1. Let’s say we have a metric as below, we want to show it as a “Singlestat”:

```
node_cpu_seconds_total{cpu="0",instance="10.226.68.144:9100",job="node_exporter",
↔mode="idle"}
node_cpu_seconds_total{cpu="0",instance="10.226.68.144:9100",job="node_exporter",
↔mode="iowait"}
node_cpu_seconds_total{cpu="0",instance="10.226.68.144:9100",job="node_exporter",
↔mode="system"}
node_cpu_seconds_total{cpu="1",instance="10.226.68.144:9100",job="node_exporter",
↔mode="idle"}
node_cpu_seconds_total{cpu="1",instance="10.226.68.144:9100",job="node_exporter",
↔mode="iowait"}
node_cpu_seconds_total{cpu="1",instance="10.226.68.144:9100",job="node_exporter",
↔mode="system"}
...
```

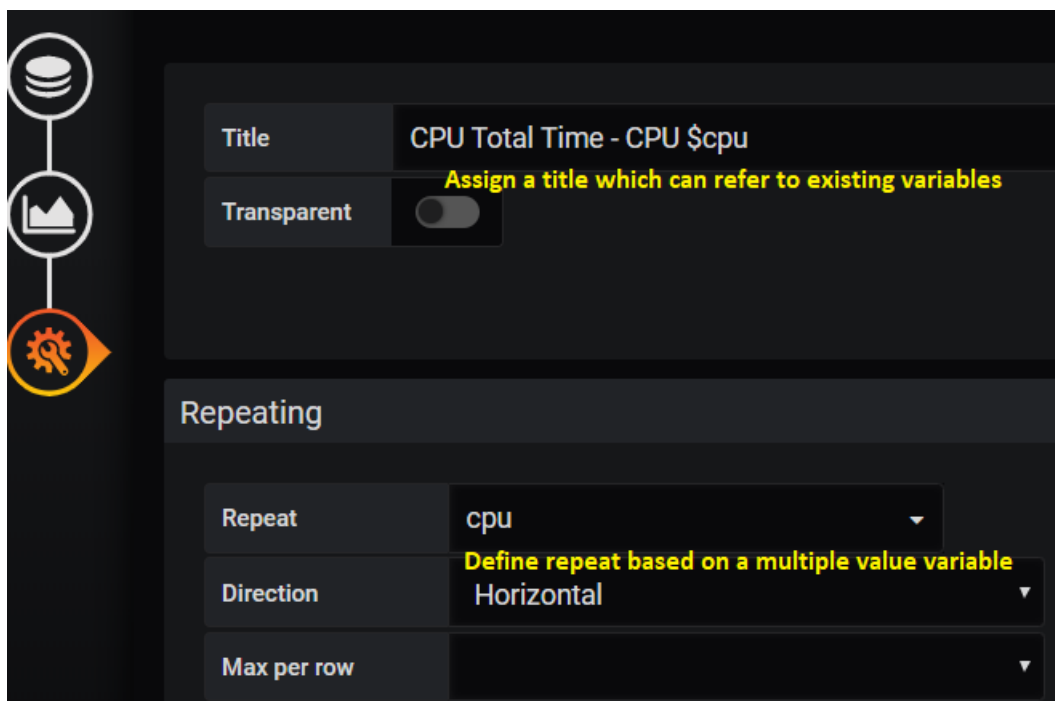
2. Add a panel by opening the dashboard->Add a panel: the page is as below, we can add our metric accordingly as the first step:



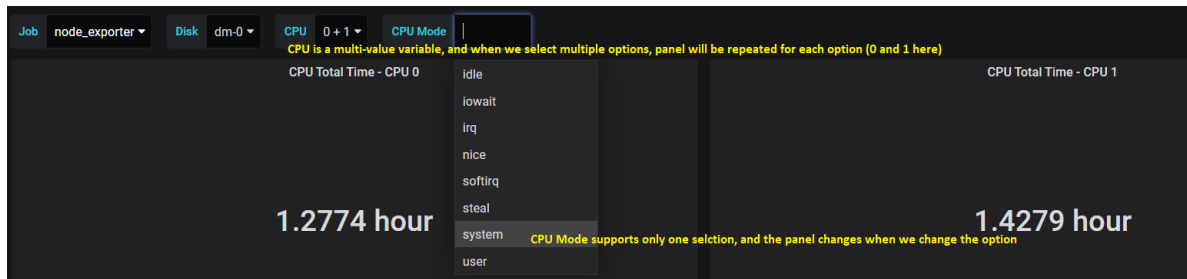
3. Then the “Singlestat” can be selected and the unit can be adjusted:



4. The last step can be used to assign a name to the grapha/panel, and the repeat scenario can be set based on defined variables:



5. After saving the graph/panel settings, we can see it from the dashboard:



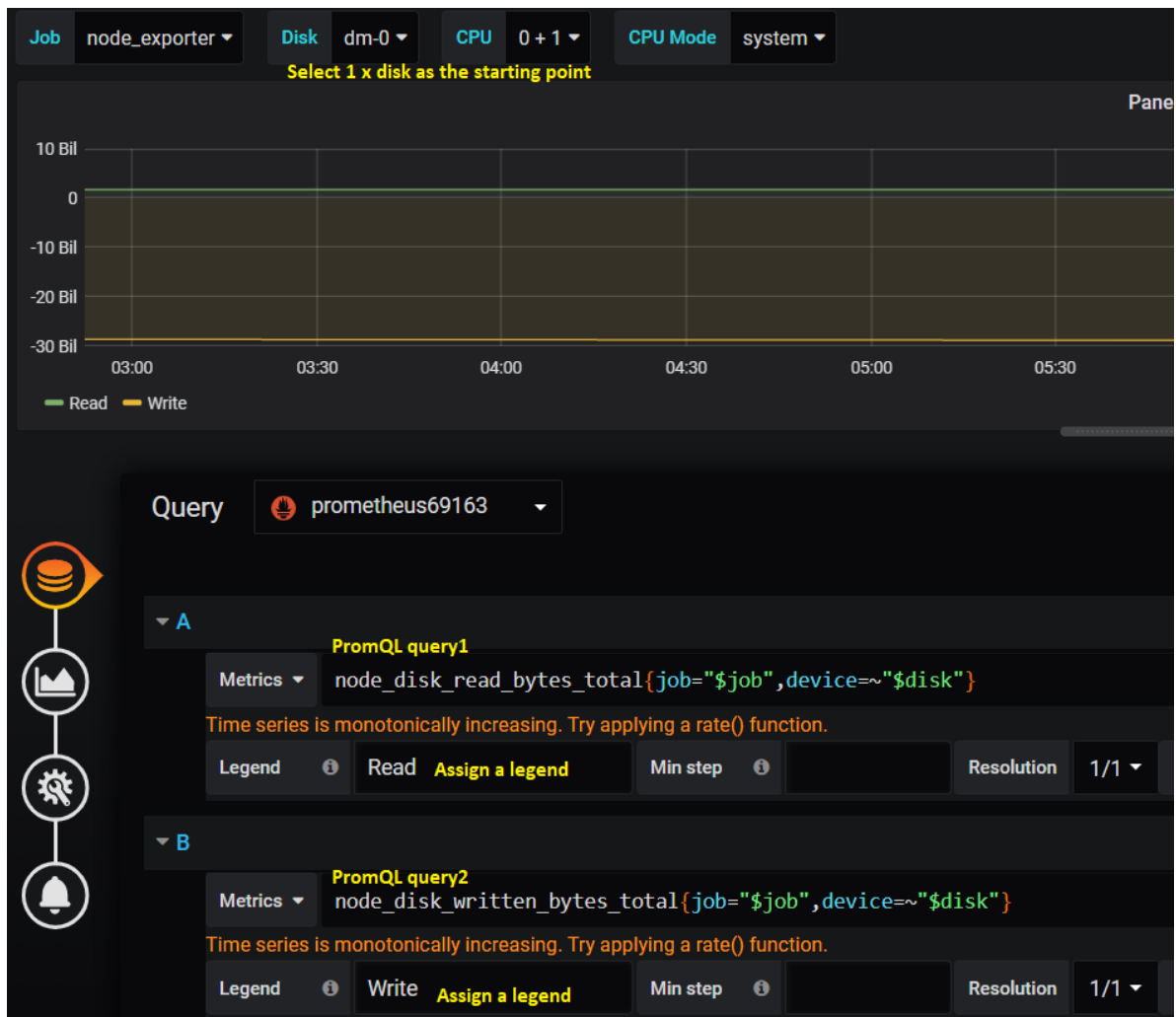
Graph

Graph actually stands for line chart in Grafana. It is used frequently to reflect metric changes. Let's explain it with an example like before.

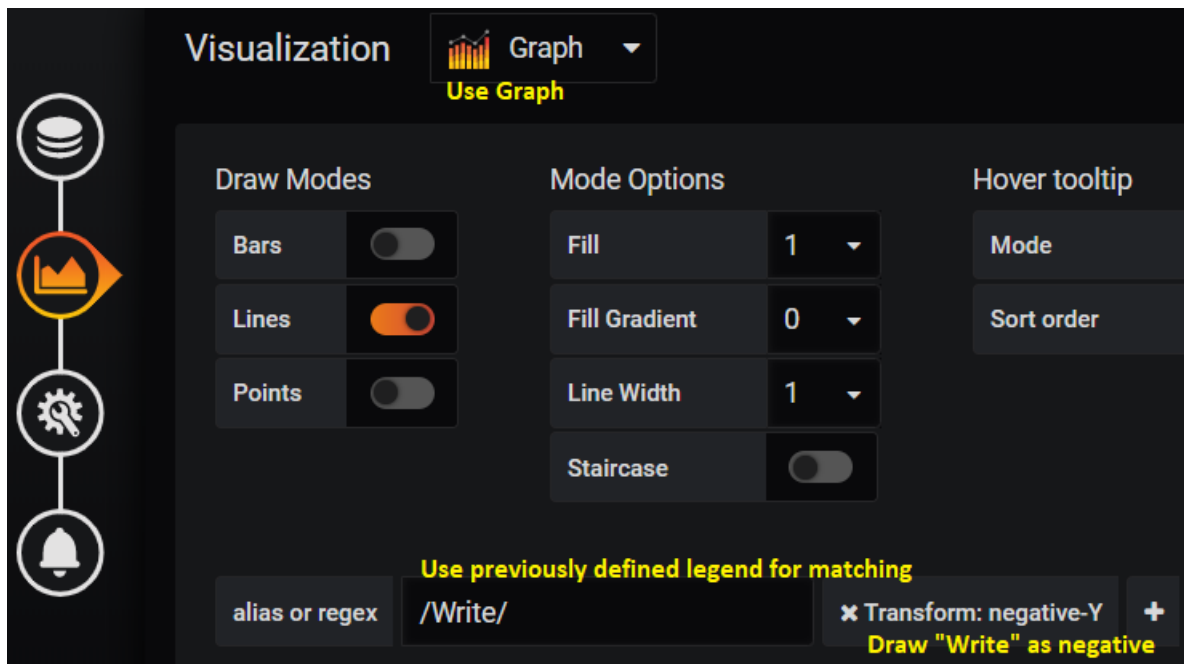
1. Let's say we have 2 x metrics (node_disk_read_bytes_total, node_disk_written_bytes_total) as below, we want to show them in the same panel as a "Graph":

```
node_disk_read_bytes_total{device="dm-0", instance="10.226.68.144:9100", job="node_
↪exporter"}
node_disk_read_bytes_total{device="dm-1", instance="10.226.68.144:9100", job="node_
↪exporter"}
.....
node_disk_written_bytes_total{device="dm-0", instance="10.226.68.144:9100", job=
↪"node_exporter"}
node_disk_written_bytes_total{device="dm-1", instance="10.226.68.144:9100", job=
↪"node_exporter"}
...
```

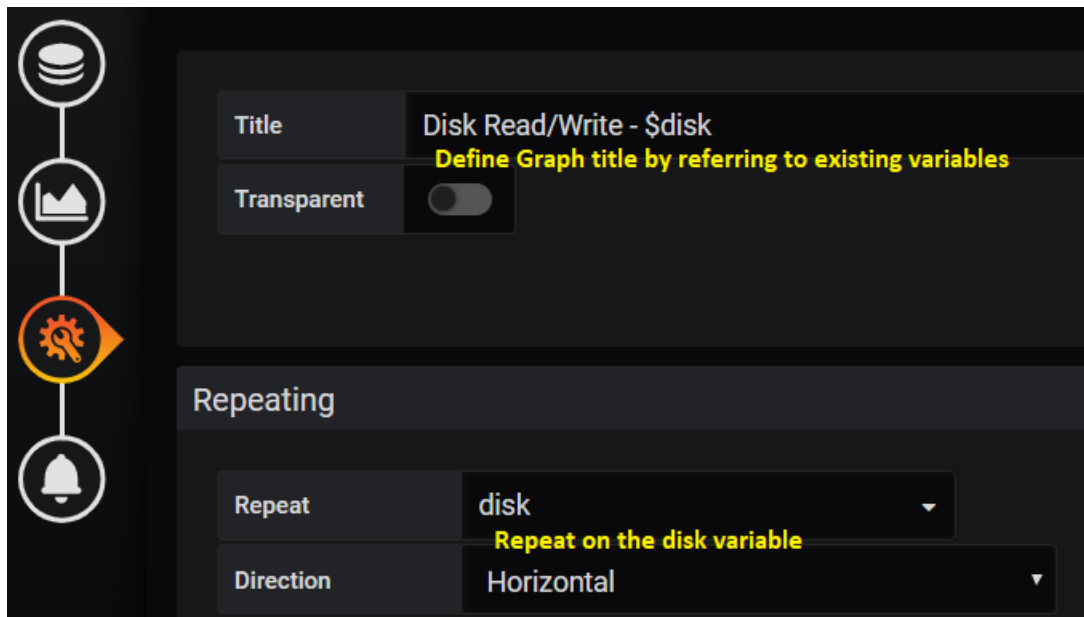
2. Add a panel and add our metrics: we define 2 x metrics here:



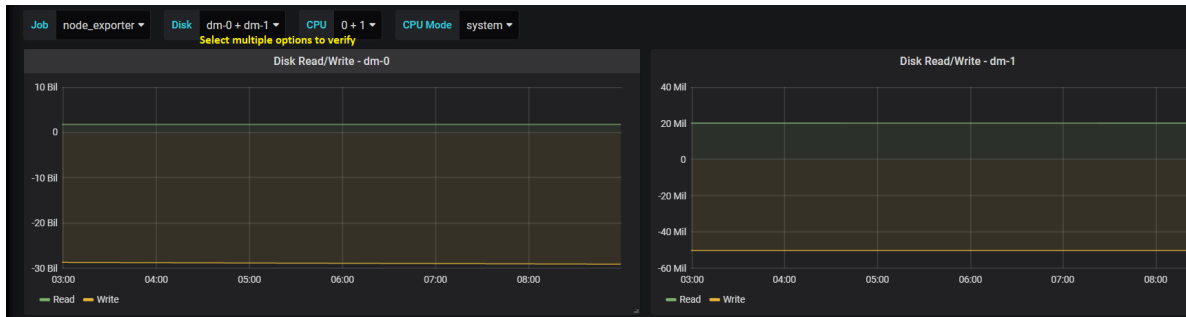
3. Select "Graph" as the visualization type and transform the data display (metric 1(Read) above y=0 and metric 2(Write) below y=0, this is not required but just a trick):



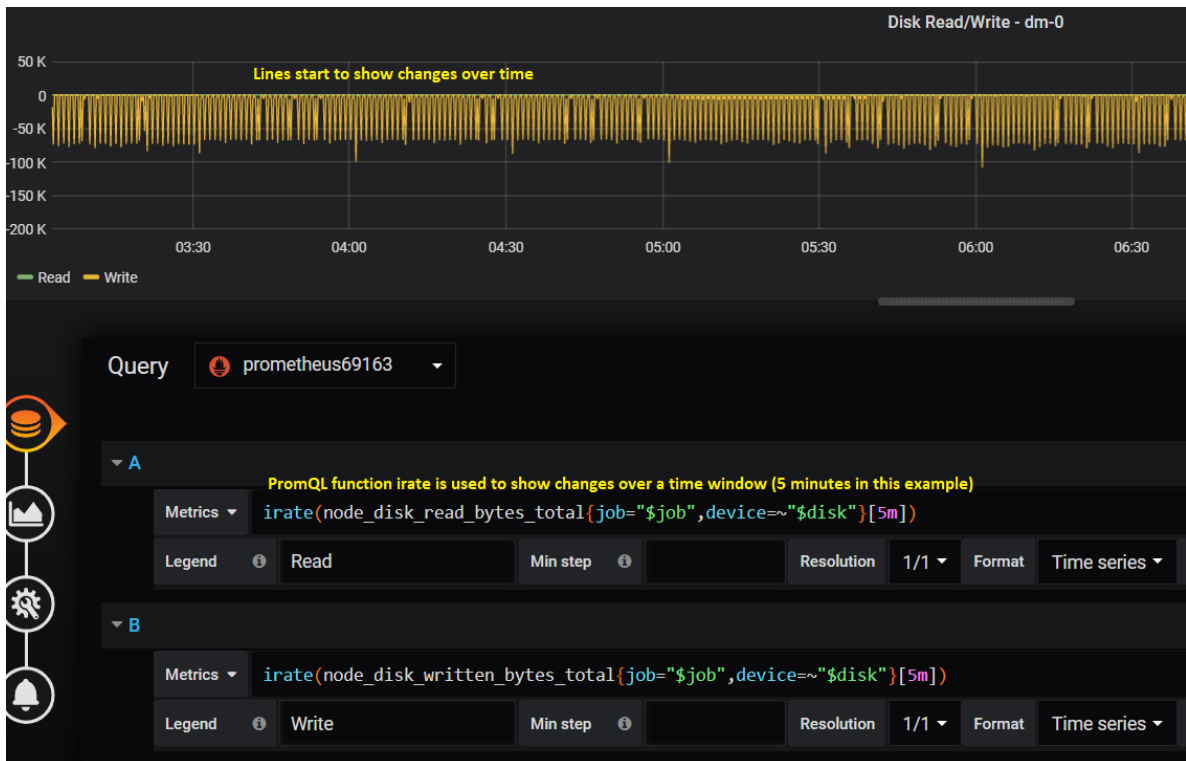
4. Define a title and specify the repeating:



5. After saving the changes, we can see the update from the dashboard:



6. This is great. However, it is easy to find the lines within the graph are flat which cannot show the change over time. Under such condition, [PromQL functions](#) can be used to achieve the goal:



4.5 Grouping and Group Repeating

While more and more panels are being added onto a dashboard, we need a mechanism to group them based on logical or other criterias in order to quick focus on the metrics we care about. In the meanwhile, we also need the functionality to repeat a set of panels over some higher level variables.

For example, we have a metric as below:

```
node_cpu_seconds_total{cpu="0",instance="10.226.68.144:9100",job="node_exporter",mode=
  ↳ "idle"}
node_cpu_seconds_total{cpu="0",instance="10.226.68.144:9100",job="node_exporter",mode=
  ↳ "iowait"}
node_cpu_seconds_total{cpu="0",instance="10.226.68.144:9100",job="node_exporter",mode=
  ↳ "irq"}
.....
```

(continues on next page)

(continued from previous page)

```
node_cpu_seconds_total{cpu="1",instance="10.226.68.144:9100",job="node_exporter",mode=
↪ "idle"}
node_cpu_seconds_total{cpu="1",instance="10.226.68.144:9100",job="node_exporter",mode=
↪ "iowait"}
node_cpu_seconds_total{cpu="1",instance="10.226.68.144:9100",job="node_exporter",mode=
↪ "irq"}
.....
```

With the knowledge we learned from the previous sections, we are already able to “repeat” panels based on a single label like “cpu” or “mode”. But how we can repeat the same for multiple labels together (repeat over both “cpu” and “mode” labels)?

Grafana supports the requirements through the use of **Row**, which is used to group panels and repeat such groups based on a variable. When **Row** repeat is used together with the normal repeat option of a **Panel**, we can achieve our goal to repeat a panel over multiple labels at the same time.

It is straightforward to use the feature:

1. Check our previously defined variable “cpu”:

Query Options

Data source	prometheus69163	Refresh	On Dashboard Load
Query	label_values(node_cpu_seconds_total{job="\$job"}, cpu)		
Regex	/.*(.*)-.* /		
Sort	Disabled		

Selection Options

Multi-value	<input checked="" type="checkbox"/>	Enable at least one of these options, otherwise, repeat won't work
Include All option	<input checked="" type="checkbox"/>	
Custom all value	blank = auto	

Value groups/tags (Experimental feature)

Enabled	<input type="checkbox"/>
---------	--------------------------

Preview of values

All	0	1
-----	---	---

2. Check our previously defined variable “cpu_mode” and make it supports multiple selection:

Query Options

Data source	prometheus69163	Refresh	On Dashboard Load
Query	label_values(node_cpu_seconds_total{job="\$job"}, mode)		
Regex	/.*-(.*)-.*		
Sort	Disabled		

Selection Options

Multi-value	<input checked="" type="checkbox"/>	Enable at least one of this option
Include All option	<input checked="" type="checkbox"/>	
Custom all value	blank = auto	

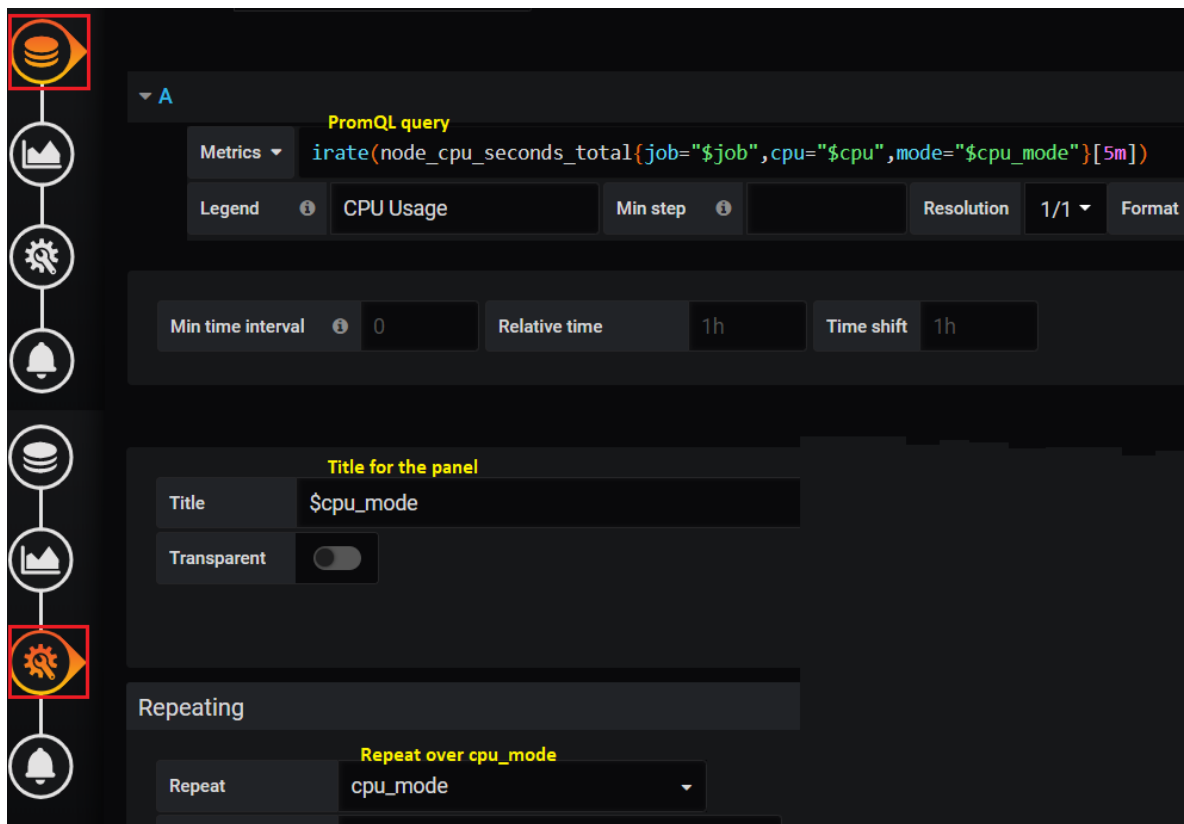
Value groups/tags (Experimental feature)

Enabled ☐

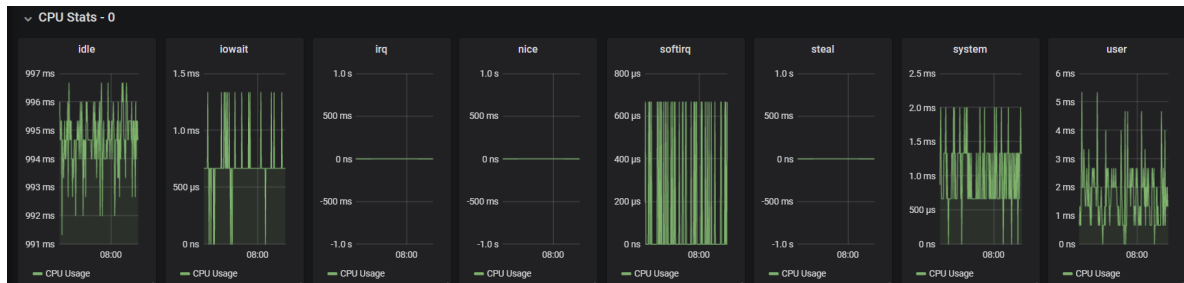
Preview of values

All idle iowait irq nice softirq steal system user

3. Select the dashboard->Add panel-> Convert to row->Unfold the row;
4. Add panel again, and drag the new panel into the Row: panels get organized into groups;
5. For the newly created panel, click "Add Query" and define it as below:

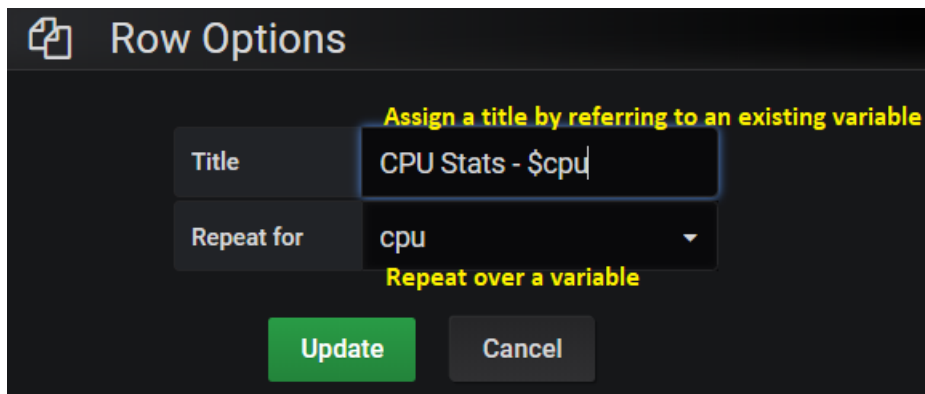


6. After saving, we have a Row as below:

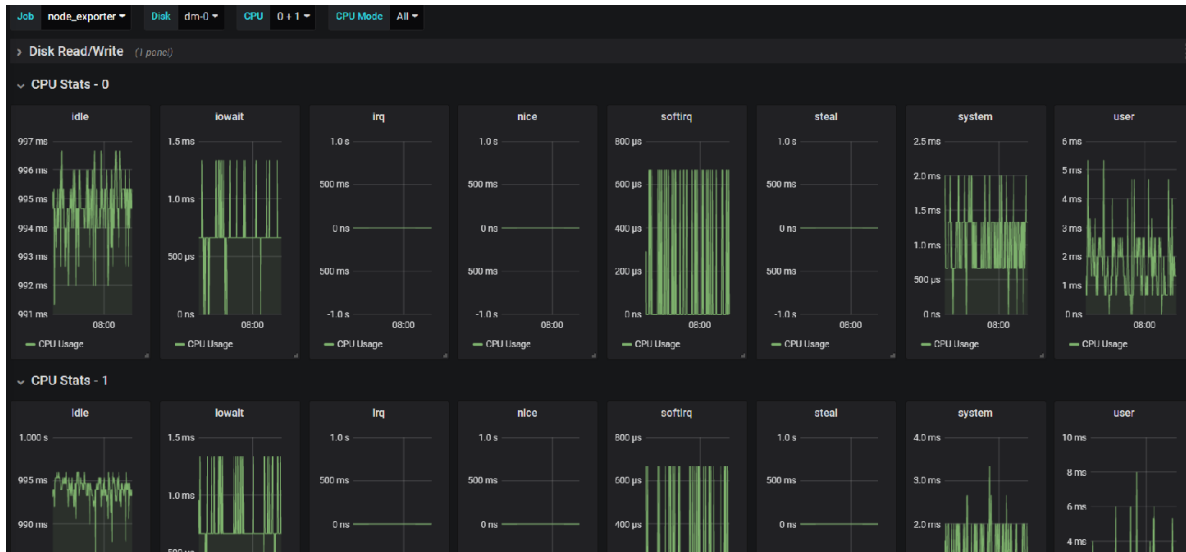


7. Fold the Row object, then itself can be dragged up/down by clicking the right end of the Row object;

8. Hover the mouse over the Row object, then click settings. Here, we can assign a name and select the variable we want to repeat the group of panels based on (cpu here):



9. Save the changes and refresh the dashboard. After selecting multiple CPU (or “ALL”), we get our dashboard changed automatically as below:



10. Done! It is time for you to practice.

4.6 Save Dashboard Settings

A dashboard can be described as a JSON document, hence it can be saved, shared and resotred easily. To export a dashboard: select the dashboard->Share dashboard->Eport->Save to file.

4.7 Reference

- [Query Prometheus](#)
- [Grafana Templating Variables](#)
- [Using Prometheus in Grafana](#)
- [Great PromQL Examples with Diagrams](#)

Alerting is an important part of monitoring. For Prometheus, alerting is separated into two parts:

- Alerting rules:
 - Defined on Prometheus servers;
 - Get triggered when rule expressions are met;
 - Sent to Alertmanager once triggered.
- Alertmanager:
 - Manage alerts, such as silencing, inhibition, aggregation;
 - Notifications.

5.1 Alertmanager Deployment

Alertmanager can be downloaded from Prometheus [official download page](#). The deployment process is as easy as:

1. Decompress the tarball;
2. Start the service: `./alertmanager`
3. The service should be accessible from [http://<FQDN or IP>:9093](#)
4. To enable notification (send emails in our example), change the configuration as below (refer to [the official example](#)):

```
# alertmanager.yml
global:
  resolve_timeout: 5m
  smtp_smarthost: '<smtp server>:<smtp server port>'
  smtp_from: '<the default sender email>'

route:
```

(continues on next page)

(continued from previous page)

```

group_by: ['alertname']
group_wait: 10s
group_interval: 10s
repeat_interval: 1h
receiver: monitor-admin

receivers:
- name: 'monitor-admin'
  email_configs:
  - to: '<receiver_email>'
    tls_config:
      insecure_skip_verify: true

```

5. Restart Alertmanager

6. Done

For more information, refer to [Alertmanager introduction](#) for details.

5.2 Alerting Rules

Alerting rules are configured on Prometheus servers:

1. Tune Prometheus configurations:

```

global:
  evaluation_interval: 30s # How frequently to evaluate rules, 1m as default

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - alertmanager:9093

# Load rules once and periodically evaluate them according to the global
↪ 'evaluation_interval'.
rule_files:
- "rules/sample.yml" # Define rules in files

```

2. Create rule definition files (rules/samples in this example):

```

groups:
- name: UnityCapacity
  rules:
  - alert: HighCapacity
    expr: physical_used{job="unity-capacity"} / physical_total{job="unity-capacity"} > 0.8
    ↪
    for: 1m
    labels:
      severity: p1
    annotations:
      summary: High Capacity Usage
      description: "High Capacity Usage {{ $labels.instance }} {{ $value }}"

```

3. Restart Prometheus or “kill -s SIGHUP *pgrep prometheus*”

4. Done

Notes: notification supports templating, A.K.A embedding variables into static paragraph. It is an important feature should be understood. Please refer to [examples](#) for the usage.

This chapter will be used to record tips during the usage of Prometheus.

6.1 Install Prometheus as a Systemd Service

Configuring a systemd service to control prometheus running is much handy than running it from the CLI as an executable binary. This tip shares the common steps to create a systemd service for Prometheus.

```
sudo useradd --no-create-home --shell /bin/false prometheus

tar -zxvf prometheus-xxxxxx.linux-amd64.tar.gz
sudo mv prometheus-xxxxxx.linux-amd64 /opt/prometheus
sudo mkdir /opt/prometheus/data
sudo chown -R prometheus:prometheus /opt/prometheus

sudo mkdir /etc/prometheus
sudo cp /opt/prometheus/prometheus.yml /etc/prometheus
sudo chown -R prometheus:prometheus /etc/prometheus

sudo cat > /etc/systemd/system/prometheus.service <<-EOF
[Unit]
Description=Prometheus
Wants=network-online.target
After=network-online.target

[Service]
User=prometheus
Group=prometheus
Type=simple
Restart=on-failure
ExecStart=/opt/prometheus/prometheus \
--config.file /etc/prometheus/prometheus.yml \
--storage.tsdb.path /opt/prometheus/data \
```

(continues on next page)

(continued from previous page)

```
--storage.tsdb.retention.time 15d

[Install]
WantedBy=multi-user.target
EOF

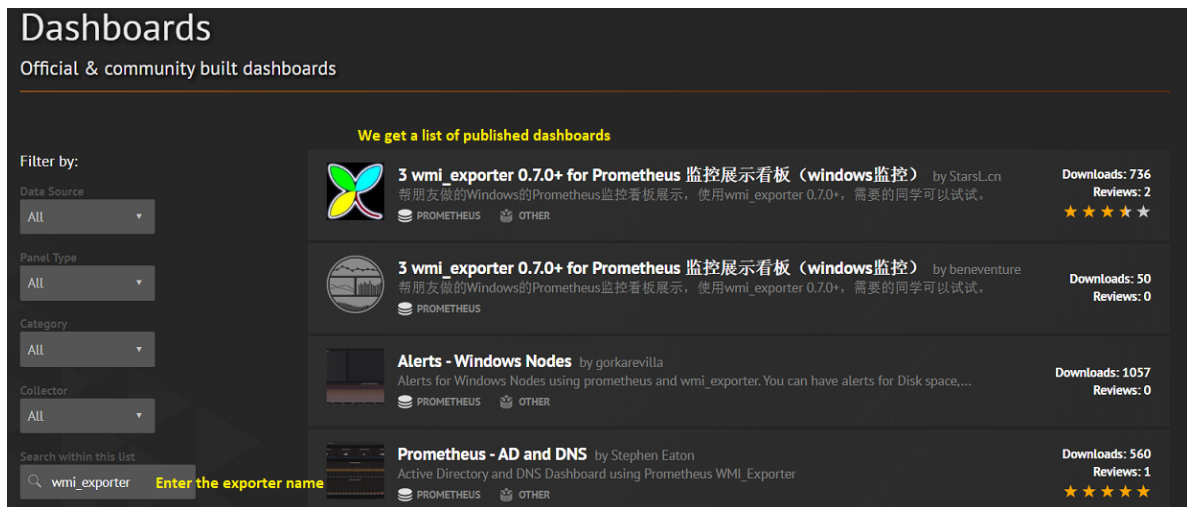
sudo systemctl daemon-reload
sudo systemctl start prometheus
```

6.2 Import Community Defined Grafana Dashboard

Grafana is powerful allowing end users define dashboards based on their requirements. However, this does not mean we have to define our dashboards from scratch all the time (or import dashboards we ourselves previously defined). Actually, we can import dashboards defined by the community easily.

Here is an example how to import a community defined dashboard:

1. Find available dashboards published by the community [here](#);
2. E.g., we want to create a dashboard for the wmi_exporter:



3. Click the first dashboard which is the most popular one and we can see its description together with a **dashboard ID**:



4. Copy the dashboard ID, then go to our dashboard GUI: Create->Import->Paste the dashboard ID->Wait for a while->Name it and select the Prometheus data source->Import;
5. We have a working dashboard now:



6. If the dashboard does not meet your requirements, modify it!
7. Done!

6.3 Add New Labels

This tip shows 2 x methods to add labels.

- The original configuration: label “node_type” is added to both targets with the same value.

```
- job_name: 'node_exporter'
  static_configs:
    - targets:
      - '192.168.10.10:9100'
      - '192.168.10.11:9100'
      labels:
        node_type: 'unity_node'
```

- Add different labels by splitting targets:

```
- job_name: 'node_exporter'
  static_configs:
    - targets:
        - '192.168.10.10:9100'
      labels:
        node_type: 'unity_node'
        node: node1
    - targets:
        - '192.168.10.11:9100'
      labels:
        node_type: 'unity_node'
        node: node2
```

- Add different labels by using `relabel_configs`:

```
- job_name: 'node_exporter'
  static_configs:
    - targets:
        - '192.168.10.10:9100'
        - '192.168.10.11:9100'
      labels:
        node_type: 'unity_node'
  relabel_configs:
    - source_labels: [__address__]
      regex: '.*?\.[10]:9100'
      target_label: 'node'
      replacement: 'node1'
    - source_labels: [__address__]
      regex: '.*?\.[11]:9100'
      target_label: 'node'
      replacement: 'node2'
```

6.4 Select Legends to Display on Grafana Panel

- Click the color icon “-” of a legend on a panel:
 - Select the color to be used
 - Customize the color to be used
 - Align the legend to left/right Y axis
- Click the name of a legend
 - Only this legend will be displayed on the panel
 - Click again, all legends will be displayed as before
- Shift + Click legends: select multiple legends to display on the panel
- Ctrl + Click legends : select multiple legends to not display

6.5 Graph Top N in Grafana

PromQL **topk** will show more than expected results on Grafana panels because of [this issue](#).

The problem can be worked around by defining a variable containing the top N results, then filter query results with this variable in Panel. The details can be found [here](#).

Below is a straightforward example:

1. Metrics:

- disk_read_average

```
disk_read_average{instance="192.168.10.11:9272", job="vcenter", vm_name="vm1"}
disk_read_average{instance="192.168.10.11:9272", job="vcenter", vm_name="vm2"}
...
disk_read_average{instance="192.168.10.11:9272", job="vcenter", vm_name="vm100"}
```

- disk_write_average

```
disk_write_average{instance="192.168.10.11:9272", job="vcenter", vm_name="vm1"}
disk_write_average{instance="192.168.10.11:9272", job="vcenter", vm_name="vm2"}
...
disk_write_average{instance="192.168.10.11:9272", job="vcenter", vm_name="vm100"
↪ }
```

2. Goal: show disk I/O (read + write) for the top 5 x VMs

3. Define a variable (top_vm_io) which returns the top 5 x VMs

```
# Query
query_result(topk(5, avg_over_time((disk_read_average + disk_write_average) [${__
↪ range_s}s:])))
# Regex
/vm_name="(.*)" /
# Enable "Multi-value" and "Include All option"
```

4. Panel query

```
disk_read_average{vm_name=~"$top_vm_io"} + disk_write_average{vm_name=~"$top_vm_io"
↪ }
```

Notes:

- PromQL functions avg_over_time/min_over_time/max_over_time: should be selected based on the use case;
- __range_s is a builtin variable, refer [here](#) for details;
- [\${__range_s}s:] is a subquery, refer [here](#) for details.

6.6 Use Telegraf as Exporters

Telegraf is a part of the [TICK Stack](#) monitoring solution. Telegraf supports collecting metrics from different sources through input plugins and shipping metrics to different destinations through output plugins.

Prometheus is a supported output destination, in other words, Telegraf can be used as Prometheus exporters. It supports a large num. of input plugins, including OSs, databases, clouds, etc.

Usage:

- List supported input plugins:

```
telegraf --input-list
```

- List supported output plugins:

```
telegraf --output-list
```

- Generate a config with vSphere input plugin and Prometheus output plugin:

```
telegraf --section-filter agent:inputs:outputs --input-filter vsphere --output-  
↪filter prometheus_client config | tee telegraf.conf
```

- Run Telegraf:

```
# After tuning the config  
telegraf --config telegraf.conf
```

6.7 Collect Metrics with Arbitrary Scripts

Sometimes, it is not cost effective to implement a Prometheus exporter with Go/Python. For example, to collect CPU related metrics on Linux - a shell script will do the work much easier and will take less time than implementing a Prometheus exporter.

Scripts are great, but it is a must to make their outputs both understandable and acceptable to Prometheus. Generally speaking, there are 2 x feasible ways:

- Prometheus Pushgateway: push metrics to Prometheus Pushgateway (then metrics are collected from Pushgateway by Prometheus “pull”) with any kind of scripts as long as their outputs are in line with the required format. It is recommended to be used for ephemeral and batch jobs. For more information, refer to [Prometheus Pushgateway](#);
- Telegraf Exec Input Plugin: Telegraf gets the capability to collect metrics by running arbitrary commands/scripts periodically. Refer to [Exec Input Plugin](#) for details.

6.8 Use Alerta to Manage Alerts

Prometheus Alertmanager is a really powerful alerting management solution, however, its GUI is not suitable for daily operations. [Alerta](#) provides a great interface consolidating alerts for main monitoring solutions like Prometheus, Nagios, Zabbix, etc.

Alerta makes use of Alertmanager which sends alerts as notifications to Alerta through webhooks. The deployment is easy and straightforward, please refer to [the official document](#) for details.

6.9 Show Diagrams on Grafana Panel

It is smart to show the relationships of monitored targets. A plugin named [Diagram Panel](#) can be used for this purpose. Below is a simple example:

1. Install the plugin:

```
pkill grafana-server  
./bin/grafana-cli --pluginsDir=./data/plugins plugins install jdbranhm-diagram-  
↪panel  
./bin/grafana-server
```

2. Add panel->Add Query->Leave the “Queries” tab as default (empty);

3. Visualization->Diagram;
4. Define a diagram with mermaid syntax:
 - Refer to [Mermaid Flowcharts Syntax](#) for syntax details;
 - Use [Mermaid Live Editor](#) to define the chart;
 - Copy the chart code (Mermaid markdown) and paste it to the panel;
5. Done.

6.10 The Built-in “up” Metric

up is a built-in Prometheus metric. Each instance has a “up” metric indicates:

- 1: the instance is healthy
- 0: the instance can not be scraped

It can be used to grab job and instance information while defining variables with Grafana. For more details, refer to [JOBS AND INSTANCES](#).

6.11 Scrape Interval Pitfall

For a single job with multiple scrapping targets, the “scrape_interval” should not be set with a too large value (the exact maximum value needs to be determined by careful tunings).

The background reason needs to be clarified with an example:

```
# Assume a blackbox_exporter job is configured with 100 x ICMP probe targets,
↪as below:
- job_name: 'blackbox_exporter'
  scrape_interval: 60m
  metrics_path: /probe
  params:
    module: [icmp]
  static_configs:
    - targets:
      - 192.168.10.101
      - ...
      - 192.168.10.200
      labels:
        type: server
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: blackbox1.lab1.local:9115
    - source_labels: [__address__]
      target_label: exporter
```

This configure works, however, PromQL (such as “probe_success” for blackbox_exporter) won’t return 100 x success/fail results as expected. The reason is that the “scrape_interval” is 60 minutes, which means the job can take

its time to complete the full scraping within this big time window. However, PromQL instant query only returns values which fall in a small time window comparing with the current timestamp (several minutes, may be related with PromQL resolution and step???) - although all targets have been scraped, but their results are distributed within the big time window (1 x hour in this example). When a query is run, some scraping results are far from the current timestamp and won't be included within the query results.

Because of the reason just mentioned as above, the scrape interval should not be set with a huge value. The exact maximum needs to be determined by tuning the scrape interval and run PromQL accordingly until the expected behavior is gotten.

6.12 Reload Prometheus through HTTP POST

Prometheus can reload its configuration file after getting a HUP signal. In the meanwhile, the same behavior can be triggered by sending as HTTP POST as below:

```
# Prometheus must be started with option "--web.enable-lifecycle"
curl -X POST http://<IP>:9090/-/reload
```

6.13 Add a Label with PromQL

PromQL can be used to add a label on the fly. This is useful for the “Outer join” transformation - make sure the fields (columns) with the same names can be referred to by different names. Refer to [Duplicate column names in table panel](#) for the background.

```
# The original metric
accessible{host="10.226.68.185",instance="192.168.56.10:9091",job="pushgateway",type=
↪ "linux" }

# PromQL
label_replace(accessible, "type1", "$1", "type", "(.*)")

# The result
accessible{host="10.226.68.185",instance="192.168.56.10:9091",job="pushgateway",type=
↪ "linux", type1="linux" }
```