Ch13. Using PromQL

PromQL은 Query Language이지만, SQL계열은 아니다. 시계열의 특징이 강한 프로메테우스에서 Label은 PromQL의 핵심이며, Label을 중심으로 각종 집계, 산술연산, 다른 메트릭과의 Join 및 수학함수의 사용까지 다양한 기능을 갖추고 있다.

우선은 PromQL의 기본 기능에 대해 알아 본다.

pre-Reference

- PromQL Cheat-sheet: https://promlabs.com/promql-cheat-sheet/

13.1. Aggregation Basics

Guage, Counter, Summary, Histogram 데이터에 대한 Aggregation(집계)를 수행하는 PromQL은 필요한 거의 모든 것을 처리할 수 있지만, 또한 대부분은 PromQL은 단순한 처리에 사용된다.

Gauge

Guage는 특정 상태(state)의 스냅샷이며, 일반적으로 값, 평균, 최소값, 최대값 등을 구하여 모니터링 하기 위해 Guage 값들을 집계한다. 아래의 내용을 Prometheus PromQL Expression에서 확인해 보자.

- Label을 없애가며, 합계를 만들어 내는 패턴. 동일한 Metric에 대해서 처리해 보자.
  - Node Exporter의 `node_filesystem_size_bytes` (추가 유사 메트릭들: `node_filesystem_avail_bytes`, `node_filesystem_free_bytes`)

```
node_filesystem_size_bytes
sum (node_filesystem_size_bytes)
sum without(device, fstype)(node_filesystem_size_bytes)
// 아래 2가지 경우는 instance, job별로 집계된 합계를 보여 준다.
sum without(device, fstype, mountpoint)(node_filesystem_size_bytes)
sum without(device, fstype, mountpoint, instance)(node_filesystem_size_bytes)

node_filesystem_free_bytes
sum without(device, fstype)(node_filesystem_free_bytes)
sum without(device, fstype, mountpoint)(node_filesystem_free_bytes)
sum without(device, fstype, mountpoint, instance)(node_filesystem_free_bytes)
```

  - `node_filesystem_size_bytes` 의 일반적인 Lable들(`device` , `instance` , `fstype` , `job` , `mountpoint`)에서 몇몇을 제외하고, 제외하지 않은 Label을 기준으로 주어진 값들의 sum을 진행한 결과이다.
  - 현실적으로는 다음이 더 낫지 않을까? 대부분의 OS에서 Filesystem들은 Virtual이 많으므로, 필요로 하는 `device` 와 `fstype` 만을 지정하여 조회해 보았다.

```
sum without(mountpoint, device) (node_filesystem_free_bytes{device=~".dev.disk[0-9]*s[0-9]*s.*|.dev.sd.*", f
stype=~"apfs|ext4"})
```

다른 Metric 데이터 형식에서도 마찬가지로, 결국 어떤 Label이 반환되도록 할 것인가에 대하 예측이 연산자를 포함하는 Vector매칭에 중요하다.

- 측정된 동일 Metric 전체에 대해서도 사용할 수 있다.

```
(process_open_fds)
sum (process_open_fds)
sum without(instance, job) (process_open_fds)
avg without(instance, job) (process_open_fds)
```

Counter

Counter는 이벤트의 갯수나 크기등을 추적하며, /metrics에서 조회되는 Count값들은 해당 Application(Node)가 시작한 이후의 총합이다. 이미 앞에서 설명되었듯이, Counter의 현재 숫자나, 총합은 큰 의미가 없으며, 우리는 단위 시간에 따라 카운터가 얼마나 빨리 증가(변화) 하는가에 관심이 있다.

`increase()`, `irate()` 함수를 사용할 수도 있지만, 보통은 `rate()`함수가 사용된다.

- 예를 들어 초단위로 수신된 네트워크 트래픽 양은 다음과 같이 확인 할 수 있다. [5m]은 5분동안의 초당 수신량 평균임에 주의하라. 초당 측정된 값들의 분당 평균 표현에는 약간의 오차가 있을 수 밖에 없음을 인지하라.

```
rate(node_network_receive_bytes_total[5m])
// NIC가 너무 많아 몇개로 추려 볼 수도 있다.
rate(node_network_receive_bytes_total{device=~"lo|lo0|en0|eno1"}[1m])
```

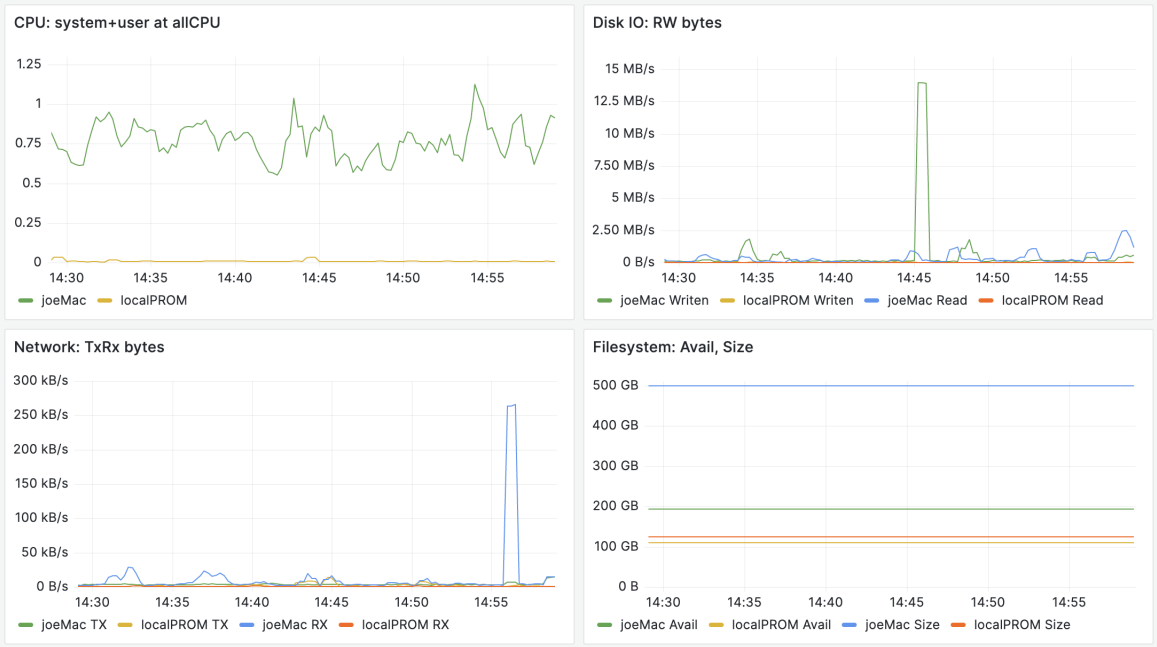
- rate()의 결과는 Guage이므로, Guage에서 사용가능한 집계연산자를 사용할 수 있다. 제외한 Label과 결과에서 남아있는 Label에 주의해서 살펴 본다.

```
sum without(device)(rate(node_network_receive_bytes_total{device=~"lo|lo0|en0|eno1"}[1m]))
sum without(instance)(rate(node_network_receive_bytes_total{device=~"lo|lo0|en0|eno1"}[1m]))
```

책에는 없지만, 해보자 1

아래에 간단히 2개 node(joeMac, localPROM)에 대한, CPU, Disk IO, Network, Filesystem에 대한 상태를 보여주는 그래프를 Grafana에 만들었고, 각각은 Node Exporter가 제공하는 Metric을 기반으로 PromQL을 적용하였다.

- Grafana는 Visual Graph를 그리게 되며, Graph의 정의에서 PromQL에서 하던 Label을 제거등은 Legend(대체)하는 방법이 추가로 있으므로 이것을 사용하는 것도 도움이 된다. 가능하다면, PromQL에서 할 수 있는 것들은 QL에서 하는 것이 좋겠다.
- Grafana 예



- 사용한 PromQL 예

```
// CPU
sum by(job) (rate(node_cpu_seconds_total{mode=~"(system|user)"}[1m]))

// DiskIO, 2개의 metric을 동시에 보여주고 있다.
rate(node_disk_written_bytes_total[1m])
rate(node_disk_read_bytes_total[1m])

// Network, 2개의 metric을 동시에 보여주고 있다.
rate(node_network_transmit_bytes_total{device=~"(en0|eno1)"}[1m])
rate(node_network_receive_bytes_total{device=~"(en0|eno1)"}[1m])

// Filesystem, 2개의 metric을 동시에 보여주고 있다.
node_filesystem_avail_bytes{mountpoint="/"}
node_filesystem_size_bytes{mountpoint="/"}

// 다음은 다른 형식의 PromQL이지만, 동일 결과 이다.
sum by(job) (rate(node_network_receive_bytes_total{device=~"(en0|eno1)"}[1m]) )
sum without(device)(rate((node_network_receive_bytes_total{device=~"(en0|eno1)"}[1m])))
sum by(job) (node_filesystem_avail_bytes{mountpoint="/"})
```

Summary

일반적으로 Summary Type의 Metric에는 `_sum`, `_count` 접미어가 모두 포함되며, 간혹 접미어 없이 Quantile(분위수) Label이 있는 시계열도 포함하는 경우도 있다.. `_sum`, `_count` 자체는 모두 Counter Type이다. (Histogram도 `_sum`, `_count`를 포함하고 있지만, Histogram은 `+Inf`를 포함하는 Bucket을 가지고 있으며, 이는 Summary의 Quantile과는 다른 것이다.)

일정 기간동안의 평균을 얻기 위해서는 (rate처리 후에) `_sum`을 `_count`로 나눈 값이다. 예를 들어, 마지막 1분동안 각각 2초, 4초, 9초가 걸리는 3개의 요청이 처리되었다면, Metric에 전체 `_count`는 3, 전체 `_sum`은 15(초)가 될 것이다. 이 경우, 평균 평균 대기 시간은 5초가 될 것이다.

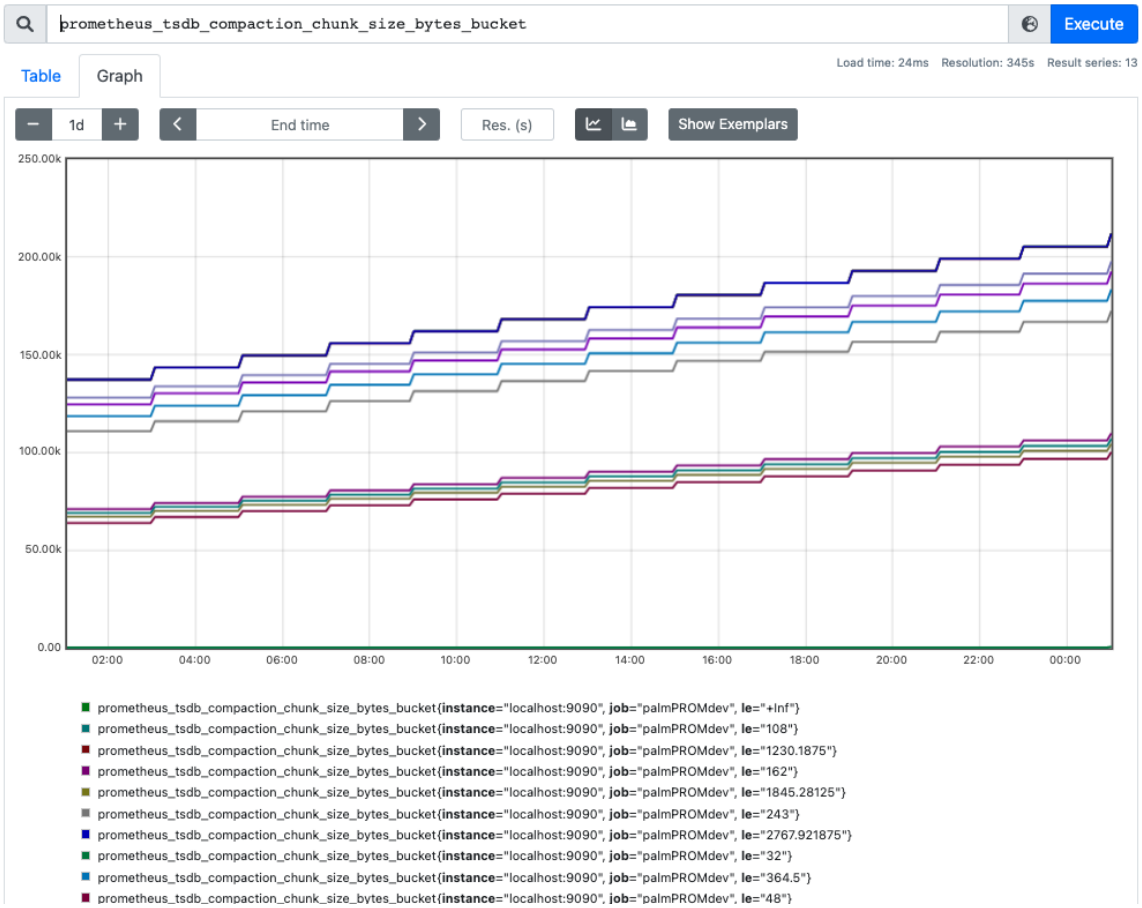
- `prometheus_http_response_size_bytes_sum`, `prometheus_http_response_size_bytes_count`를 예를 들어 살펴보자. handle Label을 무시하고, 1분간의 모든 http response 요청에 대한 `_sum`과 `_count`를 구할 수도 있다. 아래는 2개의 값을 사용하여 평균 응답크기에 대해 구한 예제이다.

```
sum without(handler)(rate(prometheus_http_response_size_bytes_sum[1m])) /
sum without(handler)(rate(prometheus_http_response_size_bytes_count[1m]))
```

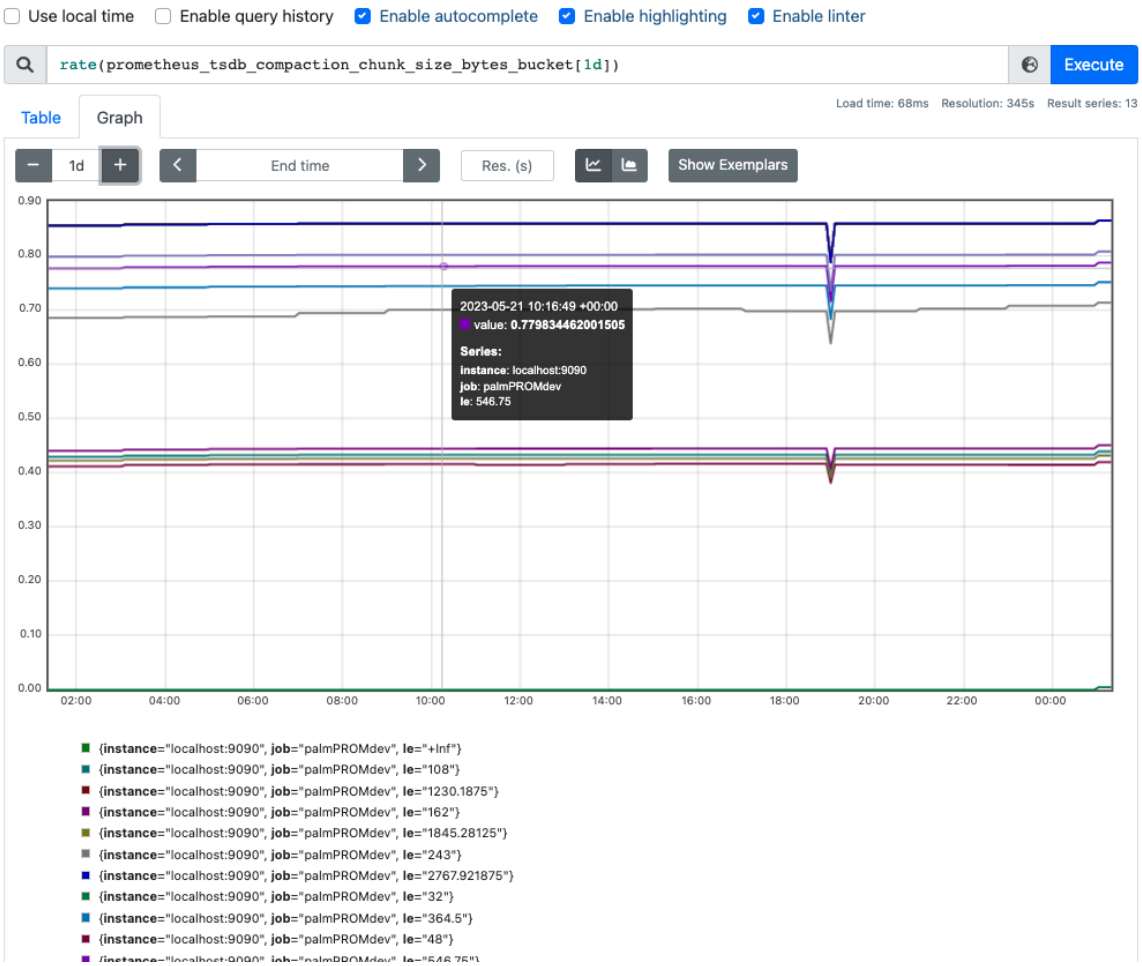
Histogram

Histogram은 Metric에 대한 분포를 추적할 수 있으며, 구분되어있는 Bucket을 기준으로 분위수(Quantile)를 계산할 수 있다. 책의 예제와 달리 `prometheus_tsdb_compaction_chunk_size_bytes_bucket`를 살펴 보자.

- 해당 메트릭은 2시간 마다 실행되는 tsdb의 Compaction작업에서 처리한 chunk의 크기들에 대한 분포를 나타낸 것이다. 아래의 조회 결과는 해당 작업이 2시간 마다 수행되므로 1d 동안의 bucket 이력으로 살펴 본 결과이다.



- 동일 메트릭에 대해 `rate()` 살펴보면 다음과 같다. bucket을 살펴 보면, 상위에 상위의 bucket들이 있고, 화면처럼 약 546 bytes정도가 0.78선에 있다.



- 해당 메트릭을 historgram으로 살펴 보면 다음과 같다. 좌측의 숫자를 잘 보면, 0.9분위의 수가 502~514 bytes사이에 있다. tsdb compaction한 size의 0.9분위수가 그것임을 알 수 있다. (그래프의 변화도를 볼때는 수치도 같이 보아야…)

- `_sum`, `_count` 메트릭도 Histogram에는 존재하는데, 그 의미와 용도는 Summary와 동일하다.
- 분위수를 0.99, 0.999로 하려면, bucket도 그렇게 되어야 하므로 너무 많은 bucket은 용량에 부담이 된다. 항상 성능, 용량과 사용성을 고려하여야 한다. 또한, 일반적으로 Histogram은 5분 또는 10분 단위의 rate()함수와 같이 사용하게 된다. range를 시간 또는 일단위로 넓히면, 이 역시도 성능,용량에 부담이 될 수 있음을 고려 한다.

책에는 없지만, 해보자 2

(A) 그래, 이기회에 다시 잘 보자: `_sum`, `_count`를 중심으로 Histogram를 제대로 이해해 보자.

우선은 prexfix처럼 사용되는 `_sum`, `_count`, `_bucket`들도 관측하고 싶은 주요한 Metric의 추가적인 항목들이다. 예를 들어, 프로메테우스 자체에 대한 Metric들 중에서 `prometheus_http_response_size_bytes` 관련되어서 3가지의 Metric(`_sum`, `_count`, `_bucket`)이 존재 한다.

그 중에서 `_bucket` Metric은 **Histogram** Type이다. 그리고, `_count`, `_sum`은 **Count** type이다.

- 아래는 해당 Metirc이 실제 계측된 내용이다. (물론, handler Label에 따라 유사한 계측 내용들이 더 있다.)
- 인터넷에 있는 여러 설명이나, 프로메테우스 시스템에 달려 있는 Comment가 오해를 불러 일으키는데, 아래의 Metirc 실제 내용을 보면 명확해 진다. `_Bucket`은 원래 목적대로 해당 handle Label별로 9개의 Bucket으로 Historgram을 구성하고 있으며, “le”는 response 한 내용의 크기(byte)에 기준한 bucket 크기이다. 그리고, `_sum`은 reponse한 내용의 크기의 합 (byte), `_count`는 reponse한 횟수이다.

```
# HELP prometheus_http_response_size_bytes Histogram of response size for HTTP requests.
# TYPE prometheus_http_response_size_bytes histogram
prometheus_http_response_size_bytes_bucket{handler="/",le="100"} 6
prometheus_http_response_size_bytes_bucket{handler="/",le="1000"} 6
prometheus_http_response_size_bytes_bucket{handler="/",le="10000"} 6
prometheus_http_response_size_bytes_bucket{handler="/",le="100000"} 6
prometheus_http_response_size_bytes_bucket{handler="/",le="1e+06"} 6
prometheus_http_response_size_bytes_bucket{handler="/",le="1e+07"} 6
prometheus_http_response_size_bytes_bucket{handler="/",le="1e+08"} 6
prometheus_http_response_size_bytes_bucket{handler="/",le="1e+09"} 6
prometheus_http_response_size_bytes_bucket{handler="/",le="+Inf"} 6
prometheus_http_response_size_bytes_sum{handler="/"} 174
prometheus_http_response_size_bytes_count{handler="/"} 6
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="100"} 5
```

```
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="1000"} 86
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="10000"} 86
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="100000"} 86
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="1e+06"} 86
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="1e+07"} 86
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="1e+08"} 86
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="1e+09"} 86
prometheus_http_response_size_bytes_bucket{handler="/api/v1/labels",le="+Inf"} 86
prometheus_http_response_size_bytes_sum{handler="/api/v1/labels"} 33571
prometheus_http_response_size_bytes_count{handler="/api/v1/labels"} 86
```

- `handle="/"`이 회신된 횟수는 6회, 6회동안 회신한 내용의 전체 크기는 `174 bytes`, Bucket `"100byte 이하"`가 6이므로 6번의 모든 회신이 100bytes 이하 였음을 알 수있다.
- 다른 관점에서 동일 내용을 살펴 보면, (사실은 29bytes로 동일 했으나,) 174/6=29 이므로 평균 회신된 내용의 크기는 29bytes 였을 것으로 추정 할 수 있다.
- `handle="/api/v1/labels"` 는 회신된 횟수는 86회, 회신한 전체 크기는 33,571 bytes, Bueckt `le="100" 5`, `le="1000" 86` 이므로, 101~1000 bytes는 경우는 81회 이다. (reponse로 회신한 내용의 크기가 그때 그때 달랐음을 알 수 있다.)
- 이것도 다른 관점에서 살펴 보면, 33,571/86=390.3604 이므로 평균 회신된 내용의 크기는 약 391 bytes였을 것으로 추정 할 수 있다. 이 경우에도 회신된 내용의 크기가 그때 그때 달랐다는 것도 예상이 된다.
- **Histogram은 이렇게 사용한다.**
  모든 기록을 남기는 것이 아니라, 안전한 최하의 정보 규모로 최대의 필수적인 정보를 알 수 있도록 한다.

(B) 조금 더, Histogram을 제대로 이해해 보자.

위의 Metirc정보들 중에는 앞의 `prometheus_http_response_size_bytes_count` 와 유사한 숫자를 가지고 있는 `prometheus_http_requests_total` Counter가 있다. 잘 살펴보면, `"code=200"` 처럼, Label이 추가 되어있는 점이 다르다.

- 관심 있는 Metric 몇 개만 보자. `code="XXX"`의 값은 HTTP Response Code이다.

```
prometheus_http_requests_total{code="200",handler="/api/v1/labels"} 86
prometheus_http_requests_total{code="302",handler="/"} 6
```

- `handler="/api/v1/labels"` 은 앞에서도 모두 86회 였고, `prometheus_http_requests_total`에서도 `code="200"` 처리된 Counter가 86으로 해당되는 모든 요청이 Code값 200, 즉 모두 정상처리 되었음을 알 수 있다.
- handler="/" 도 앞에서는 모두 6회 였으나, `prometheus_http_requests_total` 에서의 집계는 `code="302"` 처리된 Counter만 6으로되어 있다. 모든 요청이 Code값 302로, 즉 비정상 처리 되었음을 알 수 있다.
- **Histogram은 이렇게 사용한다.**
  Metric내의 다른 정보들과 조합으로 최대한 의미를 도출하고 유추하여 파악할 수 있도록 한다.

13.2. Selectors

> 💡 **VECTOR**
>
> Prometheus 및 관련영역에서 Vector라고 표현하는 것은 TSDB에서의 시계열에서 주어진 또는 기록된 하나의 1차원 List를 의미한다.

앞에서도 일부 사용예제가 있었다. 프로메테우스의 TSDB에 저장된 내용을 확인하는 과정에서 우리는 보고 싶은 메트릭, 범위, 조건들을 세세하게 설정하여 그 결과를 사용하게 된다. 그렇게 하는 방법을 Selector를 사용하게 되는데, Selector를 사용하는 주요 방법들에 대해 알아 본다.

instant vector

instant vector는 쿼리 평가 시점 전의 가장 최근의 샘플을 의미하며, 0개 이상의 시계열 List로 표현된다. 즉, 시계열 데이터 마다 하나의 샘플링된 값을 반환하는 것이다.

matcher

(대부분의 경우) 많은 Label과 Label의 내용에 대해 어떤 메트릭들을 대상으로 할 것인지를 선택하여 사용하게 된다. 여기에 사용되는 것이 matcher이고, 4가지 종류가 있다.

- (`=`) equality matcher: `job="node"` 를 예로 들수 있다. 동일한 이름의 Label을 선택한다. 빈 Label은 Label이 없는 것과 같다. `foo=""` 는 foo Label이 없음을 선택한 것이다.
- (`!=`) negative equality matcher
- (`=~`) regular expression matcher: 앞에서 설명된 RE2 엔진과 규격을 사용하는 regex 표현방식에 의한 선택이다.
- (`!~`) negative regular expression matcher

Label matcher는 동일 이름의 Label에 대해 여러개의 matcher를 사용하여, negative lookahead expression을 대체할 수도 있다. 예를 들어 "`/run/user`"를 제외하고, "`/run`"이 포함된 모든 path에 대한 파일시스템의 크기를 선택하는 경우 다음과 같이 표현 할 수 있다.

```
node_filesystem_size_bytes{job="node", mount=~"/run/.*", mount!~"/run/user/.*"}
```

- 노드익스포터에는 이와 같은 경우, 처음부터 해당 파일시스템에 대한 메트릭을 원하지 않을 경우 사용할 수 있는 옵션 flag가 있다. `--collector.filesystem.ignored-mount-points`

Label이 아니라, 메트릭이름 자체에도 matcher와 regex를 사용할 수는 있다. 내부 Reserved Label `__name__` 에 해당 메트릭 이름이 저장되는데, 예를 들어 `process_resident_memory_bytes{job="node"}` 라는 표현은 `{__name__="process_resident_memory_bytes", job="node"}` 의 편의문법(Syntatic sugar)이다. 이제 메트릭 이름에도 matcher를 사용하여 수행할 수 있지만, 권장하지 않는다.

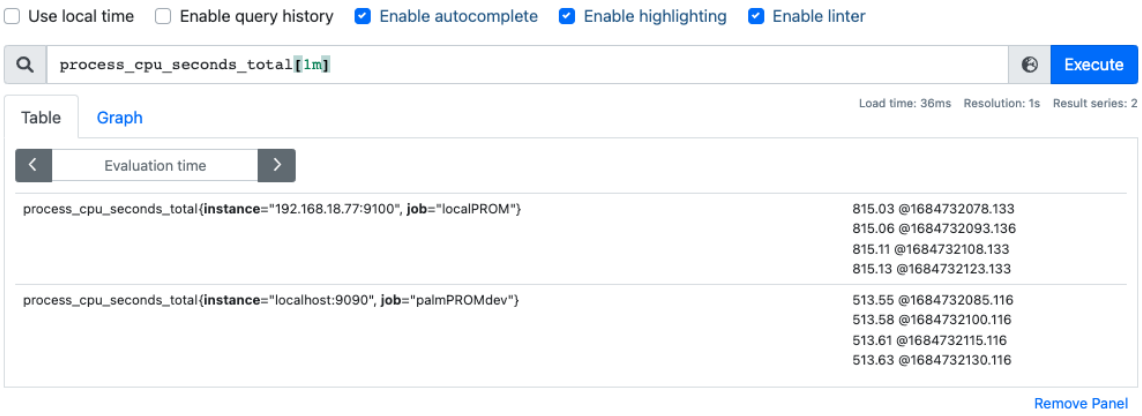
선택기의 사용에서 또 주의할 점

- regex를 사용할 때는 regex에 해당하는 Label값들이 하나로 결합되어 취급되는 것에 주의 한다.
- regex는 부담이 되므로 다른 표현 방법이 있다면 그것을 사용하라. `code=~"4.."` 대신에 `code="4xx"` 를 사용하라
- Selector `{}` 는 오류를 반환한다. 모든 시계열을 대상으로 반환하기 때문이다. 아래 예제 참고.
  - 허용되는 것: `{foo="", bar="x"}`, `{foo=~".+"}`
  - 허용되지 않는 것: `{foo=""}`, `{foo~=""}`, `{foo=~".*"}`

range vector

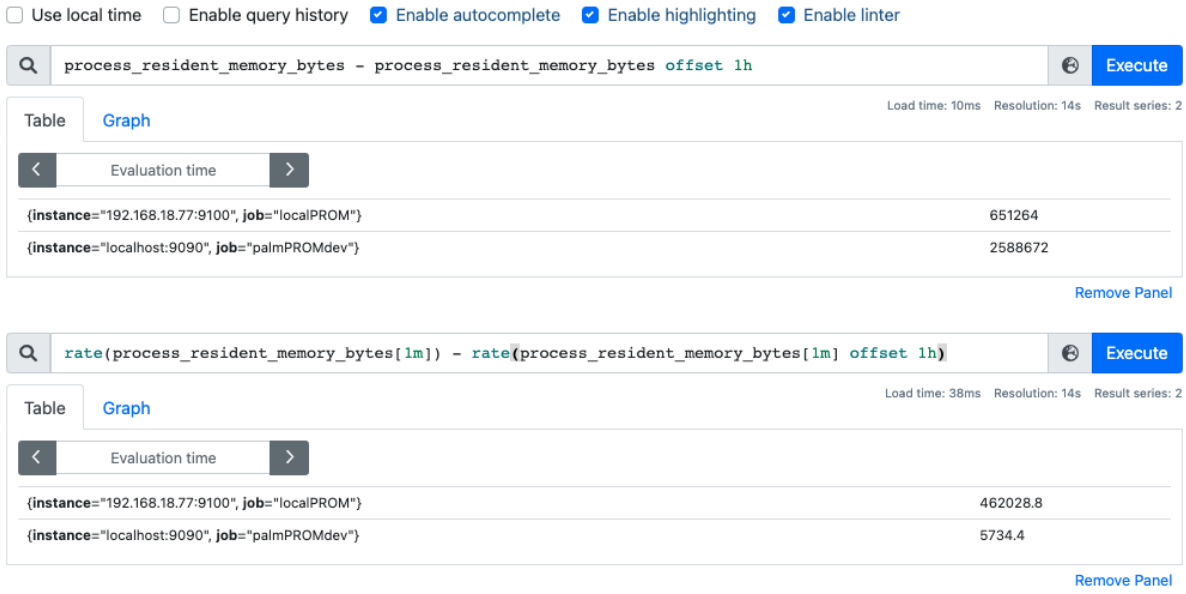
instance vector와 달리 각 시계열 데이터에 대해 더 많은 샘플링된 데이터를 반환할 수 있다. range vector는 항상 `rate()` 함수와 함께 사용한다. (정확히 표현하면, `rate()` 이외에도 range개념의 모든 함수가 해당 된다. e.g. `avg_over_time()`) 아래 쿼리에서 [1m]은 instance vector를 range vector로 지정된 시간 범위로 전환하고, 지정된 평가시간 1분 동안의 모든 시계열 데이터를 반환하도록 한다.

- `rate(process_cpu_seconds_total[1m])`
- 프로메테우스에서 `process_cpu_seconds_total[1m]` 쿼리를 실행한 결과는 아래와 같은데, 살펴보면, 15초 단위의 수집이기 때문에 [1m] 동안은 4건의 샘플의 가지고 있고, 한 node에서의 수집된 샘플은 정확히 15초 간격이지만, 각 node의 timestamp는 동일 시간은 아닌 것에 주의한다. (참고로 Table Tab에서만 `process_cpu_seconds_total[1m]` 쿼리가 가능하다. Graph Tab에서는 range 또는 instant Vector가 되어야 한다.)



offset

offset은 쿼리에 대한 평가 시간을 선택된 시간 만큼 과거로 되돌린다. 아래의 예는 `process_resident_memory_bytes`에 대해 1시간 전과의 비교 및 1시간 전의 1m 평균값과의 차를 구하는 사례이다.



13.3. HTTP API of Prometheus

프로메테우스 서버 App은 자체적으로 많은 HTTP API를 제공하는데, 많이 사용되는 것은 query와 query_range요청인데, 이것의 endpoint는 `/api/v1/` 아래에 있으며, 각각은 `/api/v1/query`, `/api/v1/query_range`이다.

query

프로메테우스 서버의 endpoint `/api/v1/query`에 아래 명령으로 특정 메트릭에 대한 쿼리를 실행하면, JSON형태의 결과가 되돌려 진다. 아래 결과를 풀어서 해석해 보면, `"resultType"`는 `"vector"` 형식이며, 2개의 mapped data를 가지고 있다. 각각은 Label들을 담고 있는 `"metric"` 맵과 **timestamp**와 결과값을 가지는 `"value"` 맵이다.

```
$ curl -s http://w77:9090/api/v1/query?query=process_resident_memory_bytes | python3 -m json.tool
$ curl -s http://w77:9090/api/v1/query?query=process_resident_memory_bytes | jq
{
  "status": "success",
  "data": {
    "resultType": "vector",
    "result": [
      {
        "metric": {
```

```
          "__name__": "process_resident_memory_bytes",
          "instance": "192.168.18.77:9100",
          "job": "localPROM"
        },
        "value": [ 1684734636.622, "20815872" ]
      },
      {
        "metric": {
          "__name__": "process_resident_memory_bytes",
          "instance": "localhost:9090",
          "job": "palmPROMdev"
        },
        "value": [ 1684734636.622, "106057728" ]
      }
    ]
  }
}
$
```

아래의 쿼리도 실행해 본다. 동일 내용을 웹브라우져에서 조회해도 된다. (URL에 escape character처리해다.) 아래는 `“resultType”`는 `“matrix”`형식이며, 조회된 `metric`에 대한 `values`를 여러개의 시계열 값을 갖는 사례이다.

```
$ curl -s "http://w77:9090/api/v1/query?query=process_resident_memory_bytes\[1m\]" | jq
{
  "status": "success",
  "data": {
    "resultType": "matrix",
    "result": [
      {
        "metric": {
          "__name__": "process_resident_memory_bytes",
          "instance": "192.168.18.77:9100",
          "job": "localPROM"
        },
        "values": [
          [ 1684911763.133, "21045248" ],
          [ 1684911778.133, "21045248" ],
          [ 1684911793.135, "20955136" ],
          [ 1684911808.136, "20955136" ]
        ]
      },
      {
        "metric": {
          "__name__": "process_resident_memory_bytes",
          "instance": "localhost:9090",
          "job": "palmPROMdev"
        },
        "values": [
          [ 1684911770.116, "107634688" ],
          [ 1684911785.116, "107634688" ],
          [ 1684911800.116, "107634688" ],
          [ 1684911815.116, "107634688" ]
        ]
      }
    ]
  }
}
$
```

간혹 “resultType”이 Scalar인 경우도 있는데, 이 경우 Label 없이 값 만을 가지는 경우다.

```
{
  "status": "success",
  "data": {
    "resultType": "scalar",
    "result": [ 1684911815.116, "42" ]
```

query_range

프로메테우스 서버의 endpoint `/api/v1/query_range`에 특정 메트릭에 대한 쿼리를 실행하면, 역시 JSON형태의 결과가 되돌려 진다. `/api/v1/query`와는 다르게 start시간, end시간, Step을 추가로 제공해 줘야 한다. (Query로 조회할 시간 range를 지정하는 것이다.)

아래의 예제 내용을 브라우져에서 조회하여 결과를 풀어서 해석해 보자. 조회의 파라미터로 시작시간, 종료시간, Interval을 지정하여 조회하였다. 결과는 `“result”`는 `“matrix”`형식이며, 해당 범위에 대한 조회 결과이다.

```
$ curl -s "http://w77:9090/api/v1/query_range?query=rate(process_resident_memory_bytes\[5m\])&start=1684681200&end=1
684681800&step=120" | jq
{
  "status": "success",
  "data": {
    "resultType": "matrix",
    "result": [
      {
        "metric": {
          "instance": "192.168.18.77:9100",
          "job": "localPROM"
        },
        "values": [
          [ 1684681200, "290186.68968445284" ],
          [ 1684681320, "289740.1385272879" ],
          [ 1684681440, "217187.84144617032" ],
          [ 1684681560, "216769.53575927267" ],
          [ 1684681680, "288329.65614035085" ],
          [ 1684681800, "432599.64139973413" ]
        ]
      },
      {
        "metric": {
          "instance": "localhost:9090",
          "job": "palmPROMdev"
        },
        "values": [
          [ 1684681200, "1882.7228070175438" ],
          [ 1684681320, "110160.84210526315" ],
          [ 1684681440, "475193.4877192982" ],
          [ 1684681560, "712502.792982456" ],
          [ 1684681680, "678010.1614035087" ],
          [ 1684681800, "675078.2877192982" ]
        ]
      }
    ]
  }
}
$
```

- rate()함수에서 지정한 5분의 범위(range)는 step으로 지정한 120초 보다 크다. 쿼리는 그때 그때 해석이 되므로 step에서 지정된 시점을 기준으로 5분의 rate()를 구한 것으로 해석해야 한다. 주의할 점은 rate()의 range보다 step이 큰 경우, 예를 들어 rate()는 1m이고, step은 300(5m)인 경우 80%의 범위에서의 값들은 무시되는 것에 주의 하여야 한다.
- query_range에 11,000개 이상의 step이 있다면, 프로메테우스는 해당 쿼리를 오류로 간주한다. 특정 리포팅을 위해 한 주에 분단위 해상도(10,080건) 이나 1년에 시간단위 해상도(8,760건)은 허용은 되지만, 권장되지 않는다.

+==