2nd Meeting Report

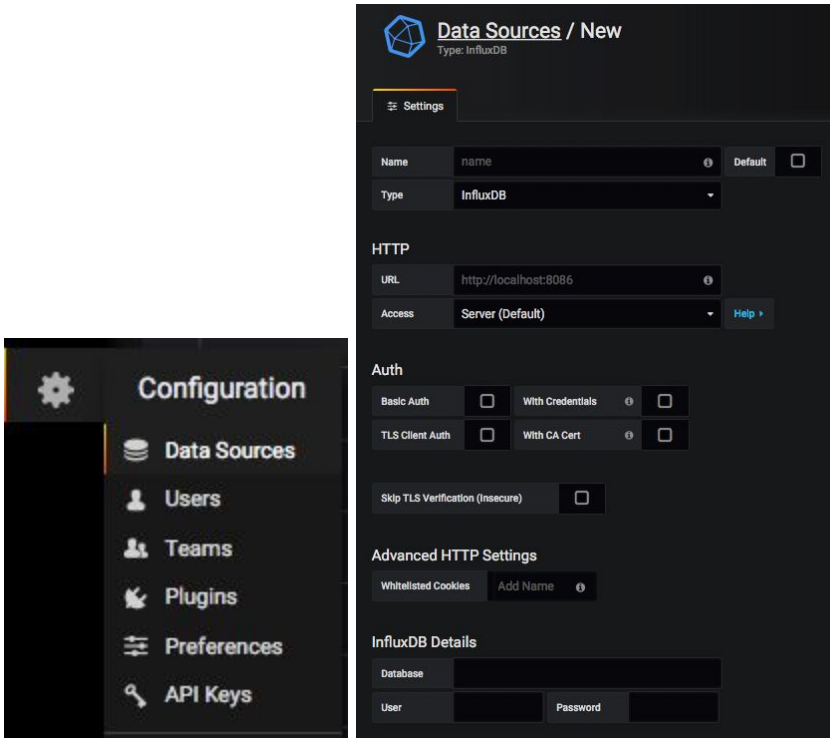
1. Grafana instead of Chronograf

1.1 Setup of data sources

We setup Grafana by following this tutorial. Grafana by default is running on port 3000 and it can be easily configured by a web UI.

For example, when we configure data sources that will display on Grafana.

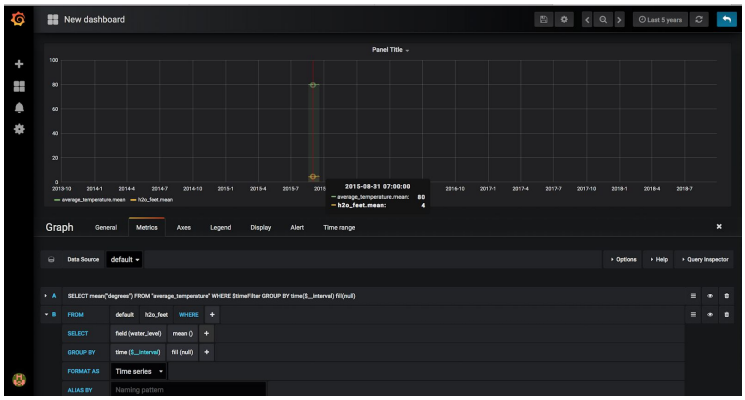
Go to localhost:3000 then use menu “Configuration > Data sources”



Example of data sources configuration which support many types of database

1.2 Query or Metrics

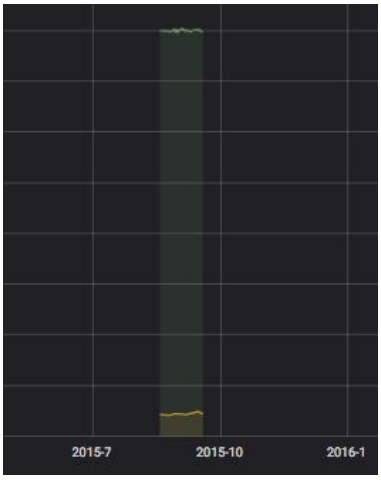
As you see in the picture below, the bottom section has a feature called “Query inspector”



Picture show “Edit dashboard feature”, Top section is visualization part, Bottom section is query part

More details:

Query A and B

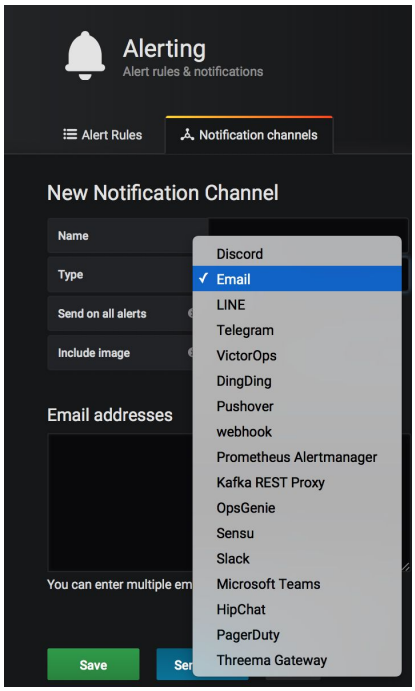


Visualize query A(Green), B(Yellow)

1.3 Notification support

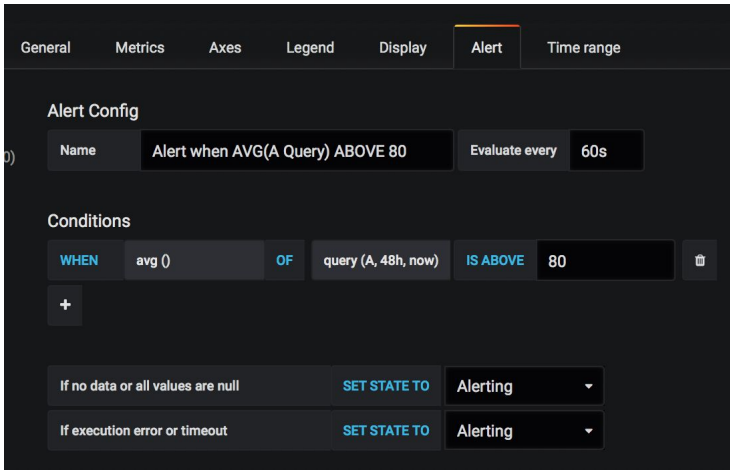
Grafana provides notification alert feature.

First, we need to custom Channel of alert (which will be used for sending notification)



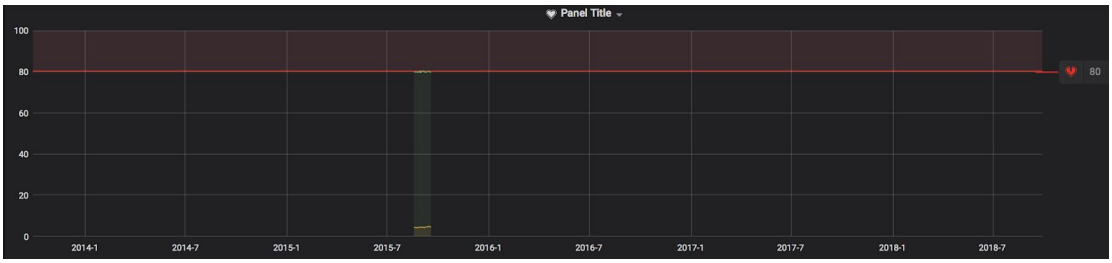
Notification support channels such as LINE, Email, Slack, Webhook, etc.

Second, setup an alert rule such as average of A query is above 80%



Setup alert when AVG is above 80%

Interesting of this feature is grafana can track if execution of the query results in error or timeout, or data that query execute are null.



Example visualization of alert rules

1.4 Query performance / Caching strategy

According to my research about "Does grafana always fetch the whole data?" My answer is 'Yes'. There still have many issues according to github discussion blog; First issue is about how to cache data on Grafana and second issue is the memory leaks when display many matrics.

Currently solutions are setting auto-refresh to never and manually refresh when you need to monitor it to reduce workload of the query or performing the cache in the database level.

About the memory leak problem when querying too much data and displaying them on a Web browser, Grafana has a feature called "MaxDataPoints" to set the amount of data that the browser can handle then the query will stop when reached MaxDataPoints.

2. InfluxDB data management

InfluxDB provides 6 functions to manage data and some of them support **retention policy** which could **manage the series data** that grows quickly **to expired automatically**; the retention policy doesn't just drop one data point at a time, it drops an entire shard group.

Functions:

2.1 CREATE DATABASE:

We can execute **CREATE DATABASE with retention policy** to specify lifetime of data such as;

```
CREATE DATABASE "NOAA_water_daabase" WITH DURATION 3d
REPLICATION 1 SHARD DURATION 1h NAME "liquid"
```

The query creates a database called "NOAA_water_database" with the duration of three days, replication of 1, shard group duration of one hour, and with the retention policy name "liquid"

2.2 DELETE:

Delete supports time intervals in the WHERE clause.

Example use;

```
DELETE FROM "h2o_quality" WHERE time < '2016-01-01'
```

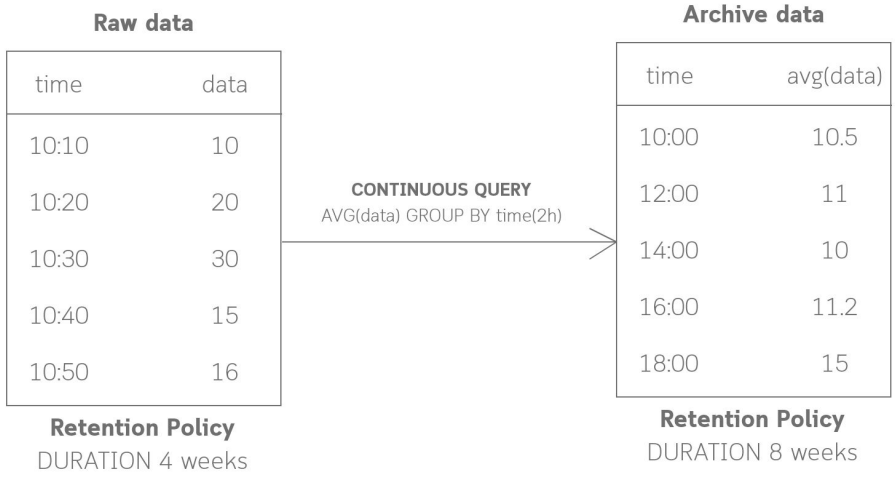
2.3 Retention policy management

Syntax:

```
CREATE RETENTION POLICY <retention_policy_name> ON
<database_name> DURATION <duration> REPLICATION <n> [SHARD
DURATION <duration>] [DEFAULT]
```

2.4 Archiving data

If we want to archive data not only remove them when expired, we can use CQ(Continuous Query) to average the fields and keep the result in another measurement.



Example archiving data from my understand

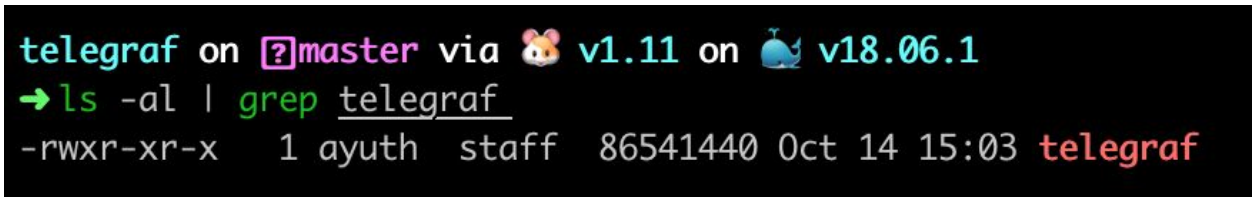
3. Collect custom metrics in Telegraf

a. Prerequisite

i. Install the Go 1.8+(currently we use go 1.11) using the method provided by this link.
ii. Set the GOPATH environment variables by following the method in this link.
iii. Clone the Telegraf source code using
$ go get -d github.com/influxdata/telegraf

b. Try to make telegraf from source code

i. Try to make the plugin to proof that our go compiler is works as expected using.
$ make
If everything works as expected 'telegraf' will be output in same folder.



c. Make the telegraf's plugin name 'pragma'

According to write your own plugin example we'll demonstrate that how to make a telegraf plugin named '**pragma'** to collect 3 custom fields. First one is **temperature** second is **humidity** and the last one is **wind.**

Currently I'm testing on my Macbook Pro 2015 (Retina, 13-inch, Early 2015)
- Processor: 2.7 GHz Intel Core i5 (real 4 physical cores)
- Memory: 8 GB 1867 MHz DDR3

I've started ~~T~~ICK (InfluxDB, Chronograf and Kapacitor without Telegraf) with docker compose(all source code available is in this repo). Each stack component is in its own individual container and all containers are talking to each others using the HTTP protocol. We report how to create a Telegraf plugin in the next section.

i. Change directory to telegraf's source code, checkout new branch name 'pragma'.
$ cd $GOPATH/src/github.com/influxdata/telegraf
$ git checkout -b pragma

ii. Make main plugin file and add boilerplate.
$ cd plugins/inputs
$ mkdir pragma
$ touch pragma/pragma.go
Paste the boilerplate below in 'pragma/pragma.go'

```
package pragma


import (
```

```
        "github.com/influxdata/telegraf"
        "github.com/influxdata/telegraf/plugins/inputs"
)

type Pragma struct{}

func (s *Pragma) SampleConfig() string {
        return ""
}

func (s *Pragma) Description() string {
        return ""
}

func (s *Pragma) Gather(acc telegraf.Accumulator) error {
        return nil
}

func init() {
        inputs.Add("Pragma", func() telegraf.Input { return &Pragma{} })
}
```

iii. Import the 'pragma' plugin in the file -> telegraf/plugins/inputs/all/all.go This imports your plugin to the main telegraf package and ensures that it can run.

```
_ "github.com/influxdata/telegraf/plugins/inputs/pragma"
```

iv. Add the properties to the struct.

```
type Pragma struct {
        Temperature float64
        Humidity    float64
        Wind        int32
}
```

v. Telegraf dynamically constructs configuration files by aggregating the configurations for all of it's plugins. This allows you to easily generate

configuration files with only certain plugins enabled by just passing a couple of CLI flags.

```
var PragmaConfig = `
 ## Initial variables
        temperature = 10.0
        humidity = 10.0
        wind = 5
`

func (s *Pragma) SampleConfig() string {
        return PragmaConfig
}
```

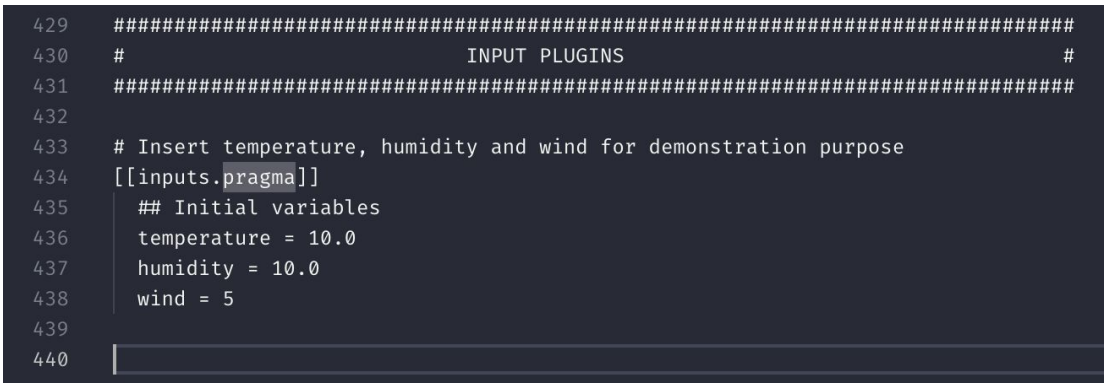
vi. Add the description of the plugin to tell what our plugin does.

```
func (s *Pragma) Description() string {
        return "Insert temperature, humidity and wind for demonstration purpose"
}
```

vii. Test it!

$ make

$ ./telegraf -sample-config -input-filter pragma -output-filter influxdb >> telegraf.conf.test



viii. The core function. The Gather function is the heart of the plugin and runs every time telegraf collects metrics. In this demonstration we'll use an input file in [[inputs.pragma]] section to set initial values of variables and randomly increase or decrease between 0 - 4.

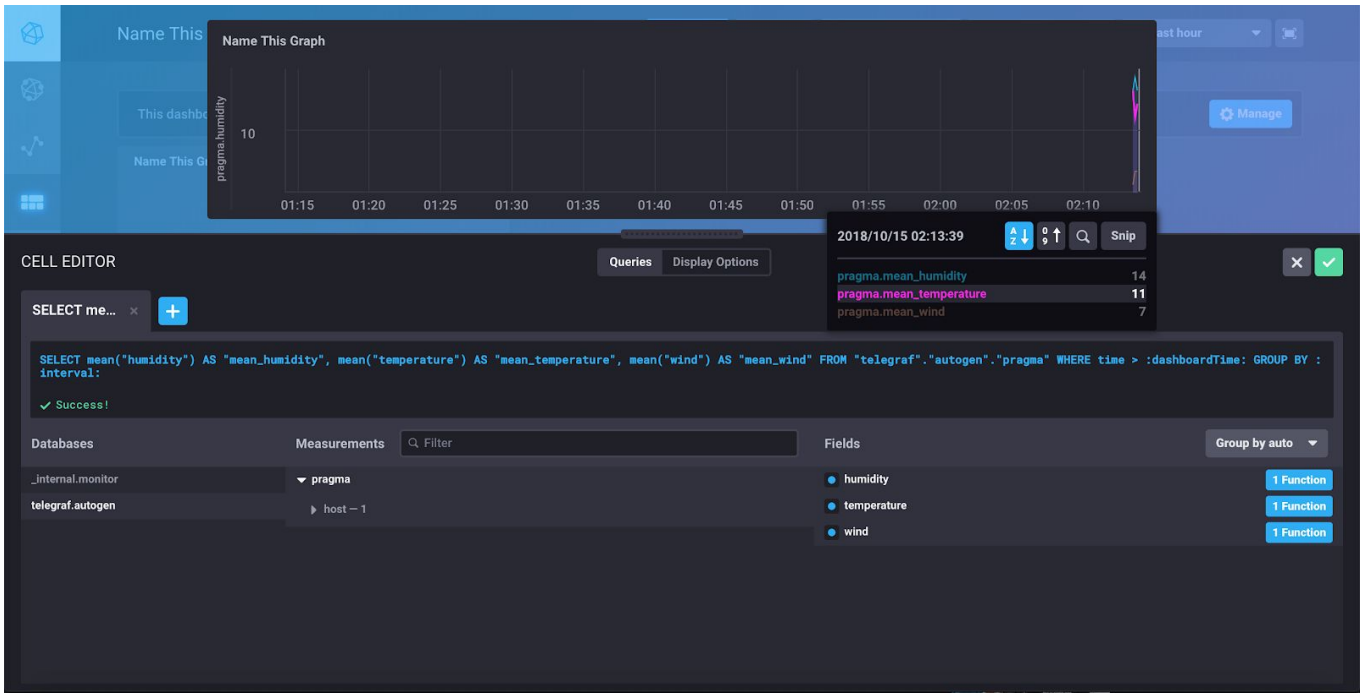
```
func (s *Pragma) Gather(acc telegraf.Accumulator) error {
        sign := int32(math.Round(rand.Float64())*2 - 1)
        temperature := s.Temperature + (sign * rand.Int31n(5))
        humidity := s.Humidity + (sign * rand.Int31n(5))
        wind := s.Wind + (sign * rand.Int31n(5))
```

```
        fields := make(map[string]interface{})
        fields["temperature"] = temperature
        fields["humidity"] = humidity
        fields["wind"] = wind

        tags := make(map[string]string)

        acc.AddFields("pragma", fields, tags)

        return nil
}
```

ix. Graph it with Chronograf

Assume that you have influxdb and chronograf set up. We'll make the telegraf from source code and run, then telegraf output will be sent to influxdb.

```
$ make
$ telegraf -sample-config -input-filter pragma -output-filter influxdb >> telegraf.conf.test
$ telegraf -config telegraf.conf.test -debug
```



x. Note: In real practice, we need to containerize the telegraf that we built from the, so that it can be portable and runable on any machine with Docker.

4. More useful references and our playground repos

- https://github.com/blacksourcez/TICK-docker-playground