

IoT Live Streaming

ΕΡΓΑΣΙΑ 1.3: Επεξεργασία ροών δεδομένων σε πραγματικό χρόνο με χρήση των open source καταναμημένων συστημάτων : Apache Kafka, Apache Flink, Redis Timeseries και Grafana

ΠΑΝΑΓΙΩΤΗΣ ΠΑΠΑΔΕΑΣ
Α.Μ. 03118039

ΘΕΟΔΩΡΟΣ ΑΡΑΠΗΣ
Α.Μ. 03118028

EMMANOYHΛ ΒΛΑΣΣΗΣ
Α.Μ 03118086

ΟΜΑΔΑ 31

Εργαλεία που χρησιμοποιήθηκαν: Apache Kafka, Apache Flink, Redis Timeseries, Grafana

I. ΕΙΣΑΓΩΓΗ

Στην παρούσα εργασία καλούμαστε να προσομοιάσουμε τη λειτουργία ενός πρότυπου συστήματος Internet of Things με χρήση σύγχρονων εργαλείων. Συγκεκριμένα παράγουμε δεδομένα που αντιστοιχούν σε διαφορετικές συσκευές με αντίστοιχα timestamps τα οποία έπειτα συλλέγονται από τον broker και αποθηκεύονται στην κατάλληλη βάση δεδομένων με χρήση ενός live streaming processing system για την επεξεργασία των δεδομένων. Τέλος τα αποτελέσματα παρουσιάζονται μέσω διαγραμμάτων που ενημερώνονται σύγχρονα με χρήση web sockets.

II. DEVICE LAYER

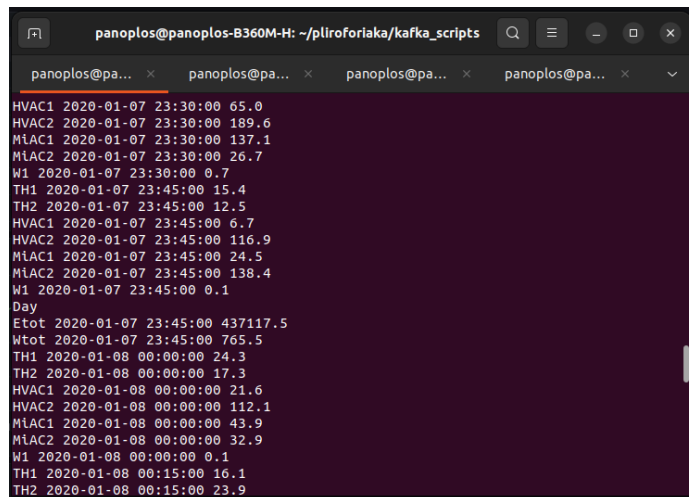
Για την παρούσα εργασία τα δεδομένα των συσκευών θα δημιουργηθούν αυτόματα με τη χρήση κώδικα καθώς δεν κατέχουμε πραγματικές συσκευές. Συγκεκριμένα δημιουργούμε το εκτελέσιμο producer.py για την δημιουργία των δεδομένων. Τα δεδομένα μας είναι στη μορφή (key, timestamp, value). Κάθε entry αυτής της μορφής περιέχει ως key το μέγεθος το οποίο αφορά . Πχ το entry με key TH1 αναφέρεται στη θερμοκρασία που παράγει ο αισθητήρας με τον αριθμό 1.

Οι αισθητήρες αντίστοιχα χωρίζονται σε αισθητήρες που παράγουν δεδομένα κάθε 15 λεπτά και σε αισθητήρες που παράγουν δεδομένα κάθε ημέρα δηλαδή κάθε 24 ώρες. Αντιστοιχίζουμε τα 15 λεπτά σε 1 δευτερόλεπτο πραγματικού χρόνου και τις 24 ώρες σε 96 δευτερόλεπτα αντίστοιχα.

Τέλος λαμβάνουμε υπόψιν ότι κάποιοι αισθητήρες είναι δυνατόν να στείλουν δεδομένα με κάποια καθυστέρηση δηλαδή με template που αντιστοιχεί σε προηγούμενες μέρες. Έτσι για τον αισθητήρα W1 προσομοιώνουμε αυτή τη λειτουργία και παράγουμε κάθε 20 δευτερόλεπτα πραγματικού χρόνου (5 ώρες δηλαδή) κάποιο entry που αντιστοιχεί σε 2 μέρες πίσω και

κάθε 120 δευτερόλεπτα πραγματικού χρόνου κάποιο entry με timestamp που αναφέρεται 10 μέρες πίσω.

Εκτελούμε το script μέσω της εντολής 'python3.8 producer.py' και βλέπουμε τα δεδομένα που παράγονται και τα οποία αποθηκεύονται σε κατάλληλα topics στον broker, τα οποία topics θα αναφερθούν αναλυτικά στην πορεία.



```
panoplos@panoplos-B360M-H: ~/p4lloforiaka/kafka_scripts
panoplos@pa... x panoplos@pa... x panoplos@pa... x panoplos@pa... x
HVAC1 2020-01-07 23:30:00 65.0
HVAC2 2020-01-07 23:30:00 189.6
M1AC1 2020-01-07 23:30:00 137.1
M1AC2 2020-01-07 23:30:00 26.7
W1 2020-01-07 23:30:00 0.7
TH1 2020-01-07 23:45:00 15.4
TH2 2020-01-07 23:45:00 12.5
HVAC1 2020-01-07 23:45:00 6.7
HVAC2 2020-01-07 23:45:00 116.9
M1AC1 2020-01-07 23:45:00 24.5
M1AC2 2020-01-07 23:45:00 138.4
W1 2020-01-07 23:45:00 0.1
Day
Etot 2020-01-07 23:45:00 437117.5
Wtot 2020-01-07 23:45:00 765.5
TH1 2020-01-08 00:00:00 24.3
TH2 2020-01-08 00:00:00 17.3
HVAC1 2020-01-08 00:00:00 21.6
HVAC2 2020-01-08 00:00:00 112.1
M1AC1 2020-01-08 00:00:00 43.9
M1AC2 2020-01-08 00:00:00 32.9
W1 2020-01-08 00:00:00 0.1
TH1 2020-01-08 00:15:00 16.1
TH2 2020-01-08 00:15:00 23.9
```

III. MESSAGING SYSTEMS-BROKER

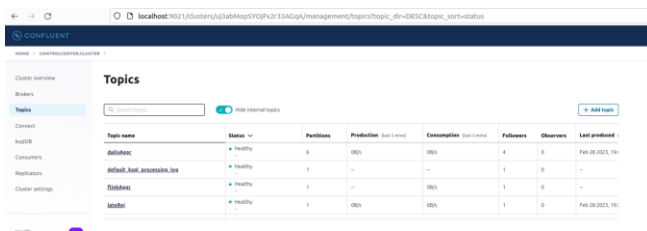
Τα δεδομένα που παράγουμε συλλέγονται σε έναν broker. Στην εργασία μας ο broker που χρησιμοποιούμε είναι το εργαλείο apache kafka [1] ενώ αργότερα εγκαθιστούμε και το confluent platform [2] καθώς παρέχει user interface πλήρως συμβατό με το apache kafka για να παρακολουθούμε τα topics που δημιουργούμε και τα messages που συλλέγουν.

A. Topics and partitions

Αξίζει σε αυτό το σημείο να αναφερθούμε στα topics που δημιουργούμε καθώς και τα partitions που αναθέτουμε σε κάθε topic. Συγκεκριμένα δημιουργούμε ένα topic που ονομάζουμε 'dailyAggr' με 6 partitions όπου συλλέγονται όλα τα σύγχρονα δεδομένα καθώς και τα δεδομένα που αντιστοιχούν σε timestamp δύο ημερών πίσω εφόσον θέλουμε να τα λάβουμε και αυτά υπόψιν στην επεξεργασία μας αργότερα. Ακόμη δημιουργούμε ένα topic 'lateRej' με 1 partition όπου αποθηκεύουμε τα δεδομένα που αντιστοιχούν σε 10 μέρες πριν. Τα δεδομένα αυτά δεν θα τα λάβουμε υπόψιν στην επεξεργασία και θα τα παρουσιάσουμε στα διαγράμματα στο τέλος σε ξεχωριστό table. Τέλος δημιουργούμε ένα topic 'flinkAggr' με 1 partition που συλλέγει τα δεδομένα αφού γίνει η κατάλληλη επεξεργασία σε αυτά. Ο λόγος που επιλέξαμε τον συγκεκριμένο αριθμό από partitions σε κάθε topic θα γίνει πιο ξεκάθαρος στο επόμενο κεφάλαιο

B. Τρόπος χρήσης

Μέσω της εντολής confluent local services start ξεκινάμε τον zookeeper και τον kafka server. Δημιουργούμε ένα εκτελέσιμο για την δημιουργία των topics με τα αντίστοιχα partitions μέσω της εντολής ./topics_creation.sh. Με http request στο localhost:9021 ελέγχουμε αν ο broker έχει στηθεί σωστά και αν τα αντίστοιχα topics έχουν δημιουργηθεί. Το επιπλέον topic που φαίνεται στην εικόνα υπάρχει default στο confluent platform και δεν χρησιμοποιείται.



Topic name	Status	Partitions	Production (last 1 hour)	Consumption (last 1 hour)	Followers	Observers	Last updated
default.topic	Healthy	1	0B/s	0B/s	1	0	Feb 28, 2023, 10:00
default.topic	Healthy	1	0B/s	0B/s	1	0	Feb 28, 2023, 10:00
default.topic	Healthy	1	0B/s	0B/s	1	0	Feb 28, 2023, 10:00
default.topic	Healthy	1	0B/s	0B/s	1	0	Feb 28, 2023, 10:00

IV. LIVE STREAMING PROCESSING SYSTEM

Στο συγκεκριμένο στάδιο ασχολούμαστε με την live επεξεργασία δεδομένων που συλλέγονται από τον broker ώστε να κάνουμε τους κατάλληλους υπολογισμούς και να τα μετατρέψουμε στη μορφή που επιθυμούμε για να τα αποθηκεύσουμε στη βάση μας αργότερα.

A. Επεξεργασία

Για την επεξεργασία των δεδομένων χρησιμοποιούμε το εργαλείο apache flink [3] και εγκαθιστούμε την απαραίτητη βιβλιοθήκη μέσω της εντολής 'pip install apache-flink'.

Υπολογίζουμε κατάλληλα aggregations για κάθε αισθητήρα. Συγκεκριμένα για τους αισθητήρες 15λέπτου υπολογίζουμε το άθροισμα των τιμών τους ανά ημέρα για κάποιους από αυτούς ενώ για κάποιους άλλους υπολογίζουμε την μέση τιμή. Για τους ημερησίους αισθητήρες υπολογίζουμε τη διαφορά της τιμής μεταξύ των δύο ημερών.

B. Χρήση του apache flink – Source

Δημιουργούμε το εκτελέσιμο 'flinkconsumer.py'. Αρχικά ορίζουμε ως source για τα streams τον broker που δημιουργήσαμε προηγουμένως (apache kafka) καθώς και τα κατάλληλα partition sets. Έτσι δημιουργούμε πολλαπλά datastreams που διαβάζουν το καθένα απο συγκεκριμένο partition ενός topic και εκτελεί την κατάλληλη επεξεργασία στα δεδομένα αυτά.

```
#TH1, TH2
partition_set = {
    KafkaTopicPartition("dailyAggr", 0),
}

#HVaC, MiaC
partition_set1 = {
    KafkaTopicPartition("dailyAggr", 1),
}
```

```
# define the source
source = KafkaSource.builder() \
    .set_bootstrap_servers('localhost:9092') \
    .set_partitions(partition_set) \
    .set_group_id("my-group") \
    .set_starting_offsets(KafkaOffsetsInitializer.earliest()) \
    .set_value_only_deserializer(SimpleStringSchema()) \
    .build()
```

C. Windows

Για την επεξεργασία των δεδομένων δημιουργούμε τα αντίστοιχα TumblingEventTimeWindows που ομαδοποιούν δεδομένα με το ίδιο key και εκτελούν aggregations στο σύνολό τους για συγκεκριμένο time duration (στην περίπτωση μας 24 ώρες ή αλλιώς 86400 δευτερόλεπτα για τους αισθητήρες 15λεπτου). Έπειτα με κατάλληλα lambda functions εκτελούμε τα κατάλληλα aggregations στα δεδομένα κάθε window.

Για τα ασύγχρονα δεδομένα που αφορούν ημερομηνίες δύο ημερών πριν, αξιοποιούμε τη συνάρτηση allowed_lateness των windows που επιτρέπει την εισαγωγή τους στο σωστό window ακόμα και αν έχει ήδη γίνει επεξεργασία των δεδομένων σε αυτό και επιτρέπει retrigger ώστε να συμπεριληφθεί το νέο δεδομένο στα aggregations του window αυτού.

```
#W1
result3 = with_timestamp_and_watermarks3.key_by(lambda i: i[0]) \
    .window(TumblingEventTimeWindows.of(Time.seconds(86400))) \
    .allowed_lateness((2*86400+10)*1e3) \
```

Για τους ημερήσιους αισθητήρες αξιοποιούμε τα SlidingEventTimeWindows με περιθώριο χρονικό διάστημα περίπου δύο ημερών και slide μία ημέρα. Έτσι μπορούμε να υπολογίσουμε την διαφορά στην τιμή μιας μέρας λαμβάνοντας υπόψιν την τιμή της παρούσας ημέρας και της προηγούμενης.

```
#Etot
result2 = with_timestamp_and_watermarks2.key_by(lambda i: i[0]) \
    .window(SlidingEventTimeWindows.of(Time.seconds(2*86400-48), Time.seconds(86400))) \
```

Τέλος για τον υπολογισμό της διαφοράς των aggregations στη μέρα (AggregationDayRest) ενώνουμε τα αντίστοιχα streams έσω της εντολής Unions και δημιουργούμε ένα τελικό παράθυρο που υπολογίζει το τελικό aggregation.

```
resultunion = result2.union(result1)
resultunion1 = result4.union(result3)
```

```
#AggDayRestEtot
resultfinal = with_timestamp_and_watermarks union \
    .map(lambda v: (v[0], v[1], v[2]) if v[0] == 'AggDayDiffEtot') \
    .window_all(TumblingEventTimeWindows.of(Time.seconds(86400)))
```

Ο τρόπος που χωρίσαμε τα streams για να διαβάζουν δεδομένα από ξεχωριστό partition έχει να κάνει με την συνάρτηση aggregation που χρησιμοποιούμε σε κάθε τύπο δεδομένων καθώς και με τον τύπο του window που χρειάζεται για κάθε επεξεργασία (tumbling και sliding). Έτσι, δεδομένα που παράγονται από αισθητήρες στα οποία θέλουμε να βρούμε το άθροισμα των τιμών τους ανά μέρα στέλνονται στο ίδιο partition ενώ δεδομένα που θέλουμε να υπολογίσουμε την μεση τιμή τους στέλνονται σε άλλο partition. Αντίστοιχα χωρίζουμε σε άλλο partition τα 15λεπτα δεδομένα σε σχέση με τα ημερήσια καθώς απαιτούν διαφορετικό τύπο window για την επεξεργασία τους

V. TIMESERIES DATABASE

Τα δεδομένα που επεξεργαζόμαστε τα αποθηκεύουμε στην redis database [4] που έχουμε εγκαταστήσει τοπικά και ακούει στο port 6379 ενώ εφόσον επεξεργαζόμαστε δεδομένα με timestamps κάνουμε install και το module redis timeseries [5]. Ξεκινάμε τον server μέσω της εντολής: redis-server --loadmodule /home/panoplos/programs/redistimeseries.so

A. Ιδέα

Αποθηκεύουμε τα δεδομένα μας αποκλειστικά με χρήση sink connectors.

Η πρώτη μας σκέψη ήταν να αποθηκεύσουμε τα δεδομένα που επεξεργαστήκαμε στο apache flink απευθείας με χρήση sink connector από το flink κατευθείαν στη redis database μας. Ωστόσο διαβάζοντας το documentation και με αρκετό ψάξιμο συνειδητοποιήσαμε ότι το flink δεν διατηρεί πλέον connector για απευθείας σύνδεση με τη βάση redis ενώ αυτός που υπήρχε παλιότερα είναι παρωχημένος και δεν αναφέρεται στις νέες εκδόσεις του flink. Μάλιστα ανακαλύψαμε ότι βρίσκονται στη διαδικασία να φτιάξουν νέο connector ο οποίος όμως δεν έχει εκδοθεί ακόμα στον παρακάτω σύνδεσμο :<https://github.com/apache/flink-connector-redis-streams>

B. Λύση

Έτσι λοιπόν η επόμενη λύση είναι να χρησιμοποιήσουμε ένα sink connector που στέλνει τα επεξεργασμένα δεδομένα από το flink στον broker του kafka και συγκεκριμένα στο topic flinkAggr .

```
sink = KafkaSink.builder() \
    .set_bootstrap_servers('localhost:9092') \
    .set_record_serializer(
        KafkaRecordSerializationSchema.builder()
            .set_topic("flinkAggr")
            .set_value_serialization_schema(SimpleStringSchema())
            .build()
    ) \
    .set_delivery_guarantee(DeliveryGuarantee.NONE) \
    .build()
```

Έτσι λοιπόν τα δεδομένα μετά την επεξεργασία αυτόματα συλλέγονται ξανά στον broker. Στη συνέχεια επιχειρούμε με νέο sink connector να αποθηκεύσουμε τα δεδομένα στη redis database μας. Ο λόγος που εγκαταστήσαμε το confluent είναι γιατί περιέχει υλοποίηση ενός sink connector με τη βάση redis, κάτι που δεν ήταν εύκολο να υλοποιηθεί διαφορετικά. Ακολουθώντας το documentation [6] παρατηρήσαμε πως ο

connector στέλνει δεδομένα στη βάση μόνο στη μορφή (key value) με την εντολή “MSET” και όχι στη μορφή (key timestamp value) με την εντολή “TS.ADD” που χρησιμοποιούμε με το module redis timeseries.

```
1677585944.646411 [0 127.0.0.1:60822] "SELECT" "1"
1677586342.378254 [0 127.0.0.1:44082] "SELECT" "1"
1677586342.594856 [1 127.0.0.1:44082] "MSET" "key1" "value1" "key2" "value2" "key3" "value3" "key1" "value1" "key2" "value2" "key3" "value3"
1677586342.595415 [1 127.0.0.1:44082] "MSET" "__kafka.offset.users.0" "{\\\"topic\\\":\\\"users\\\",\\\"partition\\\":0,\\\"offset\\\":5}"
```

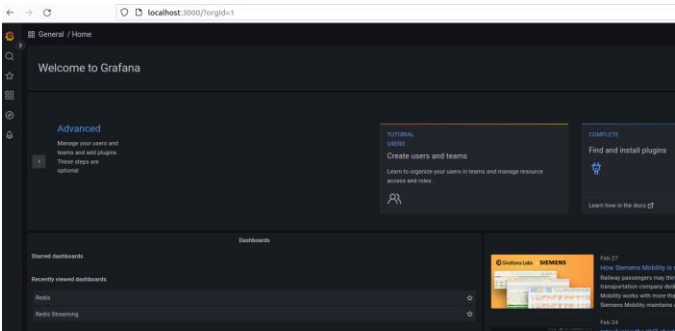
Τελικά δημιουργούμε τα scripts datains.py και datains2.py που διαβάζουν τα raw δεδομένα από το topic dailyAggr, τα επεξεργασμένα δεδομένα από το topic flinkAggr και τα ασύγχρονα δεδομένα από το topic lateRej και τα αποθηκεύουν στη βάση με το αντίστοιχο entry στη μορφή: (key timestamp value)

Με την εντολή redis-cli monitor μπορούμε να παρατηρήσουμε τις διάφορες εντολές που εκτελούνται στη βάση μας συμπεριλαμβανομένων των εντολών “ts.range” για τη λήψη δεδομένων που ανήκουν μεταξύ συγκεκριμένου χρονικού διαστήματος.

```
panoplos@panoplos-B360M-H: ~/pliroforiaka/redis$ redis-cli
127.0.0.1:6379> monitor
OK
1677608608.115767 [0 127.0.0.1:52340] "ts.range" "AggDayTH1" "1577829600000" "1578607200000"
1677608608.117356 [0 127.0.0.1:52368] "ts.range" "AggDayDiffWtot" "1577829600000" "1578607200000"
1677608609.116168 [0 127.0.0.1:52348] "ts.range" "AggDayHVAC1" "1577829600000" "1578607200000"
1677608609.118105 [0 127.0.0.1:52350] "ts.range" "Etot" "1577829600000" "1578607200000"
1677608609.121468 [0 127.0.0.1:52364] "ts.range" "LateRejW1" "1577829600000" "1578607200000"
1677608609.121742 [0 127.0.0.1:52340] "ts.range" "Mov1" "1577829600000" "1578607200000"
1677608610.125784 [0 127.0.0.1:52368] "ts.range" "AggDayRestEtot" "1577829600000" "1578607200000"
1677608610.128391 [0 127.0.0.1:52348] "ts.range" "AggDayTH1" "1577829600000" "1578607200000"
1677608611.124107 [0 127.0.0.1:52350] "ts.range" "AggDayDiffWtot" "1577829600000" "1578607200000"
1677608611.128032 [0 127.0.0.1:52364] "ts.range" "AggDayHVAC1" "1577829600000" "1578607200000"
1677608611.128232 [0 127.0.0.1:52340] "ts.range" "Etot" "1577829600000" "1578607200000"
1677608611.129165 [0 127.0.0.1:52368] "ts.range" "LateRejW1" "1577829600000" "1578607200000"
1677608611.129224 [0 127.0.0.1:52348] "ts.range" "Mov1" "1577829600000" "1578607200000"
1677608611.133894 [0 127.0.0.1:52350] "ts.range" "AggDayRestEtot" "1577829600000" "1578607200000"
```

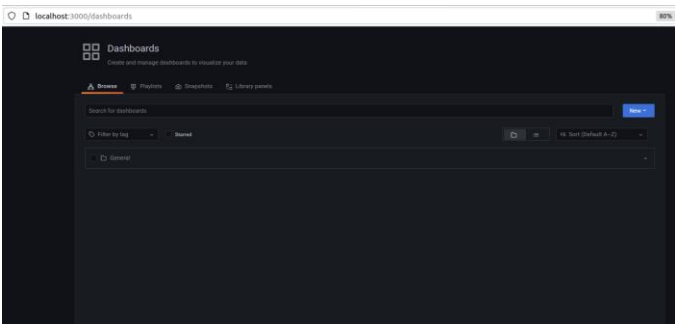
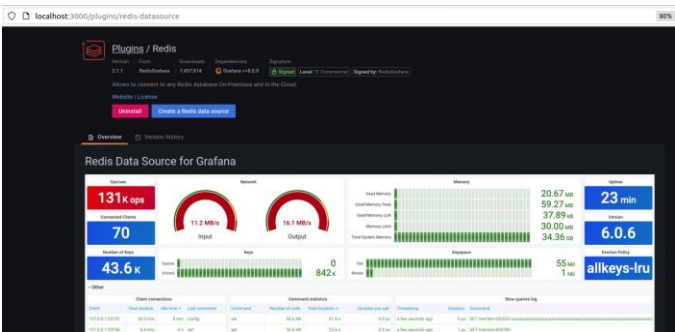
VI. PRESENTATION LAYER

Για την παρουσίαση των δεδομένων που έχουν αποθηκευτεί στη βάση χρησιμοποιούμε το εργαλείο grafana [7]. Σετάρουμε τοπικά τον grafana-server και αφού τον εκκινήσουμε αποκτούμε πρόσβαση σε αυτόν με http request στο localhost:3000.



A. Redis plugin

Αναζητούμε στα data sources το plugin για την βάση μας redis. Το κάνουμε install και προχωράμε στη δημιουργία dashboard για να παρατηρούμε τα διαγράμματα των δεδομένων καθώς αυτά εισάγονται στη βάση.



B. Παρουσίαση των δεδομένων με time series και tables

Το Grafana περιέχει ενσωματωμένα websockets για την αυτόματη ενημέρωση των διαγραμμάτων όταν προστίθενται νέα δεδομένα με καινούργιο ή παλιότερο timestamp. Δημιουργούμε 5 time charts και 2 tables. Στα time charts συμπεριλαμβάνουμε δύο διαγράμματα για raw δεδομένα: (Etot, Mov1)

και 3 διαγράμματα για aggregated δεδομένα: (AggDay(HVAC1), AggDayDiff(Wtot) και AggDayRest(Etot)).

Στα tables συμπεριλαμβανουμε ένα table για σύγχρονα aggregated δεδομένα ($AggDay(TH1) : mean_value$) και ένα table για τα ασύγχρονα δεδομένα 10 ημερών πριν ($W1$).

Time charts

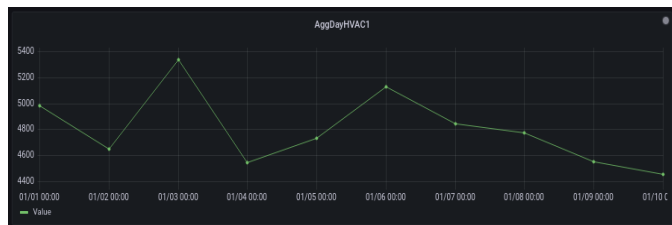
Etot (raw)



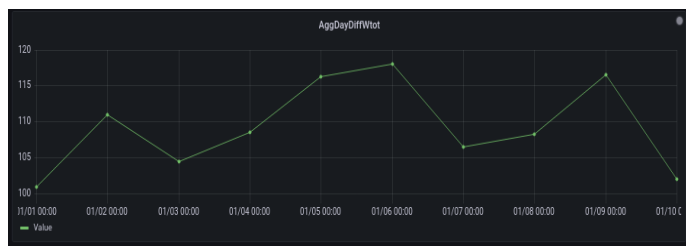
MOVI (raw)



AggDay(HVAC1)



AggDayDiff(Wtot)



AggDayRest(Etot)



Tables

AggDay(TH1)

time	Value
2020-01-01 00:00:00	22.5
2020-01-02 00:00:00	23.1
2020-01-03 00:00:00	22.9
2020-01-04 00:00:00	24.6
2020-01-05 00:00:00	23.1
2020-01-06 00:00:00	23.7

LateRejW1

time	Value
2019-12-23 05:45:00	0.800
2019-12-24 11:45:00	0.900
2019-12-25 17:45:00	0.100
2019-12-26 23:45:00	1
2019-12-28 05:45:00	0.100
2019-12-29 11:45:00	0.400

Σχόλια και συμπεράσματα σχετικά με τα διαγράμματα

Θέτουμε ως ημερομηνία εκκίνησης την 01/01/2020

Etot: παρατηρούμε γραμμική αύξηση της τιμής με τον χρόνο όπως αναμέναμε καθώς κάθε μέρα αυξάνεται με μια τιμή γύρω στο 2600x24.

MOVI: σταθερά παράγει τιμή 1 σε τυχαία timestamps μέσα στην μέρα. Βλέπουμε ότι ο ρυθμός με τον οποίο παράγονται οι άσσοι είναι απρόβλεπτος και παράγονται τυχαία.

AggDay(HVAC1) : το άθροισμα των επιμέρους τιμών κάθε μέρα. Δεν περιμέναμε κάποιο συγκεκριμένο μοτίβο

AggDayDiff(Wtot): Παρατηρούμε μικρή διακύμανση των τιμών γύρω από το 110 καθώς κάθε μέρα η τιμή αυξάνεται σε αυτό το μέγεθος. Συνεπώς υπολογίζεται σωστά η διαφορά που περιμέναμε ανά ημέρα.

AggDayRest(Etot) :

Υπολογίζει τη διαφορά $AggDayDiff(Etot) - AggDay(HVAC1) - AggDay(HVAC2) - AggDay(MiAC1) - AggDay(MiAC2)$

Γνωρίζουμε πως για το $AggDayDiff(Etot)$ ισούται με μια τιμή κοντά στο 6300 κάθε μέρα. Συνεπώς περιμένουμε το αποτέλεσμα κάθε μέρας να είναι μια θετική τιμή μικρότερη από αυτή όπως αυτή που παρουσιάζεται στο διάγραμμα (γύρω στο 30000)

AggDay(TH1):

Η μέση τιμή των τιμών του αισθητήρα TH1: παράγει τιμές μεταξύ 12-35 . Συνεπώς περιμένουμε μια μέση τιμή κοντά στο 24 όπως και συμβαίνει.

LateRejW1: Παρατηρούμε ότι παράγει και τιμές για ημερομηνίες παλιότερες της ημερομηνίας έναρξης.

ΤΕΛΙΚΑ ΣΥΜΠΕΡΑΣΜΑΤΑ

Στο παρόν άρθρο αναφερθήκαμε σε όλα τα βήματα που ακολουθήσαμε για το στήσιμο των εργαλείων, τις βιβλιοθήκες που χρησιμοποιήσαμε καθώς και τα scripts που τρέξαμε για να επιτύχουμε το στήσιμο ενός πρότυπου *iot live streaming*

system. Αναφερθήκαμε στα εμπόδια που συναντήσαμε καθώς και τους τρόπους με τον οποίο τα αντιμετωπίσαμε. Καταφέραμε τελικά να δημιουργήσουμε ένα σύγχρονο σύστημα με πλήρη επικοινωνία μεταξύ των εργαλείων που ανατεθήκαμε να χρησιμοποιήσουμε και να παρουσιάσουμε σε διαγράμματα με σύγχρονη ενημέρωση (websockets) τα τελικά αποτελέσματα.

Ο κώδικας που χρησιμοποιήθηκε καθώς και τα scripts για τα configurations των επιμέρους εργαλείων βρίσκονται στον παρακάτω σύνδεσμο [github](https://github.com).

https://github.com/PanagiotisPapadeas/Iot_project-Analysis_and_design_of_information_systems-NTUA

REFERENCES

- [1] <https://kafka.apache.org/documentation/>
- [2] <https://docs.confluent.io/platform/current/platform.html>
- [3] <https://nightlies.apache.org/flink/flink-docs-release-1.16/>
- [4] <https://redis.io/>
- [5] <https://redis.io/docs/stack/timeseries/>
- [6] <https://docs.confluent.io/kafka-connectors/redis/current/overview.html>
- [7] <https://grafana.com/docs/grafana/latest/>