SERVERLESS IOT DATA PROCESSING

Phase 4 submission :

Introduction :

Real-time data processing

1. Choose a serverless event stream processing platform. Some popular options include AWS Kinesis Firehose, Google Cloud Dataflow, and Azure Stream Analytics. These platforms can process large volumes of data in real time and scale automatically based on demand.
2. Connect your IoT devices to the serverless event stream processing platform. This can be done using a variety of protocols, such as MQTT, AMQP, and CoAP.
3. Write code to process the data in real time. The code can perform various tasks, such as data filtering, transformation, and aggregation.
4. Deploy your code to the serverless event stream processing platform. The platform will automatically scale your code up or down based on the volume of data being processed.

Automation

5. Use a serverless workflow management platform to automate your IoT data processing workflow. Some popular options include AWS Step Functions, Google Cloud Workflow Orchestration, and Azure Logic Apps. These platforms allow you to create visual workflows that can be triggered by events, such as new data arriving from your IoT devices.
6. Use serverless functions to automate specific tasks in your IoT data processing workflow. For example, you can use a serverless function to send an alert if a sensor reading exceeds a certain threshold.

Storage

7. Choose a serverless database to store your IoT data. Some popular options include Amazon DynamoDB, Google Cloud Firestore, and Azure Cosmos DB. These databases are scalable and easy to use, and they can handle both structured and unstructured data.
8. Configure your serverless event stream processing platform to store the processed data in the serverless database.
9. Use serverless functions to access and analyze the data in the serverless database.

Example

Here is an example of how to implement real-time data processing, automation, and storage in a serverless IoT data processing project:

10. Use AWS Kinesis Firehose to process the data in real time. AWS Kinesis Firehose is a serverless event stream processing platform that can process large volumes of data in real time.
11. Use AWS Step Functions to automate the IoT data processing workflow. AWS Step Functions is a serverless workflow management platform that allows you to create visual workflows that can be triggered by events.
12. Use AWS Lambda to automate specific tasks in the IoT data processing workflow. AWS Lambda is a serverless compute platform that allows you to run code without provisioning or managing servers.
13. Use Amazon DynamoDB to store the IoT data. Amazon DynamoDB is a serverless database that can handle both structured and unstructured data.

Here is a diagram of the example architecture:

[Diagram of serverless IoT data processing architecture]

Benefits of using a serverless architecture for IoT data processing

There are several benefits to using a serverless architecture for IoT data processing:

- Scalability: Serverless architectures can scale automatically based on demand, so you don't have to worry about provisioning or managing servers.
- Cost-effectiveness: Serverless architectures are pay-as-you-go, so you only pay for the resources that you use.
- Ease of use: Serverless architectures are easy to use and manage. You don't have to worry about provisioning or managing servers, and you can deploy your code with just a few clicks.

14. Choose an event source. IBM Cloud Functions supports a variety of event sources, including IBM Cloud Event Streams, IBM Cloud Object Storage, and IBM Cloud Pub/Sub.
15. Write a function to process the data. The function can perform various tasks, such as data filtering, transformation, and aggregation.
16. Deploy your function to IBM Cloud Functions.
17. Configure IBM Cloud Functions to trigger your function when an event occurs.
18. Use IBM Cloud Functions to trigger automated routines. For example, you can use a function to send an alert if a sensor reading exceeds a certain threshold.

Example

Here is an example of how to use IBM Cloud Functions to process data and trigger an automated routine in serverless IoT data processing:

19. Choose IBM Cloud Event Streams as the event source. IBM Cloud Event Streams is a fully managed event streaming service that enables you to ingest, process, and route events in real time.
20. Write a function to filter the data. The function can filter the data based on certain criteria, such as the device type or sensor reading.
21. Deploy your function to IBM Cloud Functions.
22. Configure IBM Cloud Functions to trigger your function when a new event is received from IBM Cloud Event Streams.
23. Use IBM Cloud Functions to trigger an automated routine. For example, you can use a function to send an email alert if a sensor reading exceeds a certain threshold.

Benefits of using IBM Cloud Functions for IoT data processing

There are several benefits to using IBM Cloud Functions for IoT data processing:

- Scalability: IBM Cloud Functions can scale automatically based on demand, so you don't have to worry about provisioning or managing servers.
- Cost-effectiveness: IBM Cloud Functions is pay-as-you-go, so you only pay for the resources that you use.
- Ease of use: IBM Cloud Functions is easy to use and manage. You don't have to worry about provisioning or managing servers, and you can deploy your code with just a few clicks.
- Integration with other IBM Cloud services: IBM Cloud Functions integrates with other IBM Cloud services, such as IBM Cloud Event Streams, IBM Cloud Object Storage, and IBM Cloud Pub/Sub. This makes it easy to build serverless IoT data processing applications.

24. Create an IBM Cloud Object Storage bucket.
25. Configure IBM Cloud Functions to write the processed data to the IBM Cloud Object Storage bucket.
26. Use a serverless data analytics platform to analyze the data in IBM Cloud Object Storage. Some popular options include IBM Cloud Data Engine, IBM Cloud Analytics Engine, and IBM Cloud Watson Studio.

Example

Here is an example of how to store processed data in IBM Cloud Object Storage for analysis in a serverless IoT data processing project:

27. Create an IBM Cloud Object Storage bucket. You can create an IBM Cloud Object Storage bucket from the IBM Cloud console.
28. Configure IBM Cloud Functions to write the processed data to the IBM Cloud Object Storage bucket. You can configure IBM Cloud Functions to write the processed data to the IBM Cloud Object Storage bucket in the IBM Cloud Functions console.
29. Use IBM Cloud Data Engine to analyze the data in IBM Cloud Object Storage. IBM Cloud Data Engine is a fully managed, serverless data warehouse that enables you to run SQL queries on data stored in IBM Cloud Object Storage.

To use IBM Cloud Data Engine to analyze the data in IBM Cloud Object Storage, you can follow these steps:

30. Create an IBM Cloud Data Engine instance. You can create an IBM Cloud Data Engine instance from the IBM Cloud console.
31. Connect IBM Cloud Data Engine to the IBM Cloud Object Storage bucket. You can connect IBM Cloud Data Engine to the IBM Cloud Object Storage bucket in the IBM Cloud Data Engine console.
32. Run SQL queries on the data in IBM Cloud Object Storage. You can run SQL queries on the data in IBM Cloud Object Storage from the IBM Cloud Data Engine console or from a SQL client.

Benefits of storing processed data in IBM Cloud Object Storage

There are several benefits to storing processed data in IBM Cloud Object Storage:

- Scalability: IBM Cloud Object Storage is scalable, so you can store any amount of data.
- Durability: IBM Cloud Object Storage is durable, so your data is protected from loss or corruption.
- Cost-effectiveness: IBM Cloud Object Storage is cost-effective, and you only pay for the storage that you use.
- Ease of use: IBM Cloud Object Storage is easy to use, and you can access your data from anywhere in the world.

Conclusion

Serverless architectures are a good choice for IoT data processing because they are scalable, cost-effective, and easy to use. You can use serverless event stream processing platforms, serverless workflow management platforms, and serverless databases to implement real-time data processing, automation, and storage in your IoT data processing projects.

Python

```
import json

def process_data(event):
  """This function processes the data received from the IoT
device."""

  # Get the sensor reading from the event data.
  sensor_reading = event["data"]["sensor_reading"]

  # Filter the data based on certain criteria.
  if sensor_reading > 100:
    # Trigger an automated routine.
    send_alert_email()

def send_alert_email():
  """This function sends an alert email."""

  # Create an email message.
  email_message = """
  Subject: Sensor alert!

  The sensor reading has exceeded the threshold.
  """
```

```
  # Send the email message.
  send_email(email_message)

# Register the function to be triggered when an event is
received from the IoT device.
IBMCloudFunctions.register_function(process_data,
"iot_device_event")
```

This function will be triggered whenever an event is received from the IoT device. The function will process the data and filter it based on certain criteria. If the sensor reading exceeds the threshold, the function will trigger an automated routine to send an alert email.

You can deploy this function to IBM Cloud Functions by following the steps below:

33. Create an IBM Cloud Functions account.
34. Create a new IBM Cloud Functions function.
35. Paste the Python code above into the function editor.
36. Click the Deploy button.

Once the function is deployed, you can configure IBM Cloud Functions to trigger the function when an event is received from the IoT device. You can do this by following the steps below:

37. Go to the IBM Cloud Functions console.
38. Select the function that you just deployed.
39. Click the Triggers tab.
40. Click the Add trigger button.
41. Select the IoT device event trigger type.
42. Select the IoT device and event type that you want to trigger the function.
43. Click the Save button.

Now, whenever the IoT device sends an event, the function will be triggered to process the data and trigger any necessary automated routines.

You can use this same approach to write IBM Cloud Functions source code for processing data and triggering automated routines in other types of serverless IoT data processing projects.

Creating serverless IoT data processing code involves multiple steps, and the exact implementation details can vary depending on your chosen cloud platform and programming language. Here's a simplified example using AWS Lambda in Python to get you started:

1. Set up your IoT Device: Ensure that your IoT device is configured to send data to AWS IoT Core.

2. AWS Lambda Function: Create an AWS Lambda function to process incoming IoT data. You can use the AWS Management Console or the AWS CLI to create the function. For this example, let's assume your IoT data is sent as JSON payloads.

3. Coding the Lambda Function:

```
python

import json



def lambda_handler(event, context):

    # Extract IoT data from the event

    iot_data = json.loads(event['body'])
```

```
    # Perform data processing tasks here
    processed_data = process_iot_data(iot_data)

    # Implement automation actions
    if condition_for_action(processed_data):
        perform_action(processed_data)

    # Return a response, if necessary
    response = {
        "statusCode": 200,
        "body": json.dumps({"message": "Data processed successfully"})
    }

    return response

def process_iot_data(iot_data):
    # Implement your data processing logic here
    # Example: Normalize data, detect anomalies, or perform calculations
```

```python
    processed_data = iot_data  # Replace with your processing code

    return processed_data


def condition_for_action(processed_data):

    # Implement a condition to trigger an action

    # Example: Check if a certain value exceeds a threshold

    return processed_data['value'] > 100


def perform_action(processed_data):

    # Implement the action to be taken

    # Example: Send a notification, update a database, or trigger another IoT device

    print("Action performed: ", processed_data)
```

4. Integration: Configure your AWS IoT Core to send messages to your Lambda function as a trigger. You can do this by creating a rule in AWS IoT Core that forwards the data to your Lambda function.

5. Real-Time Processing: Your Lambda function will now process the incoming IoT data in real-time according to the code you've written. You can implement data transformations, validations, and automation actions as needed.

Remember that this is a simplified example, and in a real-world scenario, you'll need to handle error handling, security, and other considerations. Additionally, the exact code and configuration may vary depending on your IoT platform and specific requirements.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import datetime
%matplotlib inline
import scipy.integrate as integrate
from scipy.optimize import curve_fit
pd.options.display.max_rows = 12
```

```
data = pd.read_csv("../input/data.txt", sep = ' ',header = None, names = ['date',
'time','epoch','moteid','temp','humidity','light','voltage'])
data.head()
```

Out[2]:

	DATE	TIME	EPOCH	MOTEID	TEMP	HUMIDITY	LIGHT	VOLTAGE
0	2004-03-31	03:38:15.757551	2	1.0	122.1530	-3.91901	11.04	2.03397
1	2004-02-28	00:59:16.02785	3	1.0	19.9884	37.09330	45.08	2.69964
2	2004-02-28	01:03:16.33393	11	1.0	19.3024	38.46290	45.08	2.68742
3	2004-02-28	01:06:16.013453	17	1.0	19.1652	38.80390	45.08	2.68742
4	2004-02-28	01:06:46.778088	18	1.0	19.1750	38.83790	45.08	2.69964

```
data.shape
```

Out[3]:

```
(2313682, 8)
```

In [4]:

```python
data.describe()
```

data.describe()

Out[4]:

	EPOCH	MOTEID	TEMP	HUMIDITY	LIGHT	VOLTAGE
count	2.313682e+06	2.313156e+06	2.312781e+06	2.312780e+06	2.219804e+06	2.313156e+06
mean	3.303993e+04	2.854412e+01	3.920700e+01	3.390814e+01	4.072110e+02	2.492552e+00
std	1.836852e+04	5.062408e+01	3.741923e+01	1.732152e+01	5.394276e+02	1.795743e-01
min	0.000000e+00	1.000000e+00	-3.840000e+01	-8.983130e+03	0.000000e+00	9.100830e-03
25%	1.757200e+04	1.700000e+01	2.040980e+01	3.187760e+01	3.956000e+01	2.385220e+00
50%	3.332700e+04	2.900000e+01	2.243840e+01	3.928030e+01	1.582400e+02	2.527320e+00
75%	4.778900e+04	4.100000e+01	2.702480e+01	4.358550e+01	5.372800e+02	2.627960e+00
max	6.553500e+04	6.540700e+04	3.855680e+02	1.375120e+02	1.847360e+03	1.856000e+01

```python
data.isnull().sum()
```

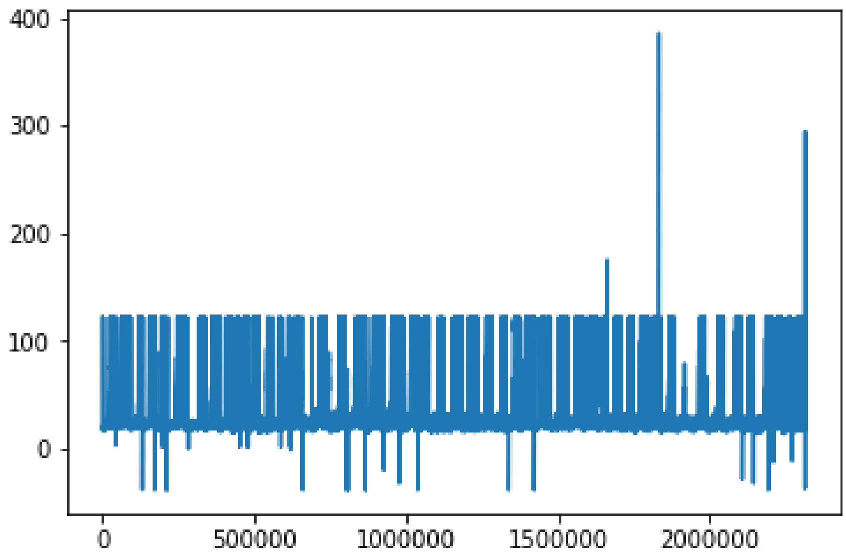
Out[5]:

```
date            0
time            0
epoch           0
moteid        526
temp          901
humidity      902
light       93878
voltage       526
dtype: int64
```

In [6]:

```python
data['temp'].plot()
```

<matplotlib.axes._subplots.AxesSubplot at 0x7f83485fb1d0>




```
data.fillna(0, inplace=True)
data['epoch'].replace(regex=True, inplace=True, to_replace=r'[^0-9.\-]',
value=r'')
data['epoch'] = data['epoch'].astype(int)
```

In [8]:

```
data.isnull().sum()
```

date 0
time 0
epoch 0
moteid 0

```
temp        0
humidity    0
light       0
voltage     0
dtype: int64
```

In [9]:

```python
data.groupby('moteid').mean()
```

epoch	temp	humidity	light	voltage	
moteid					
0.0	22031.159696	0.000000	0.000000	0.000000	0.000000
1.0	31994.793830	35.882437	34.319280	156.575828	2.519643
2.0	36491.176404	40.201817	34.298739	212.159717	2.458436
3.0	33013.771411	33.715289	34.787174	146.049674	2.517961
4.0	32688.045167	45.464410	29.991159	155.491512	2.468170
5.0	5836.428571	0.000000	0.000000	0.000000	2.516088
...	...	...	...	...	...
56.0	54586.336657	20.987656	39.261422	104.099945	2.887558
57.0	1499.000000	0.000000	17.887833	0.000000	2.482020
58.0	53830.011773	21.014753	38.541575	364.111968	2.899639
6485.0	35112.000000	262.656000	0.000000	0.000000	0.393324
33117.0	30493.000000	-36.204800	0.000000	0.000000	0.144859
65407.0	18182.000000	294.251000	0.000000	0.000000	0.013056

```python
data['Timestamp'] = data[['date', 'time']].apply(lambda x: '
'.join(x.astype(str)), axis=1)
new_data = data
```

In [11]:

```python
data.drop(['date','time'],axis=1,inplace =True)
data.set_index(pd.to_datetime(data.Timestamp), inplace=True)
```

In [12]:

```python
data[['moteid','temp','humidity','light','voltage']] =
data[['moteid','temp','humidity','light','voltage']].apply(pd.to_numeric)
```

```python
data['moteid'].value_counts()
```

Out[13]:

```
31.0       65694
29.0       64391
23.0       62440
26.0       61521
22.0       60165
21.0       58525
           ...
0.0          526
5.0           35
57.0           3
65407.0        1
6485.0         1
33117.0        1
Name: moteid, Length: 62, dtype: int64
```

```python
moteid_grp = data.groupby(['moteid'])
```

In [15]:

```
corr_id = moteid_grp.corr(method='pearson')
corr_id.fillna(0, inplace=True)
corr_id
```

epoch	humidity	light	temp	voltage		
moteid						
0.0	epoch	1.0	0.000000	0.000000	0.000000	0.000000
	humidity	0.0	0.000000	0.000000	0.000000	0.000000
	light	0.0	0.000000	0.000000	0.000000	0.000000
	temp	0.0	0.000000	0.000000	0.000000	0.000000
	voltage	0.0	0.000000	0.000000	0.000000	0.000000
1.0	epoch	1.0	-0.195395	0.014605	0.315767	-0.726957
...	...	...	...	...	...	...
33117.0	voltage	0.0	0.000000	0.000000	0.000000	0.000000
65407.0	epoch	0.0	0.000000	0.000000	0.000000	0.000000
	humidity	0.0	0.000000	0.000000	0.000000	0.000000
	light	0.0	0.000000	0.000000	0.000000	0.000000
	temp	0.0	0.000000	0.000000	0.000000	0.000000
	voltage	0.0	0.000000	0.000000	0.000000	0.000000

310 rows × 5 columns

epoch	moteid	temp	humidity	light	voltage	Timestamp	
Timestamp							
2004-03-31 03:38:15.757551	2	1.0	122.1530	-3.91901	11.04	2.03397	2004-03-31 03:38:15.757551
2004-02-28 00:59:16.027850	3	1.0	19.9884	37.09330	45.08	2.69964	2004-02-28 00:59:16.02785
2004-02-28 01:03:16.333930	11	1.0	19.3024	38.46290	45.08	2.68742	2004-02-28 01:03:16.33393

2004-02-28 01:06:16.013453	17	1.0	19.1652	38.80390	45.08	2.68742	2004-02-28 01:06:16.013453
2004-02-28 01:06:46.778088	18	1.0	19.1750	38.83790	45.08	2.69964	2004-02-28 01:06:46.778088

Temperature_data

de22	Node23	Node24	Node25	Node26	Node27	Node28	Node29					
Timestamp	Timestamp	Timestamp	Timestamp									
2004	3	1	0	18.892649	17.888361	18.682962	17.909755	17.859044	17.673663	18.159542	16.981087	17.684646
			1	17.828778	17.513253	18.301935	17.535843	17.494400	17.259217	17.705163	16.576858	17.429180
			2	17.478076	17.269043	18.038685	17.276184	17.222923	16.988568	17.443851	16.263400	17.193869
			3	17.265336	17.036493	17.833192	17.068284	17.011023	16.778655	17.242236	16.098262	17.046103
			4	16.976067	16.728080	17.526390	16.730619	16.708245	16.424560	16.829928	15.745938	16.794936
			5	16.678958	16.295632	17.130163	16.236195	16.223800	15.896133	16.312522	15.345038	16.430650
		...	...	...	...	...	...	...	...	...	...	...
		21	18	21.580820	20.955944	24.125349	32.503942	21.184272	21.099364	22.016777	20.406817	21.141881
			19	21.213658	20.334054	24.385489	35.667856	20.598625	20.551900	21.721577	19.656331	20.547413
			20	20.893717	19.951263	24.994137	40.132790	20.216815	20.256095	21.630513	19.215485	20.144796
			21	20.408241	19.569963	25.837693	45.156295	19.830846	19.937289	21.666446	18.845821	19.723914
			22	20.171971	19.167160	27.153425	51.666562	19.390084	19.724252	21.884614	18.476750	19.241492
			23	19.997640	18.620570	28.617012	59.078113	18.886997	19.474558	22.185723	17.960217	18.716850

463 rows × 9 columns

In [19]:

```
Humidity_data
```

Out[19]:

				Node14	Node22	Node23	Node24	Node25	Node26	Node27	Node28	Node29
Timestamp	Timestamp	Timestamp	Timestamp									
2004	3	1	0	42.549285	44.530996	43.168231	44.878085	44.633242	47.021085	45.492716	48.549510	46.209795
			1	45.355191	45.693293	44.364838	46.024384	45.775104	48.405524	46.959683	49.996614	46.909245
			2	46.163976	46.084623	44.710712	46.487651	46.130820	48.831494	47.188011	50.559377	47.271673
			3	46.613439	46.469025	45.070823	46.854633	46.512194	49.220666	47.632574	50.843643	47.565288
			4	47.179816	47.056620	45.602291	47.544280	47.070312	49.981491	48.445950	51.645920	48.041444
			5	47.897721	48.318876	46.838235	49.095143	48.524784	51.741095	50.494748	52.944230	49.184805
		...	...	...	...	...	...	...	...	...	...	...
		21	18	48.074626	47.581688	46.317189	48.043332	45.830167	48.557737	46.804405	48.878659	46.799120
			19	48.199534	48.236579	46.810787	48.842767	46.393657	49.506822	47.598095	50.102531	47.288588
			20	48.285621	48.535250	47.076702	49.253592	46.754700	49.895563	48.139502	50.566962	47.463996
			21	49.033441	48.811621	47.416511	49.718356	47.142598	50.570302	48.718429	50.572297	47.832622
			22	49.483676	49.228436	47.773140	50.317610	47.654925	51.163117	49.398291	50.813785	48.433595
			23	49.850957	50.196400	48.584533	51.448840	48.664630	52.343430	50.653467	51.896545	49.429506

Covariance models in time for the temperature

Temperature time lags from 0 to 9

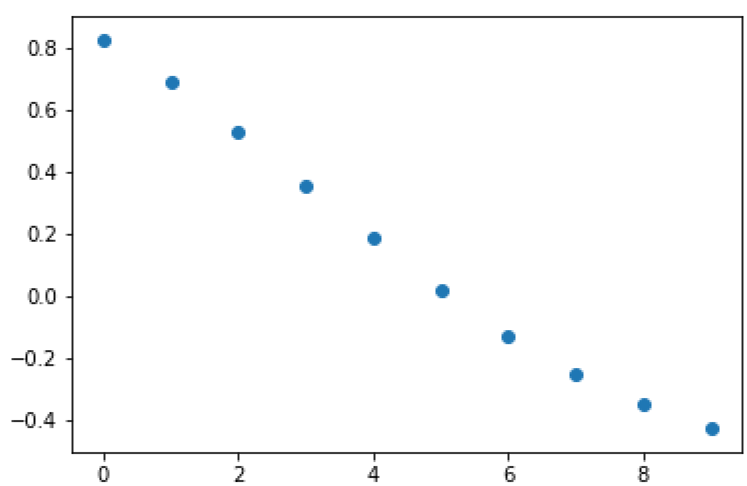
In [20]:

```
node_distance = pd.read_csv('../input/intel_nodes_distances.csv')

lists_hy = []
```

```
lists_hx = []

for z in range(10):
    # Calculation of the covariance for lag i hour for temp
    humy = [None] *45
    humx = [None] *45
    k=0
    i=0
    for first_column in Temperature_data:
        df1 = Temperature_data[first_column][:-(1+z)]
        std_test1=np.std(df1)
        j = 0
        for second_column in Temperature_data:
            if j <= i:
                df2 = Temperature_data[second_column][(1+z):]
                std_test2=np.std(df2)
                humy[k] = (np.cov(df1,df2)/(std_test1*std_test2)).item((0, 1))
                humx[k] = node_distance.iloc[i,j]
                j = j + 1
                k = k + 1
        i = i + 1
    lists_hy.append(humy)
    lists_hx.append(humx)
```

Out[21]:

<matplotlib.collections.PathCollection at 0x7f8327bc9550>



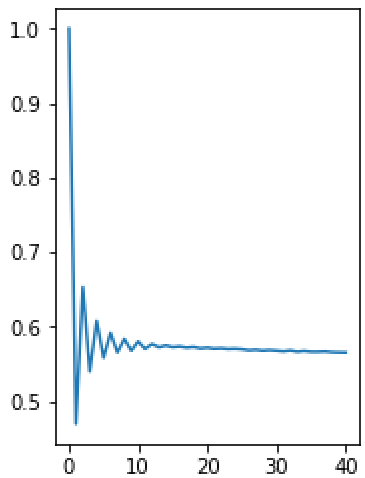
In [22]:

```
from statsmodels.tsa.stattools import acf,pacf
lag_acf = acf(data['temp'])
```

#Plot pACF: a
plt.subplot(121)
plt.plot(lag_acf)

Out[22]:

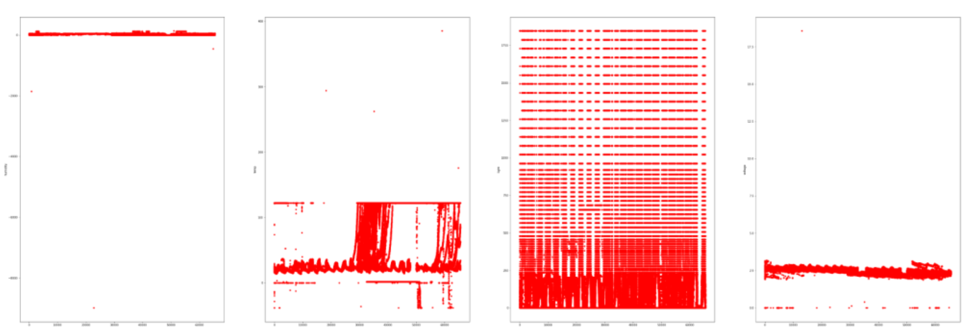
```
[<matplotlib.lines.Line2D at 0x7f8326a94470>]
```



Variation in humidity, temp, light, voltage with epoch

In [23]:

```
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(60,20))

for xcol, ax in zip(['humidity', 'temp', 'light','voltage'], axes):
    data.plot(kind='scatter', x='epoch', y=xcol, ax=ax, alpha=1, color='r')
```

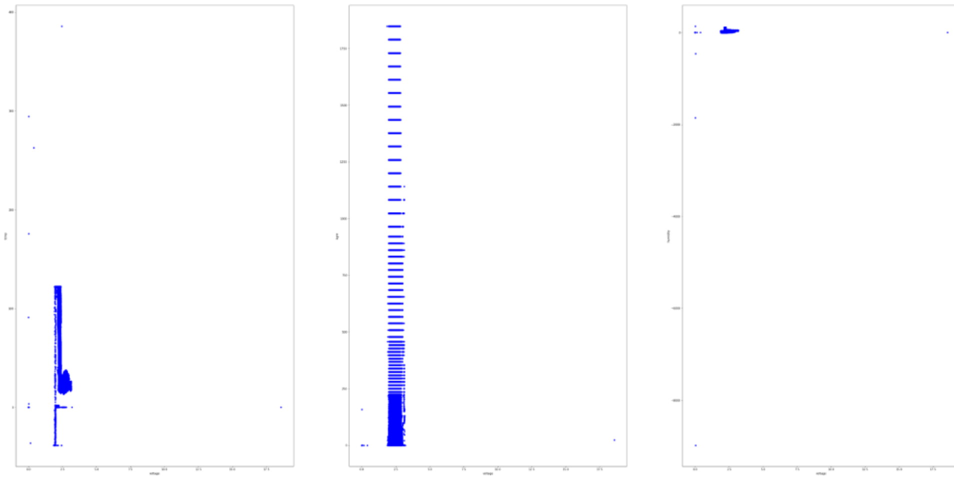


Variation with Voltage

In [24]:

```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(60,30))
for xcol, ax in zip(['temp', 'light','humidity'], axes):
```

```
    data.plot(kind='scatter', x='voltage', y=xcol, ax=ax, color='b')
```



Variation with light

In [25]:

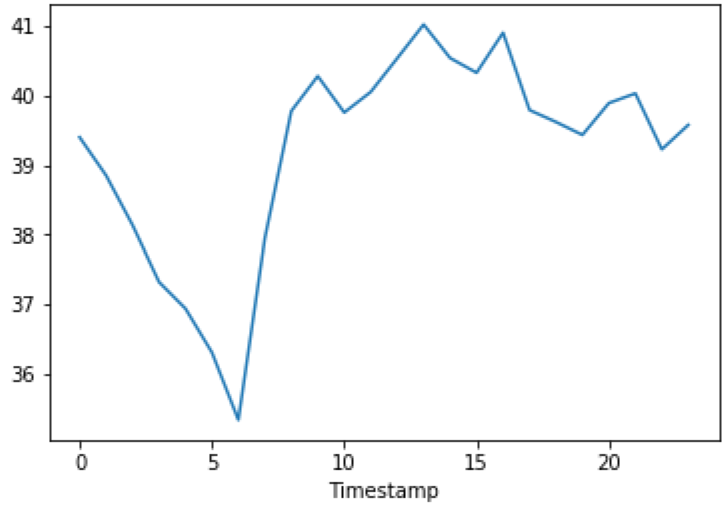
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(60,30))
for xcol, ax in zip(['temp', 'humidity','voltage'], axes):
    data.plot(kind='scatter', x='light', y=xcol, ax=ax, alpha=1, color='g')
```

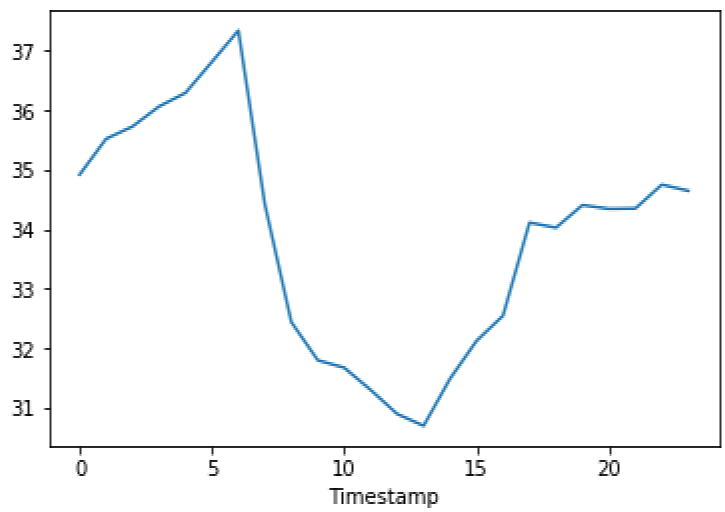
Variation with Humidity

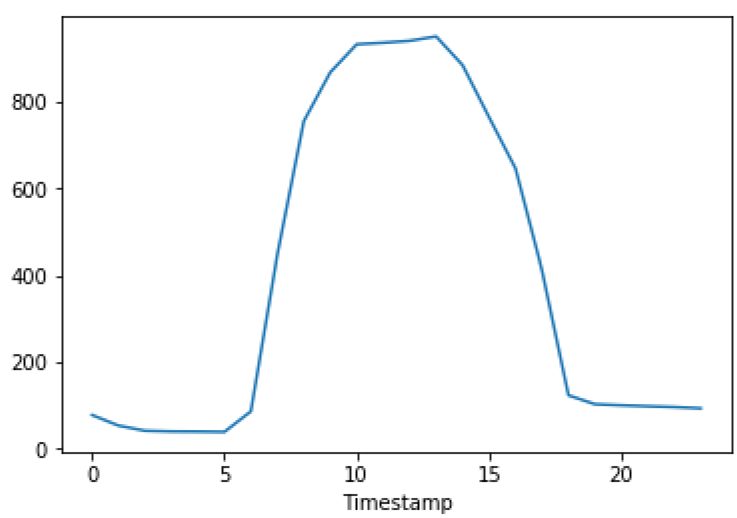
In [26]:

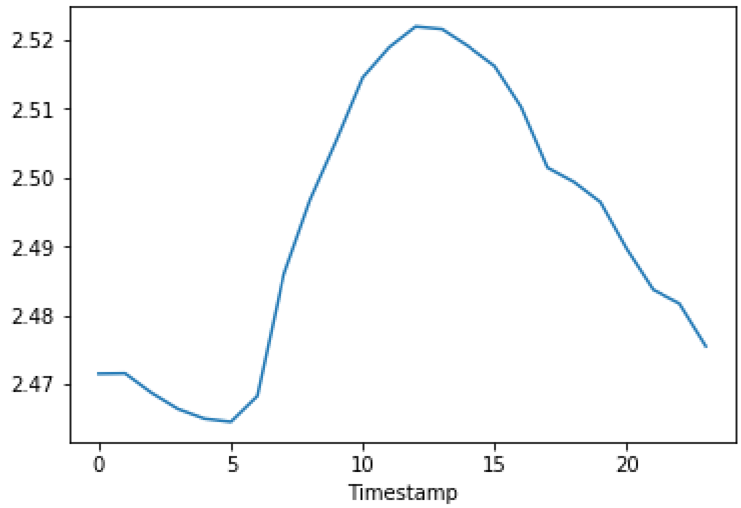
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(60,30))
for xcol, ax in zip(['temp', 'light','voltage'], axes):
    data.plot(kind='scatter', x='humidity', y=xcol, ax=ax, alpha=1,
```

`color='y'`Variation in Tempeature, Humidity, light and Voltage over time









In [29]:

```
new_data.plot(subplots=True,linewidth=0.5,
                layout=(2, 4),figsize=(60, 20),
                sharex=False,
                sharey=False)

plt.show()
```

Pearson Correlation for the multivariate time series

In [30]:

```
new_data.corr(method='pearson')
```

Out[30]:

	EPOCH	MOTEID	TEMP	HUMIDITY	LIGHT	VOLTAGE
epoch	1.000000	0.006596	0.340746	-0.221278	0.041511	-0.654189

moteid	0.006596	1.000000	-0.014062	0.023541	0.029538	0.001789
temp	0.340746	-0.014062	1.000000	-0.707251	0.021948	-0.737583
humidity	-0.221278	0.023541	-0.707251	1.000000	-0.095084	0.507445
light	0.041511	0.029538	0.021948	-0.095084	1.000000	0.055835
voltage	-0.654189	0.001789	-0.737583	0.507445	0.055835	1.000000

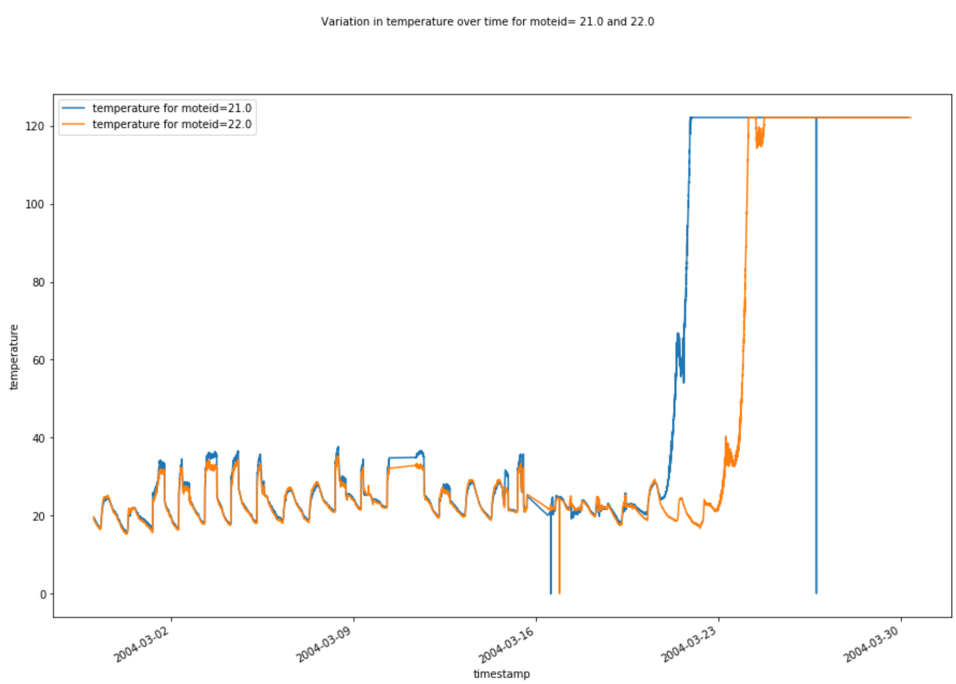
Variation in temperature readings over time for moteid's: 21 and 22

In [31]:

```
from matplotlib import pyplot as plt
d_m21 = data.loc[data['moteid'] == 21.0]
d_m22 = data.loc[data['moteid'] == 22.0]
d_m10 = data.loc[data['moteid'] == 10.0]
fig2 = plt.figure(figsize = (15,10))
d_m21['temp'].plot(label='temperature for moteid=21.0')
d_m22['temp'].plot(label='temperature for moteid=22.0')
fig2.suptitle('Variation in temperature over time for moteid= 21.0 and 22.0',
fontsize=10)
plt.xlabel('timestamp', fontsize=10)
plt.ylabel('temperature', fontsize=10)
plt.legend()
```

Out[31]:

<matplotlib.legend.Legend at 0x7f8327b78ba8>



Anomaly Detection using moving average method

For moteid:10 and window size: 20, we calculate the mean and standard deviation of the data.If the next entry in the dataframe lies between mean(+-)sd*2, it is considered normal else it is considered an anamoly.

Anomaly can be seen by blue *

In [32]:

```
from itertools import count
import matplotlib.pyplot as plt
from numpy import linspace, loadtxt, ones, convolve
import numpy as np
import pandas as pd
import collections
from random import randint
from matplotlib import style
%matplotlib inline
def mov_average(data, window_size):

    window = np.ones(int(window_size))/float(window_size)
    return np.convolve(data, window, 'same')
def find_anomalies(y, window_size, sigma=1.0):
    avg = mov_average(y, window_size).tolist()
    residual = y - avg
    std = np.std(residual)
    return {'standard_deviation': round(std, 3),
            'anomalies_dict': collections.OrderedDict([(index, y_i) for index,
y_i, avg_i in zip(count(), y, avg)
              if (y_i > avg_i + (sigma*std)) | (y_i < avg_i - (sigma*std))])}
def plot_results(x, y, window_size, sigma_value=1,
                 text_xlabel="X Axis", text_ylabel="Y Axis",
applying_rolling_std=False):

    plt.figure(figsize=(15, 8))
    plt.plot(x, y, "k.")
    y_av = mov_average(y, window_size)
    plt.plot(x, y_av, color='green')
    plt.xlim(0, 40000)
    plt.xlabel(text_xlabel)
    plt.ylabel(text_ylabel)
    events = {}
    events = find_anomalies(y, window_size=window_size, sigma=sigma_value)
```

```
    x_anom = np.fromiter(events['anomalies_dict'].keys(), dtype=int,
count=len(events['anomalies_dict']))
    y_anom = np.fromiter(events['anomalies_dict'].values(),
dtype=float,count=len(events['anomalies_dict']))
    plt.plot(x_anom, y_anom, "b*")
    print(x_anom)
    plt.grid(True)
    plt.show()
x = d_m10['epoch']
Y = d_m10['temp']
plot_results(x, y=Y, window_size=50, text_xlabel="Date",
sigma_value=3,text_ylabel="temperature")
```

```
[23743 23751 23761 23999 24206 24302 24303 24324 24350 25415 26094
26101
 26325 26336 26371 26422 26437 26549 26551 26581 26588 26622 26624
26636
 26713 26723 29505 29506 29507 36012 36017 40733 40748 40775 40779
40787
 40840 40841 40848 40982 41115 41201]
```