A 101 in time series analytics with Apache Arrow, Pandas and Parquet

Zoe Steinkamp

Zoe Steinkamp
Developer Advocate - InfluxData

LinkedIn

influxdata®

Agenda

- Why a time series database is important
- Tools to know
- What is apache parquet and apache arrow
- Apache arrow example
- Leveraging pandas for analytics
- Examples
  - Autocorrelation, Anomaly Detection, Forecasting
- Resources

 |

Audience Questions

Have you worked with time series data?

Have you worked with a time series database?

 |

influxdata®

Why a time series database is important

Types of time series data

Metrics

Quantitative values collected ***regularly*** over time

Events

State changes or values generated ***irregularly*** over time

Traces

Complete event or request propagation in a distributed system

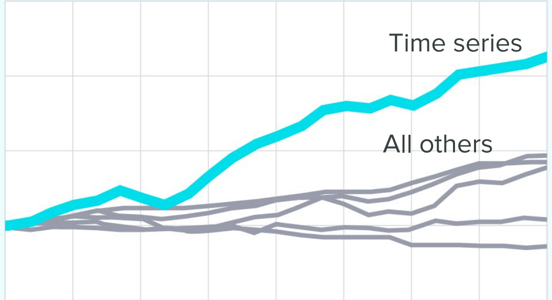
 |

influxdata®

Rise of time series as a category

RELATIONAL	DOCUMENT	SEARCH	**TIME SERIES**
• Orders • Customers • Records	• High throughput • Large document	• Distributed search • Logs • Geo	• Events, metrics, time-stamped • For IoT, analytics, cloud native
PostgreSQL	MongoDB.	elastic	influxdata®

Time series is fastest growing data category by far



source: DB Engines

 |

Typical Architecture & Deployment

InfluxDB Platform

influxdb™ 3.0

Data Sources
All time-stamped data

Metrics
Sensors
Events
Devices
& more

Data Collection
Multiple data ingest methods

Telegraf
Client libraries
& more

Data Storage & Transformation
Purpose-built time series database

Collect
Store
SQL queries

+ INTEGRATIONS

Data Visualization & Analysis

Machine learning
Analytics tools
BI tools
& more

influxdata®

Tools to know

Telegraf

The open-source agent for collecting metrics

Driven by the community (600+ contributors)

Simple to configure, extremely flexible

influxdata®

Categories of Telegraf Plugins

Consumer IoT	Industrial IoT	Databases
AMQP MQTT	KNX OPC-UA Modbus	InfluxDB Listener MySQL Elasticsearch MongoDB
Containers	**Web**	**Logging**
Docker Kubernetes Podman	NGINX Apache Cloudflare	Syslog File/Tail OpenTelemetry
Cloud	**Networking**	**Gaming / Entertainment**
AWS Cloudwatch Google Cloud Pub Sub	SNMP GNMI Jolokia TCP/UDP Listener	NVIDIA SMI AMD Minecraft CS:Go

SQL Aggregate Functions

- General aggregate functions
  - array_agg
  - avg
  - count
  - max
  - mean
  - median
  - min
  - sum

- Statistical aggregate functions
  - corr
  - covar
  - covar_pop
  - covar_samp
  - stddev
  - stddev_pop
  - stddev_samp
  - var
  - var_pop
  - var_samp

- Approximate aggregate functions
  - approx_distinct
  - approx_median
  - approx_percentile_cont
  - approx_percentile_cont_with_weight

 |

influxdata

Open Source tools

Pandas -
Python library used for data manipulation. It is common for other tools to expect a pandas dataframe data format.

TensorFlow -
machine learning and artificial intelligence platform. Data scientists use TensorFlow to build and train models using Python or JavaScript.

ADTK -
Python package for rule-based anomaly detection in time-series data. ADTK is geared toward industrial IoT use cases.

Prophet -
Python library for forecasting. It fits the forecasting problem with a curve-fitting exercise or creating a mathematical model.

What is apache parquet and apache arrow

Apache Parquet…

“Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval.”

The benefits

- Minimize disk usage while storing gigabytes of data
- Efficient retrieval and deserialization of large amounts of columnar data

What is the difference between Apache Arrow and Apache Parquet?

- Not a runtime in-memory format
- Parquet data cannot be directly operated on but must be decoded in large chunks

https://dzone.com/articles/how-to-be-a-hero-with-powerful-parquet-google-and

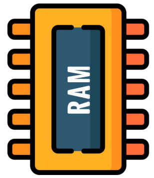
Comparison

Dataset	Size on Amazon S3	Query Run time	Data Scanned	Cost
Data stored as CSV files	1 TB	236 seconds	1.15 TB	$5.75
Data stored in Apache Parquet format*	130 GB	6.78 seconds	2.51 GB	$0.01
Savings / Speedup	87% less with Parquet	34x faster	99% less data scanned	99.7% savings

 |

influxdata®

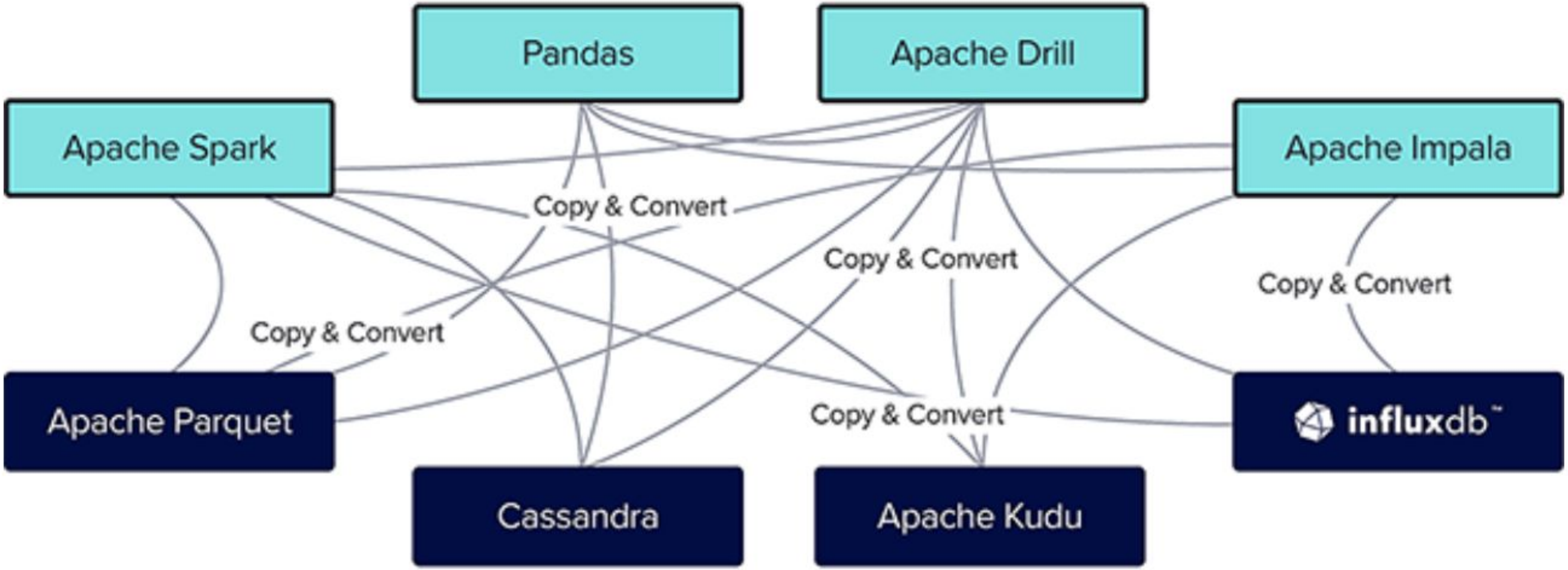
Apache Arrow is...



“Apache Arrow is a framework for defining in-memory columnar data that every processing engine can use.”

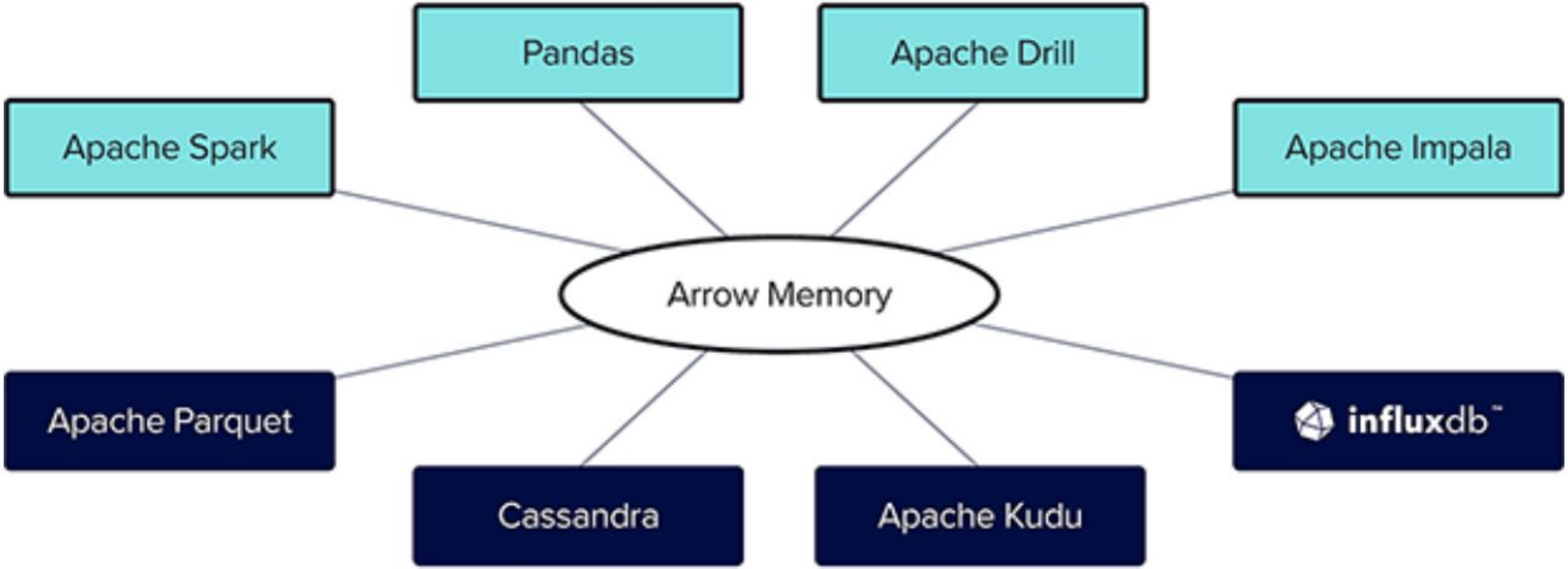
- Language-agnostic standard for columnar memory
- Efficient for running large analytical workloads on modern CPU and GPU architectures.
- It supports s a range of **programming languages including C++, Java, Python, and R.**

The problems



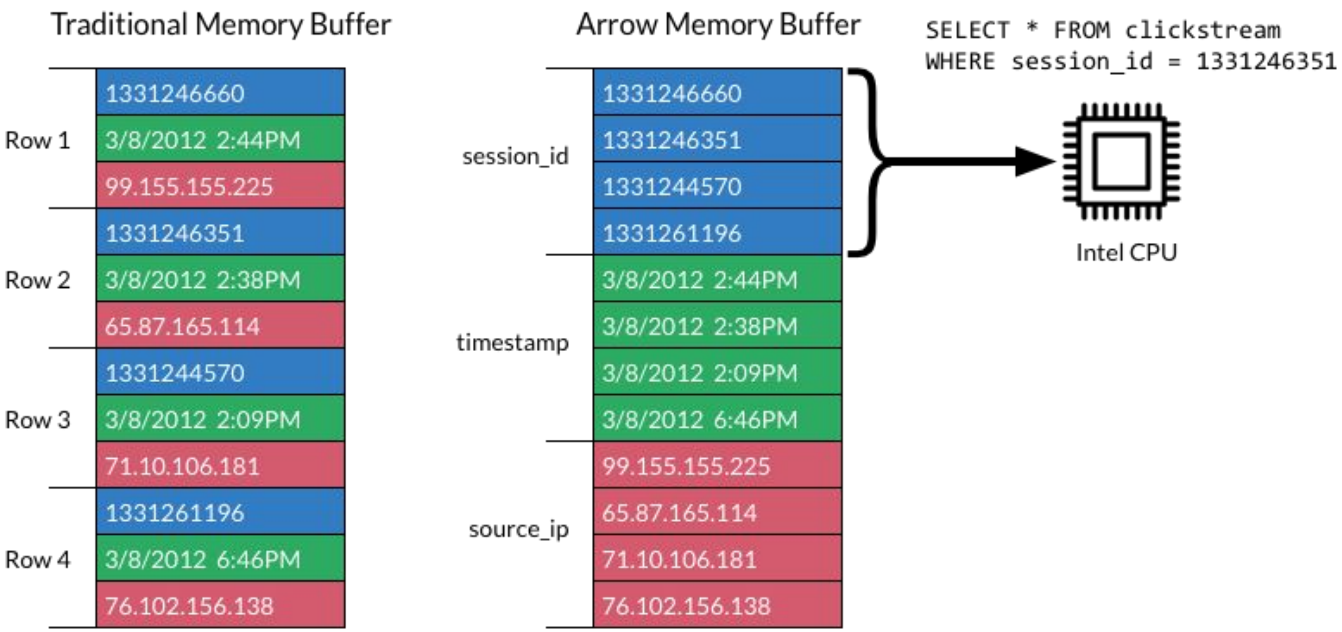
influxdata

The solution



Apache Arrow is…

	session_id	timestamp	source_ip
Row 1	1331246660	3/8/2012 2:44PM	99.155.155.225
Row 2	1331246351	3/8/2012 2:38PM	65.87.165.114
Row 3	1331244570	3/8/2012 2:09PM	71.10.106.181
Row 4	1331261196	3/8/2012 6:46PM	76.102.156.138



The Apache Arrow format allows computational routines and execution engines to maximize their efficiency when scanning and iterating large chunks of data. In particular, the contiguous columnar layout enables vectorization using the latest SIMD (Single Instruction, Multiple Data) operations included in modern processors.


APACHE ARROW
Parquet
DATA FUSION
APACHE ARROW
Flight SQL
SQL
plotly
ARUNDO
APACHE Spark
tableau
Grafana
influxdb
Java
GO
RUST

Flight vs Flight SQL

Flight

- Accepts a ticket that is implementation specific
- Query language agnostic
- Returns arrow results

FlightSQL

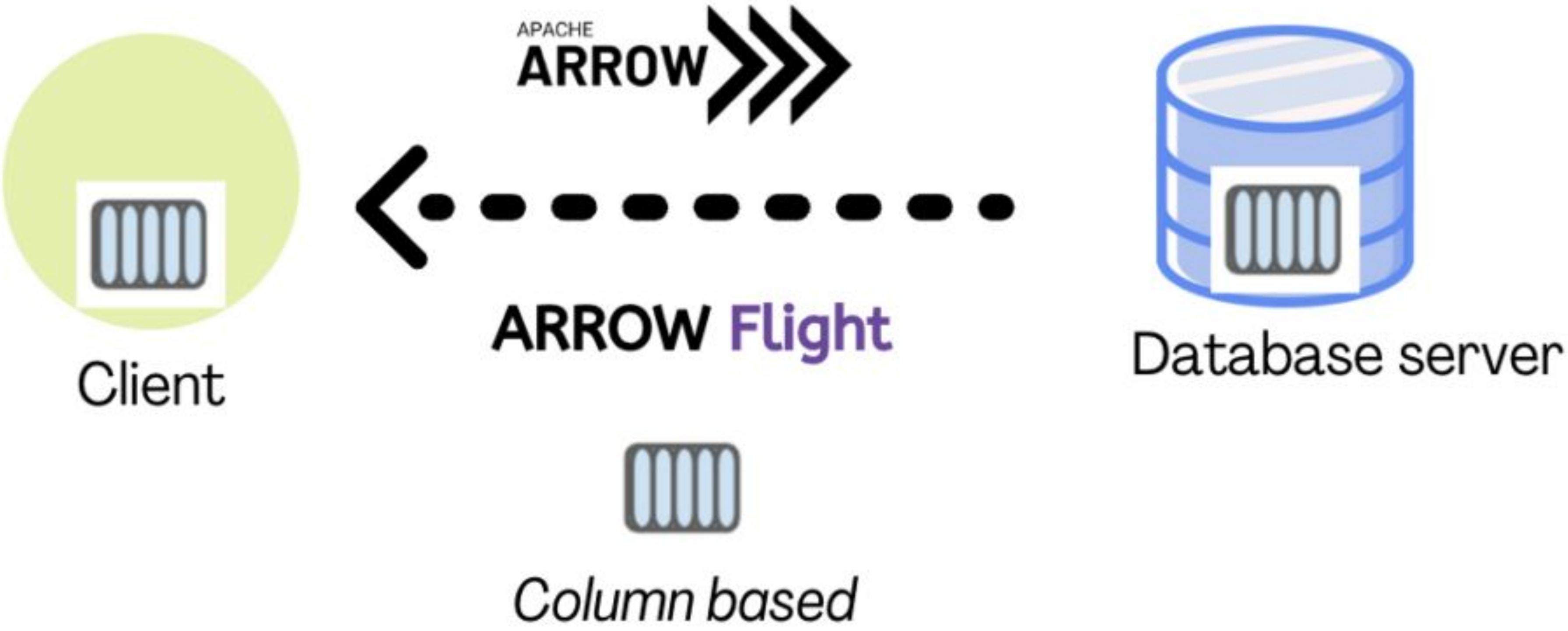
- A superset of the Flight API
- SQL specific
- Designed for ORMs and UI builders
- Implementation agnostic

What should I use in my Code?

Largely a matter of taste, but Flight supports InfluxQL

 |

influxdata®



No serialization or deserialization costs – Since Arrow data is sent directly, there is no need to serialize or deserialize the data, and no need to make an extra copy of the data.

Finally Apache Arrow Flight SQL

"A new client-server protocol developed by the Apache Arrow community for interacting with SQL databases that makes use of the Arrow in-memory columnar format"

What is Arrow Flight?

- High-volume transfers of columnar data
- Good for distributed computing and analytics

Arrow Flight SQL?

- Provides a SQL interface for Arrow Flight
- Query execution
- Returns columnar format

```python
# Experiment with 11M records
%%time
import pandas as pd
df_pandas = pd.read_csv('csv_pandas.csv')

# CPU times: user 24.3 s, sys: 8.92 s, total: 33.2 s
# Wall time: 35.5 s


----------------------------------------------------

%%time
import pyarrow as pa
from pyarrow import csv
df_arrow = csv.read_csv('csv_pandas.csv')

# CPU times: user 8.41 s, sys: 2.93 s, total: 11.3 s
# Wall time: 2.31 s
```

To understand the performance differences, let's try reading a CSV file with 11 million records and compare Pandas CSV reader (default engine) with PyArrow's.

PyArrow does it 15x faster with Arrow's in-memory columnar format!

 |

Prior to Arrow, the conversion from Spark DataFrames to Pandas was a very inefficient process since we had to go through the costly process of serialization and deserialization. With Arrow as the in-memory format, PySpark achieved two advantages. There is no need to serialize or deserialize the rows
When Python receives the Arrow data, PyArrow will create a data frame from the entire chunk of data at once instead of doing it for individual values

 |

influxdata®

Apache arrow example

Creating an Arrow table...

```python
import pyarrow as pa

# Create a array from a list of values
animal = pa.array(["sheep", "cows",
"horses", "foxes"], type=pa.string())
count = pa.array([12, 5, 2, 1],
type=pa.int8())
year = pa.array([2022, 2022, 2022,
2022], type=pa.int16())

# Create a table from the arrays
table = pa.Table.from_arrays([animal,
count, year], names=['animal','count',
'year'])
print(table)
```

=

```
pyarrow.Table
animal: string
count: int8
year: int16
----
animal:
[["sheep","cows","hors
es","foxes"]]
count: [[12,5,2,1]]
year:
[[2022,2022,2022,2022]
]
```

```python
from flightsql import FlightSQLClient

# Read only token for demo purposes
token = ""
client = FlightSQLClient(host=””,
                         token=token,
      metadata={'bucket-name':'factory'})
```

```python
# Execute a query against InfluxDB's Flight SQL endpoint
query = client.execute("SELECT * FROM iox.machine_data WHERE time > (NOW() - INTERVAL '1 DAY')")

# Create reader to consume result
reader = client.do_get(query.endpoints[0].ticket)

# Read all data into a pyarrow.Table
Table = reader.read_all()
print(Table)
```

=

```
pyarrow.Table
host: string
load: double
machineID: string
power: double
provider: string
temperature: double
time: timestamp[ns] not null
topic: string
vibration: double
----
host: [["9aa69b2d7e30"]]
load: [[50]]
machineID: [["machine1"]]
power: [[218]]
provider: [["Baird Ltd"]]
temperature: [[39]]
time: [[2023-02-14 12:30:41.989984916]]
topic: [["machine/machine1"]]
vibration: [[90]]
```

Arrow Utility functions...

```
# Convert to Pandas DataFrame
df = Table.to_pandas()
df = df.sort_values(by="time")
df.head(20)
```

```
# Load the table from the Parquet file
Table =
pq.read_table('example.parquet')
```

```
# Save the table to a Parquet file
pq.write_table(Table,
'example.parquet')
```

```
# PyArrow Aggregation
aggregation = Table.group_by("machineID").aggregate([("vibration",
"mean"), ("vibration", "max"),("vibration", "min") ]).to_pandas()
```

Table Result

	co	humidity	sensor_id	temperature	time
0	0.517422	35.139987	TLM0100	71.170208	2023-02-01 18:34:31
361	0.484613	34.874626	TLM0101	71.802606	2023-02-01 18:34:31
2527	0.394477	35.948833	TLM0203	74.779119	2023-02-01 18:34:31
722	0.513511	34.860303	TLM0102	72.012782	2023-02-01 18:34:31
2166	0.510674	35.691850	TLM0202	75.312108	2023-02-01 18:34:31
1805	0.506245	35.232817	TLM0201	74.045357	2023-02-01 18:34:31
1444	0.489873	35.769058	TLM0200	73.560425	2023-02-01 18:34:31
1083	0.415399	35.164551	TLM0103	71.340827	2023-02-01 18:34:31
723	0.517580	34.855599	TLM0102	72.055388	2023-02-01 18:34:41
1	0.530448	35.151335	TLM0100	71.180480	2023-02-01 18:34:41
362	0.467022	34.896021	TLM0101	71.835483	2023-02-01 18:34:41
1806	0.491062	35.223058	TLM0201	74.007277	2023-02-01 18:34:41
1084	0.409221	35.196949	TLM0103	71.350689	2023-02-01 18:34:41
2167	0.501382	35.710338	TLM0202	75.286829	2023-02-01 18:34:41
2528	0.413134	35.993991	TLM0203	74.819752	2023-02-01 18:34:41
1445	0.504451	35.797797	TLM0200	73.537383	2023-02-01 18:34:41
2529	0.428675	36.039681	TLM0203	74.812024	2023-02-01 18:34:51
724	0.507936	34.849071	TLM0102	72.041688	2023-02-01 18:34:51
1807	0.474696	35.180045	TLM0201	74.031949	2023-02-01 18:34:51
2	0.535308	35.104186	TLM0100	71.187633	2023-02-01 18:34:51

The resulting DataFrame looks like this. We include 20 values with the head() function just to make sure that it returns multiple time points for each sensor.

 |

Downsampling with Pandas

```
df_mean =
df.groupby(by=["sensor_id"]).resample(
'10min', on='time').mean().dropna()
# create a copy of the downsampled
data so we can write it back to InfluxDB
Cloud powered by IOx.
df_write = df_mean.reset_index()
df_mean
```

The objective here is to find the mean of our `temperature`, `co`, and `humidity` fields over 10 minute intervals. Use the groupby() function to group our dataframe by the `sensor_id` tag (or column). Then we use the resample() and mean() functions to downsample and apply a mean over the intervals, respectively.

Table Result comparison

	co	humidity	sensor_id	temperature	time
0	0.517422	35.139987	TLM0100	71.170208	2023-02-01 18:34:31
361	0.484613	34.874626	TLM0101	71.802606	2023-02-01 18:34:31
2527	0.394477	35.948833	TLM0203	74.779119	2023-02-01 18:34:31
722	0.513511	34.860303	TLM0102	72.012782	2023-02-01 18:34:31
2166	0.510674	35.691850	TLM0202	75.312108	2023-02-01 18:34:31
1805	0.506245	35.232817	TLM0201	74.045357	2023-02-01 18:34:31
1444	0.489873	35.769058	TLM0200	73.560425	2023-02-01 18:34:31
1083	0.415399	35.164551	TLM0103	71.340827	2023-02-01 18:34:31
723	0.517580	34.855599	TLM0102	72.055388	2023-02-01 18:34:41
1	0.530448	35.151335	TLM0100	71.180480	2023-02-01 18:34:41
362	0.467022	34.896021	TLM0101	71.835483	2023-02-01 18:34:41
1806	0.491062	35.223058	TLM0201	74.007277	2023-02-01 18:34:41
1084	0.409221	35.196949	TLM0103	71.350689	2023-02-01 18:34:41
2167	0.501382	35.710338	TLM0202	75.286829	2023-02-01 18:34:41
2528	0.413134	35.993991	TLM0203	74.819752	2023-02-01 18:34:41
1445	0.504451	35.797797	TLM0200	73.537383	2023-02-01 18:34:41
2529	0.428675	36.039681	TLM0203	74.812024	2023-02-01 18:34:51
724	0.507936	34.849071	TLM0102	72.041688	2023-02-01 18:34:51
1807	0.474696	35.180045	TLM0201	74.031949	2023-02-01 18:34:51
2	0.535308	35.104186	TLM0100	71.187633	2023-02-01 18:34:51

		co	humidity	temperature
sensor_id	**time**			
TLM0100	**2023-02-01 18:30:00**	0.574546	35.266605	71.171200
	2023-02-01 18:40:00	0.569610	35.358244	71.333793
	2023-02-01 18:50:00	0.516955	35.441879	71.289883
	2023-02-01 19:00:00	0.462129	35.193253	71.519450
	2023-02-01 19:10:00	0.416039	35.143414	71.556587
...	**...**	...	...	...
TLM0203	**2023-02-03 22:50:00**	0.242336	36.507586	75.153940
	2023-02-03 23:00:00	0.273699	36.562608	75.117751
	2023-02-03 23:10:00	0.258480	36.510316	74.792649
	2023-02-03 23:20:00	0.292327	36.537911	74.554340
	2023-02-03 23:30:00	0.354416	36.521472	74.556355

 |

Sending Downsampled Data back to InfluxDB

```
# write data back to InfluxDB Cloud powered by IOx
client = InfluxDBClient(url=url, token=token, org=org)
client.write_api(write_options=SYNCHRONOUS).write(bucket=bucket, record=df_write,
data_frame_measurement_name="mean_downsampled",
 data_frame_timestamp_column='time',
 data_frame_tag_columns=['sensor_id'])
```

Finally, we write our downsampled data back to InfluxDB Cloud using the InfluxDB v2 Python Client using the `.write` method and specifying the DataFrame we want to write back into InfluxDB.

 |

Leveraging pandas for analytics

Pandas Joining Dataframes

```
technologies = { 'Courses':["Spark","PySpark","Python","pandas"],
    'Fee' :[20000,25000,22000,30000],
    'Duration':['30days','40days','35days','50days'],}
index_labels=['r1','r2','r3','r4']
dataframe1 = pd.DataFrame(technologies,index=index_labels)
technologies2 = {
    'Courses':["Spark","Java","Python","Go"],
    'Discount':[2000,2300,1200,2000]
              }
index_labels2=['r1','r6','r3','r5']
dataframe2 = pd.DataFrame(technologies2,index=index_labels2)
print(df1)
print(df2)
```

```
    Courses    Fee Duration
r1    Spark  20000   30days
r2  PySpark  25000   40days
r3   Python  22000   35days
r4   pandas  30000   50days
   Courses  Discount
r1   Spark      2000
r6    Java      2300
r3  Python      1200
r5      Go      2000
```

```
     Courses    Fee Duration
r1     Spark  20000   30days
r2   PySpark  25000   40days
r3    Python  22000   35days
r4    pandas  30000   50days
    Courses  Discount
r1    Spark      2000
r6     Java      2300
r3   Python      1200
r5       Go      2000
```

```
#Join with no how parameter, defaults left
df3=df1.join(df2, lsuffix="_left", rsuffix="_right")
```

```
   Courses_left    Fee Duration Courses_right  Discount
r1        Spark  20000   30days         Spark    2000.0
r2      PySpark  25000   40days           NaN       NaN
r3       Python  22000   35days        Python    1200.0
r4       pandas  30000   50days           NaN       NaN
```

	Courses	Fee	Duration
r1	Spark	20000	30days
r2	PySpark	25000	40days
r3	Python	22000	35days
r4	pandas	30000	50days

	Courses	Discount
r1	Spark	2000
r6	Java	2300
r3	Python	1200
r5	Go	2000

```
#Join with right how parameter
df3=df1.join(df2, lsuffix="_left",
rsuffix="_right", how='right')
```

	Courses_left	Fee	Duration	Courses_right	Discount
r1	Spark	20000.0	30days	Spark	2000
r6	NaN	NaN	NaN	Java	2300
r3	Python	22000.0	35days	Python	1200
r5	NaN	NaN	NaN	Go	2000

 |

```
   Courses    Fee Duration
r1    Spark  20000   30days
r2  PySpark  25000   40days
r3   Python  22000   35days
r4   pandas  30000   50days
  Courses  Discount
r1   Spark      2000
r6    Java      2300
r3  Python      1200
r5      Go      2000
```

#Join with outer how parameter
df3=df1.join(df2, lsuffix="_left", rsuffix="_right", how=outer)

```
   Courses_left      Fee Duration Courses_right  Discount
r1        Spark  20000.0   30days         Spark    2000.0
r2      PySpark  25000.0   40days           NaN       NaN
r3       Python  22000.0   35days        Python    1200.0
r4       pandas  30000.0   50days           NaN       NaN
r5          NaN      NaN      NaN            Go    2000.0
r6          NaN      NaN      NaN          Java    2300.0
```

```
    Courses    Fee Duration
r1    Spark  20000   30days
r2  PySpark  25000   40days
r3   Python  22000   35days
r4   pandas  30000   50days
   Courses  Discount
r1   Spark      2000
r6    Java      2300
r3  Python      1200
r5      Go      2000
```

#Join with inner how parameter
df3=df1.join(df2, lsuffix="_left", rsuffix="_right", how='inner')

```
   Courses_left    Fee Duration Courses_right  Discount
r1        Spark  20000   30days         Spark      2000
r3       Python  22000   35days        Python      1200
```

 |

```
   Courses    Fee Duration
r1   Spark  20000   30days
r2 PySpark  25000   40days
r3  Python  22000   35days
r4  pandas  30000   50days
  Courses  Discount
r1   Spark      2000
r6    Java      2300
r3  Python      1200
r5      Go      2000
```

#Pandas join on column
df3=df1.set_index('Courses').join(df2.set_index('Courses'), how='inner')

```
          Fee Duration  Discount
Courses
Spark   20000   30days      2000
Python  22000   35days      1200
```

```
    Courses    Fee Duration
r1    Spark  20000   30days
r2  PySpark  25000   40days
r3   Python  22000   35days
r4   pandas  30000   50days
   Courses  Discount
r1   Spark      2000
r6    Java      2300
r3  Python      1200
r5      Go      2000
```

#Pandas join
df3=df1.join(df2.set_index('Courses'), how='inner', on='Courses')

```
    Courses    Fee Duration  Discount
r1    Spark  20000   30days      2000
r3   Python  22000   35days      1200
```

Pandas Rename Columns

```
          test         odi          t20
0        India     England     Pakistan
1  South Africa      India        India
2      England  New Zealand   Australia
3  New Zealand  South Africa    England
4    Australia    Pakistan  New Zealand
```

```
          TEST         odi          t20
0        India     England     Pakistan
1  South Africa      India        India
2      England  New Zealand   Australia
3  New Zealand  South Africa    England
4    Australia    Pakistan  New Zealand
```

```
rankings_pd.rename
(columns = {'test':'TEST'},
inplace = True)
# After renaming the
columns
print(rankings_pd)
```

Pandas Reset Index

df.reset_index()
When we reset the index, the old index is added as a column, and a new sequential index is used.
df.reset_index(drop=True)
If we add the drop, the column will not be included

```
          class  max_speed
falcon     bird      389.0
parrot     bird       24.0
lion     mammal       80.5
monkey   mammal        NaN
```

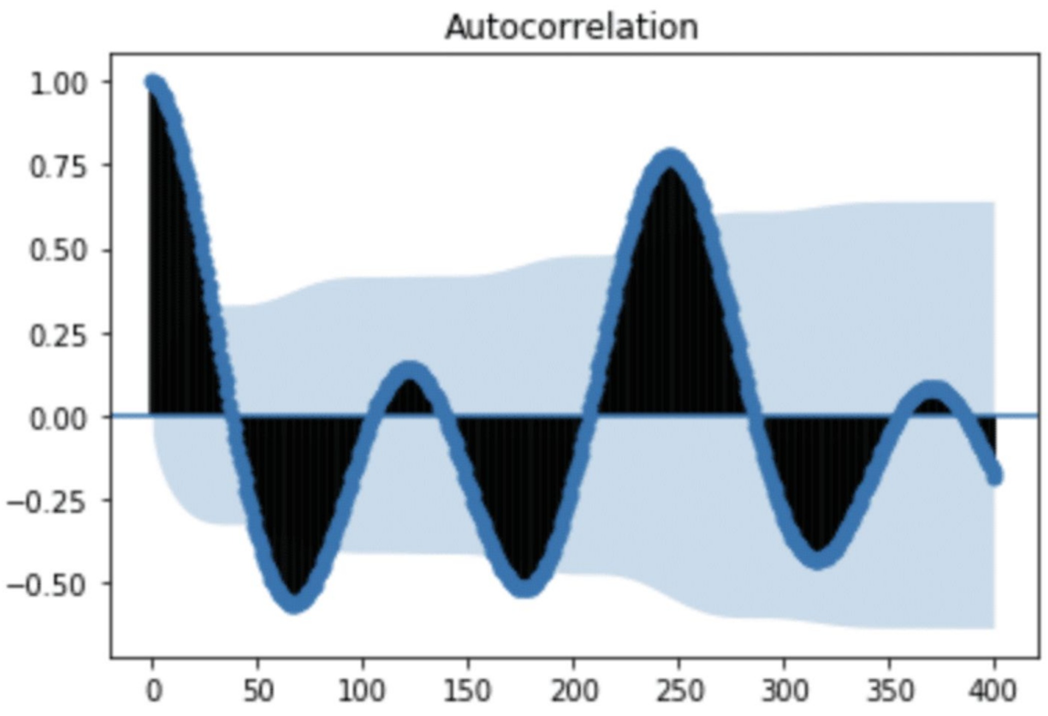
```
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

 |

Autocorrelation Example

What is autocorrelation?

The term autocorrelation refers to the degree of similarity between A) a given time series, and B) a lagged version of itself, over C) successive time intervals. In other words, autocorrelation is intended to measure the relationship between a variable's present value and any past values that you may have access to.



 |

influxdata®

Autocorrelation examples

Example 1: Regression analysis

One prominent example of how autocorrelation is commonly used takes the form of regression analysis using time series data. Here, professionals will typically use a standard auto regressive model, a moving average model or a combination that is referred to as an auto regressive integrated moving average model, or ARIMA for short.

Example 2: Scientific applications of autocorrelation

is used quite frequently in terms of fluorescence correlation spectroscopy, which is a critical part of understanding molecular-level diffusion and chemical reactions in certain scientific environments.

Examples Continued

Example 3: Global positioning systems

one of the primary mathematical techniques at the heart of the GPS chip that is embedded in smartphones

Example 4: Signal processing

a part of electrical engineering that focuses on understanding more about signals like sound, images and sometimes scientific measurements.

Example 5: Astrophysics

it helps professionals study the spatial distribution between celestial bodies in the universe like galaxies.

 |

influxdata®

Determining if your time series has autocorrelation

I am using available data from the National Oceanic and Atmospheric Administration's (NOAA) Center for Operational Oceanographic Products and Services. Specifically, I will be looking at the water levels and water temperatures of a river in Santa Monica. We will be using the influxdb python client library. A Jupyter notebook will be linked at the end of this.

Dataset:

```
curl https://s3.amazonaws.com/noaa.water-database/NOAA_data.txt -o NOAA_data.txt
influx -import -path=NOAA_data.txt -precision=s -database=NOAA_water_database
```


Next I connect to the client, query my water temperature data, and plot it.

```
client = InfluxDBClient(host='localhost', port=8086)
h20 = client.query('SELECT mean("degrees") AS "h20_temp" FROM "NOAA_water_database"."autogen"."h2o_t
h20_points = [p for p in h20.get_points()]
h20_df = pd.DataFrame(h20_points)
h20_df['time_step'] = range(0,len(h20_df['time']))
h20_df.plot(kind='line',x='time_step',y='h20_temp')
plt.show()
```

```
h20 = client.query('SELECT mean("degrees") AS
"h20_temp" FROM
"NOAA_water_database"."autogen"."h2o_temperature"
GROUP BY time(12h) LIMIT 60')
```

 |

influxdata®

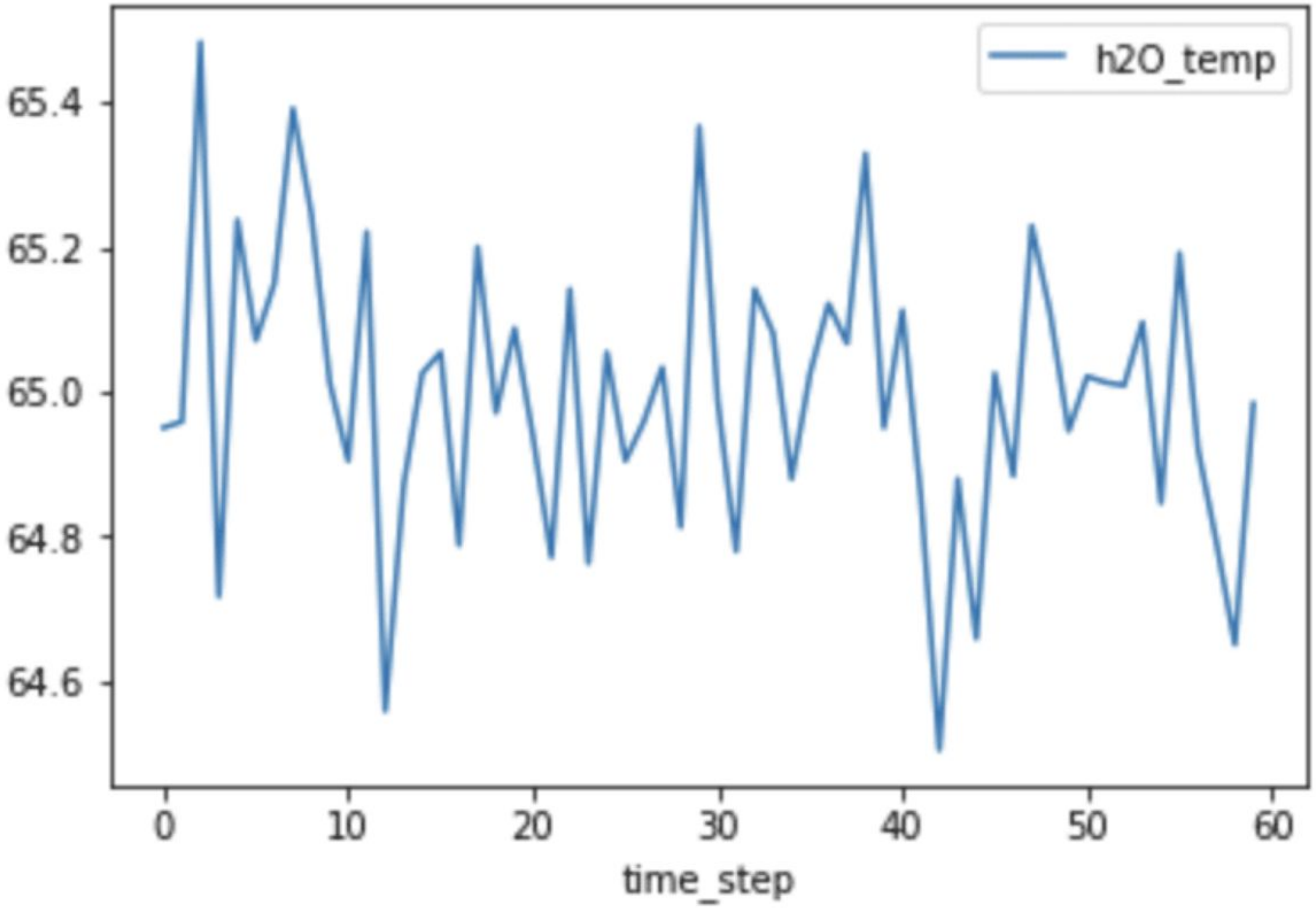


Fig 1. H2O temperature vs. timestep

From looking at the plot it's not obviously apparent whether or not our data will have any autocorrelation. For example, I can't detect the presence of seasonality, which would yield high autocorrelation.

calculate the autocorrelation

```
shift_1 = h20_df['h20_temp'].autocorr(lag=1)
print(shift_1)
-0.07205847740103073
0.17849760131784975
```

These values are very close to 0, which indicates that there is little to no correlation.

`Pandas.Sereis.autocorr()`
This method computes the Pearson correlation between the Series and its shifted self.

The Pearson correlation coefficient has a value between -1 and 1, where 0 is no linear correlation, >0 is a positive correlation, and <0 is a negative correlation.

 |

influxdata

However with autocorrelation plot

```
plot_acf(h20_df['h20_temp'], lags=20)
plt.show()
```

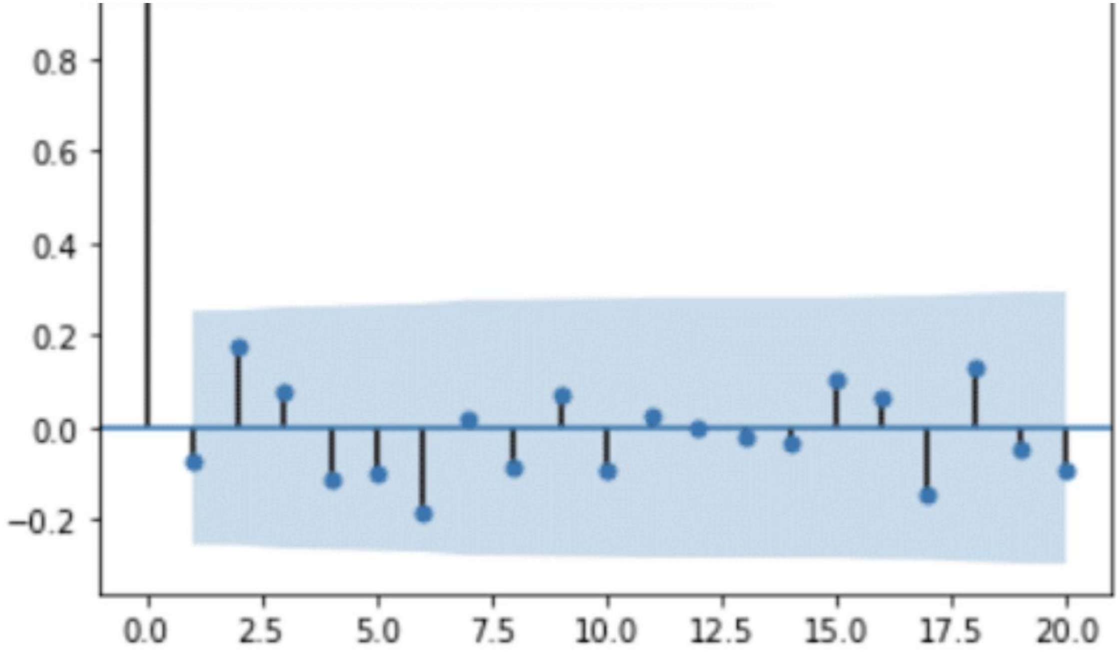


Fig 2. Autocorrelation plot for H2O temperatures

From this plot, we see that values for the ACF are within 95% confidence interval (represented by the solid gray line) for lags > 0, which verifies that our data doesn't have any autocorrelation.

Seasonality

The ACF can also be used to uncover and verify seasonality in time series data. Let’s take a look at the water levels from the same dataset

Uncovering seasonality with water levels

```
client = InfluxDBClient(host='localhost', port=8086)
h2O_level = client.query('SELECT "water_level" FROM "NOAA_water_database"."autogen"."h2o_feet" WHERE
h2O_level_points = [p for p in h2O_level.get_points()]
h2O_level_df = pd.DataFrame(h2O_level_points)
h2O_level_df['time_step'] = range(0,len(h2O_level_df['time']))
h2O_level_df.plot(kind='line',x='time_step',y='water_level')
plt.show()
```

```
h2O_level = client.query('SELECT "water_level" FROM
"NOAA_water_database"."autogen"."h2o_feet" WHERE
"location"=\'santa_monica\' AND time >= \'2015-08-22
22:12:00\' AND time <= \'2015-08-28 03:00:00\'')
```

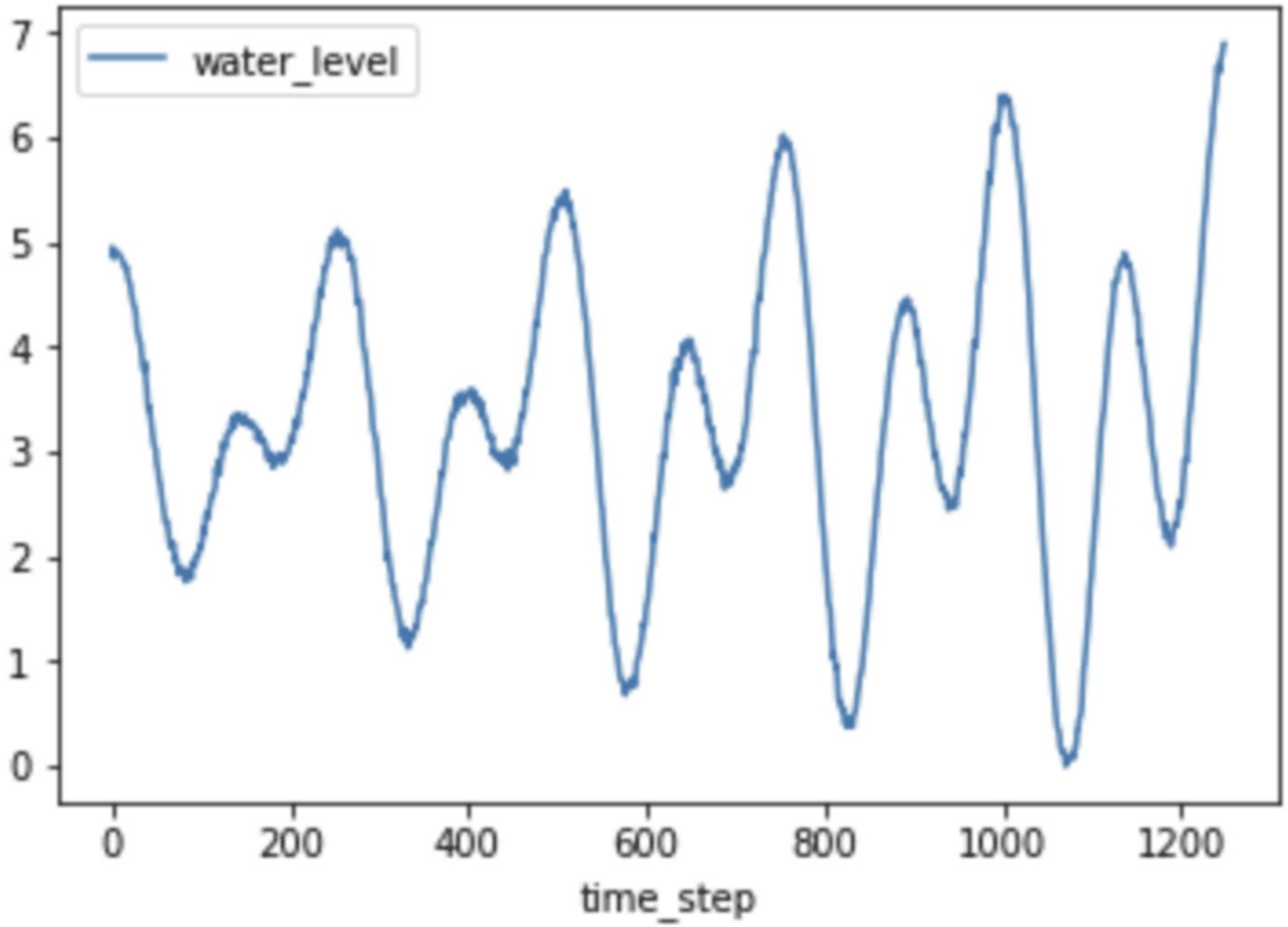


Fig 3. H2O level vs. timestep

Just by plotting the data, it's fairly obvious that seasonality probably exists, evident by the predictable pattern in the data. Let's verify this assumption by plotting the ACF.

 |

Verify by plotting the ACF

```
plot_acf(h20_level_df['water_level'], lags=400)
plt.show()
```

From the ACF plot above, we can see that our seasonal period consists of roughly 246 timesteps. While it was easily apparent from plotting time series in Figure 3 that the water level data has seasonality, that isn't always the case.

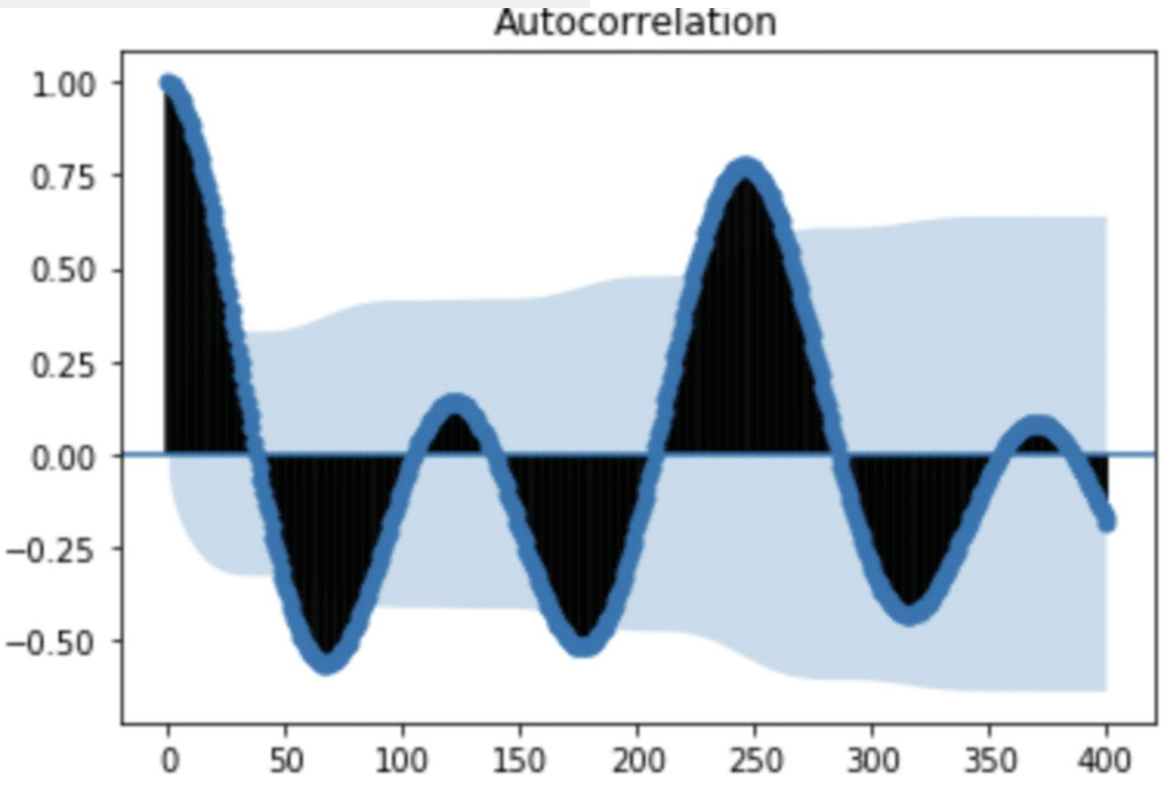


Fig. 4: Autocorrelation plot for H2O levels

 |

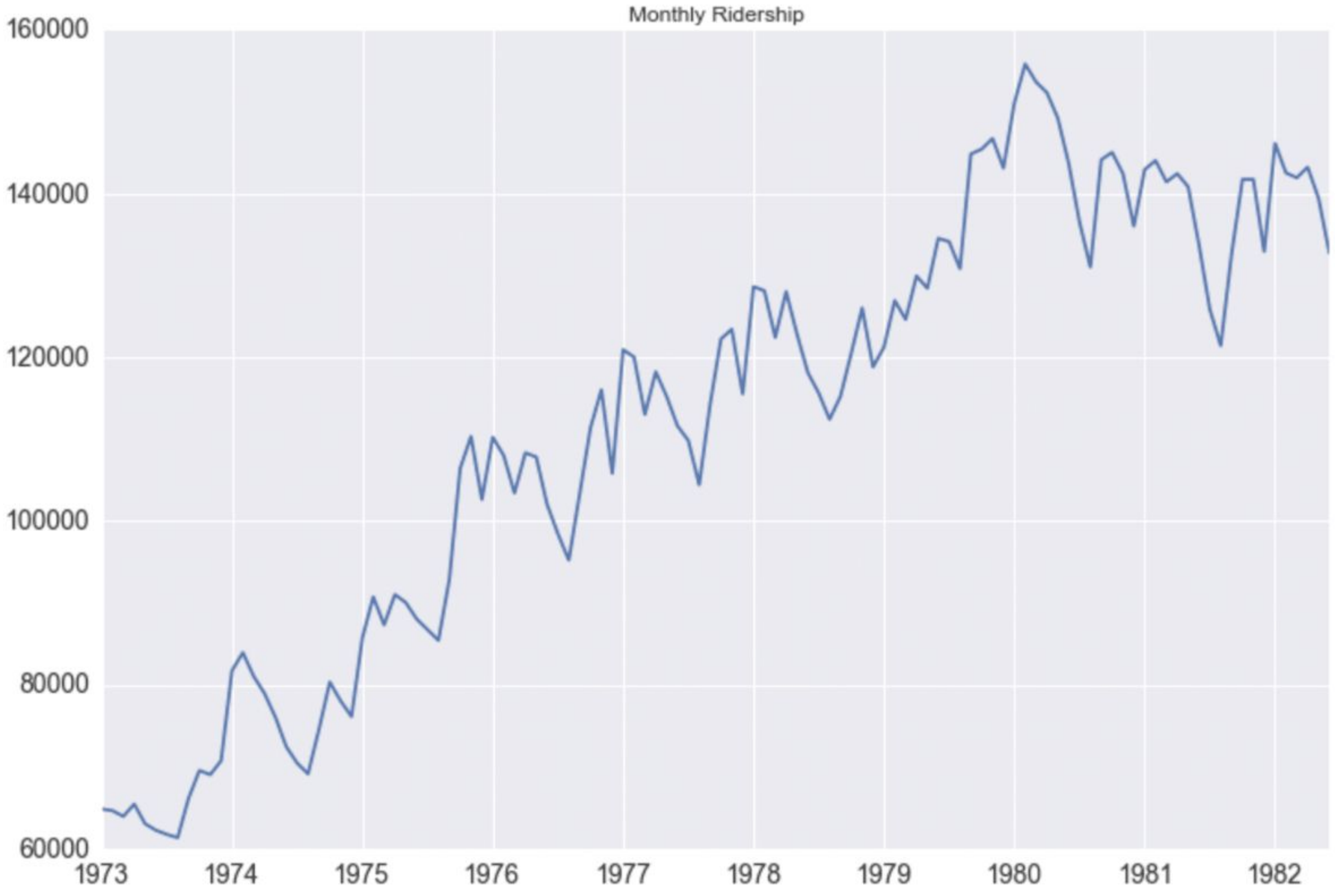


Fig. 5: Monthly Ridership vs. Year. Source: Seasonal ARIMA with Python

In Seasonal ARIMA with Python, author Sean Abu shows how he must add a seasonal component to his ARIMA method in order to account for seasonality in his dataset. It's a great example of how using ACF can help uncover hidden trends in the data.

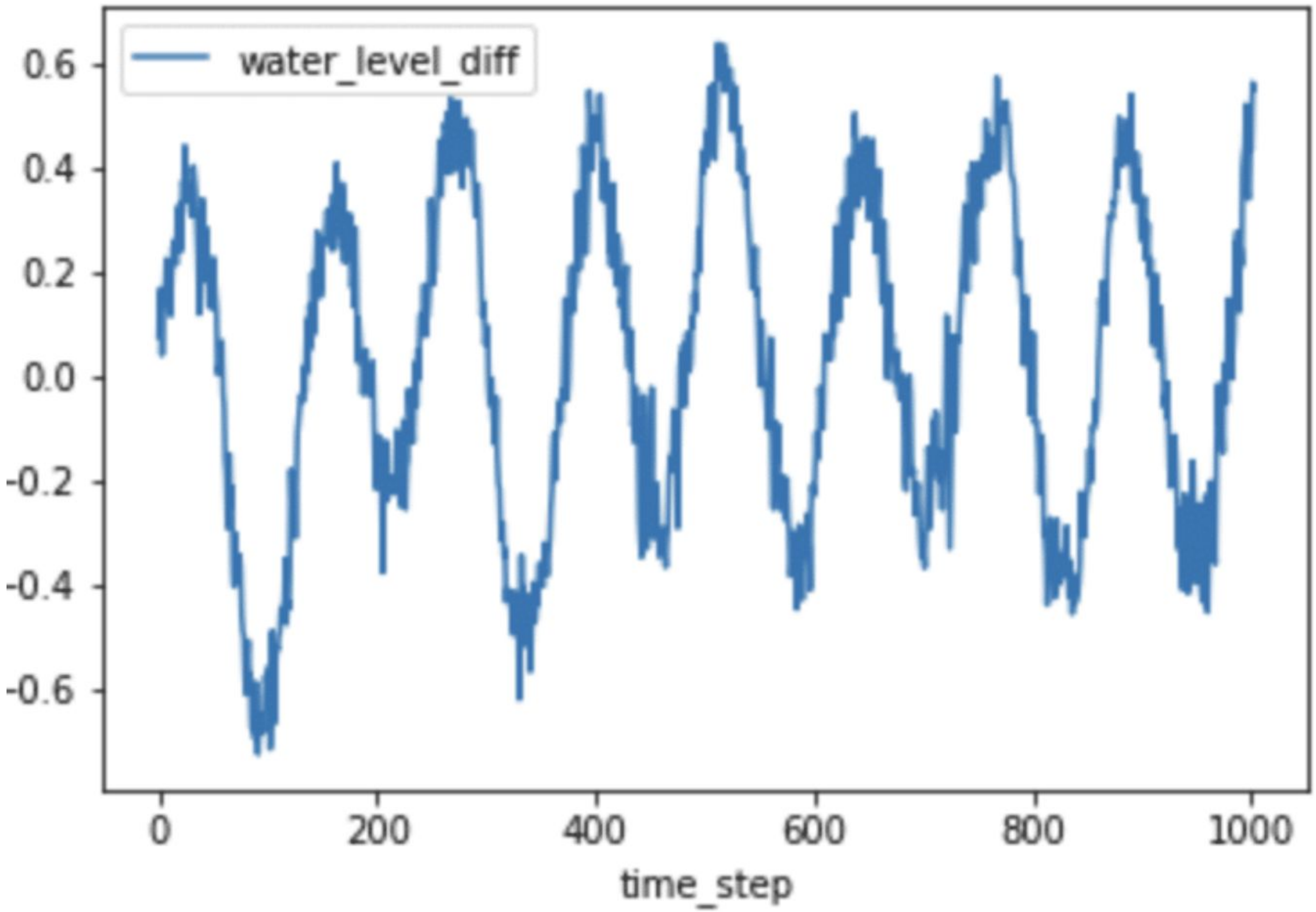
Examining trends with autocorrelation

In order to take a look at the trend of time series data, we first need to remove the seasonality. Lagged differencing is a simple transformation method that can be used to remove the seasonal component of the series. A lagged difference is defined by:

difference(t) = observation(t) - observation(t-interval)2,

where interval is the period. To calculate the lagged difference in the water level data, I used the following function:

```python
def difference(dataset, interval):
    diff = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        diff.append(value)
    return pd.DataFrame(diff, columns = ["water_level_diff"])
h2O_level_diff = difference(h2O_level_df['water_level'], 246)
h2O_level_diff['time_step'] = range(0,len(h2O_level_diff['water_level_diff']))
h2O_level_diff.plot(kind='line',x='time_step',y='water_level_diff')
plt.show()
```



This is our resulting graph

Fig. 6: Lagged difference for H2O levels

Including the ACF again

```
plot_acf(h20_level_diff['water_level_diff'], lags=300)
plt.show()
```

It might seem that we still have seasonality in our lagged difference. However, if we pay attention to the y-axis in Figure 5, we can see that the range is very small and all the values are close to 0.but there is a polynomial trend. I used seasonal_decompose to verify this.

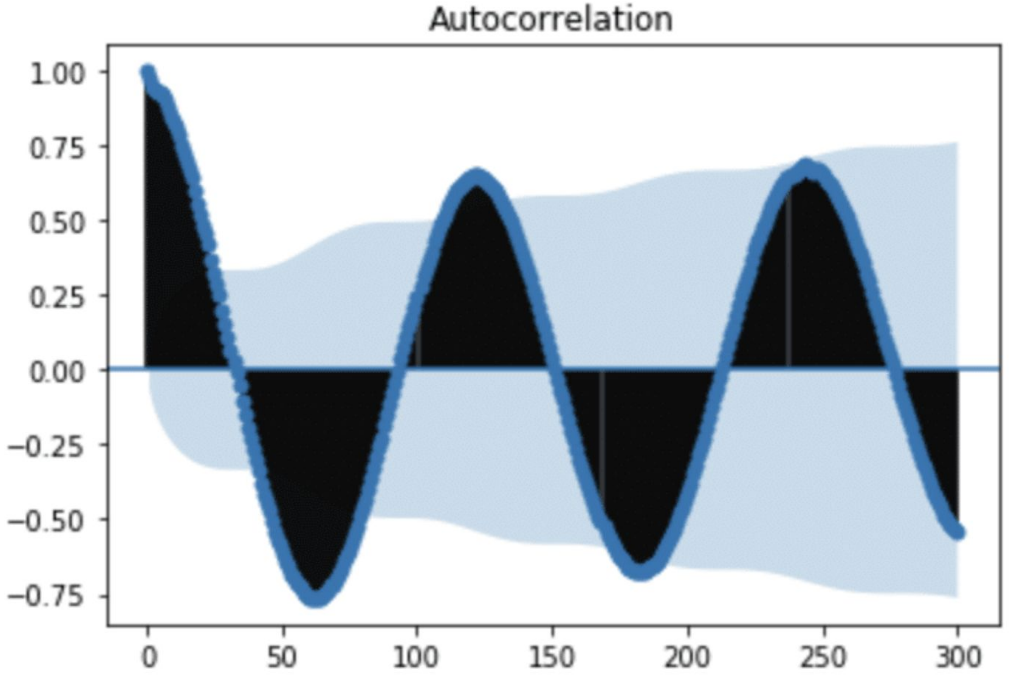


Fig. 7: ACF of lagged difference for H2O levels

 |

seasonal_decompose

```
from statsmodels.tsa.seasonal import seasonal_decompose
from matplotlib import pyplot
result = seasonal_decompose(h2O['water_level'], model='additive', freq=250)
result.plot()
pyplot.show()
```

Seasonal Decompose returns A object with seasonal, trend, and residual attributes.

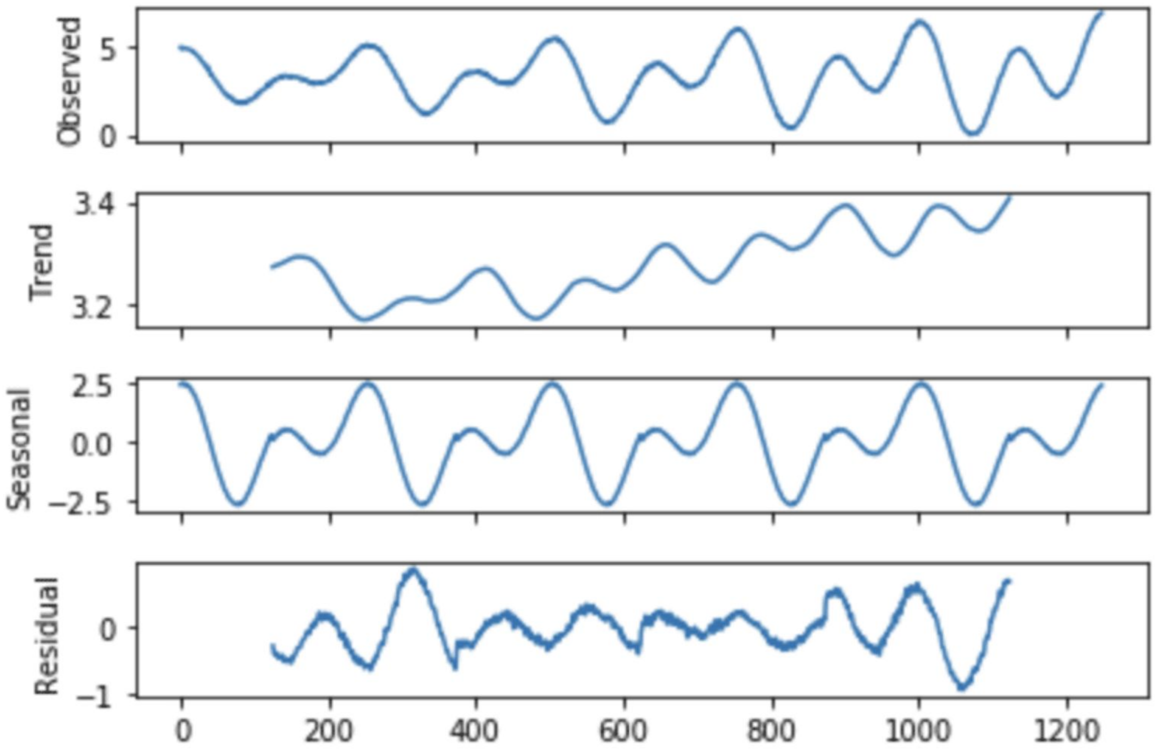


Fig. 8. Seasonal Decomposition of H2O levels

 |

Resources + Conclusion

AutoCorrelation Blog

How not to use time series for forecasting pitfalls

Jupyter Notebook

Seasonal ARIMA with Python

Anomaly detection and Forecasting

Some Examples of Anomaly Detection

For Single time series:

- Autoregression
- LevelShiftAD
- SeasonalAD

For Multiple Time series:

- BIRCH
- KMEANS
- Median Absolute Deviation(MAD)

Some Examples of Forecasting

- FBProphet
- LSTM with Keras
- statsmodels' Holt's Method.

All of the forecasting examples leverage outside libraries.

 |

AutoregressionAD algorithm

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

AutoregressionAD detects anomalous changes of autoregressive behavior in time series. AutoregressionAD can capture changes of autoregressive relationship (the relationship between a data point and points in its near past) and could be used for cyclic (but not seasonal) series in some situations.

 |

influxdata®

After we have acquired the data into a pandas format

```
s = pd.read_csv('./sample_data/sample-data.csv')
s.head()
```

	timestamp	value	label
0	1469376000	0.847300	0
1	1469376300	-0.036137	0
2	1469376600	0.074292	0
3	1469376900	0.074292	0
4	1469377200	-0.036137	0

Prepare Data for consumption

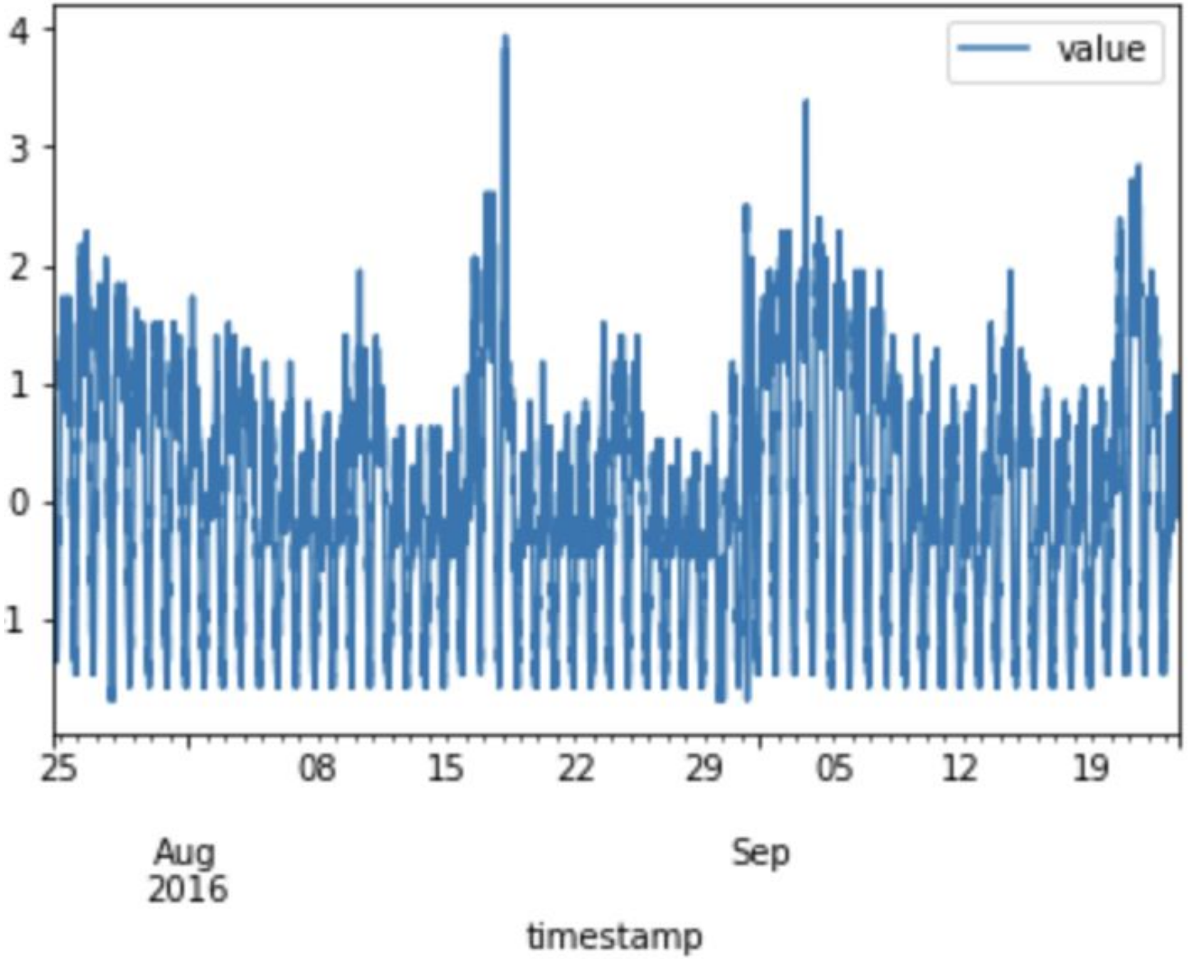
```
s.drop(['label'], axis=1, inplace=True)
s["timestamp"] = pd.to_datetime(s["timestamp"], unit='s')
s = s.set_index("timestamp")
s.head()
```

	value
timestamp	
2016-07-24 16:00:00	0.847300
2016-07-24 16:05:00	-0.036137
2016-07-24 16:10:00	0.074292
2016-07-24 16:15:00	0.074292
2016-07-24 16:20:00	-0.036137

 |

```
s.plot()
```

```
<AxesSubplot:xlabel='timestamp'>
```



Visualization before the algorithm

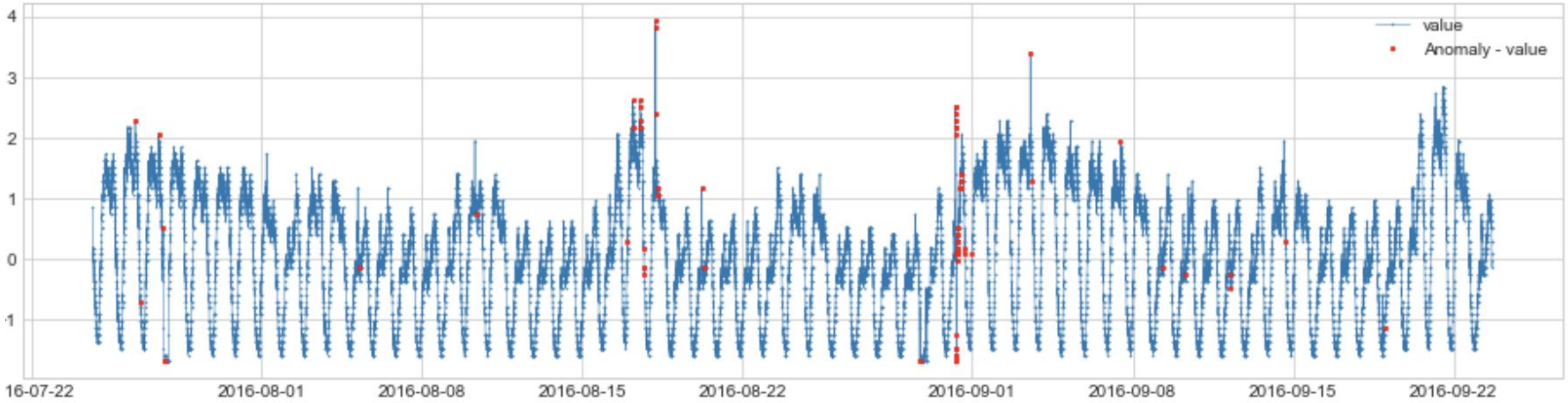
influxdata

Apply AutoregressionAD

```python
from adtk.data import validate_series
# This functoin will check some common critical issues of time series that may cause problems
s = validate_series(s)
```

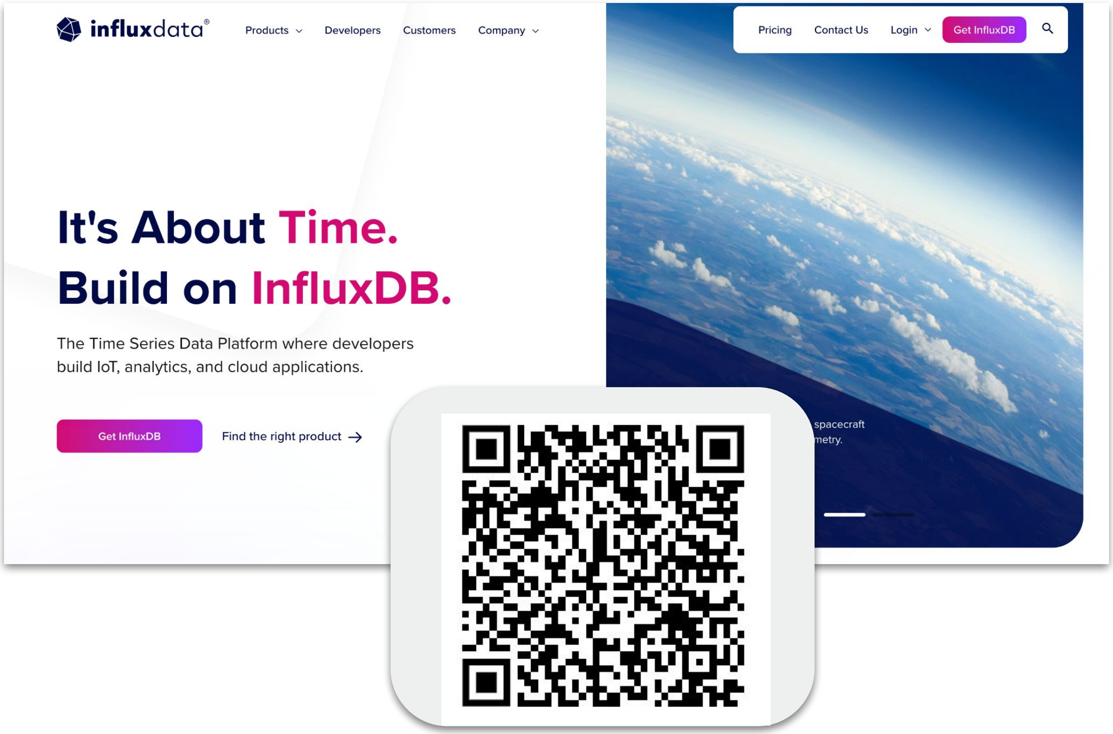
```python
from adtk.detector import AutoregressionAD
from adtk.visualization import plot
autoregression_ad = AutoregressionAD(n_steps=10, step_size=20,
c=3.0)
anomalies = autoregression_ad.fit_detect(s)
plot(s, anomaly=anomalies, ts_markersize=1, anomaly_color='red',
anomaly_tag="marker", anomaly_markersize=2);
```

Apply AutoregressionAD

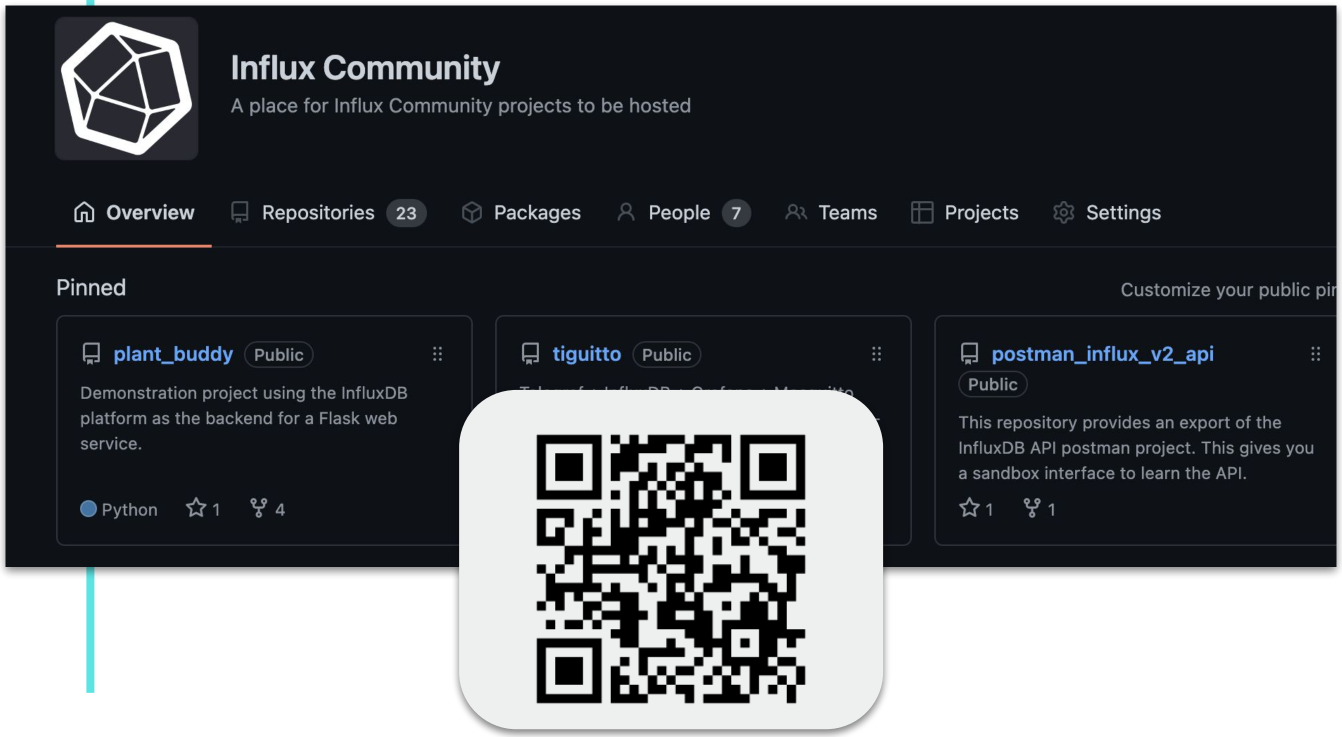


Resources

Try It Yourself



https://www.influxdata.com



https://github.com/InfluxCommunity

 |

influxdata®

influxdata®

THANK YOU