Chapter 2-1

Bonus Recipe - Reading data from a time series database

Reading data from a time series database

A time series database, a type of **NoSQL** database, is optimized for time-stamped or time series data and provides improved performance, especially when working with large datasets containing IoT data or sensor data. Initially, time series databases were mostly used with financial stock data, but their applications and use cases have expanded into various other domains. In this recipe, you will explore three popular time series databases: **InfluxDB**, **TimescaleDB,** and **TDEngine**, and learn how to connect and query data from these databases using Python.

InfluxDB is a popular open source time series database with a large community base. In this recipe, we will use InfluxDB's latest release as of this writing for Version 3 core. **InfluxDB 3** Core brings back support to **InfluxQL**, in addition to **SQL**. In InfluxDB V2, there was a shift toward using Flux Query Language, which is still supported and maintained (no major releases aside from bug fixes).

TimescaleDB is an extension of PostgreSQL specifically optimized for time series data. It leverages the power and flexibility of PostgreSQL while providing additional features tailored for handling time-stamped information efficiently. One advantage of TimescaleDB is that you can leverage SQL for querying the data. TimescaleDB is an open source time series database, and in this recipe, we will use TimescaleDB's latest release as of this writing, version 2.22.1.

TDEngine is an open source time series database designed for Internet of Things (IoT), big data, and real-time analytics. Like TimescaleDB, TDEngine utilizes SQL for querying data. In this recipe, we will work with TDEngine version 3.3.6.13, the latest as of this writing.

Getting ready

This recipe assumes that you have access to a running instance of InfluxDB, TimeseriesDB, or TDEngine. You will install the appropriate libraries to connect and interact with these databases using Python. For **InfluxDB** V3, you will need to install `influxdb3-python`; for **TimescaleDB,** you will need to install the PostgreSQL Python library `psycopg2` (recall in the *Reading from a relational database* recipe of this chapter, we used `psycopg3`); and finally, for **TDEngine,** you will need to install `taospy`.

You can install these libraries using `pip`, as follows:

```
pip install influxdb3-python
pip install 'taospy[ws]'
pip install psycopg2
```

If you do not have access to these databases, then the fastest way to get up and running is via Docker. The following are example commands for **InlfuxDB**, **TimescaleDB**, and **TDEngine**:

InfluxDB Docker container

To create an InfluxDB container, you will need to run the following command:

```
docker run -d \
  --name influxdb3-ch2 \
  -p 8181:8181 \
  -v influxdb3_data:/var/lib/influxdb3 \
  quay.io/influxdb/influxdb3-core:latest serve \
  --node-id my-node \
  --object-store file \
  --data-dir /var/lib/influxdb3
```

The `--node-id` is an identifier that you set. In this example, it is called `my-node`. We are persisting the data by specifying `--object-store file` to the `/var/lib/influxdb3` in the container, which you can change. Alternatively, you can use a memory object store (store data in RAM without persistence) by using `--object-store memory` instead.

Once the influxdb-ch2 container is up and running, you can access the shell command line using the following:

```
docker exec -it influxdb3-ch2 /bin/sh
```

Once inside the terminal of the Docker container, run the following to generate a **token**. Influxdb3 requires a token for using the CLI or API. The token generated will be shown only once in the terminal, so make sure you store it. You will not be able to retrieve the token after it is created.

```
influxdb3 create token --admin --host http://localhost:8181
```

This should print a token – for example, something like this:

```
Token: apiv3_Ee_uB-Z50N3NjVHS65kurr-11I1v8SRyd2Fe_-rriqlkdr4vXSA-H5BoiY_
z8HOj83cGRVG0dFf_Sb4hAYhOWQ
```

For this recipe, we will use the **National Oceanic and Atmospheric Administration** (**NOAA**) **Bay Area Weather** dataset, which includes daily weather measurements from three San Francisco Bay Area locations (Concord, Hayward, and San Francisco) from January 1, 2020 to December 31, 2022.

Download the dataset:

```
cd /tmp
curl -o bay-area-weather.lp https://docs.influxdata.com/downloads/bay-area-
weather.lp
```

Load the dataset into the InfluxDB database and name it NOAA:

```
influxdb3 write --database NOAA --file /tmp/bay-area-weather.lp --token
<yourtokenhere>
```

Validate that the database is created:

```
influxdb3 show databases --token <yourtokenhere>
>>
+---------------+
| iox::database |
+---------------+
| NOAA          |
| _internal     |
+---------------+
```

The following command will show all the tables available in the NOAA database:

```
influxdb3 query --database NOAA 'SHOW TABLES' --token <yourtokenhere>
```

If you set the `INFLUXDB3_AUTH_TOKEN` environment variable, you can omit the --token flag in the CLI command:

```
export INFLUXDB3_AUTH_TOKEN=<yourtokenhere>
influxdb3 show databases
```

For instructions on how to load the sample data or other provided *sample datasets*, please refer to the InfluxDB official documentation at `https://docs.influxdata.com/influxdb3/enterprise/reference/sample-data/`.

TimescaleDB Docker container

To create a TimescaleDB container, you will need to run the following command:

```
docker run -d \
  --name timescaledb-ch2 \
  -p 5432:5432 \
  -e POSTGRES_PASSWORD=your_password \
  timescale/timescaledb:2.22.1-pg17
```

For more information, you can access the official Docker Hub page: `https://hub.docker.com/r/timescale/timescaledb`

Once the timescaledb-ch3 container is up and running, you can load the `MSFT.csv` file using the same instructions as in the *Getting ready* section of the *Reading data from a relational database* recipe.

Note, the default username is `postgres` and the password is whatever password you set up in the Docker command. In the *Reading data from a relational database* recipe, you used **DBeaver Community Edition** to interact with PostgreSQL. DBeaver supports many different databases, including both **TimescaleDB** and **TDEngine**.

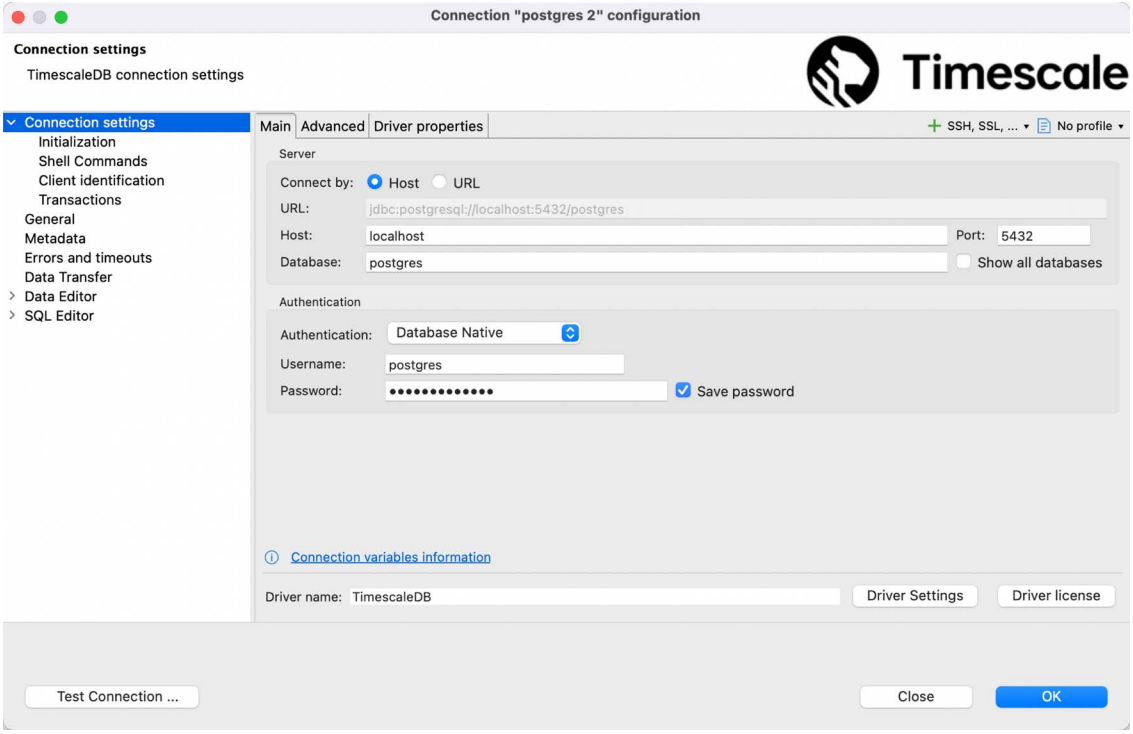

Figure 2.1: DBeaver TimescaleDB/Postgres connection settings

Since TimescaleDB is based on PostgreSQL, it also defaults to port 5432. So, if you are already running a local instance of PostgreSQL database, which also defaults to port 5432, you may run into an issue with TimescaleDB. In such a case, you may opt to change the Docker run configuration and change the port.

Once you are connected to the TimescaleDB instance, you can use DBeaver to upload the `MSFT.csv` file (similar to the steps from the *Reading data from a relational* database recipe). You can use the default `postgres` database, then go to the public schema, and right-click to import the `MSFT.csv` file.

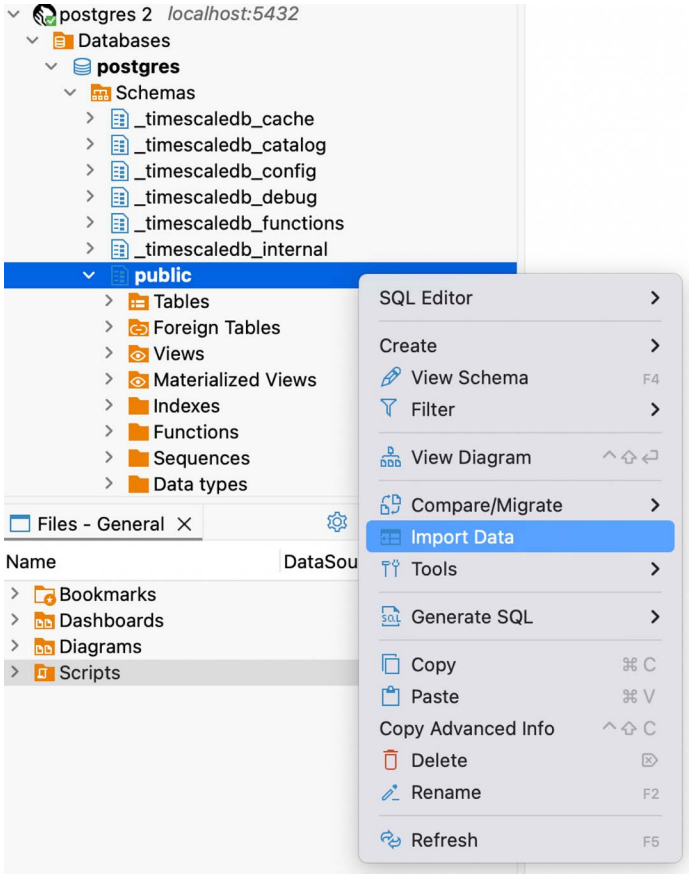

Figure 2.2: Importing data to TimescaleDB using DBeaver

TDEngine Docker container

To create a TDEngine container, you will need to run the following command:

```
docker run -d \
   --name tdengine-ch2 \
   -p 6030-6060:6030-6060 \
   -p 6030-6060:6030-6060/udp \
   tdengine/tdengine:3.3.6.13
```

For more information, you can access the official Docker Hub page: `https://hub.docker.com/r/tdengine/tdengine`

Once the tdengine-ch2 container is up and running, you can create a demo dataset by running the `taosBenchmark` command from inside the container shell. Here are the steps to access the shell from inside the running container to run the needed command to install and set up the demo dataset:

```
docker exec -it tdengine-ch2 /bin/bash
```

From inside the container's shell, run the following command:

```
taosBenchmark
```

You might be prompted with the following:

```
Press enter key to continue or Ctrl-C to stop
```

If so, just hit **Enter** to continue.

Once the demo set is created, you can exit the terminal. You can now leverage DBeaver to verify that the data has been created. You can use the same instructions as in the *Getting ready* section of the *Reading data from a relational database* recipe.

Note, the default username is `root` and the default password is `taosdata`. You can use **DBeaver Community Edition** to connect to TDEngine and verify the data being loaded.

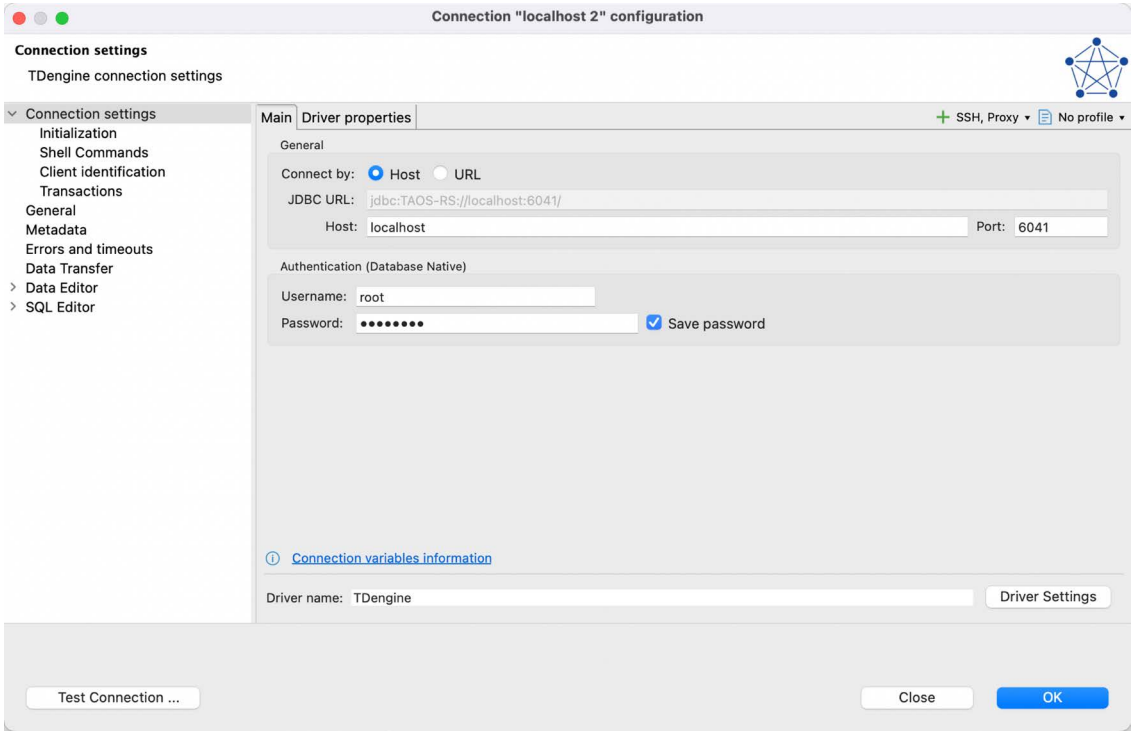

Figure 2.3: DBeaver TDEngine connection settings

You should now see a `test` **database** created, a `meters` **supertable** with 10,000 **subtables** named `d0` to `d9999`, with each table containing around 10,000 rows and six columns (`ts`, `current`, `voltage`, `phase`, `groupid`, and `location`). You can verify this by executing the following SQL query in DBeaver:

```
SELECT COUNT(*) FROM test.meters;
```

This should output 100,000,000 (count of rows) in the meters **supertable** (10,000 **subtables** multiplied by 10,000 rows in each).

How to do it...

This recipe will demonstrate how you can connect and interact with three popular time series database systems.

InfluxDB

We will be leveraging the `Influxdb_client_3` Python SDK for InfluxDB 3.x, which provides support for pandas DataFrames in terms of both read and write functionality. Let's get started:

1. First, let's import the necessary libraries:

```
from influxdb_client_3 import InfluxDBClient3
import pandas as pd
```

2. Establish your connection using `InfluxDBClient3`:

```
token= 'yourtokenhere'
client = InfluxDBClient3(host="http://localhost:8181",
                         database="NOAA",
                         token=token)
```

   InfluxDB V3 supports both **SQL** and **InfluxQL** (SQL-like syntax) and natively supports both pandas and polars DataFrames. Use the `query` method, pass your SQL query, and specify `mode='pandas'` to get a pandas DataFrame returned:

```
df = client.query(query=query, language="sql", mode='pandas')
print(df.head())
>>
  location  precip  temp_avg  temp_max  temp_min       time  wind_avg
0  Concord     0.0      52.0      66.0      44.0 2020-01-01      3.13
1  Concord     0.0      53.0      66.0      42.0 2020-01-02      3.13
2  Concord     0.0      49.0      60.0      38.0 2020-01-03      2.68
3  Concord     0.0      51.0      61.0      41.0 2020-01-04      4.25
4  Concord     0.0      49.0      61.0      38.0 2020-01-05      4.70
```

The query method has a language parameter and accepts either `"influxql"` or `"sql"`. The `mode` parameter supports `"all"`, `"pandas"`, `"polars"`, `"chunk"`, `"reader"`, or `"schema"`. The default **mode** is `"all"`, which returns a **PyArrow** table.

```
table = client.query(query=query, language="sql", mode='all')
type(table)
>>
pyarrow.lib.Table
```

You can convert the PyArrow table to a pandas DataFrame using the `to_pandas()` method:

```
df = table.to_pandas()
```

3. Close the connection to release resources:

```
client.close()
```

For better resource management, you can use the following:

```
query = 'SELECT * FROM weather'
with InfluxDBClient3(
    host="http://localhost:8181",
    database="NOAA",
    token=token
) as client:
    df = client.query(query=query, language="sql", mode='pandas')
```

TimescaleDB

Since **TimescaleDB** is based on **PostgreSQL**, you can use the `psycopg2` library for querying the database, and it should be similar to the approach used in the *Reading data from a relational database* recipe. You will utilize **SQLAlchemy** and `psycopg2` to leverage pandas' `read_sql` method.

1. Import SQLAlchemy and pandas:

```
import pandas as pd
from sqlalchemy import create_engine
```

2. Create the engine object with the proper connection string to the PostgreSQL backend:

```
engine =\
    create_engine(
        "postgresql+psycopg2://postgres:your_password@localhost:5432/
postgres"
)
```

3. Use the `read_sql` method to retrieve the result set of your query into a pandas DataFrame:

```
query = "SELECT * FROM msft"
df = pd.read_sql(query,
                 engine,
                 index_col='date',
                 parse_dates={'date': '%Y-%m-%d'})

print(df.head())
```

TimescaleDB offers many advantages over PostgreSQL, and you will explore some of these in *Chapter 4*. Querying and interacting with TimescaleDB brings a similar experience to those familiar with SQL and PostgreSQL.

TDEngine

Start by establishing a connection to the **TDEngine** database, and then run a query against the demo dataset from taosBenchmark, described in the *Getting ready* section.

1. Start by importing the required libraries:

```
import taosrest
import pandas as pd
```

2. Establish a connection to the server:

```
user='root'
password='taosdata'
url='http://localhost:6041'

conn = taosrest.connect(
    user=user,
    password=password,
    url=url
)
```

3. Run the following SQL query using the `query` method from the connection object, `conn`:

```
query = """
    SELECT *
    FROM test.meters
    WHERE location = 'California.LosAngles'
    LIMIT 100000;
"""
results = conn.query(query)
```

4. You can verify the number of rows and column names in the result set:

```
results.rows
>>
100000

results.fields
>>
[{'name': 'ts', 'type': 'TIMESTAMP', 'bytes': 8},
 {'name': 'current', 'type': 'FLOAT', 'bytes': 4},
 {'name': 'voltage', 'type': 'INT', 'bytes': 4},
 {'name': 'phase', 'type': 'FLOAT', 'bytes': 4},
 {'name': 'groupid', 'type': 'INT', 'bytes': 4},
 {'name': 'location', 'type': 'VARCHAR', 'bytes': 24}]
```

5. The `results.data` contains the values from the result set but without column headers. Before we load our result set into a pandas DataFrame, we need to capture the column names in a list from `results.fields`:

```
cols = [col['name'] for col in results.fields ]
df = pd.DataFrame(results.data, columns=cols)
df = df.set_index('ts')
df.info()
>>
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 100000 entries, 2017-07-14 05:40:00 to 2017-07-14
05:40:05.903000
Data columns (total 5 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   current   100000 non-null  float64
 1   voltage   100000 non-null  int64
 2   phase     100000 non-null  float64
 3   groupid   100000 non-null  int64
 4   location  100000 non-null  object
dtypes: float64(2), int64(2), object(1)
memory usage: 4.6+ MB
```

6. Close the connection and release resources:

```
conn.close()
```

How it works...

All three time series databases (InfluxDB, TimescaleDB, and TDEngine) introduced allow you to use SQL to query data.

InfluxDB

InfluxDB introduced **Flux query language** in version 2, but opted to move to support **InfluxQL** and **SQL** in version 3. **InfluxDB 3** introduces several innovations to the InfluxDB design, such as the decoupled architecture between compute and storage for better scalability. **InfluxDB 3** stores data as **Apache Parquet** files (physical storage), with **Apache Arrow** as the in-memory representation, and **Apache DataFusion** as the query planner and engine (SQL) that leverages Arrow.

In this recipe, we started by creating an instance of `InfluxDbClient3`, which gave us access to the `query` method. The following are key parameters to consider when using the query method:

- `query`: The SQL or InfluxQL statement to execute.
- `language`: The query language that is being used and passed. It can be either `"sql"` or `"influxql"`.
- `mode`: Used to specify the output returned from InfluxDB. The default is `"all"`.
    - `all`: Returns the entire dataset as a `pyarrow.table`.
    - `chunk`: Returns a `pyarrow._flight.FlightStreamReader` object that provides raw data chunks.
    - `pandas`: Returns the result as a pandas DataFrame.
    - `reader`: Returns a `pyarrow.lib.RecordBatchReader` object for processing large datasets incrementally.
    - `schema`: Returns the schema of the result without fetching the actual data.

When deciding which `mode` to specify, consider these guidelines:

- Use `"all"` for small datasets that can fit comfortably in memory
- Use `"chunk"` or `"reader"` for large datasets to manage memory efficiently
- Use `"pandas"` when integrating with pandas-based workflows for data analysis
- Use `"schema"` to inspect the result structure before fetching data, which can be helpful to understand the data structure

TimescaleDB

TimescaleDB is a PostgreSQL extension designed specifically for time-series data. It uses a time-series-aware storage model and indexing techniques to improve performance when working with time-series data. The extension divides data into chunks based on time intervals, allowing it to scale efficiently, especially for large datasets.

In this recipe, we utilized SQLAlchemy with `psycopg2` to connect to TimescaleDB, leveraging the familiar pandas `read_sql` method. Since TimescaleDB is built on PostgreSQL, you can use standard SQL syntax while benefiting from TimescaleDB's optimizations for time-series data.

TimescaleDB introduces a couple of key concepts that enhance its performance:

- **Hypertables**: These are virtual tables that automatically partition data by time, functioning like conventional PostgreSQL tables but with additional features for time-series data management
- **Chunks**: Data within hypertables is divided into multiple chunks, each containing data for a specific time period

TDEngine

TDEngine is an open source, high-performance time-series database designed specifically for IoT, connected cars, and Industrial IoT applications. A unique advantage of TDEngine since its first release is its support for standard SQL queries, which significantly reduces the learning curve for users.

In this recipe, we used the `taosrest` library to connect to TDEngine and execute SQL queries. The connection object provides a query method that returns results in a format that can be easily converted to a pandas DataFrame.

There's more...

Since InfluxDB 3.0 uses Apache Arrow as its in-memory representation format, you can take advantage of PyArrow's powerful data processing capabilities directly on your query results:

```
# Group weather data by location and calculate average temperature
query = "select * from weather"
table = client.query(query=query, language="sql", mode='all')
location_avg_temps = table.group_by(
    'location'
).aggregate([('temp_avg', 'mean')])
print(location_avg_temps)
>>
pyarrow.Table
location: string
temp_avg_mean: double
----
location: [["Concord","Hayward","San Francisco"]]
temp_avg_mean: [[61.52098540145985,58.81204379562044,58.238138686131386]]
```

The code groups NOAA weather data by location and calculates the mean temperature for each location, returning a new PyArrow table with the results.

To more efficiently process large datasets from InfluxDB using batch processing techniques, you can use either **chunk** or **reader** modes. This approach allows you to analyze time series data without loading everything into memory at once, which is particularly valuable when working with extensive weather datasets such as the NOAA Bay Area collection:

```
# Query with reader mode for memory efficiency
reader = client.query(query=query, language="sql", mode='reader')

# Process data in batches
total_rows = 0
sum_temp = 0

for batch in reader:
    df = batch.to_pandas()
    total_rows += len(df)
    sum_temp += df['temp_avg'].sum()

avg_temp = sum_temp / total_rows
print(f"Average temperature across {total_rows} records: {avg_temp:.2f} F")
>>
Average temperature across 3288 records: 59.52 F
```

See also

Please refer to the official **Influxdb3-python** library documentation on GitHub at `https://github.com/InfluxCommunity/influxdb3-python`

To learn more about the **TDEngine** Python library, refer to the official documentation at `https://docs.tdengine.com/cloud/programming/client-libraries/python/`

To learn more about TimescaleDB and Python, refer to the official documentation at `https://docs.timescale.com/quick-start/latest/python/`.