Chapter 4-1

Bonus Recipe - Writing time series data to InfluxDB

Writing time series data to InfluxDB

When working with large time series data, such as sensor or **Internet of Things** (**IoT**) data, you will need a more efficient way to store and query such data for further analytics. This is where **time series databases** shine, as they are built exclusively to work with complex and very large time series datasets.

In this recipe, we will work with **InfluxDB** as an example of how to write to a time series database.

Getting ready

Refer to the *Reading data from a time series database* recipe in *Chapter 2*, on how to set up your InfluxDB instance and the different ways to connect to InfluxDB.

You will be using the `ExtraSensory` dataset, a mobile sensory dataset made available by the University of California, San Diego (*Vaizman, Y., Ellis, K., and Lanckriet, G. "Recognizing Detailed Human Context In-the-Wild from Smartphones and Smartwatches". IEEE Pervasive Computing, vol. 16, no. 4, October-December 2017, pp. 62-74. doi:10.1109/MPRV.2017.3971131*).

You can download the dataset here: `http://extrasensory.ucsd.edu/#download`.

The dataset consists of 60 files, each representing a participant, each identified by a unique identifier (UUID). Each file contains a total of 278 columns: 225 features, 51 labels, and 2 (timestamp and label_source).

This recipe aims to demonstrate how to write a time series DataFrame to InfluxDB. In this recipe, two columns are selected: the timestamp (date ranges from `2015-07-23` to `2016-06-02`, covering 152 days) and the watch accelerometer reading (measured in milli G-forces or milli-G).

Before you can interact with InfluxDB in Python, you will need to install the InfluxDB Python library.

You can install the library with **pip** by running the following command:

```
$ pip install influxdb3-python
```

You will also need to create a new database in InfluxDB3 using the CLI. Refer to *Chapter 2* on how to run a Docker container, generate a token, and run some CLI commands.

The following command will let you go into the container's terminal:

```
docker exec -it influxdb3-ch3 /bin/sh
```

Once inside, you can run the create database command as shown:

```
influxdb3 create database extrasensory --token <yourtokenhere>
```

To confirm that the database is created, you can run the following:

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

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 create database sensors
influxdb3 show databases
```

How to do it...

You will start this recipe by reading a file from the ExtraSensory dataset (for a specific UUID), focusing on one feature column—the watch accelerometer. You will be performing some data transformations to prepare the data before writing the time series DataFrame to InfluxDB:

1. Start by loading the required libraries:

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

2. The data consists of 60 compressed CSV files (`csv.gz`), which you can read using `pandas.read_csv()`. The default `compression` parameter in `read_csv` is set to `infer`. This means that pandas will infer based on the file extension which compression or decompression protocol to use. The files have a .gz extension, which will be used to infer which decompression protocol to use. Alternatively, you can indicate which compression protocol to use with `compression='gzip'`.

   In the following code, you will read one of these files, select both the `timestamp` and `watch_acceleration:magnitude_stats:mean` columns, rename the columns, and, finally, perform a **backfill** operation for all **na** (missing) values:

```
path = Path('../../datasets/Ch4/ExtraSensory/')
file = '0A986513-7828-4D53-AA1F-E02D6DF9561B.features_labels.csv.gz'
columns = ['timestamp',
           'watch_acceleration:magnitude_stats:mean']
df = pd.read_csv(path.joinpath(file),
                 usecols=columns,
                 compression='gzip')
df = df.bfill()
df.columns = ['timestamp','wacc']
df.shape
>>
(3960, 2)
```

   The backfill operation, `bfill()`, is applied to handle any missing values by propagating the next valid observation backward to fill the gaps, which is important for continuous time series analysis. From the preceding output, you should have 3,960 sensor readings from that one file.

3. To write the data to InfluxDB, you need at least one `measurement` column and a `timestamp` column. Currently, the timestamp is a Unix timestamp (**epoch**) captured in seconds, which is an acceptable format for writing out data to InfluxDB. For example, `2015-12-08 7:06:37 PM` is stored as `1449601597` in the dataset.

   InfluxDB stores timestamps in epoch nanoseconds on disk, but when querying data, InfluxDB will display the data in **RFC3339 UTC format** to make it more human-readable. So, `1449601597` in **RFC3339** would be represented as `2015-12-08T19:06:37+00:00.000Z`. Note that the precision in InfluxDB is in *nanoseconds*.

   In the following step, you will convert the Unix timestamp to a format that is more human-readable for your analysis in **pandas**, which is also an acceptable format with InfluxDB:

```
df['timestamp'] = pd.to_datetime(df['timestamp'],
                                  origin='unix',
                                  unit='s',
                                  utc=True)
df.set_index('timestamp', inplace=True)
print(df.head())
```

```
>>
                                  wacc
timestamp
2015-12-08 19:06:37+00:00   995.369977
2015-12-08 19:07:37+00:00   995.369977
2015-12-08 19:08:37+00:00   995.369977
2015-12-08 19:09:37+00:00   996.406005
2015-12-08 19:10:55+00:00  1034.180063
```

In the preceding code, the `unit` parameter is set to `'s'` for **seconds.** This instructs pandas to calculate the number of seconds based on the origin. The `origin` parameter is set to `unix` by default, so the conversion will calculate the number of seconds to the Unix epoch start provided. The `utc` parameter is set to `True`, which will return a **UTC** `DatetimeIndex` type. The `dtype` of our DataFrame index is now `datetime64[ns, UTC]`.

You can learn more about Unix epoch timestamps in the *Working with Unix epoch timestamps* recipe from *Chapter 5.*

4. Split the DataFrame into two (**batch1** and **batch2**) to showcase how we can write and then append to the sensor database in InfluxDB:

```
n = len(df)
half = n // 2

batch1 = df.iloc[:half, :].copy()
batch2 = df.iloc[half:, :].copy()
```

The code splits the DataFrame into two equal parts (1,980 rows each for a total of 3,960 total rows combined). We are splitting the data to demonstrate how to append to an existing **measurement** (analogous to a table) in **InfluxDB.**

5. Establish a connection with the InfluxDB database instance. All you need is to pass your **token** and specify the **database** if it exists. In this example, you want to connect to the `extrasensory` database created in the *Getting ready* section:

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

If you did not create the `extrasensory` database, you can omit the database name and use the following:

```
client = InfluxDBClient3(host="http://localhost:8181", token=token)
```

This should work. But if you attempt to query or write to InfluxDB, you will be required to provide a database name. If the database name was provided when creating the **client** object, then the database name will be inferred when writing or reading, if not explicitly provided.

> While the Python client auto-creates databases during writes, this behavior is not guaranteed in future releases.
>
> When writing data, the database (extrasensory, in this example) will be auto-created if it doesn't exist. However, for production environments, it's recommended to explicitly create databases (pre-create databases) using the CLI or HTTP API for better control.

6. To write the **batch1** DataFrame, you can use the `client.write()` method:

```
client.write(batch1, data_frame_measurement_name="watch_acc")
```

   If you did not specify the database name when you created the `client` object, then you will need to pass it as shown:

```
client.write(batch1,
             database='extrasensory',
             data_frame_measurement_name='watch_acc')
```

7. Verify that the data has been written to the `watch_acc` measurement:

```
query = "SELECT * FROM watch_acc"

sensor_df = client.query(query=query, language="sql", mode='pandas')
print(sensor_df.shape)
>>
(1980, 2)
```

8. To append **batch2** to the same `watch_acc` measurement in the `extrasensory` database, you simply run `client.write()` and pass the batch2 DataFrame. If the measurement exists, then the default behavior is to append to it (if there are no duplicate timestamps):

```
client.write(batch2,
             database='extrasensory',
             data_frame_measurement_name="watch_acc")
```

   Again, you can query the database to verify that the total number of records has changed:

```
query = "SELECT * FROM watch_acc"

sensor_df = client.query(query=query, language="sql", mode='pandas')
print(sensor_df.shape)
>>
```

```
(3960, 2)

print(sensor_df.head())
>>
                 time         wacc
0 2015-12-08 19:06:37   995.369977
1 2015-12-08 19:07:37   995.369977
2 2015-12-08 19:08:37   995.369977
3 2015-12-08 19:09:37   996.406005
4 2015-12-08 19:10:55  1034.180063

sensor_df.info()
>>
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3960 entries, 0 to 3959
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   time    3960 non-null   datetime64[ns]
 1   wacc    3960 non-null   float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 62.0 KB
```

9. Now that you are done, you can close the client as shown:

```
client.close()
```

How it works...

Before writing a pandas DataFrame to InfluxDB using the `InfluxDBClient3` library, you will need to either write to an existing database or create a new one. The Python library currently only supports read and write operations with minimal administrative tasks. Such tasks can be done either via the InfluxDB3 CLI or InfluxDB HTTP API.

Once you are ready to write your pandas DataFrame, you need to create your client object by connecting to the database:

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

The client provides access to several read and write operations, including `query`, `query_async`, `write`, `write_file`, and `close`. The main difference between the `write()` and `write_file()` methods is that `write()` is used to write data that is already in memory (e.g., pandas DataFrame), in which you pass the data directly as the `record` parameter, and supports a Point object, dictionary, line protocol string, and DataFrames, including pandas or Polars DataFrames. The `write_file()` method is used when you want to write directly from a file. The method takes a `file` parameter (path to file) and supports various file formats, including `feather`, `parquet`, `csv`, `json`, and `orc`.

In the recipe, you used the `write()` method, which takes these parameters:

- **data_frame_measurement_name**: This is required when writing a DataFrame. The parameter specifies the name of the InfluxDB measurements (analogous to a table) where all rows in the DataFrame are written to the specified measurement.
- **data_frame_timestamp_column** (optional): This specifies the DataFrame column that contains timestamp values. If not provided, it is assumed that the timestamp is the index of the DataFrame (e.g., DatetimeIndex).
- **data_frame_tag_columns** (optional): This parameter takes a list of column names from the DataFrame that should be treated as tags in InfluxDB. **Tags** are indexed metadata that are useful for filtering and grouping data efficiently (e.g., sensor ID, location). Generally, columns not specified as timestamp or tags are treated as fields. **Fields** contain actual measured values (e.g., temperature, pressure). Non-measurement values, such as strings, are often treated as tags.

In the recipe, we used the following:

```
client.write(batch1,
             database='extrasensory',
             data_frame_measurement_name='watch_acc')
```

This would write the **batch1** DataFrame (1,980 records) to a measurement named `watch_acc` (table) in the `extrasensory` database. If the database did not exist, then it would create one. If the `watch_acc` measurement did not exist, then it would create one; otherwise, it would append to it. InfluxDB ignores duplicate timestamps for the same measurement. If you rerun `client.write()` with the same data, no new records will be appended.

There is another way to accomplish the same, using `client._write_api.write`:

```
client = InfluxDBClient3(host="http://localhost:8181",
                         token=token)
client._write_api.write(bucket='extrasensory',
                        record=df,
                        data_frame_measurement_name='wacc')
```

Here, bucket is equivalent to a database, but uses the legacy terminology (e.g., InfluxDB v2). In **InfluxDB v3**, database replaces the v2 term bucket. The `write()` method uses `database`, not `bucket`. In the previous example, you are creating a new measurement named `wacc` in the `extrasensory` database.

If you want to delete the extrasensory database, then you can run the following CLI command inside the terminal of the running Docker container:

```
influxdb3 delete database extrasensory --token <yourtokenhere>
```

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

When querying data from InfluxDB, you'll notice that the timestamp column is returned as "`time`" in the query results, regardless of what you named it in your original DataFrame. For example, in our recipe, we named our column "`timestamp`" and set it as the index, but when we queried the data back from InfluxDB, it appeared as "`time`". This is by design in InfluxDB's data model, where "`time`" is a reserved column name that automatically stores timestamp information.

There's more...

In the previous example, we had to manually close the connection using `client.close()`. For better resource management (for example, automatically closing the connection) and exception handling, you can benefit from using the `with` statement. The following example shows how you can rewrite the same code in a cleaner and more efficient format:

```
with InfluxDBClient3(
    token=token,
    host="http://localhost:8181",
    database="sensor"
) as client:
    client.write(df, data_frame_measurement_name="wacc")
```

See also

- To learn more about the InfluxDB line protocol, please refer to their documentation here: `https://docs.influxdata.com/influxdb/v2.0/reference/syntax/line-protocol/`
- To learn more about the Python API for InfluxDB 2.x, please refer to the official documentation here: `https://docs.influxdata.com/influxdb/cloud/tools/client-libraries/python/`