In mathematics, time series is a series of data points listed with respect to time; most commonly, it is a sequence taken at successive equal intervals point in time. Common examples of time series are daily closing values of the stock market, counts of sunspots etc. Time series analysis comprises methods for analysing time-series data to extract meaningful statistical information and other data characteristics. In contrast, time series forecasting uses a model to predict future values based on previously observed values. In this article, we are going to explore the following regression techniques used for time series forecasting; The dataset we are using for all the techniques remains the same and can be found here. The dataset contains weather data collected for the city of Delhi for four years, from 2013 to 2017. import pandas as pd data = pd.read_csv('DailyDelhiClimateTrain.csv') data.head() Lets plot the line chart for humidity. import plotly.express as px fig = px.line(data, x=data.date, y='humidity', title='Humidity with slider') fig.update_xaxes(rangeslider_visible=True) fig.show() In multiple regression models, we forecast variables of interest using a linear combination of predictors. Here in the autoregressive model, we forecast the variable of interest using a linear combination of past values of the variable. The term autoregression indicates it is a regression of variables against itself. The model can be formulated as; Where: Yt is the value of time series at time t C is the intercept Ø is the slope coefficient Yt-p is the lagged values of time series ε is the error term This method is suitable for univariate time series without trend and a seasonal component. Code Implementation # AR example from statsmodels.tsa.ar_model import AutoReg # fit model train,test = data[0:1000],data[1000:] model = AutoReg(train.humidity, lags=350) model_fit = model.fit() # make prediction pred = model_fit.predict(len(train),len(test)+len(train)-1,dynamic=False) plt.plot(test.humidity) plt.plot(pred,color='red') Rather than using past forecast values in regression, a moving average model uses past forecast errors in a regression-like model. In other words, the moving average models the next sequence as a linear function of residual error from the mean process at an earlier time step. Thus, it combines both autoregressive and moving average models. This method is suitable for univariate time series without trend and seasonal component. Code Implementation: #MA model from statsmodels.tsa.arima.model import ARIMA # fit model model = ARIMA(train.humidity,order=(300,0,0)) model_fit = model.fit() # make prediction pred = model_fit.predict(len(train),len(test)+len(train)-1) plt.plot(test.humidity) plt.plot(pred,color='red') It explicitly creates a suite of standard structure in time series data and it provides a simple and powerful method for forecasting. It combines both autoregressive and moving average models as well as a differencing pre-processing step of the sequence to make the sequence stationary. This method supports univariate time series with trend and without seasonal component. The statsmodel library provides the capability to fit ARIMA models. Code Implementation: from statsmodels.tsa.arima.model import ARIMA train,test = data.humidity[0:1000],data.humidity[1000:] X = train size = int(len(X) * 0.66) train, test = X[0:size], X[size:len(X)] history = [x for x in train] predictions = list() for i in range(len(test)): model = ARIMA(history, order=(5,1,0)) model_fit = model.fit() output = model_fit.forecast() pred = output[0] predictions.append(pred) true = test[i] history.append(obs) print('predicted=%f, expected=%f' % (pred, true)) plt.plot(test) plt.plot(predictions, color='red') An extension of ARIMA that supports the direct modeling of the seasonal component of the series is called SARIMA. The problem with ARIMA is that it does not support seasonal data i.e repeating cycles. ARIMA expects data that is not seasonal or seasonal component removed SARIMA adds the three hyperparameters to specify the AR, differencing and moving average for the seasonal component of series This model suitable for univariate time series with trend and seasonal component. Code Implementation: from statsmodels.tsa.statespace.sarimax import SARIMAX size = int(len(X) * 0.66) train, test = X[0:size], X[size:len(X)] history = [x for x in train] predictions = list() # walk-forward validation for t in range(len(test)): model = SARIMAX(history,seasonal_order=(3, 1, 0, 2)) model_fit = model.fit() output = model_fit.forecast() pred = output[0] predictions.append(pred) true = test[t] history.append(true) print('predicted=%f, expected=%f' % (pred, true)) plt.plot(test) plt.plot(predictions, color='red') The vector autoregression model can predict when two or more time series influence each other means the relationship involved in time series is bi-directional. This model considers each variable as a function of past values that are to be predicted, nothing but the time lag of the series. For all this, it considers an autoregressive model. The main difference between the previous model and VAR is, those models are unidirectional, where predictors influence the Y but not vice-versa. Whereas the VAR model is bidirectional, variables influence each other. This model is suitable for multivariate time series without trend and seasonal components. Code Implementation: Load multiple variables: x1 = data.humidity.values x2 = data.meantemp.values list1 = list() for i in range(len(x1)): x3 = x1[i] x4 = x2[i] row1 = [x3,x4] list1.append(row1) Fit and forecast to few steps from statsmodels.tsa.vector_ar.var_model import VAR # fit model model = VAR(list1) model_fit = model.fit() # make prediction forecast = model_fit.forecast(model_fit.y, steps=5) print(forecast) Output: [[95.76561271 10.57589906] [92.08148688 11.10511153] [88.87374484 11.59330815] [86.07847799 12.04540676] [83.64040052 12.46567364]] This article has seen the major techniques used to forecast time series entities with a practical use case. The most time-consuming thing in the univariate techniques is adjusting the lag values; the proper lag value decides the nature of forecasting. The rest of the techniques are straightforward. A time series, as the name suggests, is a series of data points that are listed in chronological order. More often than not, time series are used to track the changes of certain things over short and long periods – with the price of stocks or even other commodities being a prime example. Regardless, you're taking a closer look at how something changes at regular intervals over time – which is important when attempting to use the past to forecast the future. Why time series analysis is important If you can see exactly how the price of a security has changed over time, for example, you can make a more educated guess about what might happen to the price over the same interval in the future. This can lead to better and more informed decision making, which is what makes time series analysis so important to so many sectors. Why time series data is unique A time series is a series of data points indexed in time. The fact that time series data is ordered makes it unique in the data space because it often displays serial dependence. Serial dependence occurs when the value of a datapoint at one time is statistically dependent on another datapoint in another time. However, this attribute of time series data violates one of the fundamental assumptions of many statistical analyses — that data is statistically independent. What is autocorrelation in time series? 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. Therefore, a time series autocorrelation attempts to measure the current values of a variable against the historical data of that variable. It ultimately plots one series over the other, and determines the degree of similarity between the two. For the sake of comparison, autocorrelation is essentially the exact same process that you would go through when calculating the correlation between two different sets of time series values on your own. The major difference here is that autocorrelation uses the same time series two times: once in its original values, and then again once a few different time periods have occurred. Autocorrelation is also known as serial correlation, time series correlation and lagged correlation. Regardless of how it's being used, autocorrelation is an ideal method for uncovering trends and patterns in time series data that would have otherwise gone undiscovered. Autocorrelation examples It's important to note that autocorrelation in time series data is that not all fields use this technique in exactly the same way. It's nothing if not malleable – meaning that the simple principle of comparing data with a delayed copy of itself is equally valuable in a wide array of contexts. Likewise, not all of the applications of autocorrelation in various fields are equivalent – meaning that they're using a simple process to arrive at a totally different end result. 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 Autocorrelation is also 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. Example 3: Global positioning systems Autocorrelation is also one of the primary mathematical techniques at the heart of the GPS chip that is embedded in smartphones or other mobile devices. Here, autocorrelation is used to correct for propagation delay — meaning the time shift that happens when a carrier signal is transmitted and before it is ultimately received by the GPS device in question. This is how the GPS always knows exactly where you are and tells you when and where to turn just before you arrive at that precise location. Example 4: Signal processing Autocorrelation is also a very important technique in signal processing, which is a part of electrical engineering that focuses on understanding more about (and even modifying or synthesizing) signals like sound, images and sometimes scientific measurements. In this context, autocorrelation can help people better understand repeating events like musical beats — which itself is important for determining the proper tempo of a song. Many also use it to estimate a very specific pitch in a musical tone, too. Example 5: Astrophysics Astrophysics is that branch of astronomy that takes our known principles of both physics and chemistry and applies them in a way that helps us better understand the nature of objects in outer space, rather than simply remaining satisfied with knowing their relative position or how they're moving. This is another important way in which autocorrelation is used, as it helps professionals study the spatial distribution between celestial bodies in the universe like galaxies. It can also be helpful when making multi-wavelength observations of low mass x-ray binaries, too. Why autocorrelation matters Often, one of the first steps in any data analysis is performing regression analysis. However, one of the assumptions of regression analysis is that the data has no autocorrelation. This can be frustrating because if you try to do a regression analysis on data with autocorrelation, then your analysis will be misleading. Additionally, some time series forecasting methods (specifically regression modeling) rely on the assumption that there isn't any autocorrelation in the residuals (the difference between the fitted model and the data). People often use the residuals to assess whether their model is a good fit while ignoring that assumption that the residuals have no autocorrelation (or that the errors are independent and identically distributed or i.i.d). This mistake can mislead people into believing that their model is a good fit when in fact it isn't. I highly recommend reading this article about How (not) to use Machine Learning for time series forecasting: Avoiding the pitfalls in which the author demonstrates how the increasingly popular LSTM (Long Short Term Memory) Network can appear to be an excellent univariate time series predictor, when in reality it's just overfitting the data. He goes further to explain how this misconception is the result of accuracy metrics failing due to the presence of autocorrelation. Finally, perhaps the most compelling aspect of autocorrelation analysis is how it can help us uncover hidden patterns in our data and help us select the correct forecasting methods. Specifically, we can use it to help identify seasonality and trend in our time series data. Additionally, analyzing the autocorrelation function (ACF) and partial autocorrelation function (PACF) in conjunction is necessary for selecting the appropriate ARIMA model for your time series prediction. Testing for autocorrelation Any autocorrelation that may be present in time series data is determined using a correlogram, also known as an ACF plot. This is used to help you determine whether your series of numbers is exhibiting autocorrelation at all, at which point you can then begin to better understand the pattern that the values in the series may be predicting. The most common autocorrelation test is called the Durbin-Watson test, which was named after James Durbin and Geoffrey Watson and was derived back in the early 1950s. Autocorrelation statistics and test Also commonly referred to as the Durbin-Watson statistic, this test is used to detect the presence of autocorrelation at a lag of one in any prediction errors uncovered from a regression analysis. The precise calculation used to conduct this test can be found here. Once you have successfully plugged your numbers into the Durbin-Watson test, it reports a statistic on a value of 0 to 4. If the value returned is 2, there is no autocorrelation in your time series to speak of. If the value is between 0 and 2, you're seeing what is known as positive autocorrelation - something that is very common in time series data. If the value is anywhere between 2 and 4, that means there is a negative correlation — something that is less common in time series data, but that does occur under certain circumstances. How to determine if your time series data has autocorrelation For this exercise, I'm using InfluxDB and the InfluxDB Python CL. 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. Dataset: curl -o NOAA_data.txt influx -import -path=NOAA_data.txt -precision=s -database=NOAA_water_database This analysis and code is included in a jupyter notebook in this repo. First, I import all of my dependencies. import pandas as pd import numpy as np import matplotlib import matplotlib.pyplot as plt from influxdb import InfluxDBClient from statsmodels.graphics.tsaplots import plot_pacf from statsmodels.graphics.tsaplots import plot_acf from scipy.stats import linregress Next I connect to the client, query my water temperature data, and plot it. client = InfluxDBClient(host='localhost', port=8086) h2O = client.query('SELECT mean("degrees") AS "h2O_temp" FROM "NOAA_water_database"."autogen"."h2o_temperature" GROUP BY time(12h) LIMIT 60') h2O_points = [p for p in h2O.get_points()] h2O_df = pd.DataFrame(h2O_points) h2O_df['time_step'] = range(0,len(h2O_df['time'])) h2O_df.plot(kind='line',x='time_step',y='h2O_temp') plt.show() Fig 1. H2O temperature vs. timestep From looking at the plot above, 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. I can calculate the autocorrelation with Pandas.Sereis.autocorr() function which returns the value of the Pearson correlation coefficient. The Pearson correlation coefficient is a measure of the linear correlation between two variables. The Pearson correlation coefficient has a value between -1 and 1, where 0 is no linear correlation, >0 is a positive correlation, and 0, which verifies that our data doesn't have any autocorrelation. At first, I found this result surprising, because usually the air temperature on one day is highly correlated with the temperature the day before. I assumed the same would be true about water temperature. This result reminded me that streams and rivers don't have the same system behavior as air. I'm no hydrologist, but I know spring fed streams or snowmelt can often be the same temperature year-round. Perhaps they exhibit a stationary temperature profile day to day where the mean, variance, and autocorrelation are all constant (where autocorrelation is = 0). Uncovering seasonality with autocorrelation in time series data 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. client = InfluxDBClient(host='localhost', port=8086) 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 = now() - INTERVAL '90 days' GROUP BY room, _time ORDER BY _time''') print(table.to_pandas().to_markdown()) client.close() querySQL() c from influxdb_client_3 import InfluxDBClient3 import os database = os.getenv('INFLUX_DATABASE') token = os.getenv('INFLUX_TOKEN') host=" " def write_line_protocol(): client = InfluxDBClient3(host, database=database, token=token) record = "home,room=Living\\ Room temp=22.2,hum=36.4,co=17i" print("Writing record:", record) client.write(record) client.close() write_line_protocol() c @main struct QueryCpuData: AsyncParsableCommand { @Option(name: .shortAndLong, help: "The name or id of the bucket destination.") private var bucket: String @Option(name: .shortAndLong, help: "The name or id of the organization destination.") private var org: String @Option(name: .shortAndLong, help: "Authentication token.") private var token: String @Option(name: .shortAndLong, help: "HTTP address of InfluxDB.") private var url: String } extension QueryCpuData { mutating func run() async throws { // // Initialize Client with default Bucket and Organization // let client = InfluxDBClient(url: url, token: token, options: InfluxDBClient.InfluxDBOptions(bucket: bucket, org: org)) // Flux query let query = """ from(bucket: "\(self.bucket)") |> range(start: -10m) |> filter(fn: (r) => r["_measurement"] == "cpu") |> filter(fn: (r) => r["cpu"] == "cpu-total") |> filter(fn: (r) => r["_field"] == "usage_user" or r["_field"] == "usage_system") |> last() """ print("Query to execute:\(query)") let response = try await client.queryAPI.queryRaw(query: query) let csv = String(decoding: response, as: UTF8.self) print("InfluxDB response: \(csv)") client.close() } } c import ArgumentParser import Foundation import InfluxDBSwift import InfluxDBSwiftApis @main struct WriteData: AsyncParsableCommand { @Option(name: .shortAndLong, help: "The name or id of the bucket destination.") private var bucket: String @Option(name: .shortAndLong, help: "The name or id of the organization destination.") private var org: String @Option(name: .shortAndLong, help: "Authentication token.") private var token: String @Option(name: .shortAndLong, help: "HTTP address of InfluxDB.") private var url: String } extension WriteData { mutating func run() async throws { // // Initialize Client with default Bucket and Organization // let client = InfluxDBClient(url: url, token: token, options: InfluxDBClient.InfluxDBOptions(bucket: bucket, org: org)) // // Record defined as Data Point // let recordPoint = InfluxDBClient .Point("demo") .addTag(key: "type", value: "point") .addField(key: "value", value: .int(2)) // // Record defined as Data Point with Timestamp // let recordPointDate = InfluxDBClient .Point("demo") .addTag(key: "type", value: "point-timestamp") .addField(key: "value", value: .int(2)) .time(time: .date(Date())) try await client.makeWriteAPI().write(points: [recordPoint, recordPointDate]) print("Written data:\([recordPoint, recordPointDate].map { "\t- \($0)" }.joined(separator: ""))") print("Success!") client.close() } } c import {InfluxDBClient} from '@influxdata/influxdb3-client' import {tableFromArrays} from 'apache-arrow'; const database = process.env.INFLUX_DATABASE; const token = process.env.INFLUX_TOKEN; const host = " "; async function main() { const client = new InfluxDBClient({host, token}) const query = ` SELECT room, DATE_BIN(INTERVAL '1 day', time) AS _time, AVG(temp) AS temp, AVG(hum) AS hum, AVG(co) AS co FROM home WHERE time >= now() - INTERVAL '90 days' GROUP BY room, _time ORDER BY _time ` const result = await client.query(query, database) const data = {room: [], day: [], temp: []} for await (const row of result) { data.day.push(new Date(row._time).toISOString()) data.room.push(row.room) data.temp.push(row.temp) } console.table([...tableFromArrays(data)]) client.close() } main() c import {InfluxDBClient} from '@influxdata/influxdb3-client' const database = process.env.INFLUX_DATABASE; const token = process.env.INFLUX_TOKEN; const host = " "; async function main() { const client = new InfluxDBClient({host, token}) const record = "home,room=Living\\ Room temp=22.2,hum=36.4,co=17i" await client.write(record, database) client.close() } main() c package com.influxdb3.examples; import com.influxdb.v3.client.InfluxDBClient; import java.util.stream.Stream; public final class Query { private Query() { //not called } /** * @throws Exception */ public static void main() throws Exception { final String hostUrl = " "; final char[] authToken = (System.getenv("INFLUX_TOKEN")).toCharArray(); final String database = System.getenv("INFLUX_DATABASE"); try (InfluxDBClient client = InfluxDBClient.getInstance(hostUrl, authToken, database)) { String sql = """ SELECT room, DATE_BIN(INTERVAL '1 day', time) AS _time, AVG(temp) AS temp, AVG(hum) AS hum, AVG(co) AS co FROM home WHERE time >= now() - INTERVAL '90 days' GROUP BY room, _time ORDER BY _time"""; String layoutHeading = "| %-16s | %-12s | %-6s |%n"; System.out.printf("--%n"); System.out.printf(layoutHeading, "day", "room", "temp"); System.out.printf("--%n"); String layout = "| %-16s | %-12s | %.2f |%n"; try (Stream stream = client.query(sql)) { stream.forEach(row -> System.out.printf(layout, row[1], row[0], row[2])); } } } } c package com.influxdb3.examples; import com.influxdb.v3.client.InfluxDBClient; public final class Write { public static void main() throws Exception { final String hostUrl = " "; final char[] authToken = (System.getenv("INFLUX_TOKEN")).toCharArray(); final String database = System.getenv("INFLUX_DATABASE"); try (InfluxDBClient client = InfluxDBClient.getInstance(hostUrl, authToken, database)) { String record = "home,room=Living\\ Room temp=22.2,hum=36.4,co=17i"; System.out.printf("Write record: %s%n", record); client.writeRecord(record); } } } c InfluxDB2::Client.use(' ', 'my-token', org: 'my-org') do |client| result = client .create_query_api .query_raw(query: 'from(bucket:"my-bucket") |> range(start: 1970-01-01) |> last()') puts result end c InfluxDB2::Client.use(' ', 'my-token', bucket: 'my-bucket', org: 'my-org', precision: InfluxDB2::WritePrecision::NANOSECOND) do |client| write_api = client.create_write_api write_api.write(data: 'h2o,location=west value=33i 15') end c package example import org.apache.pekko.actor.ActorSystem import org.apache.pekko.stream.scaladsl.Sink import com.influxdb.client.scala.InfluxDBClientScalaFactory import com.influxdb.query.FluxRecord import scala.concurrent.Await import scala.concurrent.duration.Duration object InfluxDB2ScalaExample { implicit val system: ActorSystem = ActorSystem("it-tests") def main(args: Array[String]): Unit = { val influxDBClient = InfluxDBClientScalaFactory .create(" ", "my-token".toCharArray, "my-org") val fluxQuery = ("from(bucket: \"my-bucket\")" + " |> range(start: -1d)" + " |> filter(fn: (r) => (r[\"_measurement\"] == \"cpu\" and r[\"_field\"] == \"usage_system\"))") //Result is returned as a stream val results = influxDBClient.getQueryScalaApi().query(fluxQuery) //Example of additional result stream processing on client side val sink = results //filter on client side using `filter` built-in operator .filter(it => "cpu0" == it.getValueByKey("cpu")) //take first 20 records .take(20) //print results .runWith(Sink.foreach[FluxRecord](it => println(s"Measurement: ${it.getMeasurement}, value: ${it.getValue}"))) // wait to finish Await.result(sink, Duration.Inf) influxDBClient.close() system.terminate() } } c package com.influxdb.client.scala.internal import org.apache.pekko.Done import org.apache.pekko.stream.scaladsl.{Flow, Keep, Sink, Source} import com.influxdb.client.InfluxDBClientOptions import com.influxdb.client.domain.WritePrecision import com.influxdb.client.internal.{AbstractWriteBlockingClient, AbstractWriteClient} import com.influxdb.client.scala.WriteScalaApi import com.influxdb.client.service.WriteService import com.influxdb.client.write.{Point, WriteParameters} import javax.annotation.Nonnull import scala.collection.immutable.ListMap import scala.concurrent.Future import scala.jdk.CollectionConverters._ class WriteScalaApiImpl(@Nonnull service: WriteService, @Nonnull options: InfluxDBClientOptions) extends AbstractWriteBlockingClient(service, options) with WriteScalaApi { override def writeRecord(precision: Option[WritePrecision], bucket: Option[String], org: Option[String]): Sink[String, Future[Done]] = { Flow[String] .map(record => Seq(new AbstractWriteClient.BatchWriteDataRecord(record))) .toMat(Sink.foreach(batch => writeHttp(precision, bucket, org, batch)))(Keep.right) } override def writeRecords(precision: Option[WritePrecision], bucket: Option[String], org: Option[String]): Sink[Seq[String], Future[Done]] = { writeRecords(toWriteParameters(precision, bucket, org)) } override def writeRecords(parameters: WriteParameters): Sink[Seq[String], Future[Done]] = { Flow[Seq[String]] .map(records => records.map(record => new AbstractWriteClient.BatchWriteDataRecord(record))) .toMat(Sink.foreach(batch => writeHttp(parameters, batch)))(Keep.right) } override def writePoint(bucket: Option[String], org: Option[String]): Sink[Point, Future[Done]] = { Flow[Point] .map(point => (point.getPrecision, Seq(new AbstractWriteClient.BatchWriteDataPoint(point, options)))) .toMat(Sink.foreach(batch => writeHttp(Some(batch._1), bucket, org, batch._2)))(Keep.right) } override def writePoints(bucket: Option[String], org: Option[String]): Sink[Seq[Point], Future[Done]] = { writePoints(new WriteParameters(bucket.orNull, org.orNull, null, null)) } override def writePoints(parameters: WriteParameters): Sink[Seq[Point], Future[Done]] = { Flow[Seq[Point]] // create ordered Map .map(records => records.foldRight(ListMap.empty[WritePrecision, Seq[Point]]) { case (point, map) => map.updated(point.getPrecision, point +: map.getOrElse(point.getPrecision, Seq())) }.toList.reverse) .map(grouped => grouped.map(group => (group._1, group._2.map(point => new AbstractWriteClient.BatchWriteDataPoint(point, options))))) .flatMapConcat(batches => Source(batches)) .toMat(Sink.foreach(batch => writeHttp(parameters.copy(batch._1, options), batch._2)))(Keep.right) } override def writeMeasurement[M](precision: Option[WritePrecision], bucket: Option[String], org: Option[String]): Sink[M, Future[Done]] = { Flow[M] .map(measurement => { val parameters = toWriteParameters(precision, bucket, org) Seq(toMeasurementBatch(measurement, parameters.precisionSafe(options))) }) .toMat(Sink.foreach(batch => writeHttp(precision, bucket, org, batch)))(Keep.right) } override def writeMeasurements[M](precision: Option[WritePrecision], bucket: Option[String], org: Option[String]): Sink[Seq[M], Future[Done]] = { writeMeasurements(toWriteParameters(precision, bucket, org)) } override def writeMeasurements[M](parameters: WriteParameters): Sink[Seq[M], Future[Done]] = { Flow[Seq[M]] .map(records => records.map(record => toMeasurementBatch(record, parameters.precisionSafe(options)))) .toMat(Sink.foreach(batch => writeHttp(parameters, batch)))(Keep.right) } private def writeHttp(precision: Option[WritePrecision], bucket: Option[String], org: Option[String], batch: Seq[AbstractWriteClient.BatchWriteData]): Done = { writeHttp(toWriteParameters(precision, bucket, org), batch) } private def writeHttp(parameters: WriteParameters, batch: Seq[AbstractWriteClient.BatchWriteData]): Done = { write(parameters, batch.toList.asJava.stream()) Done.done() } private def toWriteParameters(precision: Option[WritePrecision], bucket: Option[String], org: Option[String]): WriteParameters = { val parameters = new WriteParameters(bucket.orNull, org.orNull, precision.orNull, null) parameters.check(options) parameters } } c package influxdbv3 import ("context" "fmt" "io" "os" "text/tabwriter" "github.com/apache/arrow/go/v12/arrow" "github.com/InfluxCommunity/influxdb3-go/influx") func QuerySQL() error { url := " " token := os.Getenv("INFLUX_TOKEN") database := os.Getenv("INFLUX_DATABASE") client, err := influx.New(influx.Configs{ HostURL: url, AuthToken: token, }) defer func (client *influx.Client) { err := client.Close() if err != nil { panic(err) } }(client) query := ` SELECT room, DATE_BIN(INTERVAL '1 day', time) AS _time, AVG(temp) AS temp, AVG(hum) AS hum, AVG(co) AS co FROM home WHERE time >= now() - INTERVAL '90 days' GROUP BY room, _time ORDER BY _time ` iterator, err := client.Query(context.Background(), database, query) if err != nil { panic(err) } w := tabwriter.NewWriter(io.Discard, 4, 4, 1, ' ', 0) w.Init(os.Stdout, 0, 8, 0, '\t', 0) fmt.Fprintln(w, "day\troom\ttemp") for iterator.Next() { row := iterator.Value() day := (row["_time"].(arrow.Timestamp)).ToTime(arrow.TimeUnit(arrow.Nanosecond)) fmt.Fprintf(w, "%s\t%s\t%.2f", day, row["room"], row["temp"]) } w.Flush() return nil } c package influxdbv3 import ("context" "os" "fmt" "github.com/InfluxCommunity/influxdb3-go/influx") func WriteLineProtocol() error { url := " " token := os.Getenv("INFLUX_TOKEN") database := os.Getenv("INFLUX_DATABASE") client, err := influx.New(influx.Configs{ HostURL: url, AuthToken: token, }) defer func (client *influx.Client) { err := client.Close() if err != nil { panic(err) } }(client) record := "home,room=Living\\ Room temp=22.2,hum=36.4,co=17i" fmt.Println("Writing record: ", record) err = client.Write(context.Background(), database, []byte(record)) if err != nil { panic(err) } return nil } c using System; using System.Threading.Tasks; using InfluxDB3.Client; using InfluxDB3.Client.Query; namespace InfluxDBv3; public class Query { public static async Task QuerySQL() { const string hostUrl = " "; string? database = System.Environment.GetEnvironmentVariable("INFLUX_DATABASE"); string? authToken = System.Environment.GetEnvironmentVariable("INFLUX_TOKEN"); using var client = new InfluxDBClient(hostUrl, authToken: authToken, database: database); const string sql = @" SELECT room, DATE_BIN(INTERVAL '1 day', time) AS _time, AVG(temp) AS temp, AVG(hum) AS hum, AVG(co) AS co FROM home WHERE time >= now() - INTERVAL '90 days' GROUP BY room, _time ORDER BY _time "; Console.WriteLine("{0,-30}{1,-15}{2,-15}", "day", "room", "temp"); await foreach (var row in client.Query(query: sql)) { Console.WriteLine("{0,-30}{1,-15}{2,-15}", row[1], row[0], row[2]); } Console.WriteLine(); } } c using System; using System.Threading.Tasks; using InfluxDB3.Client; using InfluxDB3.Client.Query; namespace InfluxDBv3; public class Write { public static async Task WriteLineProtocol() { const string hostUrl = " "; string? database = System.Environment.GetEnvironmentVariable("INFLUX_DATABASE"); string? authToken = System.Environment.GetEnvironmentVariable("INFLUX_TOKEN"); using var client = new InfluxDBClient(hostUrl, authToken: authToken, database: database); const string record = "home,room=Living\\ Room temp=22.2,hum=36.4,co=17i"; Console.WriteLine("Write record: {0,-30}", record); await client.WriteRecordAsync(record: record); } } c client range(start: -1h) |> drop(columns: ["_start", "_stop"])') data c