
AWS SDK for pandas

Release 2.20.0

Amazon Web Services

Mar 03, 2023

CONTENTS

1	Read The Docs	3
1.1	What is AWS SDK for pandas?	3
1.2	Install	3
1.3	Tutorials	9
1.4	API Reference	132
	Index	369

An AWS Professional Service open source initiative | aws-proserve-opensource@amazon.com

AWS Data Wrangler is now **AWS SDK for pandas (awswrangler)**. We're changing the name we use when we talk about the library, but everything else will stay the same. You'll still be able to install using `pip install awswrangler` and you won't need to change any of your code. As part of this change, we've moved the library from AWS Labs to the main AWS GitHub organisation but, thanks to the GitHub's redirect feature, you'll still be able to access the project by its old URLs until you update your bookmarks. Our documentation has also moved to aws-sdk-pandas.readthedocs.io, but old bookmarks will redirect to the new site.

```
>>> pip install awswrangler
```

```
import awswrangler as wr
import pandas as pd
from datetime import datetime

df = pd.DataFrame({"id": [1, 2], "value": ["foo", "boo"]})

# Storing data on Data Lake
wr.s3.to_parquet(
    df=df,
    path="s3://bucket/dataset/",
    dataset=True,
    database="my_db",
    table="my_table"
)

# Retrieving the data directly from Amazon S3
df = wr.s3.read_parquet("s3://bucket/dataset/", dataset=True)

# Retrieving the data from Amazon Athena
df = wr.athena.read_sql_query("SELECT * FROM my_table", database="my_db")

# Get a Redshift connection from Glue Catalog and retrieving data from Redshift Spectrum
con = wr.redshift.connect("my-glue-connection")
df = wr.redshift.read_sql_query("SELECT * FROM external_schema.my_table", con=con)
con.close()

# Amazon Timestream Write
df = pd.DataFrame({
    "time": [datetime.now(), datetime.now()],
    "my_dimension": ["foo", "boo"],
    "measure": [1.0, 1.1],
})
rejected_records = wr.timestream.write(df,
    database="sampleDB",
    table="sampleTable",
    time_col="time",
    measure_col="measure",
    dimensions_cols=["my_dimension"],
)

# Amazon Timestream Query
wr.timestream.query("""
SELECT time, measure_value::double, my_dimension
```

(continues on next page)

(continued from previous page)

```
FROM "sampleDB"."sampleTable" ORDER BY time DESC LIMIT 3  
""")
```

READ THE DOCS

1.1 What is AWS SDK for pandas?

An [AWS Professional Service open source](#) python initiative that extends the power of [Pandas](#) library to AWS connecting **DataFrames** and AWS data related services.

Easy integration with Athena, Glue, Redshift, Timestream, OpenSearch, Neptune, QuickSight, Chime, CloudWatchLogs, DynamoDB, EMR, SecretManager, PostgreSQL, MySQL, SQLServer and S3 (Parquet, CSV, JSON and EXCEL).

Built on top of other open-source projects like [Pandas](#), [Apache Arrow](#) and [Boto3](#), it offers abstracted functions to execute usual ETL tasks like load/unload data from **Data Lakes**, **Data Warehouses** and **Databases**.

Check our [tutorials](#) or the [list of functionalities](#).

1.2 Install

AWS SDK for pandas runs on Python 3.7, 3.8, 3.9 and 3.10, and on several platforms (AWS Lambda, AWS Glue Python Shell, EMR, EC2, on-premises, Amazon SageMaker, local, etc).

Some good practices to follow for options below are:

- Use new and isolated Virtual Environments for each project ([venv](#)).
- On Notebooks, always restart your kernel after installations.

Note: If you want to use `awswrangler` to connect to Microsoft SQL Server or Oracle, some additional configuration is needed. Please have a look at the corresponding section below.

1.2.1 PyPI (pip)

```
>>> pip install awswrangler
```

1.2.2 Conda

```
>>> conda install -c conda-forge awswrangler
```

1.2.3 AWS Lambda Layer

Managed Layer

Note: There is a one week minimum delay between version release and layers being available in the AWS Lambda console.

AWS SDK for pandas is available as an AWS Lambda Managed layer in all AWS commercial regions.

It can be accessed in the AWS Lambda console directly:

The screenshot shows the 'Choose a layer' dialog in the AWS Lambda console. It has three tabs: 'AWS layers', 'Custom layers', and 'Specify an ARN'. The 'AWS layers' tab is active. Below the tabs, it says 'Layers provided by AWS that are compatible with your function's runtime.' and shows a dropdown menu with 'AWSSDKPandas-Python39'. Below that is a 'Version' dropdown menu with '1'. There is a search bar with a magnifying glass icon. Below the search bar, a list shows '1' with a checkmark. At the bottom right are 'Cancel' and 'Add' buttons.

Or via its ARN: `arn:aws:lambda:<region>:336392948345:layer:AWSSDKPandas-Python<python-version>:<layer-version>`.

For example: `arn:aws:lambda:us-east-1:336392948345:layer:AWSSDKPandas-Python37:1`.

The full list of ARNs is available [here](#).

Custom Layer

You can also create your own Lambda layer with these instructions:

- 1 - Go to [GitHub's release section](#) and download the zipped layer for to the desired version. Alternatively, you can download the zip from the [public artifacts bucket](#).
- 2 - Go to the AWS Lambda console, open the layer section (left side) and click **create layer**.
- 3 - Set name and python version, upload your downloaded zip file and press **create**.
- 4 - Go to your Lambda function and select your new layer!

Serverless Application Repository (SAR)

AWS SDK for pandas layers are also available in the [AWS Serverless Application Repository \(SAR\)](#).

The app deploys the Lambda layer version in your own AWS account and region via a CloudFormation stack. This option provides the ability to use semantic versions (i.e. library version) instead of Lambda layer versions.

Table 1: AWS SDK for pandas Layer Apps

App	ARN	Description
aws-sdk-pandas-layer-py3-7	arn:aws:serverlessrepo:us-east-1:336392948345:application:aws-sdk-pandas-layer-py3-7	Layer for Python 3.7.x runtimes
aws-sdk-pandas-layer-py3-8	arn:aws:serverlessrepo:us-east-1:336392948345:application:aws-sdk-pandas-layer-py3-8	Layer for Python 3.8.x runtimes
aws-sdk-pandas-layer-py3-9	arn:aws:serverlessrepo:us-east-1:336392948345:application:aws-sdk-pandas-layer-py3-9	Layer for Python 3.9.x runtimes

Here is an example of how to create and use the AWS SDK for pandas Lambda layer in your CDK app:

```
from aws_cdk import core, aws_sam as sam, aws_lambda

class AWSSDKPandasApp(core.Construct):
    def __init__(self, scope: core.Construct, id_: str):
        super().__init__(scope, id)

        aws_sdk_pandas_layer = sam.CfnApplication(
            self,
            "awssdkpandas-layer",
            location=sam.CfnApplication.ApplicationLocationProperty(
                application_id="arn:aws:serverlessrepo:us-east-1:336392948345:applications/aws-
↪ sdk-pandas-layer-py3-8",
                semantic_version="2.20.0", # Get the latest version from https://github.com/aws/
↪ aws-sdk-pandas/releases
            ),
        )

        aws_sdk_pandas_layer_arn = aws_sdk_pandas_layer.get_attr("Outputs.WranglerLayer38Arn
↪").to_string()
        aws_sdk_pandas_layer_version = aws_lambda.LayerVersion.from_layer_version_arn(self,
↪ "awssdkpandas-layer-version", aws_sdk_pandas_layer_arn)

        aws_lambda.Function(
            self,
            "awssdkpandas-function",
            runtime=aws_lambda.Runtime.PYTHON_3_8,
            function_name="sample-awssdk-pandas-lambda-function",
            code=aws_lambda.Code.from_asset("./src/awssdk-pandas-lambda"),
```

(continues on next page)

(continued from previous page)

```
handler='lambda_function.lambda_handler',
layers=[aws_sdk_pandas_layer_version]
)
```

1.2.4 AWS Glue Python Shell Jobs

Note: Glue Python Shell Python3.9 has version 2.15.1 of awswrangler [baked in](#). If you need a different version, follow instructions below:

- 1 - Go to [GitHub's release page](#) and download the wheel file (.whl) related to the desired version. Alternatively, you can download the wheel from the [public artifacts bucket](#).
- 2 - Upload the wheel file to the Amazon S3 location of your choice.
- 3 - Go to your Glue Python Shell job and point to the S3 wheel file in the *Python library path* field.

[Official Glue Python Shell Reference](#)

1.2.5 AWS Glue PySpark Jobs

Note: AWS SDK for pandas has compiled dependencies (C/C++) so support is only available for Glue PySpark Jobs ≥ 2.0 .

Go to your Glue PySpark job and create a new *Job parameters* key/value:

- Key: `--additional-python-modules`
- Value: `pyarrow==2,awswrangler`

To install a specific version, set the value for the above Job parameter as follows:

- Value: `cython==0.29.21,pg8000==1.21.0,pyarrow==2,pandas==1.3.0,awswrangler==2.20.0`

Note: Pyarrow 3 is not currently supported in Glue PySpark Jobs, which is why an installation of pyarrow 2 is required.

[Official Glue PySpark Reference](#)

1.2.6 Public Artifacts

Lambda zipped layers and Python wheels are stored in a publicly accessible S3 bucket for all versions.

- Bucket: `aws-data-wrangler-public-artifacts`
- Prefix: `releases/<version>/`
 - Lambda layer: `awswrangler-layer-<version>-py<py-version>.zip`
 - Python wheel: `awswrangler-<version>-py3-none-any.whl`

For example: `s3://aws-data-wrangler-public-artifacts/releases/2.20.0/awswrangler-layer-2.20.0-py3.8.zip`

1.2.7 Amazon SageMaker Notebook

Run this command in any Python 3 notebook cell and then make sure to **restart the kernel** before importing the **aws wrangler** package.

```
>>> !pip install aws wrangler
```

1.2.8 Amazon SageMaker Notebook Lifecycle

Open the AWS SageMaker console, go to the lifecycle section and use the below snippet to configure AWS SDK for pandas for all compatible SageMaker kernels ([Reference](#)).

```
#!/bin/bash

set -e

# OVERVIEW
# This script installs a single pip package in all SageMaker conda environments, apart
# from the JupyterSystemEnv which
# is a system environment reserved for Jupyter.
# Note this may timeout if the package installations in all environments take longer
# than 5 mins, consider using
# "nohup" to run this as a background process in that case.

sudo -u ec2-user -i <<'EOF'

# PARAMETERS
PACKAGE=aws wrangler

# Note that "base" is special environment name, include it there as well.
for env in base /home/ec2-user/anaconda3/envs/*; do
    source /home/ec2-user/anaconda3/bin/activate $(basename "$env")
    if [ $env = 'JupyterSystemEnv' ]; then
        continue
    fi
    nohup pip install --upgrade "$PACKAGE" &
    source /home/ec2-user/anaconda3/bin/deactivate
done
EOF
```

1.2.9 EMR Cluster

Despite not being a distributed library, AWS SDK for pandas could be used to complement Big Data pipelines.

- Configure Python 3 as the default interpreter for PySpark on your cluster configuration [ONLY REQUIRED FOR EMR < 6]

```
[
{
  "Classification": "spark-env",
  "Configurations": [
```

(continues on next page)

(continued from previous page)

```
{
  "Classification": "export",
  "Properties": {
    "PYSPARK_PYTHON": "/usr/bin/python3"
  }
}
]
```

- Keep the bootstrap script above on S3 and reference it on your cluster.
 - For EMR Release < 6

```
#!/usr/bin/env bash
set -ex

sudo pip-3.6 install pyarrow==2 awswrangler
```

- For EMR Release >= 6

```
#!/usr/bin/env bash
set -ex

sudo pip install pyarrow==2 awswrangler
```

Note: Make sure to freeze the library version in the bootstrap for production environments (e.g. `awswrangler==2.20.0`)

Note: Pyarrow 3 is not currently supported in the default EMR image, which is why an installation of pyarrow 2 is required.

1.2.10 From Source

```
>>> git clone https://github.com/aws/aws-sdk-pandas.git
>>> cd aws-sdk-pandas
>>> pip install .
```

1.2.11 Notes for Microsoft SQL Server

awswrangler uses `pyodbc` for interacting with Microsoft SQL Server. To install this package you need the ODBC header files, which can be installed, with the following commands:

```
>>> sudo apt install unixodbc-dev
>>> yum install unixODBC-devel
```

After installing these header files you can either just install `pyodbc` or `awswrangler` with the `sqlserver` extra, which will also install `pyodbc`:

```
>>> pip install pyodbc
>>> pip install awswrangler[sqlserver]
```

Finally you also need the correct ODBC Driver for SQL Server. You can have a look at the [documentation from Microsoft](#) to see how they can be installed in your environment.

If you want to connect to Microsoft SQL Server from AWS Lambda, you can build a separate Layer including the needed ODBC drivers and *pyodbc*.

If you maintain your own environment, you need to take care of the above steps. Because of this limitation usage in combination with Glue jobs is limited and you need to rely on the provided [functionality inside Glue itself](#).

1.2.12 Notes for Oracle Database

awswrangler is using the [oracledb](#) for interacting with Oracle Database. For installing this package you do not need the Oracle Client libraries unless you want to use the Thick mode. You can have a look at the [documentation from Oracle](#) to see how they can be installed in your environment.

After installing these client libraries you can either just install *oracledb* or *awswrangler* with the *oracle* extra, which will also install *oracledb*:

```
>>> pip install oracledb
>>> pip install awswrangler[oracle]
```

If you maintain your own environment, you need to take care of the above steps. Because of this limitation usage in combination with Glue jobs is limited and you need to rely on the provided [functionality inside Glue itself](#).

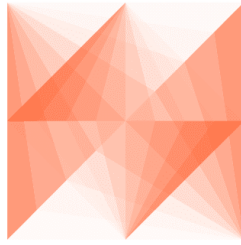
1.2.13 Notes for SPARQL support

To be able to use SPARQL either just install *SPARQLWrapper* or *awswrangler* with the *sparql* extra, which will also install *SPARQLWrapper*:

```
>>> pip install SPARQLWrapper
>>> pip install awswrangler[sparql]
```

1.3 Tutorials

Note: You can also find all Tutorial Notebooks on [GitHub](#).



AWS SDK for pandas

1.3.1 1 - Introduction

What is AWS SDK for pandas?

An [open-source](#) Python package that extends the power of [Pandas](#) library to AWS connecting **DataFrames** and AWS data related services (**Amazon Redshift**, **AWS Glue**, **Amazon Athena**, **Amazon Timestream**, **Amazon EMR**, etc).

Built on top of other open-source projects like [Pandas](#), [Apache Arrow](#) and [Boto3](#), it offers abstracted functions to execute usual ETL tasks like load/unload data from **Data Lakes**, **Data Warehouses** and **Databases**.

Check our [list of functionalities](#).

How to install?

awswrangler runs almost anywhere over Python 3.7, 3.8, 3.9 and 3.10, so there are several different ways to install it in the desired environment.

- [PyPi \(pip\)](#)
- [Conda](#)
- [AWS Lambda Layer](#)
- [AWS Glue Python Shell Jobs](#)
- [AWS Glue PySpark Jobs](#)
- [Amazon SageMaker Notebook](#)
- [Amazon SageMaker Notebook Lifecycle](#)
- [EMR Cluster](#)
- [From source](#)

Some good practices for most of the above methods are: - Use new and individual Virtual Environments for each project ([venv](#)) - On Notebooks, always restart your kernel after installations.

Let's Install it!

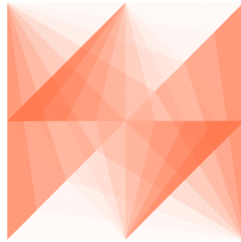
```
[ ]: !pip install awswrangler
```

Restart your kernel after the installation!

```
[1]: import awswrangler as wr
```

```
wr.__version__
```

```
[1]: '2.0.0'
```



AWS SDK for pandas

1.3.2 2 - Sessions

How awswrangler handles Sessions and AWS credentials?

After version 1.0.0 awswrangler relies on `Boto3.Session()` to manage AWS credentials and configurations.

awswrangler will not store any kind of state internally. Users are in charge of managing Sessions.

Most awswrangler functions receive the optional `boto3_session` argument. If `None` is received, the default `boto3` Session will be used.

```
[1]: import awswrangler as wr  
import boto3
```

Using the default Boto3 Session

```
[2]: wr.s3.does_object_exist("s3://noaa-ghcn-pds/fake")
```

```
[2]: False
```

Customizing and using the default Boto3 Session

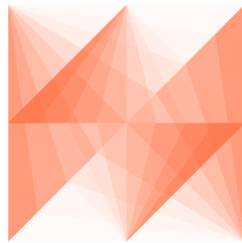
```
[3]: boto3.setup_default_session(region_name="us-east-2")  
  
wr.s3.does_object_exist("s3://noaa-ghcn-pds/fake")
```

```
[3]: False
```

Using a new custom Boto3 Session

```
[4]: my_session = boto3.Session(region_name="us-east-2")  
  
wr.s3.does_object_exist("s3://noaa-ghcn-pds/fake", boto3_session=my_session)
```

```
[4]: False
```



AWS SDK for pandas

1.3.3 3 - Amazon S3

Table of Contents

- 1. CSV files
 - 1.1 Writing CSV files
 - 1.2 Reading single CSV file
 - 1.3 Reading multiple CSV files
 - * 1.3.1 Reading CSV by list
 - * 1.3.2 Reading CSV by prefix
- 2. JSON files
 - 2.1 Writing JSON files
 - 2.2 Reading single JSON file
 - 2.3 Reading multiple JSON files
 - * 2.3.1 Reading JSON by list
 - * 2.3.2 Reading JSON by prefix
- 3. Parquet files

- 3.1 Writing Parquet files
- 3.2 Reading single Parquet file
- 3.3 Reading multiple Parquet files
 - * 3.3.1 Reading Parquet by list
 - * 3.3.2 Reading Parquet by prefix
- 4. Fixed-width formatted files (only read)
 - 4.1 Reading single FWF file
 - 4.2 Reading multiple FWF files
 - * 4.2.1 Reading FWF by list
 - * 4.2.2 Reading FWF by prefix
- 5. Excel files
 - 5.1 Writing Excel file
 - 5.2 Reading Excel file
- 6. Reading with lastModified filter
 - 6.1 Define the Date time with UTC Timezone
 - 6.2 Define the Date time and specify the Timezone
 - 6.3 Read json using the LastModified filters
- 7. Download Objects
 - 7.1 Download object to a file path
 - 7.2 Download object to a file-like object in binary mode
- 8. Upload Objects
 - 8.1 Upload object from a file path
 - 8.2 Upload object from a file-like object in binary mode
- 9. Delete objects

```
[1]: import awswrangler as wr
import pandas as pd
import boto3
import pytz
from datetime import datetime

df1 = pd.DataFrame({
    "id": [1, 2],
    "name": ["foo", "boo"]
})

df2 = pd.DataFrame({
    "id": [3],
    "name": ["bar"]
})
```

Enter your bucket name:

```
[2]: import getpass
      bucket = getpass.getpass()
```

1. CSV files

1.1 Writing CSV files

```
[3]: path1 = f"s3://{bucket}/csv/file1.csv"
      path2 = f"s3://{bucket}/csv/file2.csv"

      wr.s3.to_csv(df1, path1, index=False)
      wr.s3.to_csv(df2, path2, index=False)
```

1.2 Reading single CSV file

```
[4]: wr.s3.read_csv([path1])
```

```
[4]:   id name
0    1  foo
1    2  boo
```

1.3 Reading multiple CSV files

1.3.1 Reading CSV by list

```
[5]: wr.s3.read_csv([path1, path2])
```

```
[5]:   id name
0    1  foo
1    2  boo
2    3  bar
```

1.3.2 Reading CSV by prefix

```
[6]: wr.s3.read_csv(f"s3://{bucket}/csv/")
```

```
[6]:   id name
0    1  foo
1    2  boo
2    3  bar
```

2. JSON files

2.1 Writing JSON files

```
[7]: path1 = f"s3://{bucket}/json/file1.json"
      path2 = f"s3://{bucket}/json/file2.json"

      wr.s3.to_json(df1, path1)
      wr.s3.to_json(df2, path2)

[7]: ['s3://woodadw-test/json/file2.json']
```

2.2 Reading single JSON file

```
[8]: wr.s3.read_json([path1])

[8]:   id name
0    1  foo
1    2  boo
```

2.3 Reading multiple JSON files

2.3.1 Reading JSON by list

```
[9]: wr.s3.read_json([path1, path2])

[9]:   id name
0    1  foo
1    2  boo
0    3  bar
```

2.3.2 Reading JSON by prefix

```
[10]: wr.s3.read_json(f"s3://{bucket}/json/")

[10]:   id name
0    1  foo
1    2  boo
0    3  bar
```

3. Parquet files

For more complex features related to Parquet Dataset check the tutorial number 4.

3.1 Writing Parquet files

```
[11]: path1 = f"s3://{bucket}/parquet/file1.parquet"
      path2 = f"s3://{bucket}/parquet/file2.parquet"

      wr.s3.to_parquet(df1, path1)
      wr.s3.to_parquet(df2, path2)
```

3.2 Reading single Parquet file

```
[12]: wr.s3.read_parquet([path1])
```

```
[12]:   id name
0    1  foo
1    2  boo
```

3.3 Reading multiple Parquet files

3.3.1 Reading Parquet by list

```
[13]: wr.s3.read_parquet([path1, path2])
```

```
[13]:   id name
0    1  foo
1    2  boo
2    3  bar
```

3.3.2 Reading Parquet by prefix

```
[14]: wr.s3.read_parquet(f"s3://{bucket}/parquet/")
```

```
[14]:   id name
0    1  foo
1    2  boo
2    3  bar
```

4. Fixed-width formatted files (only read)

As of today, Pandas doesn't implement a `to_fwf` functionality, so let's manually write two files:

```
[15]: content = "1  Herfelingen 27-12-18\n"\
               "2   Lambusart 14-06-18\n"\
               "3 Spormaggiore 15-04-18"
boto3.client("s3").put_object(Body=content, Bucket=bucket, Key="fwf/file1.txt")

content = "4   Buizingen 05-09-19\n"\
          "5   San Rafael 04-09-19"
boto3.client("s3").put_object(Body=content, Bucket=bucket, Key="fwf/file2.txt")

path1 = f"s3://{bucket}/fwf/file1.txt"
path2 = f"s3://{bucket}/fwf/file2.txt"
```

4.1 Reading single FWF file

```
[16]: wr.s3.read_fwf([path1], names=["id", "name", "date"])
```

```
[16]:   id      name      date
0    1  Herfelingen 27-12-18
1    2   Lambusart 14-06-18
2    3 Spormaggiore 15-04-18
```

4.2 Reading multiple FWF files

4.2.1 Reading FWF by list

```
[17]: wr.s3.read_fwf([path1, path2], names=["id", "name", "date"])
```

```
[17]:   id      name      date
0    1  Herfelingen 27-12-18
1    2   Lambusart 14-06-18
2    3 Spormaggiore 15-04-18
3    4   Buizingen 05-09-19
4    5   San Rafael 04-09-19
```

4.2.2 Reading FWF by prefix

```
[18]: wr.s3.read_fwf(f"s3://{bucket}/fwf/", names=["id", "name", "date"])
```

```
[18]:   id      name      date
0    1  Herfelingen 27-12-18
1    2   Lambusart 14-06-18
2    3 Spormaggiore 15-04-18
3    4   Buizingen 05-09-19
4    5   San Rafael 04-09-19
```

5. Excel files

5.1 Writing Excel file

```
[19]: path = f"s3://{bucket}/file0.xlsx"

      wr.s3.to_excel(df1, path, index=False)

[19]: 's3://woodadw-test/file0.xlsx'
```

5.2 Reading Excel file

```
[20]: wr.s3.read_excel(path)

[20]:    id name
0    1  foo
1    2  boo
```

6. Reading with lastModified filter

Specify the filter by LastModified Date.

The filter needs to be specified as datetime with time zone

Internally the path needs to be listed, after that the filter is applied.

The filter compare the s3 content with the variables lastModified_begin and lastModified_end

<https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html>

6.1 Define the Date time with UTC Timezone

```
[21]: begin = datetime.strptime("20-07-31 20:30", "%y-%m-%d %H:%M")
      end = datetime.strptime("21-07-31 20:30", "%y-%m-%d %H:%M")

      begin_utc = pytz.utc.localize(begin)
      end_utc = pytz.utc.localize(end)
```

6.2 Define the Date time and specify the Timezone

```
[22]: begin = datetime.strptime("20-07-31 20:30", "%y-%m-%d %H:%M")
      end = datetime.strptime("21-07-31 20:30", "%y-%m-%d %H:%M")

      timezone = pytz.timezone("America/Los_Angeles")

      begin_Los_Angeles = timezone.localize(begin)
      end_Los_Angeles = timezone.localize(end)
```

6.3 Read json using the LastModified filters

```
[23]: wr.s3.read_fwf(f"s3://{bucket}/fwf/", names=["id", "name", "date"], last_modified_
      ↪begin=begin_utc, last_modified_end=end_utc)
wr.s3.read_json(f"s3://{bucket}/json/", last_modified_begin=begin_utc, last_modified_
      ↪end=end_utc)
wr.s3.read_csv(f"s3://{bucket}/csv/", last_modified_begin=begin_utc, last_modified_
      ↪end=end_utc)
wr.s3.read_parquet(f"s3://{bucket}/parquet/", last_modified_begin=begin_utc, last_
      ↪modified_end=end_utc)
```

7. Download objects

Objects can be downloaded from S3 using either a path to a local file or a file-like object in binary mode.

7.1 Download object to a file path

```
[24]: local_file_dir = getpass.getpass()
```

```
[25]: import os

path1 = f"s3://{bucket}/csv/file1.csv"
local_file = os.path.join(local_file_dir, "file1.csv")
wr.s3.download(path=path1, local_file=local_file)

pd.read_csv(local_file)
```

```
[25]:    id name
0     1  foo
1     2  boo
```

7.2 Download object to a file-like object in binary mode

```
[26]: path2 = f"s3://{bucket}/csv/file2.csv"
local_file = os.path.join(local_file_dir, "file2.csv")
with open(local_file, mode="wb") as local_f:
    wr.s3.download(path=path2, local_file=local_f)

pd.read_csv(local_file)
```

```
[26]:    id name
0     3  bar
```

8. Upload objects

Objects can be uploaded to S3 using either a path to a local file or a file-like object in binary mode.

8.1 Upload object from a file path

```
[27]: local_file = os.path.join(local_file_dir, "file1.csv")  
wr.s3.upload(local_file=local_file, path=path1)
```

```
wr.s3.read_csv(path1)
```

```
[27]:    id name  
0    1  foo  
1    2  boo
```

8.2 Upload object from a file-like object in binary mode

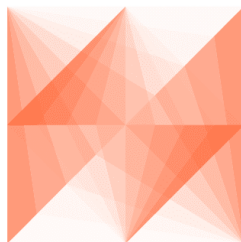
```
[28]: local_file = os.path.join(local_file_dir, "file2.csv")  
with open(local_file, "rb") as local_f:  
    wr.s3.upload(local_file=local_f, path=path2)
```

```
wr.s3.read_csv(path2)
```

```
[28]:    id name  
0    3  bar
```

9. Delete objects

```
[29]: wr.s3.delete_objects(f"s3://{bucket}/")
```



AWS SDK for pandas

1.3.4 4 - Parquet Datasets

aws wrangler has 3 different write modes to store Parquet Datasets on Amazon S3.

- **append** (Default)
Only adds new files without any delete.
- **overwrite**
Deletes everything in the target directory and then add new files. If writing new files fails for any reason, old files are *not* restored.
- **overwrite_partitions** (Partition Upsert)
Only deletes the paths of partitions that should be updated and then writes the new partitions files. It's like a "partition Upsert".

```
[1]: from datetime import date
import aws wrangler as wr
import pandas as pd
```

Enter your bucket name:

```
[2]: import getpass
bucket = getpass.getpass()
path = f"s3://{bucket}/dataset/"
.....
```

Creating the Dataset

```
[3]: df = pd.DataFrame({
    "id": [1, 2],
    "value": ["foo", "boo"],
    "date": [date(2020, 1, 1), date(2020, 1, 2)]
})

wr.s3.to_parquet(
    df=df,
    path=path,
    dataset=True,
    mode="overwrite"
)

wr.s3.read_parquet(path, dataset=True)
```

```
[3]:   id value      date
0    1  foo  2020-01-01
1    2  boo  2020-01-02
```

Appending

```
[4]: df = pd.DataFrame({
      "id": [3],
      "value": ["bar"],
      "date": [date(2020, 1, 3)]
    })

    wr.s3.to_parquet(
        df=df,
        path=path,
        dataset=True,
        mode="append"
    )

    wr.s3.read_parquet(path, dataset=True)
```

```
[4]:   id value      date
0    3   bar 2020-01-03
1    1   foo 2020-01-01
2    2   boo 2020-01-02
```

Overwriting

```
[5]: wr.s3.to_parquet(
      df=df,
      path=path,
      dataset=True,
      mode="overwrite"
    )

    wr.s3.read_parquet(path, dataset=True)
```

```
[5]:   id value      date
0    3   bar 2020-01-03
```

Creating a Partitioned Dataset

```
[6]: df = pd.DataFrame({
      "id": [1, 2],
      "value": ["foo", "boo"],
      "date": [date(2020, 1, 1), date(2020, 1, 2)]
    })

    wr.s3.to_parquet(
        df=df,
        path=path,
        dataset=True,
        mode="overwrite",
        partition_cols=["date"]
    )
```

(continues on next page)

(continued from previous page)

```
wr.s3.read_parquet(path, dataset=True)
```

```
[6]:
```

	id	value	date
0	1	foo	2020-01-01
1	2	boo	2020-01-02

Upserting partitions (overwrite_partitions)

```
[7]: df = pd.DataFrame({
      "id": [2, 3],
      "value": ["xoo", "bar"],
      "date": [date(2020, 1, 2), date(2020, 1, 3)]
    })

wr.s3.to_parquet(
    df=df,
    path=path,
    dataset=True,
    mode="overwrite_partitions",
    partition_cols=["date"]
)

wr.s3.read_parquet(path, dataset=True)
```

```
[7]:
```

	id	value	date
0	1	foo	2020-01-01
1	2	xoo	2020-01-02
2	3	bar	2020-01-03

BONUS - Glue/Athena integration

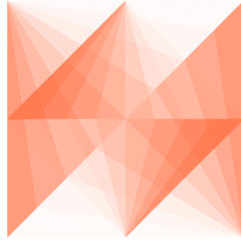
```
[8]: df = pd.DataFrame({
      "id": [1, 2],
      "value": ["foo", "boo"],
      "date": [date(2020, 1, 1), date(2020, 1, 2)]
    })

wr.s3.to_parquet(
    df=df,
    path=path,
    dataset=True,
    mode="overwrite",
    database="aws_sdk_pandas",
    table="my_table"
)

wr.athena.read_sql_query("SELECT * FROM my_table", database="aws_sdk_pandas")
```

```
[8]:
```

	id	value	date
0	1	foo	2020-01-01
1	2	boo	2020-01-02



AWS SDK for pandas

1.3.5 5 - Glue Catalog

`aws wrangler` makes heavy use of `Glue Catalog` to store metadata of tables and connections.

```
[1]: import awswrangler as wr
import pandas as pd
```

Enter your bucket name:

```
[2]: import getpass
bucket = getpass.getpass()
path = f"s3://{bucket}/data/"
```

.....

Creating a Pandas DataFrame

```
[3]: df = pd.DataFrame({
    "id": [1, 2, 3],
    "name": ["shoes", "tshirt", "ball"],
    "price": [50.3, 10.5, 20.0],
    "in_stock": [True, True, False]
})
df
```

```
[3]:
```

	id	name	price	in_stock
0	1	shoes	50.3	True
1	2	tshirt	10.5	True
2	3	ball	20.0	False

Checking Glue Catalog Databases

```
[4]: databases = wr.catalog.databases()
     print(databases)
```

	Database	Description
0	aws_sdk_pandas	AWS SDK for pandas Test Arena - Glue Database
1	default	Default Hive database

Create the database awswrangler_test if not exists

```
[5]: if "awswrangler_test" not in databases.values:
     wr.catalog.create_database("awswrangler_test")
     print(wr.catalog.databases())
     else:
     print("Database awswrangler_test already exists")
```

	Database	Description
0	aws_sdk_pandas	AWS SDK for pandas Test Arena - Glue Database
1	awswrangler_test	
2	default	Default Hive database

Checking the empty database

```
[6]: wr.catalog.tables(database="awswrangler_test")
```

```
[6]: Empty DataFrame
Columns: [Database, Table, Description, Columns, Partitions]
Index: []
```

Writing DataFrames to Data Lake (S3 + Parquet + Glue Catalog)

```
[7]: desc = "This is my product table."

param = {
    "source": "Product Web Service",
    "class": "e-commerce"
}

comments = {
    "id": "Unique product ID.",
    "name": "Product name",
    "price": "Product price (dollar)",
    "in_stock": "Is this product available in the stock?"
}

res = wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/products/",
```

(continues on next page)

(continued from previous page)

```

dataset=True,
database="awswrangler_test",
table="products",
mode="overwrite",
description=desc,
parameters=param,
columns_comments=comments
)

```

Checking Glue Catalog (AWS Console)

Tables > products

Last updated 18 Sep 2020 Table Version (Current version) ▾

[Edit table](#) [Delete table](#) [View properties](#) [Compare versions](#) [Edit schema](#)

Name	products
Description	This is my product table.
Database	awswrangler_test
Classification	parquet
Location	s3://igor-tavares/products/
Connection	
Deprecated	No
Last updated	Fri Sep 18 19:03:00 GMT-300 2020
Input format	org.apache.hadoop.hive.q1.io.parquet.MapredParquetInputFormat
Output format	org.apache.hadoop.hive.q1.io.parquet.MapredParquetOutputFormat
Serde serialization lib	org.apache.hadoop.hive.q1.io.parquet.serde.ParquetHiveSerDe
Serde parameters	serialization.format 1

Table properties

compressionType	snappy	source	Product Web Service	projection.enabled	false	class	e-commerce	typeOfData	file
-----------------	--------	--------	---------------------	--------------------	-------	-------	------------	------------	------

Schema

Column name	Data type	Partition key	Comment
1 id	bigint		Unique product ID.
2 name	string		Product name
3 price	double		Product price (dollar)
4 in_stock	boolean		Is this product available in the stock?

Showing: 1 - 4 of 4 < >

Looking Up for the new table!

```
[8]: wr.catalog.tables(name_contains="roduc")
```

```

[8]:      Database      Table      Description \
0  awswrangler_test  products  This is my product table.

      Columns Partitions
0  id, name, price, in_stock

```

```
[9]: wr.catalog.tables(name_prefix="pro")
```

```

[9]:      Database      Table      Description \
0  awswrangler_test  products  This is my product table.

      Columns Partitions
0  id, name, price, in_stock

```

```
[10]: wr.catalog.tables(name_suffix="ts")
```

```
[10]:      Database      Table      Description \
0  awswrangler_test  products  This is my product table.

      Columns Partitions
0  id, name, price, in_stock
```

```
[11]: wr.catalog.tables(search_text="This is my")
```

```
[11]:      Database      Table      Description \
0  awswrangler_test  products  This is my product table.

      Columns Partitions
0  id, name, price, in_stock
```

Getting tables details

```
[12]: wr.catalog.table(database="awswrangler_test", table="products")
```

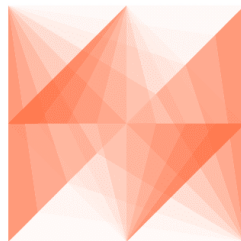
```
[12]:  Column Name      Type  Partition      Comment
0      id      bigint      False      Unique product ID.
1     name      string      False      Product name
2    price      double      False      Product price (dollar)
3  in_stock  boolean      False  Is this product available in the stock?
```

Cleaning Up the Database

```
[13]: for table in wr.catalog.get_tables(database="awswrangler_test"):
      wr.catalog.delete_table_if_exists(database="awswrangler_test", table=table["Name"])
```

Delete Database

```
[14]: wr.catalog.delete_database('awswrangler_test')
```



AWS SDK for pandas

1.3.6 6 - Amazon Athena

`aws wrangler` has three ways to run queries on Athena and fetch the result as a `DataFrame`:

- **`ctas_approach=True`** (Default)

Wraps the query with a CTAS and then reads the table data as parquet directly from s3.

- PROS:
 - * Faster for mid and big result sizes.
 - * Can handle some level of nested types.
- CONS:
 - * Requires create/delete table permissions on Glue.
 - * Does not support timestamp with time zone
 - * Does not support columns with repeated names.
 - * Does not support columns with undefined data types.
 - * A temporary table will be created and then deleted immediately.
 - * Does not support custom `data_source/catalog_id`.

- **`unload_approach=True` and `ctas_approach=False`**

Does an UNLOAD query on Athena and parse the Parquet result on s3.

- PROS:
 - * Faster for mid and big result sizes.
 - * Can handle some level of nested types.
 - * Does not modify Glue Data Catalog.
- CONS:
 - * Output S3 path must be empty.
 - * Does not support timestamp with time zone
 - * Does not support columns with repeated names.
 - * Does not support columns with undefined data types.

- **`ctas_approach=False`**

Does a regular query on Athena and parse the regular CSV result on s3.

- PROS:
 - * Faster for small result sizes (less latency).
 - * Does not require create/delete table permissions on Glue
 - * Supports timestamp with time zone.
 - * Support custom `data_source/catalog_id`.
- CONS:
 - * Slower (But stills faster than other libraries that uses the regular Athena API)
 - * Does not handle nested types at all.


```
[1]: import awswrangler as wr
```

Enter your bucket name:

```
[2]: import getpass
bucket = getpass.getpass()
path = f"s3://{bucket}/data/"
```

Checking/Creating Glue Catalog Databases

```
[3]: if "awswrangler_test" not in wr.catalog.databases().values:
      wr.catalog.create_database("awswrangler_test")
```

Creating a Parquet Table from the NOAA's CSV files

Reference

```
[ ]: cols = ["id", "dt", "element", "value", "m_flag", "q_flag", "s_flag", "obs_time"]

df = wr.s3.read_csv(
    path="s3://noaa-ghcn-pds/csv/by_year/189",
    names=cols,
    parse_dates=["dt", "obs_time"]) # Read 10 files from the 1890 decade (~1GB)

df
```

```
[ ]: wr.s3.to_parquet(
    df=df,
    path=path,
    dataset=True,
    mode="overwrite",
    database="awswrangler_test",
    table="noaa"
)
```

```
[ ]: wr.catalog.table(database="awswrangler_test", table="noaa")
```

Reading with ctas_approach=False

```
[ ]: %%time

wr.athena.read_sql_query("SELECT * FROM noaa", database="awswrangler_test", ctas_
    ↪approach=False)
```

Default with ctas_approach=True - 13x faster (default)

```
[ ]: %%time

wr.athena.read_sql_query("SELECT * FROM noaa", database="aws wrangler_test")
```

Using categories to speed up and save memory - 24x faster

```
[ ]: %%time

wr.athena.read_sql_query("SELECT * FROM noaa", database="aws wrangler_test", categories=[
    ↪ "id", "dt", "element", "value", "m_flag", "q_flag", "s_flag", "obs_time"])
```

Reading with unload_approach=True

```
[ ]: %%time

wr.athena.read_sql_query("SELECT * FROM noaa", database="aws wrangler_test", ctas_
    ↪ approach=False, unload_approach=True, s3_output=f"s3://{bucket}/unload/")
```

Batching (Good for restricted memory environments)

```
[ ]: %%time

dfs = wr.athena.read_sql_query(
    "SELECT * FROM noaa",
    database="aws wrangler_test",
    chunksize=True # Chunksize calculated automatically for ctas_approach.
)

for df in dfs: # Batching
    print(len(df.index))
```

```
[ ]: %%time

dfs = wr.athena.read_sql_query(
    "SELECT * FROM noaa",
    database="aws wrangler_test",
    chunksize=100_000_000
)

for df in dfs: # Batching
    print(len(df.index))
```

Cleaning Up S3

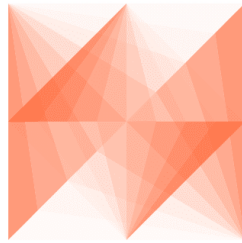
```
[ ]: wr.s3.delete_objects(path)
```

Delete table

```
[ ]: wr.catalog.delete_table_if_exists(database="awswrangler_test", table="noaa")
```

Delete Database

```
[ ]: wr.catalog.delete_database('awswrangler_test')
```



AWS SDK for pandas

1.3.7 7 - Redshift, MySQL, PostgreSQL, SQL Server and Oracle

awswrangler's Redshift, MySQL and PostgreSQL have two basic functions in common that try to follow Pandas conventions, but add more data type consistency.

- wr.redshift.to_sql()
- wr.redshift.read_sql_query()
- wr.mysql.to_sql()
- wr.mysql.read_sql_query()
- wr.postgresql.to_sql()
- wr.postgresql.read_sql_query()
- wr.sqlserver.to_sql()
- wr.sqlserver.read_sql_query()
- wr.oracle.to_sql()
- wr.oracle.read_sql_query()

```
[1]: import awswrangler as wr
import pandas as pd
```

```
df = pd.DataFrame({
```

(continues on next page)

(continued from previous page)

```

    "id": [1, 2],
    "name": ["foo", "boo"]
})

```

Connect using the Glue Catalog Connections

- `wr.redshift.connect()`
- `wr.mysql.connect()`
- `wr.postgresql.connect()`
- `wr.sqlserver.connect()`
- `wr.oracle.connect()`

```

[2]: con_redshift = wr.redshift.connect("aws-sdk-pandas-redshift")
    con_mysql = wr.mysql.connect("aws-sdk-pandas-mysql")
    con_postgresql = wr.postgresql.connect("aws-sdk-pandas-postgresql")
    con_sqlserver = wr.sqlserver.connect("aws-sdk-pandas-sqlserver")
    con_oracle = wr.oracle.connect("aws-sdk-pandas-oracle")

```

Raw SQL queries (No Pandas)

```

[3]: with con_redshift.cursor() as cursor:
    for row in cursor.execute("SELECT 1"):
        print(row)

```

```

[1]

```

Loading data to Database

```

[4]: wr.redshift.to_sql(df, con_redshift, schema="public", table="tutorial", mode="overwrite")
    wr.mysql.to_sql(df, con_mysql, schema="test", table="tutorial", mode="overwrite")
    wr.postgresql.to_sql(df, con_postgresql, schema="public", table="tutorial", mode=
    ↪ "overwrite")
    wr.sqlserver.to_sql(df, con_sqlserver, schema="dbo", table="tutorial", mode="overwrite")
    wr.oracle.to_sql(df, con_oracle, schema="test", table="tutorial", mode="overwrite")

```

Unloading data from Database

```

[5]: wr.redshift.read_sql_query("SELECT * FROM public.tutorial", con=con_redshift)
    wr.mysql.read_sql_query("SELECT * FROM test.tutorial", con=con_mysql)
    wr.postgresql.read_sql_query("SELECT * FROM public.tutorial", con=con_postgresql)
    wr.sqlserver.read_sql_query("SELECT * FROM dbo.tutorial", con=con_sqlserver)
    wr.oracle.read_sql_query("SELECT * FROM test.tutorial", con=con_oracle)

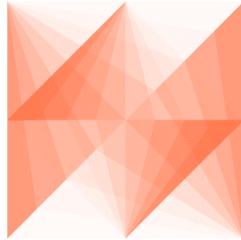
```

```

[5]:   id name
0    1  foo
1    2  boo

```

```
[6]: con_redshift.close()
      con_mysql.close()
      con_postgresql.close()
      con_sqlserver.close()
      con_oracle.close()
```



AWS SDK for pandas

1.3.8 8 - Redshift - COPY & UNLOAD

Amazon Redshift has two SQL command that help to load and unload large amount of data staging it on Amazon S3:

1 - COPY

2 - UNLOAD

Let's take a look and how awswrangler can use it.

```
[1]: import awswrangler as wr

      con = wr.redshift.connect("aws-sdk-pandas-redshift")
```

Enter your bucket name:

```
[2]: import getpass
      bucket = getpass.getpass()
      path = f"s3://{bucket}/stage/"
```

.....

Enter your IAM ROLE ARN:

```
[3]: iam_role = getpass.getpass()
```

.....

Creating a Dataframe from the NOAA's CSV files

Reference

```
[4]: cols = ["id", "dt", "element", "value", "m_flag", "q_flag", "s_flag", "obs_time"]
```

```
df = wr.s3.read_csv(
    path="s3://noaa-ghcn-pds/csv/by_year/1897.csv",
    names=cols,
    parse_dates=["dt", "obs_time"]) # ~127MB, ~4MM rows
```

```
df
```

```
[4]:
```

	id	dt	element	value	m_flag	q_flag	s_flag	obs_time
0	AG000060590	1897-01-01	TMAX	170	NaN	NaN	E	NaN
1	AG000060590	1897-01-01	TMIN	-14	NaN	NaN	E	NaN
2	AG000060590	1897-01-01	PRCP	0	NaN	NaN	E	NaN
3	AGE00135039	1897-01-01	TMAX	140	NaN	NaN	E	NaN
4	AGE00135039	1897-01-01	TMIN	40	NaN	NaN	E	NaN
...
3923594	UZM00038457	1897-12-31	TMIN	-145	NaN	NaN	r	NaN
3923595	UZM00038457	1897-12-31	PRCP	4	NaN	NaN	r	NaN
3923596	UZM00038457	1897-12-31	TAVG	-95	NaN	NaN	r	NaN
3923597	UZM00038618	1897-12-31	PRCP	66	NaN	NaN	r	NaN
3923598	UZM00038618	1897-12-31	TAVG	-45	NaN	NaN	r	NaN

```
[3923599 rows x 8 columns]
```

Load and Unload with COPY and UNLOAD commands

Note: Please use a empty S3 path for the COPY command.

```
[5]: %%time
```

```
wr.redshift.copy(
    df=df,
    path=path,
    con=con,
    schema="public",
    table="commands",
    mode="overwrite",
    iam_role=iam_role,
)
```

```
CPU times: user 2.78 s, sys: 293 ms, total: 3.08 s
Wall time: 20.7 s
```

```
[6]: %%time
```

```
wr.redshift.unload(
    sql="SELECT * FROM public.commands",
    con=con,
    iam_role=iam_role,
```

(continues on next page)

(continued from previous page)

```

    path=path,
    keep_files=True,
)

```

```

CPU times: user 10 s, sys: 1.14 s, total: 11.2 s
Wall time: 27.5 s

```

```

[6]:
      id      dt element  value m_flag q_flag s_flag obs_time
0  AG000060590 1897-01-01  TMAX    170  <NA>  <NA>      E  <NA>
1  AG000060590 1897-01-01  PRCP      0  <NA>  <NA>      E  <NA>
2  AGE00135039 1897-01-01  TMIN     40  <NA>  <NA>      E  <NA>
3  AGE00147705 1897-01-01  TMAX    164  <NA>  <NA>      E  <NA>
4  AGE00147705 1897-01-01  PRCP      0  <NA>  <NA>      E  <NA>
...
3923594 USW00094967 1897-12-31  TMAX   -144  <NA>  <NA>      6  <NA>
3923595 USW00094967 1897-12-31  PRCP      0      P  <NA>      6  <NA>
3923596 UZM00038457 1897-12-31  TMAX   -49  <NA>  <NA>      r  <NA>
3923597 UZM00038457 1897-12-31  PRCP      4  <NA>  <NA>      r  <NA>
3923598 UZM00038618 1897-12-31  PRCP     66  <NA>  <NA>      r  <NA>

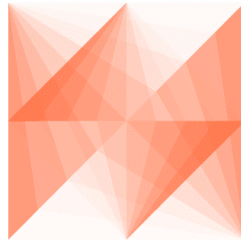
[7847198 rows x 8 columns]

```

```

[7]: con.close()

```



AWS SDK for pandas

1.3.9 9 - Redshift - Append, Overwrite and Upsert

aws wrangler's `copy/to_sql` function has three different mode options for Redshift.

- 1 - append
- 2 - overwrite
- 3 - upsert

```

[2]: import awswrangler as wr
import pandas as pd
from datetime import date

con = wr.redshift.connect("aws-sdk-pandas-redshift")

```

Enter your bucket name:

```
[3]: import getpass
      bucket = getpass.getpass()
      path = f"s3://{bucket}/stage/"
```

```
.....
```

Enter your IAM ROLE ARN:

```
[4]: iam_role = getpass.getpass()
```

```
.....
```

Creating the table (Overwriting if it exists)

```
[10]: df = pd.DataFrame({
      "id": [1, 2],
      "value": ["foo", "boo"],
      "date": [date(2020, 1, 1), date(2020, 1, 2)]
      })

      wr.redshift.copy(
          df=df,
          path=path,
          con=con,
          schema="public",
          table="my_table",
          mode="overwrite",
          iam_role=iam_role,
          primary_keys=["id"]
      )

      wr.redshift.read_sql_table(table="my_table", schema="public", con=con)
```

```
[10]:   id value      date
      0   2   boo  2020-01-02
      1   1   foo  2020-01-01
```

Appending

```
[11]: df = pd.DataFrame({
      "id": [3],
      "value": ["bar"],
      "date": [date(2020, 1, 3)]
      })

      wr.redshift.copy(
          df=df,
```

(continues on next page)

(continued from previous page)

```

    path=path,
    con=con,
    schema="public",
    table="my_table",
    mode="append",
    iam_role=iam_role,
    primary_keys=["id"]
)

wr.redshift.read_sql_table(table="my_table", schema="public", con=con)

```

```

[11]:
   id value      date
0    1  foo  2020-01-01
1    2  boo  2020-01-02
2    3  bar  2020-01-03

```

Upserting

```

[12]: df = pd.DataFrame({
        "id": [2, 3],
        "value": ["xoo", "bar"],
        "date": [date(2020, 1, 2), date(2020, 1, 3)]
    })

wr.redshift.copy(
    df=df,
    path=path,
    con=con,
    schema="public",
    table="my_table",
    mode="upsert",
    iam_role=iam_role,
    primary_keys=["id"]
)

wr.redshift.read_sql_table(table="my_table", schema="public", con=con)

```

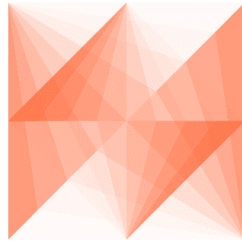
```

[12]:
   id value      date
0    1  foo  2020-01-01
1    2  xoo  2020-01-02
2    3  bar  2020-01-03

```

Cleaning Up

```
[13]: with con.cursor() as cursor:
        cursor.execute("DROP TABLE public.my_table")
        con.close()
```



AWS SDK for pandas

1.3.10 10 - Parquet Crawler

`awswrangler` can extract only the metadata from Parquet files and Partitions and then add it to the Glue Catalog.

```
[1]: import awswrangler as wr
```

Enter your bucket name:

```
[2]: import getpass
        bucket = getpass.getpass()
        path = f"s3://{bucket}/data/"
        .....
```

Creating a Parquet Table from the NOAA's CSV files

Reference

```
[3]: cols = ["id", "dt", "element", "value", "m_flag", "q_flag", "s_flag", "obs_time"]

        df = wr.s3.read_csv(
            path="s3://noaa-ghcn-pds/csv/by_year/189",
            names=cols,
            parse_dates=["dt", "obs_time"]) # Read 10 files from the 1890 decade (~1GB)

        df
```

	id	dt	element	value	m_flag	q_flag	s_flag	obs_time
0	AGE00135039	1890-01-01	TMAX	160	NaN	NaN	E	NaN
1	AGE00135039	1890-01-01	TMIN	30	NaN	NaN	E	NaN
2	AGE00135039	1890-01-01	PRCP	45	NaN	NaN	E	NaN

(continues on next page)

(continued from previous page)

```

3      AGE00147705 1890-01-01  TMAX    140    NaN    NaN    E    NaN
4      AGE00147705 1890-01-01  TMIN     74    NaN    NaN    E    NaN
...      ...      ...      ...      ...      ...      ...      ...
29249753 UZM00038457 1899-12-31  PRCP     16    NaN    NaN    r    NaN
29249754 UZM00038457 1899-12-31  TAVG    -73    NaN    NaN    r    NaN
29249755 UZM00038618 1899-12-31  TMIN    -76    NaN    NaN    r    NaN
29249756 UZM00038618 1899-12-31  PRCP     0    NaN    NaN    r    NaN
29249757 UZM00038618 1899-12-31  TAVG   -60    NaN    NaN    r    NaN

```

[29249758 rows x 8 columns]

```
[4]: df["year"] = df["dt"].dt.year
```

```
df.head(3)
```

```

[4]:      id      dt element  value  m_flag  q_flag  s_flag  obs_time  year
0  AGE00135039 1890-01-01  TMAX    160    NaN    NaN    E    NaN  1890
1  AGE00135039 1890-01-01  TMIN     30    NaN    NaN    E    NaN  1890
2  AGE00135039 1890-01-01  PRCP     45    NaN    NaN    E    NaN  1890

```

```

[5]: res = wr.s3.to_parquet(
      df=df,
      path=path,
      dataset=True,
      mode="overwrite",
      partition_cols=["year"],
    )

```

```
[6]: [ x.split("data/", 1)[1] for x in wr.s3.list_objects(path)]
```

```

[6]: ['year=1890/06a519afc8e48c9b08c8908f30adcfe.snappy.parquet',
      'year=1891/5a99c28dbef54008bfc770c946099e02.snappy.parquet',
      'year=1892/9b1ea5d1cfad40f78c920f93540ca8ec.snappy.parquet',
      'year=1893/92259b49c134401eaf772506ee802af6.snappy.parquet',
      'year=1894/c734469ffff944f69dc277c630064a16.snappy.parquet',
      'year=1895/cf7ccde86aaf4d138f86c379c0817aa6.snappy.parquet',
      'year=1896/ce02f4c2c554438786b766b33db451b6.snappy.parquet',
      'year=1897/e04de04ad3c444deadcc9c410ab97ca1.snappy.parquet',
      'year=1898/acb0e02878f04b56a6200f4b5a97be0e.snappy.parquet',
      'year=1899/a269bdbb0f6a48faac55f3bcfef7df7a.snappy.parquet']

```

Crawling!

```

[7]: %%time

res = wr.s3.store_parquet_metadata(
      path=path,
      database="awswrangler_test",
      table="crawler",
      dataset=True,
      mode="overwrite",

```

(continues on next page)

(continued from previous page)

```
dtype={"year": "int"}
)
```

```
CPU times: user 1.81 s, sys: 528 ms, total: 2.33 s
Wall time: 3.21 s
```

Checking

```
[8]: wr.catalog.table(database="awswrangler_test", table="crawler")
```

```
[8]:
```

	Column Name	Type	Partition	Comment
0	id	string	False	
1	dt	timestamp	False	
2	element	string	False	
3	value	bigint	False	
4	m_flag	string	False	
5	q_flag	string	False	
6	s_flag	string	False	
7	obs_time	string	False	
8	year	int	True	

```
[9]: %%time
```

```
wr.athena.read_sql_query("SELECT * FROM crawler WHERE year=1890", database="awswrangler_
↪test")
```

```
CPU times: user 3.52 s, sys: 811 ms, total: 4.33 s
Wall time: 9.6 s
```

```
[9]:
```

	id	dt	element	value	m_flag	q_flag	s_flag	obs_time	\
0	USC00195145	1890-01-01	TMIN	-28	<NA>	<NA>	6	<NA>	
1	USC00196770	1890-01-01	PRCP	0	P	<NA>	6	<NA>	
2	USC00196770	1890-01-01	SNOW	0	<NA>	<NA>	6	<NA>	
3	USC00196915	1890-01-01	PRCP	0	P	<NA>	6	<NA>	
4	USC00196915	1890-01-01	SNOW	0	<NA>	<NA>	6	<NA>	
...	
6139	ASN00022006	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	
6140	ASN00022007	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	
6141	ASN00022008	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	
6142	ASN00022009	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	
6143	ASN00022011	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	
	year								
0	1890								
1	1890								
2	1890								
3	1890								
4	1890								
...	...								
6139	1890								
6140	1890								
6141	1890								

(continues on next page)

(continued from previous page)

```
6142  1890
6143  1890

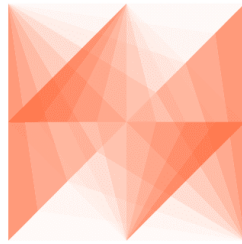
[1276246 rows x 9 columns]
```

Cleaning Up S3

```
[10]: wr.s3.delete_objects(path)
```

Cleaning Up the Database

```
[11]: for table in wr.catalog.get_tables(database="awswrangler_test"):
        wr.catalog.delete_table_if_exists(database="awswrangler_test", table=table["Name"])
```



AWS SDK for pandas

1.3.11 11 - CSV Datasets

awswrangler has 3 different write modes to store CSV Datasets on Amazon S3.

- **append** (Default)
Only adds new files without any delete.
- **overwrite**
Deletes everything in the target directory and then add new files.
- **overwrite_partitions** (Partition Upsert)
Only deletes the paths of partitions that should be updated and then writes the new partitions files. It's like a "partition Upsert".

```
[1]: from datetime import date
import awswrangler as wr
import pandas as pd
```

Enter your bucket name:

```
[2]: import getpass
      bucket = getpass.getpass()
      path = f"s3://{bucket}/dataset/"
      .....
```

Checking/Creating Glue Catalog Databases

```
[3]: if "awswrangler_test" not in wr.catalog.databases().values:
      wr.catalog.create_database("awswrangler_test")
```

Creating the Dataset

```
[4]: df = pd.DataFrame({
      "id": [1, 2],
      "value": ["foo", "boo"],
      "date": [date(2020, 1, 1), date(2020, 1, 2)]
    })

    wr.s3.to_csv(
        df=df,
        path=path,
        index=False,
        dataset=True,
        mode="overwrite",
        database="awswrangler_test",
        table="csv_dataset"
    )

    wr.athena.read_sql_table(database="awswrangler_test", table="csv_dataset")

[4]:   id value      date
0    1  foo  2020-01-01
1    2  boo  2020-01-02
```

Appending

```
[5]: df = pd.DataFrame({
      "id": [3],
      "value": ["bar"],
      "date": [date(2020, 1, 3)]
    })

    wr.s3.to_csv(
        df=df,
        path=path,
        index=False,
```

(continues on next page)

(continued from previous page)

```

dataset=True,
mode="append",
database="awswrangler_test",
table="csv_dataset"
)

wr.athena.read_sql_table(database="awswrangler_test", table="csv_dataset")

```

```

[5]:   id value      date
     0   3   bar  2020-01-03
     1   1   foo  2020-01-01
     2   2   boo  2020-01-02

```

Overwriting

```

[6]: wr.s3.to_csv(
      df=df,
      path=path,
      index=False,
      dataset=True,
      mode="overwrite",
      database="awswrangler_test",
      table="csv_dataset"
    )

wr.athena.read_sql_table(database="awswrangler_test", table="csv_dataset")

```

```

[6]:   id value      date
     0   3   bar  2020-01-03

```

Creating a Partitioned Dataset

```

[7]: df = pd.DataFrame({
      "id": [1, 2],
      "value": ["foo", "boo"],
      "date": [date(2020, 1, 1), date(2020, 1, 2)]
    })

wr.s3.to_csv(
    df=df,
    path=path,
    index=False,
    dataset=True,
    mode="overwrite",
    database="awswrangler_test",
    table="csv_dataset",
    partition_cols=["date"]
)

wr.athena.read_sql_table(database="awswrangler_test", table="csv_dataset")

```

```
[7]:
```

	id	value	date
0	2	boo	2020-01-02
1	1	foo	2020-01-01

Upserting partitions (overwrite_partitions)

```
[8]:
```

```
df = pd.DataFrame({
    "id": [2, 3],
    "value": ["xoo", "bar"],
    "date": [date(2020, 1, 2), date(2020, 1, 3)]
})

wr.s3.to_csv(
    df=df,
    path=path,
    index=False,
    dataset=True,
    mode="overwrite_partitions",
    database="awswrangler_test",
    table="csv_dataset",
    partition_cols=["date"]
)

wr.athena.read_sql_table(database="awswrangler_test", table="csv_dataset")
```

```
[8]:
```

	id	value	date
0	1	foo	2020-01-01
1	2	xoo	2020-01-02
0	3	bar	2020-01-03

BONUS - Glue/Athena integration

```
[9]:
```

```
df = pd.DataFrame({
    "id": [1, 2],
    "value": ["foo", "boo"],
    "date": [date(2020, 1, 1), date(2020, 1, 2)]
})

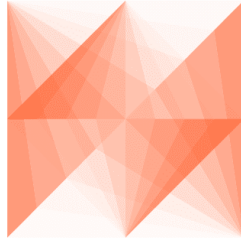
wr.s3.to_csv(
    df=df,
    path=path,
    dataset=True,
    index=False,
    mode="overwrite",
    database="aws_sdk_pandas",
    table="my_table",
    compression="gzip"
)

wr.athena.read_sql_query("SELECT * FROM my_table", database="aws_sdk_pandas")
```



```
[9]:
```

	id	value	date
0	1	foo	2020-01-01
1	2	boo	2020-01-02



AWS SDK for pandas

1.3.12 12 - CSV Crawler

`aws wrangler` can extract only the metadata from a Pandas DataFrame and then add it can be added to Glue Catalog as a table.

```
[1]: import awswrangler as wr
      from datetime import datetime
      import pandas as pd
```

Enter your bucket name:

```
[2]: import getpass
      bucket = getpass.getpass()
      path = f"s3://{bucket}/csv_crawler/"
      .....
```

Creating a Pandas DataFrame

```
[3]: ts = lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S.%f") # noqa
      dt = lambda x: datetime.strptime(x, "%Y-%m-%d").date() # noqa

      df = pd.DataFrame(
          {
              "id": [1, 2, 3],
              "string": ["foo", None, "boo"],
              "float": [1.0, None, 2.0],
              "date": [dt("2020-01-01"), None, dt("2020-01-02")],
              "timestamp": [ts("2020-01-01 00:00:00.0"), None, ts("2020-01-02 00:00:01.0")],
              "bool": [True, None, False],
              "par0": [1, 1, 2],
              "par1": ["a", "b", "b"],
```

(continues on next page)

(continued from previous page)

```

    }
)

df

```

```

[3]:
   id string  float      date      timestamp  bool  par0 par1
0   1   foo    1.0  2020-01-01  2020-01-01 00:00:00  True    1   a
1   2  None    NaN      None           NaT   None    1   b
2   3   boo    2.0  2020-01-02  2020-01-02 00:00:01  False   2   b

```

Extracting the metadata

```

[4]: columns_types, partitions_types = wr.catalog.extract_athena_types(
      df=df,
      file_format="csv",
      index=False,
      partition_cols=["par0", "par1"]
    )

```

```

[5]: columns_types

```

```

[5]: {'id': 'bigint',
      'string': 'string',
      'float': 'double',
      'date': 'date',
      'timestamp': 'timestamp',
      'bool': 'boolean'}

```

```

[6]: partitions_types

```

```

[6]: {'par0': 'bigint', 'par1': 'string'}

```

Creating the table

```

[7]: wr.catalog.create_csv_table(
      table="csv_crawler",
      database="awswrangler_test",
      path=path,
      partitions_types=partitions_types,
      columns_types=columns_types,
    )

```

Checking

```
[8]: wr.catalog.table(database="awswrangler_test", table="csv_crawler")
```

```
[8]:
```

	Column Name	Type	Partition	Comment
0	id	bigint	False	
1	string	string	False	
2	float	double	False	
3	date	date	False	
4	timestamp	timestamp	False	
5	bool	boolean	False	
6	par0	bigint	True	
7	par1	string	True	

We can still using the extracted metadata to ensure all data types consistence to new data

```
[9]: df = pd.DataFrame(
    {
        "id": [1],
        "string": ["1"],
        "float": [1],
        "date": [ts("2020-01-01 00:00:00.0")],
        "timestamp": [dt("2020-01-02")],
        "bool": [1],
        "par0": [1],
        "par1": ["a"],
    }
)

df
```

```
[9]:
```

	id	string	float	date	timestamp	bool	par0	par1
0	1	1	1	2020-01-01	2020-01-02	1	1	a

```
[10]: res = wr.s3.to_csv(
    df=df,
    path=path,
    index=False,
    dataset=True,
    database="awswrangler_test",
    table="csv_crawler",
    partition_cols=["par0", "par1"],
    dtype=columns_types
)
```

You can also extract the metadata directly from the Catalog if you want

```
[11]: dtype = wr.catalog.get_table_types(database="awswrangler_test", table="csv_crawler")
```

```
[12]: res = wr.s3.to_csv(  
    df=df,  
    path=path,  
    index=False,  
    dataset=True,  
    database="awswrangler_test",  
    table="csv_crawler",  
    partition_cols=["par0", "par1"],  
    dtype=dtype  
)
```

Checking out

```
[13]: df = wr.athena.read_sql_table(database="awswrangler_test", table="csv_crawler")
```

df

```
[13]:
```

	id	string	float	date	timestamp	bool	par0	par1
0	1	1	1.0	None	2020-01-02	True	1	a
1	1	1	1.0	None	2020-01-02	True	1	a

```
[14]: df.dtypes
```

```
[14]: id                Int64  
string              string  
float               float64  
date                object  
timestamp           datetime64[ns]  
bool                boolean  
par0                Int64  
par1                string  
dtype: object
```

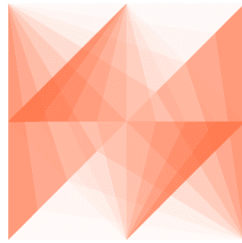
Cleaning Up S3

```
[15]: wr.s3.delete_objects(path)
```

Cleaning Up the Database

```
[16]: wr.catalog.delete_table_if_exists(database="awswrangler_test", table="csv_crawler")
```

```
[16]: True
```



AWS SDK for pandas

1.3.13 13 - Merging Datasets on S3

awswrangler has 3 different copy modes to store Parquet Datasets on Amazon S3.

- **append** (Default)
Only adds new files without any delete.
- **overwrite**
Deletes everything in the target directory and then add new files.
- **overwrite_partitions** (Partition Upsert)
Only deletes the paths of partitions that should be updated and then writes the new partitions files. It's like a "partition Upsert".

```
[1]: from datetime import date
import awswrangler as wr
import pandas as pd
```

Enter your bucket name:

```
[2]: import getpass
bucket = getpass.getpass()
path1 = f"s3://{bucket}/dataset1/"
path2 = f"s3://{bucket}/dataset2/"
.....
```

Creating Dataset 1

```
[3]: df = pd.DataFrame({
      "id": [1, 2],
      "value": ["foo", "boo"],
      "date": [date(2020, 1, 1), date(2020, 1, 2)]
    })

wr.s3.to_parquet(
    df=df,
    path=path1,
    dataset=True,
    mode="overwrite",
    partition_cols=["date"]
)

wr.s3.read_parquet(path1, dataset=True)
```

```
[3]:   id value      date
0    1  foo  2020-01-01
1    2  boo  2020-01-02
```

Creating Dataset 2

```
[4]: df = pd.DataFrame({
      "id": [2, 3],
      "value": ["xoo", "bar"],
      "date": [date(2020, 1, 2), date(2020, 1, 3)]
    })

dataset2_files = wr.s3.to_parquet(
    df=df,
    path=path2,
    dataset=True,
    mode="overwrite",
    partition_cols=["date"]
)["paths"]

wr.s3.read_parquet(path2, dataset=True)
```

```
[4]:   id value      date
0    2  xoo  2020-01-02
1    3  bar  2020-01-03
```

Merging (Dataset 2 -> Dataset 1) (APPEND)

```
[5]: wr.s3.merge_datasets(
      source_path=path2,
      target_path=path1,
      mode="append"
    )

wr.s3.read_parquet(path1, dataset=True)
```

```
[5]:   id value      date
0    1  foo  2020-01-01
1    2  xoo  2020-01-02
2    2  boo  2020-01-02
3    3  bar  2020-01-03
```

Merging (Dataset 2 -> Dataset 1) (OVERWRITE_PARTITIONS)

```
[6]: wr.s3.merge_datasets(
      source_path=path2,
      target_path=path1,
      mode="overwrite_partitions"
    )

wr.s3.read_parquet(path1, dataset=True)
```

```
[6]:   id value      date
0    1  foo  2020-01-01
1    2  xoo  2020-01-02
2    3  bar  2020-01-03
```

Merging (Dataset 2 -> Dataset 1) (OVERWRITE)

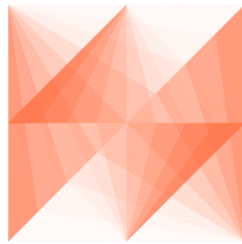
```
[7]: wr.s3.merge_datasets(
      source_path=path2,
      target_path=path1,
      mode="overwrite"
    )

wr.s3.read_parquet(path1, dataset=True)
```

```
[7]:   id value      date
0    2  xoo  2020-01-02
1    3  bar  2020-01-03
```

Cleaning Up

```
[8]: wr.s3.delete_objects(path1)
     wr.s3.delete_objects(path2)
```



AWS SDK for pandas

1.3.14 14 - Schema Evolution

aws wrangler supports new **columns** on Parquet and CSV datasets through:

- `wr.s3.to_parquet()`
- `wr.s3.store_parquet_metadata()` i.e. “Crawler”
- `wr.s3.to_csv()`

```
[1]: from datetime import date
     import aws wrangler as wr
     import pandas as pd
```

Enter your bucket name:

```
[2]: import getpass
     bucket = getpass.getpass()
     path = f"s3://{bucket}/dataset/"
```

.....

Creating the Dataset

Parquet

```
[3]: df = pd.DataFrame({
     "id": [1, 2],
     "value": ["foo", "boo"],
     })

     wr.s3.to_parquet(
         df=df,
```

(continues on next page)

(continued from previous page)

```
    path=path,  
    dataset=True,  
    mode="overwrite",  
    database="aws_sdk_pandas",  
    table="my_table"  
)  
  
wr.s3.read_parquet(path, dataset=True)
```

```
[3]:   id value  
0    1   foo  
1    2   boo
```

CSV

```
[ ]: df = pd.DataFrame({  
    "id": [1, 2],  
    "value": ["foo", "boo"],  
})  
  
wr.s3.to_csv(  
    df=df,  
    path=path,  
    dataset=True,  
    mode="overwrite",  
    database="aws_sdk_pandas",  
    table="my_table"  
)  
  
wr.s3.read_csv(path, dataset=True)
```

Schema Version 0 on Glue Catalog (AWS Console)

Tables > my_table

[Edit table](#) [Delete table](#)

Name my_table
Description
Database aws_data_wrangler
Classification parquet
Location [s3://\[REDACTED\]/dataset/](#)
Connection
Deprecated No
Last updated Tue May 19 19:07:03 GMT-300 2020
Input format org.apache.hadoop.hive.q1.io.parquet.MapredParquetInputFormat
Output format org.apache.hadoop.hive.q1.io.parquet.MapredParquetOutputFormat
Serde serialization lib org.apache.hadoop.hive.q1.io.parquet.serde.ParquetHiveSerDe

Serde parameters serialization.format 1**Table properties** compressionType **snappy** typeOfData **file**Last updated 19 May 2020 Table **Version (Current version)**

Showing: 1 - 1 < >

Version	Created:	Created by:
0	19 May 2020 7:0...	[REDACTED]

Schema

Showing: 1 - 2 of 2 < >

	Column name	Data type	Partition key	Comment
1	id	bigint		
2	value	string		

Appending with NEW COLUMNS

Parquet

```
[4]: df = pd.DataFrame({
    "id": [3, 4],
    "value": ["bar", None],
    "date": [date(2020, 1, 3), date(2020, 1, 4)],
    "flag": [True, False]
})

wr.s3.to_parquet(
    df=df,
    path=path,
    dataset=True,
    mode="append",
    database="aws_sdk_pandas",
    table="my_table",
    catalog_versioning=True # Optional
)

wr.s3.read_parquet(path, dataset=True, validate_schema=False)
```

```
[4]:   id value      date  flag
0    3   bar 2020-01-03  True
1    4  None 2020-01-04 False
```

(continues on next page)

(continued from previous page)

2	1	foo	NaN	NaN
3	2	boo	NaN	NaN

CSV

Note: for CSV datasets due to [column ordering](#), by default, schema evolution is disabled. Enable it by passing `schema_evolution=True` flag

```
[ ]: df = pd.DataFrame({
    "id": [3, 4],
    "value": ["bar", None],
    "date": [date(2020, 1, 3), date(2020, 1, 4)],
    "flag": [True, False]
})

wr.s3.to_csv(
    df=df,
    path=path,
    dataset=True,
    mode="append",
    database="aws_sdk_pandas",
    table="my_table",
    schema_evolution=True,
    catalog_versioning=True # Optional
)

wr.s3.read_csv(path, dataset=True, validate_schema=False)
```

Schema Version 1 on Glue Catalog (AWS Console)

Tables > my_table

Edit tableDelete table

Name

my_table

Description

aws_data_wrangler

Database

aws_data_wrangler

Classification

parquet

Location

s3://[REDACTED]dataset/

Connection

Deprecated

No

Last updated

Tue May 19 19:11:15 GMT-300 2020

Input format

org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat

Output format

org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat

Serde serialization lib

org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe

Serde parameters

serialization.format1

Table properties

compressionTypesnappytypeOfDatafile

Schema

	Column name	Data type	Partition key	Comment
1	id	bigint		
2	value	string		
3	date	date		
4	flag	boolean		

Last updated 19 May 2020

Table Version (Current version)

Showing: 1 - 2

Version	Created:	Created by:
1	19 May 2020 7:1...	
0	19 May 2020 7:1...	

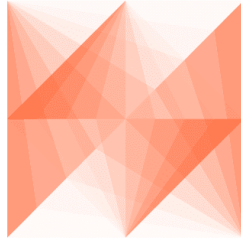
Reading from Athena

```
[5]: wr.athena.read_sql_table(table="my_table", database="aws_sdk_pandas")
[5]:   id value      date  flag
0   3  bar  2020-01-03  True
1   4  None 2020-01-04 False
2   1  foo      None  <NA>
3   2  boo      None  <NA>
```

Cleaning Up

```
[6]: wr.s3.delete_objects(path)
wr.catalog.delete_table_if_exists(table="my_table", database="aws_sdk_pandas")

[6]: True
```



AWS SDK for pandas

1.3.15 15 - EMR

```
[1]: import awswrangler as wr
import boto3
```

Enter your bucket name:

```
[2]: import getpass
bucket = getpass.getpass()
```

.....

Enter your Subnet ID:

```
[8]: subnet = getpass.getpass()
```

.....

Creating EMR Cluster

```
[9]: cluster_id = wr.emr.create_cluster(subnet)
```

Uploading our PySpark script to Amazon S3

```
[10]: script = """
      from pyspark.sql import SparkSession
      spark = SparkSession.builder.appName("docker-awswrangler").getOrCreate()
      sc = spark.sparkContext

      print("Spark Initialized")
      """

      _ = boto3.client("s3").put_object(
          Body=script,
          Bucket=bucket,
          Key="test.py"
      )
```

Submit PySpark step

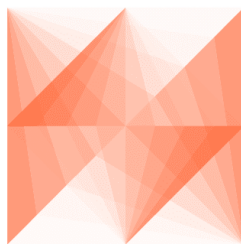
```
[11]: step_id = wr.emr.submit_step(cluster_id, command=f"spark-submit s3://{bucket}/test.py")
```

Wait Step

```
[12]: while wr.emr.get_step_state(cluster_id, step_id) != "COMPLETED":
      pass
```

Terminate Cluster

```
[13]: wr.emr.terminate_cluster(cluster_id)
```



AWS SDK for pandas

1.3.16 16 - EMR & Docker

```
[ ]: import awswrangler as wr
import boto3
import getpass
```

Enter your bucket name:

```
[2]: bucket = getpass.getpass()
```

```
.....
```

Enter your Subnet ID:

```
[3]: subnet = getpass.getpass()
```

```
.....
```

Build and Upload Docker Image to ECR repository

Replace the {ACCOUNT_ID} placeholder.

```
[ ]: %%writefile Dockerfile

FROM amazoncorretto:8

RUN yum -y update
RUN yum -y install yum-utils
RUN yum -y groupinstall development

RUN yum list python3*
RUN yum -y install python3 python3-dev python3-pip python3-virtualenv

RUN python -V
RUN python3 -V

ENV PYSPARK_DRIVER_PYTHON python3
ENV PYSPARK_PYTHON python3

RUN pip3 install --upgrade pip
RUN pip3 install awswrangler

RUN python3 -c "import awswrangler as wr"
```

```
[ ]: %%bash

docker build -t 'local/emr-wrangler' .
aws ecr create-repository --repository-name emr-wrangler
docker tag local/emr-wrangler {ACCOUNT_ID}.dkr.ecr.us-east-1.amazonaws.com/emr-wrangler:
```

(continues on next page)

(continued from previous page)

```
↪emr-wrangler
eval $(aws ecr get-login --region us-east-1 --no-include-email)
docker push {ACCOUNT_ID}.dkr.ecr.us-east-1.amazonaws.com/emr-wrangler:emr-wrangler
```

Creating EMR Cluster

```
[4]: cluster_id = wr.emr.create_cluster(subnet, docker=True)
```

Refresh ECR credentials in the cluster (expiration time: 12h)

```
[5]: wr.emr.submit_ecr_credentials_refresh(cluster_id, path=f"s3://{bucket}/")
```

```
[5]: 's-1B0045RWJL8CL'
```

Uploading application script to Amazon S3 (PySpark)

```
[7]: script = """
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("docker-awswrangler").getOrCreate()
sc = spark.sparkContext

print("Spark Initialized")

import awswrangler as wr

print(f"awswrangler version: {wr.__version__}")
"""

boto3.client("s3").put_object(Body=script, Bucket=bucket, Key="test_docker.py")
```

Submit PySpark step

```
[8]: DOCKER_IMAGE = f"{wr.get_account_id()}.dkr.ecr.us-east-1.amazonaws.com/emr-wrangler:emr-
↪wrangler"

step_id = wr.emr.submit_spark_step(
    cluster_id,
    f"s3://{bucket}/test_docker.py",
    docker_image=DOCKER_IMAGE
)
```


Wait Step

```
[ ]: while wr.emr.get_step_state(cluster_id, step_id) != "COMPLETED":
    pass
```

Terminate Cluster

```
[ ]: wr.emr.terminate_cluster(cluster_id)
```

Another example with custom configurations

```
[9]: cluster_id = wr.emr.create_cluster(
    cluster_name="my-demo-cluster-v2",
    logging_s3_path=f"s3://{bucket}/emr-logs/",
    emr_release="emr-6.7.0",
    subnet_id=subnet,
    emr_ec2_role="EMR_EC2_DefaultRole",
    emr_role="EMR_DefaultRole",
    instance_type_master="m5.2xlarge",
    instance_type_core="m5.2xlarge",
    instance_ebs_size_master=50,
    instance_ebs_size_core=50,
    instance_num_on_demand_master=0,
    instance_num_on_demand_core=0,
    instance_num_spot_master=1,
    instance_num_spot_core=2,
    spot_bid_percentage_of_on_demand_master=100,
    spot_bid_percentage_of_on_demand_core=100,
    spot_provisioning_timeout_master=5,
    spot_provisioning_timeout_core=5,
    spot_timeout_to_on_demand_master=False,
    spot_timeout_to_on_demand_core=False,
    python3=True,
    docker=True,
    spark_glue_catalog=True,
    hive_glue_catalog=True,
    presto_glue_catalog=True,
    debugging=True,
    applications=["Hadoop", "Spark", "Hive", "Zeppelin", "Livy"],
    visible_to_all_users=True,
    maximize_resource_allocation=True,
    keep_cluster_alive_when_no_steps=True,
    termination_protected=False,
    spark_pyarrow=True
)

wr.emr.submit_ecr_credentials_refresh(cluster_id, path=f"s3://{bucket}/emr/")

DOCKER_IMAGE = f"{wr.get_account_id()}.dkr.ecr.us-east-1.amazonaws.com/emr-wrangler:emr-
↪wrangler"
```

(continues on next page)

(continued from previous page)

```

step_id = wr.emr.submit_spark_step(
    cluster_id,
    f"s3://{bucket}/test_docker.py",
    docker_image=DOCKER_IMAGE
)

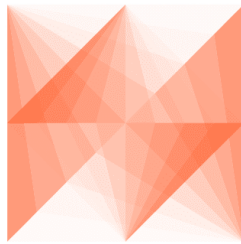
```

```

[ ]: while wr.emr.get_step_state(cluster_id, step_id) != "COMPLETED":
    pass

wr.emr.terminate_cluster(cluster_id)

```



AWS SDK for pandas

1.3.17 17 - Partition Projection

<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>

```

[1]: import awswrangler as wr
import pandas as pd
from datetime import datetime
import getpass

```

Enter your bucket name:

```

[2]: bucket = getpass.getpass()

```

.....

Integer projection

```

[3]: df = pd.DataFrame({
    "value": [1, 2, 3],
    "year": [2019, 2020, 2021],
    "month": [10, 11, 12],
    "day": [25, 26, 27]
})

```

(continues on next page)

(continued from previous page)

df

```
[3]:
```

	value	year	month	day
0	1	2019	10	25
1	2	2020	11	26
2	3	2021	12	27

```
[4]: wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/table_integer/",
    dataset=True,
    partition_cols=["year", "month", "day"],
    database="default",
    table="table_integer",
    projection_enabled=True,
    projection_types={
        "year": "integer",
        "month": "integer",
        "day": "integer"
    },
    projection_ranges={
        "year": "2000,2025",
        "month": "1,12",
        "day": "1,31"
    },
)
```

```
[5]: wr.athena.read_sql_query(f"SELECT * FROM table_integer", database="default")
```

```
[5]:
```

	value	year	month	day
0	3	2021	12	27
1	2	2020	11	26
2	1	2019	10	25

Enum projection

```
[6]: df = pd.DataFrame({
    "value": [1, 2, 3],
    "city": ["São Paulo", "Tokio", "Seattle"],
})
```

df

```
[6]:
```

	value	city
0	1	São Paulo
1	2	Tokio
2	3	Seattle

```
[7]: wr.s3.to_parquet(
    df=df,
```

(continues on next page)

(continued from previous page)

```

path=f"s3://{bucket}/table_enum/",
dataset=True,
partition_cols=["city"],
database="default",
table="table_enum",
projection_enabled=True,
projection_types={
    "city": "enum",
},
projection_values={
    "city": "São Paulo,Tokio,Seattle"
},
)

```

```
[8]: wr.athena.read_sql_query(f"SELECT * FROM table_enum", database="default")
```

```
[8]:
```

	value	city
0	1	São Paulo
1	3	Seattle
2	2	Tokio

Date projection

```
[9]: ts = lambda x: datetime.strptime(x, "%Y-%m-%d %H:%M:%S")
dt = lambda x: datetime.strptime(x, "%Y-%m-%d").date()

df = pd.DataFrame({
    "value": [1, 2, 3],
    "dt": [dt("2020-01-01"), dt("2020-01-02"), dt("2020-01-03")],
    "ts": [ts("2020-01-01 00:00:00"), ts("2020-01-01 00:00:01"), ts("2020-01-01 00:00:02
↪")],
})

```

df

```
[9]:
```

	value	dt	ts
0	1	2020-01-01	2020-01-01 00:00:00
1	2	2020-01-02	2020-01-01 00:00:01
2	3	2020-01-03	2020-01-01 00:00:02

```
[10]: wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/table_date/",
    dataset=True,
    partition_cols=["dt", "ts"],
    database="default",
    table="table_date",
    projection_enabled=True,
    projection_types={
        "dt": "date",
        "ts": "date",
    }
)

```

(continues on next page)

(continued from previous page)

```

    },
    projection_ranges={
        "dt": "2020-01-01,2020-01-03",
        "ts": "2020-01-01 00:00:00,2020-01-01 00:00:02"
    },
)

```

```
[11]: wr.athena.read_sql_query(f"SELECT * FROM table_date", database="default")
```

```
[11]:
```

	value	dt	ts
0	1	2020-01-01	2020-01-01 00:00:00
1	2	2020-01-02	2020-01-01 00:00:01
2	3	2020-01-03	2020-01-01 00:00:02

Injected projection

```
[12]: df = pd.DataFrame({
    "value": [1, 2, 3],
    "uuid": ["761e2488-a078-11ea-bb37-0242ac130002", "b89ed095-8179-4635-9537-88592c0f6bc3", "87adc586-ce88-4f0a-b1c8-bf8e00d32249"],
})

df
```

```
[12]:
```

	value	uuid
0	1	761e2488-a078-11ea-bb37-0242ac130002
1	2	b89ed095-8179-4635-9537-88592c0f6bc3
2	3	87adc586-ce88-4f0a-b1c8-bf8e00d32249

```
[13]: wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/table_injected/",
    dataset=True,
    partition_cols=["uuid"],
    database="default",
    table="table_injected",
    projection_enabled=True,
    projection_types={
        "uuid": "injected",
    }
)

```

```
[14]: wr.athena.read_sql_query(
    sql=f"SELECT * FROM table_injected WHERE uuid='b89ed095-8179-4635-9537-88592c0f6bc3'",
    database="default"
)

```

```
[14]:
```

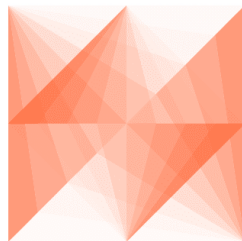
	value	uuid
0	2	b89ed095-8179-4635-9537-88592c0f6bc3

Cleaning Up

```
[15]: wr.s3.delete_objects(f"s3://{bucket}/table_integer/")
wr.s3.delete_objects(f"s3://{bucket}/table_enum/")
wr.s3.delete_objects(f"s3://{bucket}/table_date/")
wr.s3.delete_objects(f"s3://{bucket}/table_injected/")
```

```
[16]: wr.catalog.delete_table_if_exists(table="table_integer", database="default")
wr.catalog.delete_table_if_exists(table="table_enum", database="default")
wr.catalog.delete_table_if_exists(table="table_date", database="default")
wr.catalog.delete_table_if_exists(table="table_injected", database="default")
```

```
[ ]:
```



AWS SDK for pandas

1.3.18 18 - QuickSight

For this tutorial we will use the public AWS COVID-19 data lake.

References:

- [A public data lake for analysis of COVID-19 data](#)
- [Exploring the public AWS COVID-19 data lake](#)
- [CloudFormation template](#)

Please, install the CloudFormation template above to have access to the public data lake.

P.S. To be able to access the public data lake, you must allow explicitly QuickSight to access the related external bucket.

```
[1]: import awswrangler as wr
from time import sleep
```

List users of QuickSight account

```
[2]: [{"username": user["UserName"], "role": user["Role"]} for user in wr.quicksight.list_
↪users('default')]
```

```
[2]: [{'username': 'dev', 'role': 'ADMIN'}]
```

```
[3]: wr.catalog.databases()
```

```
[3]:
```

	Database	Description
0	aws_sdk_pandas	AWS SDK for pandas Test Arena - Glue Database
1	awsrangler_test	
2	covid-19	
3	default	Default Hive database

```
[4]: wr.catalog.tables(database="covid-19")
```

```
[4]:
```

	Database	Table \	Description \
0	covid-19	alleninstitute_comprehend_medical	Comprehend Medical results run against Allen I...
1	covid-19	alleninstitute_metadata	Metadata on papers pulled from the Allen Insti...
2	covid-19	country_codes	Lookup table for country codes
3	covid-19	county_populations	Lookup table for population for each county ba...
4	covid-19	covid_knowledge_graph_edges	AWS Knowledge Graph for COVID-19 data
5	covid-19	covid_knowledge_graph_nodes_author	AWS Knowledge Graph for COVID-19 data
6	covid-19	covid_knowledge_graph_nodes_concept	AWS Knowledge Graph for COVID-19 data
7	covid-19	covid_knowledge_graph_nodes_institution	AWS Knowledge Graph for COVID-19 data
8	covid-19	covid_knowledge_graph_nodes_paper	AWS Knowledge Graph for COVID-19 data
9	covid-19	covid_knowledge_graph_nodes_topic	AWS Knowledge Graph for COVID-19 data
10	covid-19	covid_testing_states_daily	USA total test daily trend by state. Sourced ...
11	covid-19	covid_testing_us_daily	USA total test daily trend. Sourced from covi...
12	covid-19	covid_testing_us_total	USA total tests. Sourced from covidtracking.c...
13	covid-19	covidcast_data	CMU Delphi's COVID-19 Surveillance Data
14	covid-19	covidcast_metadata	CMU Delphi's COVID-19 Surveillance Metadata
15	covid-19	enigma_jhu	Johns Hopkins University Consolidated data on ...
16	covid-19	enigma_jhu_timeseries	
17	covid-19	hospital_beds	
18	covid-19	nytimes_counties	
19	covid-19	nytimes_states	
20	covid-19	prediction_models_county_predictions	
21	covid-19	prediction_models_severity_index	
22	covid-19	tableau_covid_datahub	
23	covid-19	tableau_jhu	
24	covid-19	us_state_abbreviations	
25	covid-19	world_cases_deaths_testing	

(continues on next page)

(continued from previous page)

```

16 Johns Hopkins University data on COVID-19 case...
17 Data on hospital beds and their utilization in...
18 Data on COVID-19 cases from NY Times at US cou...
19 Data on COVID-19 cases from NY Times at US sta...
20 County-level Predictions Data. Sourced from Yu...
21 Severity Index models. Sourced from Yu Group a...
22 COVID-19 data that has been gathered and unifi...
23 Johns Hopkins University data on COVID-19 case...
24     Lookup table for US state abbreviations
25 Data on confirmed cases, deaths, and testing. ...

```

Columns Partitions

```

0 paper_id, date, dx_name, test_name, procedure...
1 cord_uid, sha, source_x, title, doi, pmcid, pu...
2 country, alpha-2 code, alpha-3 code, numeric c...
3 id, id2, county, state, population estimate 2018
4 id, label, from, to, score
5 id, label, first, last, full_name
6 id, label, entity, concept
7 id, label, institution, country, settlement
8 id, label, doi, sha_code, publish_time, source...
9 id, label, topic, topic_num
10 date, state, positive, negative, pending, hosp...
11 date, states, positive, negative, posneg, pend...
12 positive, negative, posneg, hospitalized, deat...
13 data_source, signal, geo_type, time_value, geo...
14 data_source, signal, time_type, geo_type, min...
15 fips, admin2, province_state, country_region, ...
16 uid, fips, iso2, iso3, code3, admin2, latitude...
17 objectid, hospital_name, hospital_type, hq_add...
18 date, county, state, fips, cases, deaths
19 date, state, fips, cases, deaths
20 countyfips, countyname, statename, severity_co...
21 severity_1-day, severity_2-day, severity_3-day...
22 country_short_name, country_alpha_3_code, coun...
23 case_type, cases, difference, date, country_re...
24 state, abbreviation
25 iso_code, location, date, total_cases, new_cas...

```

Create data source of QuickSight Note: data source stores the connection information.

```

[5]: wr.quicksight.create_athena_data_source(
      name="covid-19",
      workgroup="primary",
      allowed_to_manage=["dev"]
    )

```

```

[6]: wr.catalog.tables(database="covid-19", name_contains="nyt")

```

```

[6]: Database      Table \
0 covid-19  nytimes_counties
1 covid-19  nytimes_states

```

(continues on next page)

(continued from previous page)

```

                                Description \
0  Data on COVID-19 cases from NY Times at US cou...
1  Data on COVID-19 cases from NY Times at US sta...

                                Columns Partitions
0  date, county, state, fips, cases, deaths
1      date, state, fips, cases, deaths

```

```
[7]: wr.athena.read_sql_query("SELECT * FROM nytimes_counties limit 10", database="covid-19",
    ↪ ctas_approach=False)
```

```
[7]:
```

	date	county	state	fips	cases	deaths
0	2020-01-21	Snohomish	Washington	53061	1	0
1	2020-01-22	Snohomish	Washington	53061	1	0
2	2020-01-23	Snohomish	Washington	53061	1	0
3	2020-01-24	Cook	Illinois	17031	1	0
4	2020-01-24	Snohomish	Washington	53061	1	0
5	2020-01-25	Orange	California	06059	1	0
6	2020-01-25	Cook	Illinois	17031	1	0
7	2020-01-25	Snohomish	Washington	53061	1	0
8	2020-01-26	Maricopa	Arizona	04013	1	0
9	2020-01-26	Los Angeles	California	06037	1	0

```
[8]: sql = """
SELECT
    j.*,
    co.Population,
    co.county AS county2,
    hb.*
FROM
    (
        SELECT
            date,
            county,
            state,
            fips,
            cases as confirmed,
            deaths
        FROM "covid-19".nytimest_counties
    ) j
LEFT OUTER JOIN (
    SELECT
        DISTINCT county,
        state,
        "population estimate 2018" AS Population
    FROM
        "covid-19".county_populations
    WHERE
        state IN (
            SELECT
                DISTINCT state

```

(continues on next page)

(continued from previous page)

```

        FROM
            "covid-19".nytimes_counties
        )
        AND county IN (
            SELECT
                DISTINCT county as county
            FROM "covid-19".nytimes_counties
        )
    ) co ON co.county = j.county
    AND co.state = j.state
    LEFT OUTER JOIN (
        SELECT
            count(objectid) as Hospital,
            fips as hospital_fips,
            sum(num_licensed_beds) as licensed_beds,
            sum(num_staffed_beds) as staffed_beds,
            sum(num_icu_beds) as icu_beds,
            avg(bed_utilization) as bed_utilization,
            sum(
                potential_increase_in_bed_capac
            ) as potential_increase_bed_capacity
        FROM "covid-19".hospital_beds
        WHERE
            fips in (
                SELECT
                    DISTINCT fips
                FROM
                    "covid-19".nytimes_counties
            )
        GROUP BY
            2
    ) hb ON hb.hospital_fips = j.fips
    """

wr.athena.read_sql_query(sql, database="covid-19", ctas_approach=False)

```

```

[8]:
   date      county      state  fips  confirmed  deaths  population \
0  2020-04-12    Park    Montana  30067         7         0      16736
1  2020-04-12   Ravalli    Montana  30081         3         0      43172
2  2020-04-12 Silver Bow    Montana  30093        11         0      34993
3  2020-04-12     Clay  Nebraska  31035         2         0       6214
4  2020-04-12    Cuming  Nebraska  31039         2         0       8940
...      ...      ...      ...      ...      ...      ...
227684 2020-06-11   Hockley    Texas  48219        28         1      22980
227685 2020-06-11  Hudspeth    Texas  48229        11         0       4795
227686 2020-06-11     Jones    Texas  48253       633         0      19817
227687 2020-06-11  La Salle    Texas  48283         4         0       7531
227688 2020-06-11  Limestone    Texas  48293        36         1      23519

   county2  Hospital  hospital_fips  licensed_beds  staffed_beds \
0      Park         0         30067           25           25
1   Ravalli         0         30081           25           25

```

(continues on next page)

(continued from previous page)

2	Silver Bow	0	30093	98	71
3	Clay	<NA>	<NA>	<NA>	<NA>
4	Cuming	0	31039	25	25
...
227684	Hockley	0	48219	48	48
227685	Hudspeth	<NA>	<NA>	<NA>	<NA>
227686	Jones	0	48253	45	7
227687	La Salle	<NA>	<NA>	<NA>	<NA>
227688	Limestone	0	48293	78	69

	icu_beds	bed_utilization	potential_increase_bed_capacity
0	4	0.432548	0
1	5	0.567781	0
2	11	0.551457	27
3	<NA>	NaN	<NA>
4	4	0.204493	0
...
227684	8	0.120605	0
227685	<NA>	NaN	<NA>
227686	1	0.718591	38
227687	<NA>	NaN	<NA>
227688	9	0.163940	9

[227689 rows x 15 columns]

Create Dataset with custom SQL option

```
[9]: wr.quicksight.create_athena_dataset(
    name="covid19-nytimes-usa",
    sql=sql,
    sql_name='CustomSQL',
    data_source_name="covid-19",
    import_mode='SPICE',
    allowed_to_manage=["dev"]
)
```

```
[10]: ingestion_id = wr.quicksight.create_ingestion("covid19-nytimes-usa")
```

Wait ingestion

```
[11]: while wr.quicksight.describe_ingestion(ingestion_id=ingestion_id, dataset_name="covid19-
    nytimes-usa")["IngestionStatus"] not in ["COMPLETED", "FAILED"]:
    sleep(1)
```

Describe last ingestion

```
[12]: wr.quicksight.describe_ingestion(ingestion_id=ingestion_id, dataset_name="covid19-
    nytimes-usa")["RowInfo"]
```

```
[12]: {'RowsIngested': 227689, 'RowsDropped': 0}
```

List all ingestions

```
[13]: [{"time": user["CreatedTime"], "source": user["RequestSource"]} for user in wr.
      ↪quicksight.list_ingestions("covid19-nytimes-usa")]

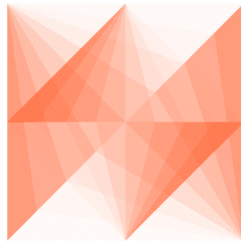
[13]: [{'time': datetime.datetime(2020, 6, 12, 15, 13, 46, 996000, tzinfo=tzlocal()),
      'source': 'MANUAL'},
      {'time': datetime.datetime(2020, 6, 12, 15, 13, 42, 344000, tzinfo=tzlocal()),
      'source': 'MANUAL'}]
```

Create new dataset from a table directly

```
[14]: wr.quicksight.create_athena_dataset(
      name="covid-19-tableau_jhu",
      table="tableau_jhu",
      data_source_name="covid-19",
      database="covid-19",
      import_mode='DIRECT_QUERY',
      rename_columns={
          "cases": "Count_of_Cases",
          "combined_key": "County"
      },
      cast_columns_types={
          "Count_of_Cases": "INTEGER"
      },
      tag_columns={
          "combined_key": [{"ColumnGeographicRole": "COUNTY"}]
      },
      allowed_to_manage=["dev"]
    )
```

Cleaning up

```
[15]: wr.quicksight.delete_data_source("covid-19")
      wr.quicksight.delete_dataset("covid19-nytimes-usa")
      wr.quicksight.delete_dataset("covid-19-tableau_jhu")
```



AWS SDK for pandas

1.3.19 19 - Amazon Athena Cache

`aws wrangler` has a cache strategy that is disabled by default and can be enabled by passing `max_cache_seconds` bigger than 0. This cache strategy for Amazon Athena can help you to **decrease query times and costs**.

When calling `read_sql_query`, instead of just running the query, we now can verify if the query has been run before. If so, and this last run was within `max_cache_seconds` (a new parameter to `read_sql_query`), we return the same results as last time if they are still available in S3. We have seen this increase performance more than 100x, but the potential is pretty much infinite.

The detailed approach is: - When `read_sql_query` is called with `max_cache_seconds > 0` (it defaults to 0), we check for the last queries run by the same workgroup (the most we can get without pagination). - By default it will check the last 50 queries, but you can customize it through the `max_cache_query inspections` argument. - We then sort those queries based on `CompletionDateTime`, descending - For each of those queries, we check if their `CompletionDateTime` is still within the `max_cache_seconds` window. If so, we check if the query string is the same as now (with some smart heuristics to guarantee coverage over both `ctas_approaches`). If they are the same, we check if the last one's results are still on S3, and then return them instead of re-running the query. - During the whole cache resolution phase, if there is anything wrong, the logic falls back to the usual `read_sql_query` path.

P.S. The ``cache scope is bounded for the current workgroup``, so you will be able to reuse queries results from others colleagues running in the same environment.

```
[1]: import aws wrangler as wr
```

Enter your bucket name:

```
[2]: import getpass
bucket = getpass.getpass()
path = f"s3://{bucket}/data/"
```

```
.....
```

Checking/Creating Glue Catalog Databases

```
[3]: if "aws wrangler_test" not in wr.catalog.databases().values:
      wr.catalog.create_database("aws wrangler_test")
```

Creating a Parquet Table from the NOAA's CSV files

Reference

```
[4]: cols = ["id", "dt", "element", "value", "m_flag", "q_flag", "s_flag", "obs_time"]

df = wr.s3.read_csv(
    path="s3://noaa-ghcn-pds/csv/by_year/189",
    names=cols,
    parse_dates=["dt", "obs_time"]) # Read 10 files from the 1890 decade (~1GB)

df
```

```
[4]:
```

	id	dt	element	value	m_flag	q_flag	s_flag	obs_time
0	AGE00135039	1890-01-01	TMAX	160	NaN	NaN	E	NaN
1	AGE00135039	1890-01-01	TMIN	30	NaN	NaN	E	NaN
2	AGE00135039	1890-01-01	PRCP	45	NaN	NaN	E	NaN
3	AGE00147705	1890-01-01	TMAX	140	NaN	NaN	E	NaN
4	AGE00147705	1890-01-01	TMIN	74	NaN	NaN	E	NaN
...
29240014	UZM00038457	1899-12-31	PRCP	16	NaN	NaN	r	NaN
29240015	UZM00038457	1899-12-31	TAVG	-73	NaN	NaN	r	NaN
29240016	UZM00038618	1899-12-31	TMIN	-76	NaN	NaN	r	NaN
29240017	UZM00038618	1899-12-31	PRCP	0	NaN	NaN	r	NaN
29240018	UZM00038618	1899-12-31	TAVG	-60	NaN	NaN	r	NaN

[29240019 rows x 8 columns]

```
[5]: wr.s3.to_parquet(
      df=df,
      path=path,
      dataset=True,
      mode="overwrite",
      database="awswrangler_test",
      table="noaa"
    )
```

```
[6]: wr.catalog.table(database="awswrangler_test", table="noaa")
```

```
[6]:
```

	Column Name	Type	Partition	Comment
0	id	string	False	
1	dt	timestamp	False	
2	element	string	False	
3	value	bigint	False	
4	m_flag	string	False	
5	q_flag	string	False	
6	s_flag	string	False	
7	obs_time	string	False	

The test query

The more computational resources the query needs, the more the cache will help you. That's why we're doing it using this long running query.

```
[7]: query = """
      SELECT
        n1.element,
        count(1) as cnt
      FROM
        noaa n1
      JOIN
        noaa n2
      ON
        n1.id = n2.id
      GROUP BY
```

(continues on next page)

(continued from previous page)

```
n1.element
"""
```

First execution...

```
[8]: %%time
```

```
wr.athena.read_sql_query(query, database="awsrangler_test")
```

```
CPU times: user 5.31 s, sys: 232 ms, total: 5.54 s
```

```
Wall time: 6min 42s
```

```
[8]:
```

	element	cnt
0	WDMV	49755046
1	SNWD	5089486328
2	DATN	10817510
3	DAPR	102579666
4	MDTN	10817510
5	WT03	71184687
6	WT09	584412
7	TOBS	146984266
8	DASF	7764526
9	WT04	9648963
10	WT18	92635444
11	WT01	87526136
12	WT16	323354156
13	PRCP	71238907298
14	SNOW	21950890838
15	WT06	307339
16	TAVG	2340863803
17	TMIN	41450979633
18	MDTX	11210687
19	WT07	4486872
20	WT10	137873
21	EVAP	970404
22	WT14	8073701
23	DATX	11210687
24	WT08	33933005
25	WT05	8211491
26	TMAX	39876132467
27	MDPR	114320989
28	WT11	22212890
29	DWPR	69005655
30	MDSF	12004843

Second execution with CACHE (400x faster)

[9]: %%time

```
wr.athena.read_sql_query(query, database="awsrangler_test", max_cache_seconds=900)
```

```
CPU times: user 493 ms, sys: 34.9 ms, total: 528 ms
```

```
Wall time: 975 ms
```

[9]:

	element	cnt
0	WDMV	49755046
1	SNWD	5089486328
2	DATN	10817510
3	DAPR	102579666
4	MDTN	10817510
5	WT03	71184687
6	WT09	584412
7	TOBS	146984266
8	DASF	7764526
9	WT04	9648963
10	WT18	92635444
11	WT01	87526136
12	WT16	323354156
13	PRCP	71238907298
14	SNOW	21950890838
15	WT06	307339
16	TAVG	2340863803
17	TMIN	41450979633
18	MDTX	11210687
19	WT07	4486872
20	WT10	137873
21	EVAP	970404
22	WT14	8073701
23	DATX	11210687
24	WT08	33933005
25	WT05	8211491
26	TMAX	39876132467
27	MDPR	114320989
28	WT11	22212890
29	DWPR	69005655
30	MDSF	12004843

Allowing awswrangler to inspect up to 500 historical queries to find same result to reuse.

[10]: %%time

```
wr.athena.read_sql_query(query, database="awsrangler_test", max_cache_seconds=900, max_
↪cache_query inspections=500)
```

```
CPU times: user 504 ms, sys: 44 ms, total: 548 ms
```

```
Wall time: 1.19 s
```

[10]:

	element	cnt
0	WDMV	49755046

(continues on next page)

(continued from previous page)

1	SNWD	5089486328
2	DATN	10817510
3	DAPR	102579666
4	MDTN	10817510
5	WT03	71184687
6	WT09	584412
7	TOBS	146984266
8	DASF	7764526
9	WT04	9648963
10	WT18	92635444
11	WT01	87526136
12	WT16	323354156
13	PRCP	71238907298
14	SNOW	21950890838
15	WT06	307339
16	TAVG	2340863803
17	TMIN	41450979633
18	MDTX	11210687
19	WT07	4486872
20	WT10	137873
21	EVAP	970404
22	WT14	8073701
23	DATX	11210687
24	WT08	33933005
25	WT05	8211491
26	TMAX	39876132467
27	MDPR	114320989
28	WT11	22212890
29	DWPR	69005655
30	MDSF	12004843

Cleaning Up S3

```
[11]: wr.s3.delete_objects(path)
```

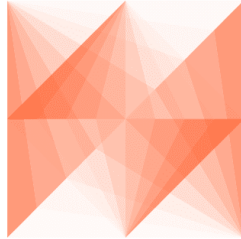
Delete table

```
[12]: wr.catalog.delete_table_if_exists(database="awswrangler_test", table="noaa")
```

```
[12]: True
```

Delete Database

```
[13]: wr.catalog.delete_database('awswrangler_test')
```



AWS SDK for pandas

1.3.20 20 - Spark Table Interoperability

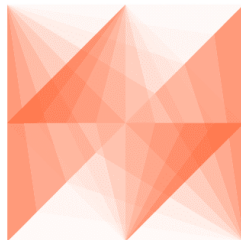
`awswrangler` has no difficulty to insert, overwrite or do any other kind of interaction with a Table created by Apache Spark.

But if you want to do the opposite (Spark interacting with a table created by `awswrangler`) you should be aware that `awswrangler` follows the Hive's format and you must be explicit when using the Spark's `saveAsTable` method:

```
[ ]: spark_df.write.format("hive").saveAsTable("database.table")
```

Or just move forward using the `insertInto` alternative:

```
[ ]: spark_df.write.insertInto("database.table")
```



AWS SDK for pandas

1.3.21 21 - Global Configurations

`aws wrangler` has two ways to set global configurations that will override the regular default arguments configured in functions signatures.

- **Environment variables**
- **`wr.config`**

P.S. Check the [function API doc](#) to see if your function has some argument that can be configured through Global configurations.

P.P.S. One exception to the above mentioned rules is the `botocore_config` property. It cannot be set through environment variables but only via `wr.config`. It will be used as the `botocore.config.Config` for all underlying `boto3` calls. The default config is `botocore.config.Config(retries={"max_attempts": 5}, connect_timeout=10, max_pool_connections=10)`. If you only want to change the retry behavior, you can use the environment variables `AWS_MAX_ATTEMPTS` and `AWS_RETRY_MODE`. (see [Boto3 documentation](#))

Environment Variables

```
[1]: %env WR_DATABASE=default
      %env WR_CTAS_APPROACH=False
      %env WR_MAX_CACHE_SECONDS=900
      %env WR_MAX_CACHE_QUERY_INSPECTIONS=500
      %env WR_MAX_REMOTE_CACHE_ENTRIES=50
      %env WR_MAX_LOCAL_CACHE_ENTRIES=100
```

```
env: WR_DATABASE=default
env: WR_CTAS_APPROACH=False
env: WR_MAX_CACHE_SECONDS=900
env: WR_MAX_CACHE_QUERY_INSPECTIONS=500
env: WR_MAX_REMOTE_CACHE_ENTRIES=50
env: WR_MAX_LOCAL_CACHE_ENTRIES=100
```

```
[1]: import aws wrangler as wr
      import botocore
```

```
[3]: wr.athena.read_sql_query("SELECT 1 AS FOO")
```

```
[3]:   foo
      0    1
```

Resetting

```
[4]: # Specific
      wr.config.reset("database")
      # All
      wr.config.reset()
```

wr.config

```
[5]: wr.config.database = "default"
wr.config.ctas_approach = False
wr.config.max_cache_seconds = 900
wr.config.max_cache_query_inspections = 500
wr.config.max_remote_cache_entries = 50
wr.config.max_local_cache_entries = 100
# Set botocore.config.Config that will be used for all boto3 calls
wr.config.botocore_config = botocore.config.Config(
    retries={"max_attempts": 10},
    connect_timeout=20,
    max_pool_connections=20
)
```

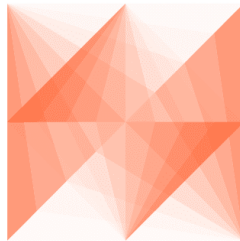
```
[6]: wr.athena.read_sql_query("SELECT 1 AS FOO")
```

```
[6]:    foo
0     1
```

Visualizing

```
[2]: wr.config
```

```
[2]: <aws wrangler._config._Config at 0x125f53710>
```



AWS SDK for pandas

1.3.22 22 - Writing Partitions Concurrently

- `concurrent_partitioning` argument:

```
If True will increase the parallelism level during the partitions writing. It will
↪ decrease the
writing time and increase memory usage.
```

P.S. Check the [function API doc](#) to see it has some argument that can be configured through Global configurations.

```
[1]: %reload_ext memory_profiler
```

```
import awswrangler as wr
```

Enter your bucket name:

```
[2]: import getpass
      bucket = getpass.getpass()
      path = f"s3://{bucket}/data/"
      .....
```

Reading 4 GB of CSV from NOAA's historical data and creating a year column

```
[3]: noaa_path = "s3://noaa-ghcn-pds/csv/by_year/193"

cols = ["id", "dt", "element", "value", "m_flag", "q_flag", "s_flag", "obs_time"]
dates = ["dt", "obs_time"]
dtype = {x: "category" for x in ["element", "m_flag", "q_flag", "s_flag"]}

df = wr.s3.read_csv(noaa_path, names=cols, parse_dates=dates, dtype=dtype)

df["year"] = df["dt"].dt.year

print(f"Number of rows: {len(df.index)}")
print(f"Number of columns: {len(df.columns)}")

Number of rows: 125407761
Number of columns: 9
```

Default Writing

```
[4]: %%time
      %%memit

      wr.s3.to_parquet(
          df=df,
          path=path,
          dataset=True,
          mode="overwrite",
          partition_cols=["year"],
      )

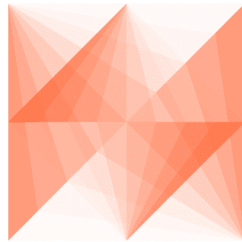
      peak memory: 22169.04 MiB, increment: 11119.68 MiB
      CPU times: user 49 s, sys: 12.5 s, total: 1min 1s
      Wall time: 1min 11s
```

Concurrent Partitioning (Decreasing writing time, but increasing memory usage)

```
[5]: %%time
      %%memit

      wr.s3.to_parquet(
          df=df,
          path=path,
          dataset=True,
          mode="overwrite",
          partition_cols=["year"],
          concurrent_partitioning=True # <-----
      )

      peak memory: 27819.48 MiB, increment: 15743.30 MiB
      CPU times: user 52.3 s, sys: 13.6 s, total: 1min 5s
      Wall time: 41.6 s
```



AWS SDK for pandas

1.3.23 23 - Flexible Partitions Filter (PUSH-DOWN)

- `partition_filter` argument:

- Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter).
- This function MUST receive a single argument (Dict[str, str]) where keys are ↵ partitions names and values are partitions values.
- This function MUST return a bool, True to read the partition or False to ignore ↵ it.
- Ignored if ``dataset=False``.

P.S. Check the [function API doc](#) to see it has some argument that can be configured through Global configurations.

```
[1]: import awswrangler as wr
      import pandas as pd
```

Enter your bucket name:

```
[2]: import getpass
      bucket = getpass.getpass()
      path = f"s3://{bucket}/dataset/"
      .....
```

Creating the Dataset (PARQUET)

```
[3]: df = pd.DataFrame({
      "id": [1, 2, 3],
      "value": ["foo", "boo", "bar"],
      })

      wr.s3.to_parquet(
          df=df,
          path=path,
          dataset=True,
          mode="overwrite",
          partition_cols=["value"]
      )

      wr.s3.read_parquet(path, dataset=True)
```

```
[3]:   id value
      0    3   bar
      1    2   boo
      2    1   foo
```

Example 1

```
[4]: my_filter = lambda x: x["value"].endswith("oo")

      wr.s3.read_parquet(path, dataset=True, partition_filter=my_filter)
```

```
[4]:   id value
      0    2   boo
      1    1   foo
```

Example 2

```
[5]: from Levenshtein import distance

      def my_filter(partitions):
          return distance("boo", partitions["value"]) <= 1

      wr.s3.read_parquet(path, dataset=True, partition_filter=my_filter)
```

```
[5]:      id value
0     2   boo
1     1   foo
```

Creating the Dataset (CSV)

```
[6]: df = pd.DataFrame({
      "id": [1, 2, 3],
      "value": ["foo", "boo", "bar"],
    })

wr.s3.to_csv(
    df=df,
    path=path,
    dataset=True,
    mode="overwrite",
    partition_cols=["value"],
    compression="gzip",
    index=False
)

wr.s3.read_csv(path, dataset=True)
```

```
[6]:      id value
0     3   bar
1     2   boo
2     1   foo
```

Example 1

```
[7]: my_filter = lambda x: x["value"].endswith("oo")

wr.s3.read_csv(path, dataset=True, partition_filter=my_filter)
```

```
[7]:      id value
0     2   boo
1     1   foo
```

Example 2

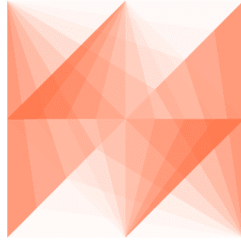
```
[8]: from Levenshtein import distance

def my_filter(partitions):
    return distance("boo", partitions["value"]) <= 1

wr.s3.read_csv(path, dataset=True, partition_filter=my_filter)
```



```
[8]:      id value
0     2   boo
1     1   foo
```



AWS SDK for pandas

1.3.24 24 - Athena Query Metadata

For `wr.athena.read_sql_query()` and `wr.athena.read_sql_table()` the resulting `DataFrame` (or every `DataFrame` in the returned Iterator for chunked queries) have a `query_metadata` attribute, which brings the query result metadata returned by Boto3/Athena.

The expected `query_metadata` format is the same returned by:

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.get_query_execution

Environment Variables

```
[1]: %env WR_DATABASE=default
env: WR_DATABASE=default
```

```
[2]: import awswrangler as wr
```

```
[5]: df = wr.athena.read_sql_query("SELECT 1 AS foo")
```

```
df
```

```
[5]:      foo
0     1
```

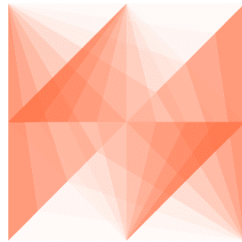
Getting statistics from query metadata

```
[6]: print(f'DataScannedInBytes:           {df.query_metadata["Statistics"][
      ↪ "DataScannedInBytes"]}')
      print(f'TotalExecutionTimeInMillis:   {df.query_metadata["Statistics"][
      ↪ "TotalExecutionTimeInMillis"]}')
      print(f'QueryQueueTimeInMillis:       {df.query_metadata["Statistics"][
      ↪ "QueryQueueTimeInMillis"]}')
      print(f'QueryPlanningTimeInMillis:    {df.query_metadata["Statistics"][
      ↪ "QueryPlanningTimeInMillis"]}')
      print(f'ServiceProcessingTimeInMillis: {df.query_metadata["Statistics"][
      ↪ "ServiceProcessingTimeInMillis"]}')

```

```
DataScannedInBytes:           0
TotalExecutionTimeInMillis:   2311
QueryQueueTimeInMillis:       121
QueryPlanningTimeInMillis:    250
ServiceProcessingTimeInMillis: 37

```



AWS SDK for pandas

1.3.25 25 - Redshift - Loading Parquet files with Spectrum

Enter your bucket name:

```
[1]: import getpass
      bucket = getpass.getpass()
      PATH = f"s3://{bucket}/files/"

```

.....

Mocking some Parquet Files on S3

```
[2]: import awswrangler as wr
      import pandas as pd

      df = pd.DataFrame({
          "col0": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
          "col1": ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"],
      })

```

(continues on next page)

(continued from previous page)

df

```
[2]:      col0 col1
0         0    a
1         1    b
2         2    c
3         3    d
4         4    e
5         5    f
6         6    g
7         7    h
8         8    i
9         9    j
```

```
[3]: wr.s3.to_parquet(df, PATH, max_rows_by_file=2, dataset=True, mode="overwrite")
```

Crawling the metadata and adding into Glue Catalog

```
[4]: wr.s3.store_parquet_metadata(
      path=PATH,
      database="aws_sdk_pandas",
      table="test",
      dataset=True,
      mode="overwrite"
    )
```

```
[4]: ({'col0': 'bigint', 'col1': 'string'}, None, None)
```

Running the CTAS query to load the data into Redshift storage

```
[5]: con = wr.redshift.connect(connection="aws-sdk-pandas-redshift")
```

```
[6]: query = "CREATE TABLE public.test AS (SELECT * FROM aws_sdk_pandas_external.test)"
```

```
[7]: with con.cursor() as cursor:
      cursor.execute(query)
```

Running an INSERT INTO query to load MORE data into Redshift storage

```
[8]: df = pd.DataFrame({
      "col0": [10, 11],
      "col1": ["k", "l"],
    })
wr.s3.to_parquet(df, PATH, dataset=True, mode="overwrite")
```

```
[9]: query = "INSERT INTO public.test (SELECT * FROM aws_sdk_pandas_external.test)"
```

```
[10]: with con.cursor() as cursor:
      cursor.execute(query)
```

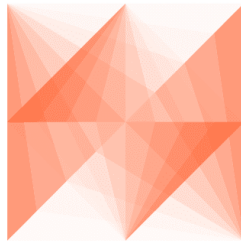
Checking the result

```
[11]: query = "SELECT * FROM public.test"
```

```
[13]: wr.redshift.read_sql_table(con=con, schema="public", table="test")
```

```
[13]:   col0  col1
0      5    f
1      1    b
2      3    d
3      6    g
4      8    i
5     10    k
6      4    e
7      0    a
8      2    c
9      7    h
10     9    j
11    11    l
```

```
[14]: con.close()
```



AWS SDK for pandas

1.3.26 26 - Amazon Timestream

Creating resources

```
[10]: import awswrangler as wr
      import pandas as pd
      from datetime import datetime

      wr.timestream.create_database("sampleDB")
      wr.timestream.create_table("sampleDB", "sampleTable", memory_retention_hours=1, magnetic_
      ↪retention_days=1)
```

Write

```
[11]: df = pd.DataFrame(
    {
        "time": [datetime.now(), datetime.now(), datetime.now()],
        "dim0": ["foo", "boo", "bar"],
        "dim1": [1, 2, 3],
        "measure": [1.0, 1.1, 1.2],
    }
)

rejected_records = wr.timestream.write(
    df=df,
    database="sampleDB",
    table="sampleTable",
    time_col="time",
    measure_col="measure",
    dimensions_cols=["dim0", "dim1"],
)

print(f"Number of rejected records: {len(rejected_records)}")
```

Number of rejected records: 0

Query

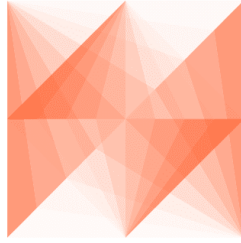
```
[12]: wr.timestream.query(
    'SELECT time, measure_value::double, dim0, dim1 FROM "sampleDB"."sampleTable" ORDER_
    ↪BY time DESC LIMIT 3'
)

[12]:
```

	time	measure_value::double	dim0	dim1
0	2020-12-08 19:15:32.468	1.0	foo	1
1	2020-12-08 19:15:32.468	1.2	bar	3
2	2020-12-08 19:15:32.468	1.1	boo	2

Deleting resources

```
[13]: wr.timestream.delete_table("sampleDB", "sampleTable")
wr.timestream.delete_database("sampleDB")
```



AWS SDK for pandas

1.3.27 27 - Amazon Timestream - Example 2

Reading test data

```
[1]: import awswrangler as wr
import pandas as pd
from datetime import datetime

df = pd.read_csv(
    "https://raw.githubusercontent.com/aws/amazon-timestream-tools/master/sample_apps/
    ↪data/sample.csv",
    names=[
        "ignore0",
        "region",
        "ignore1",
        "az",
        "ignore2",
        "hostname",
        "measure_kind",
        "measure",
        "ignore3",
        "ignore4",
        "ignore5",
    ],
    usecols=["region", "az", "hostname", "measure_kind", "measure"],
)
df["time"] = datetime.now()
df.reset_index(inplace=True, drop=False)

df
```

```
[1]:
```

	index	region	az	hostname	measure_kind \
0	0	us-east-1	us-east-1a	host-fj2hx	cpu_utilization
1	1	us-east-1	us-east-1a	host-fj2hx	memory_utilization
2	2	us-east-1	us-east-1a	host-6kMPE	cpu_utilization
3	3	us-east-1	us-east-1a	host-6kMPE	memory_utilization
4	4	us-east-1	us-east-1a	host-sxj7X	cpu_utilization
...
125995	125995	eu-north-1	eu-north-1c	host-De8RB	memory_utilization

(continues on next page)

(continued from previous page)

```

125996 125996 eu-north-1 eu-north-1c host-2z8tn memory_utilization
125997 125997 eu-north-1 eu-north-1c host-2z8tn cpu_utilization
125998 125998 eu-north-1 eu-north-1c host-9FcZw memory_utilization
125999 125999 eu-north-1 eu-north-1c host-9FcZw cpu_utilization

```

```

      measure                                time
0      21.394363 2020-12-08 16:18:47.599597
1      68.563420 2020-12-08 16:18:47.599597
2      17.144579 2020-12-08 16:18:47.599597
3      73.507870 2020-12-08 16:18:47.599597
4      26.584865 2020-12-08 16:18:47.599597
...      ...
125995 68.063468 2020-12-08 16:18:47.599597
125996 72.203680 2020-12-08 16:18:47.599597
125997 29.212219 2020-12-08 16:18:47.599597
125998 71.746134 2020-12-08 16:18:47.599597
125999  1.677793 2020-12-08 16:18:47.599597

```

```
[126000 rows x 7 columns]
```

Creating resources

```

[2]: wr.timestream.create_database("sampleDB")
wr.timestream.create_table("sampleDB", "sampleTable", memory_retention_hours=1, magnetic_
    ↪ retention_days=1)

```

Write CPU_UTILIZATION records

```

[3]: df_cpu = df[df.measure_kind == "cpu_utilization"].copy()
df_cpu.rename(columns={"measure": "cpu_utilization"}, inplace=True)
df_cpu

```

```

[3]:
   index  region  az  hostname  measure_kind \
0      0  us-east-1  us-east-1a  host-fj2hx  cpu_utilization
2      2  us-east-1  us-east-1a  host-6kMPE  cpu_utilization
4      4  us-east-1  us-east-1a  host-sxj7X  cpu_utilization
6      6  us-east-1  us-east-1a  host-ExOui  cpu_utilization
8      8  us-east-1  us-east-1a  host-Bwb3j  cpu_utilization
...    ...    ...    ...    ...    ...
125990 125990  eu-north-1  eu-north-1c  host-aPtc6  cpu_utilization
125992 125992  eu-north-1  eu-north-1c  host-7ZF9L  cpu_utilization
125994 125994  eu-north-1  eu-north-1c  host-De8RB  cpu_utilization
125997 125997  eu-north-1  eu-north-1c  host-2z8tn  cpu_utilization
125999 125999  eu-north-1  eu-north-1c  host-9FcZw  cpu_utilization

      cpu_utilization                                time
0      21.394363 2020-12-08 16:18:47.599597
2      17.144579 2020-12-08 16:18:47.599597
4      26.584865 2020-12-08 16:18:47.599597

```

(continues on next page)

(continued from previous page)

```

6          52.930970 2020-12-08 16:18:47.599597
8          99.134110 2020-12-08 16:18:47.599597
...          ...          ...
125990     89.566125 2020-12-08 16:18:47.599597
125992     75.510598 2020-12-08 16:18:47.599597
125994       2.771261 2020-12-08 16:18:47.599597
125997     29.212219 2020-12-08 16:18:47.599597
125999       1.677793 2020-12-08 16:18:47.599597

```

```
[63000 rows x 7 columns]
```

```

[4]: rejected_records = wr.timestream.write(
      df=df_cpu,
      database="sampleDB",
      table="sampleTable",
      time_col="time",
      measure_col="cpu_utilization",
      dimensions_cols=["index", "region", "az", "hostname"],
    )

assert len(rejected_records) == 0

```

Write MEMORY_UTILIZATION records

```

[5]: df_memory = df[df.measure_kind == "memory_utilization"].copy()
      df_memory.rename(columns={"measure": "memory_utilization"}, inplace=True)

```

```
df_memory
```

```

[5]:
      index  region  az  hostname  measure_kind \
1         1  us-east-1  us-east-1a  host-fj2hx  memory_utilization
3         3  us-east-1  us-east-1a  host-6kMPE  memory_utilization
5         5  us-east-1  us-east-1a  host-sxj7X  memory_utilization
7         7  us-east-1  us-east-1a  host-ExOui  memory_utilization
9         9  us-east-1  us-east-1a  host-Bwb3j  memory_utilization
...      ...      ...      ...      ...      ...
125991  125991  eu-north-1  eu-north-1c  host-aPtc6  memory_utilization
125993  125993  eu-north-1  eu-north-1c  host-7ZF9L  memory_utilization
125995  125995  eu-north-1  eu-north-1c  host-De8RB  memory_utilization
125996  125996  eu-north-1  eu-north-1c  host-2z8tn  memory_utilization
125998  125998  eu-north-1  eu-north-1c  host-9FcZw  memory_utilization

      memory_utilization  time
1         68.563420 2020-12-08 16:18:47.599597
3         73.507870 2020-12-08 16:18:47.599597
5         22.401424 2020-12-08 16:18:47.599597
7         45.440135 2020-12-08 16:18:47.599597
9         15.042701 2020-12-08 16:18:47.599597
...      ...      ...
125991     75.686739 2020-12-08 16:18:47.599597
125993     18.386152 2020-12-08 16:18:47.599597

```

(continues on next page)

(continued from previous page)

```

125995          68.063468 2020-12-08 16:18:47.599597
125996          72.203680 2020-12-08 16:18:47.599597
125998          71.746134 2020-12-08 16:18:47.599597

```

```
[63000 rows x 7 columns]
```

```

[6]: rejected_records = wr.timestream.write(
    df=df_memory,
    database="sampleDB",
    table="sampleTable",
    time_col="time",
    measure_col="memory_utilization",
    dimensions_cols=["index", "region", "az", "hostname"],
)

assert len(rejected_records) == 0

```

Querying CPU_UTILIZATION

```

[7]: wr.timestream.query("""
    SELECT
        hostname, region, az, measure_name, measure_value::double, time
    FROM "sampleDB"."sampleTable"
    WHERE measure_name = 'cpu_utilization'
    ORDER BY time DESC
    LIMIT 10
""")

```

```

[7]:
  hostname      region      az      measure_name \
0 host-0gvFx    us-west-1    us-west-1a  cpu_utilization
1 host-rZUNx    eu-north-1    eu-north-1a  cpu_utilization
2 host-t1kAB    us-east-2      us-east-2b  cpu_utilization
3 host-RdQRf    us-east-1      us-east-1c  cpu_utilization
4 host-4Llhu    us-east-1      us-east-1c  cpu_utilization
5 host-2plqa    us-west-1      us-west-1a  cpu_utilization
6 host-J3Q4z    us-east-1      us-east-1b  cpu_utilization
7 host-VIR5T    ap-east-1      ap-east-1a  cpu_utilization
8 host-G042D    us-east-1      us-east-1c  cpu_utilization
9 host-8EBHm    us-west-2      us-west-2c  cpu_utilization

  measure_value::double      time
0          39.617911 2020-12-08 19:18:47.600
1          30.793332 2020-12-08 19:18:47.600
2          74.453239 2020-12-08 19:18:47.600
3          76.984448 2020-12-08 19:18:47.600
4          41.862733 2020-12-08 19:18:47.600
5          34.864762 2020-12-08 19:18:47.600
6          71.574266 2020-12-08 19:18:47.600
7          14.017491 2020-12-08 19:18:47.600
8          60.199068 2020-12-08 19:18:47.600
9          96.631624 2020-12-08 19:18:47.600

```

Querying MEMORY_UTILIZATION

```
[8]: wr.timestream.query("""
      SELECT
        hostname, region, az, measure_name, measure_value::double, time
      FROM "sampleDB"."sampleTable"
      WHERE measure_name = 'memory_utilization'
      ORDER BY time DESC
      LIMIT 10
      """)
```

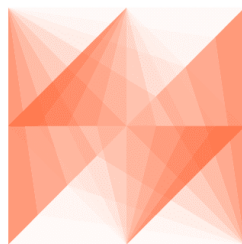
```
[8]:
```

	hostname	region	az	measure_name \
0	host-7c897	us-west-2	us-west-2b	memory_utilization
1	host-2z8tn	eu-north-1	eu-north-1c	memory_utilization
2	host-J3Q4z	us-east-1	us-east-1b	memory_utilization
3	host-mjrQb	us-east-1	us-east-1b	memory_utilization
4	host-AyWSI	us-east-1	us-east-1c	memory_utilization
5	host-Axf0g	us-west-2	us-west-2a	memory_utilization
6	host-ilMBa	us-east-2	us-east-2b	memory_utilization
7	host-CWdXX	us-west-2	us-west-2c	memory_utilization
8	host-8EBHm	us-west-2	us-west-2c	memory_utilization
9	host-dRIJj	us-east-1	us-east-1c	memory_utilization

	measure_value::double	time
0	63.427726	2020-12-08 19:18:47.600
1	41.071368	2020-12-08 19:18:47.600
2	23.944388	2020-12-08 19:18:47.600
3	69.173431	2020-12-08 19:18:47.600
4	75.591467	2020-12-08 19:18:47.600
5	29.720739	2020-12-08 19:18:47.600
6	71.544134	2020-12-08 19:18:47.600
7	79.792799	2020-12-08 19:18:47.600
8	66.082554	2020-12-08 19:18:47.600
9	86.748960	2020-12-08 19:18:47.600

Deleting resources

```
[9]: wr.timestream.delete_table("sampleDB", "sampleTable")
      wr.timestream.delete_database("sampleDB")
```



AWS SDK for pandas

1.3.28 28 - Amazon DynamoDB

Writing Data

```
[23]: from datetime import datetime
      from decimal import Decimal
      from pathlib import Path

      import awswrangler as wr
      import pandas as pd
      from boto3.dynamodb.conditions import Attr, Key
```

Writing DataFrame

```
[27]: table_name = "movies"

df = pd.DataFrame({
    "title": ["Titanic", "Snatch", "The Godfather"],
    "year": [1997, 2000, 1972],
    "genre": ["drama", "caper story", "crime"],
})
wr.dynamodb.put_df(df=df, table_name=table_name)
```

Writing CSV file

```
[3]: filepath = Path("items.csv")
      df.to_csv(filepath, index=False)
      wr.dynamodb.put_csv(path=filepath, table_name=table_name)
      filepath.unlink()
```

Writing JSON files

```
[4]: filepath = Path("items.json")
      df.to_json(filepath, orient="records")
      wr.dynamodb.put_json(path="items.json", table_name=table_name)
      filepath.unlink()
```

Writing list of items

```
[5]: items = df.to_dict(orient="records")
      wr.dynamodb.put_items(items=items, table_name=table_name)
```

Reading Data

Read PartiQL

```
[ ]: # PartiQL Query
wr.dynamodb.read_partiql_query(
    query=f"SELECT * FROM {table_name} WHERE title=? AND year=?",
    parameters=["Snatch", 2000],
)
```

Read Items

```
[ ]: # Limit Read to 5 items
wr.dynamodb.read_items(table_name=table_name, max_items_evaluated=5)

# Limit Read to Key expression
wr.dynamodb.read_items(
    table_name=table_name,
    key_condition_expression=(Key("title").eq("Snatch") & Key("year").eq(2000))
)
```

Executing statements

```
[29]: title = "The Lord of the Rings: The Fellowship of the Ring"
year = datetime.now().year
genre = "epic"
rating = Decimal('9.9')
plot = "The fate of Middle-earth hangs in the balance as Frodo and eight companions_
↳begin their journey to Mount Doom in the land of Mordor."

# Insert items
wr.dynamodb.execute_statement(
    statement=f"INSERT INTO {table_name} VALUE {{'title': ?, 'year': ?, 'genre': ?, 'info
↳': ?}}",
    parameters=[title, year, genre, {"plot": plot, "rating": rating}],
)

# Select items
wr.dynamodb.execute_statement(
    statement=f"SELECT * FROM \"{table_name}\" WHERE title=? AND year=?",
    parameters=[title, year],
)

# Update items
wr.dynamodb.execute_statement(
    statement=f"UPDATE \"{table_name}\" SET info.rating=? WHERE title=? AND year=?",
    parameters=[Decimal(10), title, year],
)
```

(continues on next page)

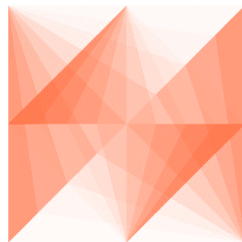
(continued from previous page)

```
# Delete items
wr.dynamodb.execute_statement(
    statement=f"DELETE FROM \'{table_name}\' WHERE title=? AND year=?",
    parameters=[title, year],
)
```

```
[29]: []
```

Deleting items

```
[6]: wr.dynamodb.delete_items(items=items, table_name="table")
```



AWS SDK for pandas

1.3.29 29 - S3 Select

AWS SDK for pandas supports [Amazon S3 Select](#), enabling applications to use SQL statements in order to query and filter the contents of a single S3 object. It works on objects stored in CSV, JSON or Apache Parquet, including compressed and large files of several TBs.

With S3 Select, the query workload is delegated to Amazon S3, leading to lower latency and cost, and to higher performance (up to 400% improvement). This is in comparison with other awswrangler operations such as `read_parquet` where the S3 object is downloaded and filtered on the client-side.

This feature has a number of limitations however, and should be used for specific scenarios only:

- It operates on a single S3 object
- The maximum length of a record in the input or result is 1 MB
- The maximum uncompressed row group size is 256 MB (Parquet only)
- It can only emit nested data in JSON format
- Certain SQL operations are not supported (e.g. ORDER BY)

Read full CSV file

```
[2]: import awswrangler as wr

df = wr.s3.select_query(
    sql="SELECT * FROM s3object",
    path="s3://nyc-tlc/csv_backup/fhv_tripdata_2019-09.csv", # 58 MB
    input_serialization="CSV",
    input_serialization_params={
        "FileHeaderInfo": "Use",
        "RecordDelimiter": "\r\n",
    },
    use_threads=True,
)
df.head()
```

```
[2]:
```

	dispatching_base_num	pickup_datetime	dropoff_datetime	PULocationID	\
0	B00009	2019-09-01 00:35:00	2019-09-01 00:59:00	264	
1	B00009	2019-09-01 00:48:00	2019-09-01 01:09:00	264	
2	B00014	2019-09-01 00:16:18	2019-09-02 00:35:37	264	
3	B00014	2019-09-01 00:55:03	2019-09-01 01:09:35	264	
4	B00014	2019-09-01 00:13:08	2019-09-02 01:12:31	264	

	DOLocationID	SR_Flag
0	264	
1	264	
2	264	
3	264	
4	264	

Filter JSON file

```
[2]: wr.s3.select_query(
    sql="SELECT * FROM s3object[*] s where s.\"family_name\" = \"Biden\"",
    path="s3://aws-glue-datasets/examples/us-legislators/all/persons.json",
    input_serialization="JSON",
    input_serialization_params={
        "Type": "Document",
    },
)
```

```
[2]:
```

	family_name	contact_details	name	\
0	Biden	[{'type': 'twitter', 'value': 'joebiden'}]	Joseph Biden, Jr.	

	links	gender	\
0	[{'note': 'Wikipedia (ace)', 'url': 'https://a...'}]	male	

	image	\
0	https://theunitedstates.io/images/congress/ori...	

	identifiers	\
0	[{'scheme': 'bioguide', 'identifier': 'B000444...'}]	

(continues on next page)

(continued from previous page)

```

                                other_names      sort_name \
0  [{'note': 'alternate', 'name': 'Joe Biden'}, {...  Biden, Joseph

                                images given_name  birth_date \
0  [{'url': 'https://theunitedstates.io/images/co...  Joseph  1942-11-20

                                id
0  64239edf-8e06-4d2d-acc0-33d96bc79774

```

Read Snappy compressed Parquet

```

[3]: df = wr.s3.select_query(
        sql="SELECT * FROM s3object s where s.\"star_rating\" >= 5",
        path="s3://amazon-reviews-pds/parquet/product_category=Gift_Card/part-000000-
        ↪495c48e6-96d6-4650-aa65-3c36a3516ddd.c000.snappy.parquet",
        input_serialization="Parquet",
        input_serialization_params={},
        use_threads=True,
    )
df.loc[:, df.columns != "product_title"].head()

```

```

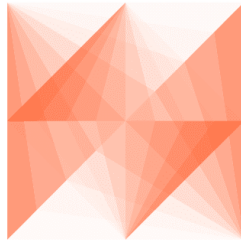
[3]: marketplace customer_id      review_id  product_id product_parent \
0      US      52670295    RGPOFKORD8RTU    B0002CZPPG      867256265
1      US      29964102    R2U8X8V5KPB4J3    B00H5BMF00      373287760
2      US      25173351    R15XV3LXUMLTXL    B00PG40C04      137115061
3      US      12516181    R3G6G7H8TX4H0T    B0002CZPPG      867256265
4      US      38355314    R2NJ7WNBUI6YTQ    B00B2TFS06      89375983

    star_rating  helpful_votes  total_votes  vine  verified_purchase \
0             5             105           107    N                 N
1             5              0              0    N                 Y
2             5              0              0    N                 Y
3             5              6              6    N                 N
4             5              0              0    N                 Y

    review_headline      review_body \
0  Excellent Gift Idea  I wonder if the other reviewer actually read t...
1      Five Stars      convenience is the name of the game.
2  Birthday Gift      This gift card was handled with accuracy in de...
3      Love 'em.      Gotta love these iTunes Prepaid Card thingys. ...
4      Five Stars      perfect

    review_date  year
0  2005-02-08  2005
1  2015-05-03  2015
2  2015-05-03  2015
3  2005-10-15  2005
4  2015-05-03  2015

```



AWS SDK for pandas

1.3.30 30 - Data Api

The Data Api simplifies access to Amazon Redshift and RDS by removing the need to manage database connections and credentials. Instead, you can execute SQL commands to an Amazon Redshift cluster or Amazon Aurora cluster by simply invoking an HTTPS API endpoint provided by the Data API. It takes care of managing database connections and returning data. Since the Data API leverages IAM user credentials or database credentials stored in AWS Secrets Manager, you don't need to pass credentials in API calls.

Connect to the cluster

- `wr.data_api.redshift.connect()`
- `wr.data_api.rds.connect()`

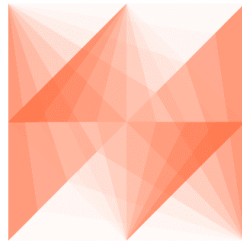
```
[ ]: con_redshift = wr.data_api.redshift.connect(  
    cluster_id="aws-sdk-pandas-1xn5lqxrdrv3",  
    database="test_redshift",  
    secret_arn="arn:aws:secretsmanager:us-east-1:111111111111:secret:aws-sdk-pandas/  
↳redshift-ewn43d"  
)  
  
con_redshift_serverless = wr.data_api.redshift.connect(  
    workgroup_name="aws-sdk-pandas",  
    database="test_redshift",  
    secret_arn="arn:aws:secretsmanager:us-east-1:111111111111:secret:aws-sdk-pandas/  
↳redshift-f3en4w"  
)  
  
con_mysql = wr.data_api.rds.connect(  
    resource_arn="arn:aws:rds:us-east-1:111111111111:cluster:mysql-serverless-cluster-  
↳wrangler",  
    database="test_rds",  
    secret_arn="arn:aws:secretsmanager:us-east-1:111111111111:secret:aws-sdk-pandas/  
↳mysql-23df3"  
)
```


Read from database

- `wr.data_api.redshift.read_sql_query()`
- `wr.data_api.rds.read_sql_query()`

```
[ ]: df = wr.data_api.redshift.read_sql_query(
    sql="SELECT * FROM public.test_table",
    con=con_redshift,
)

df = wr.data_api.rds.read_sql_query(
    sql="SELECT * FROM test.test_table",
    con=con_rds,
)
```



AWS SDK for pandas

1.3.31 31 - OpenSearch

Table of Contents

- 1. Initialize
 - Connect to your Amazon OpenSearch domain
 - Enter your bucket name
 - Initialize sample data
- 2. Indexing (load)
 - Index documents (no Pandas)
 - Index json file
 - Index CSV
- 3. Search
 - 3.1 Search by DSL
 - 3.2 Search by SQL
- 4. Delete Indices
- 5. Bonus - Prepare data and index from DataFrame
 - Prepare the data for indexing

- Create index with mapping
- Index dataframe
- Execute geo query

1. Initialize

```
[1]: import awswrangler as wr
```

Connect to your Amazon OpenSearch domain

```
[2]: client = wr.opensearch.connect(  
    host='OPENSEARCH-ENDPOINT',  
    # username='FGAC-USERNAME(OPTIONAL)',  
    # password='FGAC-PASSWORD(OPTIONAL)'  
)  
client.info()
```

Enter your bucket name

```
[3]: bucket = 'BUCKET'
```

Initialize sample data

```
[4]: sf_restaurants_inspections = [  
    {  
        "inspection_id": "24936_20160609",  
        "business_address": "315 California St",  
        "business_city": "San Francisco",  
        "business_id": "24936",  
        "business_location": {"lon": -122.400152, "lat": 37.793199},  
        "business_name": "San Francisco Soup Company",  
        "business_postal_code": "94104",  
        "business_state": "CA",  
        "inspection_date": "2016-06-09T00:00:00.000",  
        "inspection_score": 77,  
        "inspection_type": "Routine - Unscheduled",  
        "risk_category": "Low Risk",  
        "violation_description": "Improper food labeling or menu misrepresentation",  
        "violation_id": "24936_20160609_103141",  
    },  
    {  
        "inspection_id": "60354_20161123",  
        "business_address": "10 Mason St",  
        "business_city": "San Francisco",  
        "business_id": "60354",  
        "business_location": {"lon": -122.409061, "lat": 37.783527},  
    },  
]
```

(continues on next page)

(continued from previous page)

```

    "business_name": "Soup Unlimited",
    "business_postal_code": "94102",
    "business_state": "CA",
    "inspection_date": "2016-11-23T00:00:00.000",
    "inspection_type": "Routine",
    "inspection_score": 95,
  },
  {
    "inspection_id": "1797_20160705",
    "business_address": "2872 24th St",
    "business_city": "San Francisco",
    "business_id": "1797",
    "business_location": {"lon": -122.409752, "lat": 37.752807},
    "business_name": "TIO CHILOS GRILL",
    "business_postal_code": "94110",
    "business_state": "CA",
    "inspection_date": "2016-07-05T00:00:00.000",
    "inspection_score": 90,
    "inspection_type": "Routine - Unscheduled",
    "risk_category": "Low Risk",
    "violation_description": "Unclean nonfood contact surfaces",
    "violation_id": "1797_20160705_103142",
  },
  {
    "inspection_id": "66198_20160527",
    "business_address": "1661 Tennessee St Suite 3B",
    "business_city": "San Francisco Whard Restaurant",
    "business_id": "66198",
    "business_location": {"lon": -122.388478, "lat": 37.75072},
    "business_name": "San Francisco Restaurant",
    "business_postal_code": "94107",
    "business_state": "CA",
    "inspection_date": "2016-05-27T00:00:00.000",
    "inspection_type": "Routine",
    "inspection_score": 56,
  },
  {
    "inspection_id": "5794_20160907",
    "business_address": "2162 24th Ave",
    "business_city": "San Francisco",
    "business_id": "5794",
    "business_location": {"lon": -122.481299, "lat": 37.747228},
    "business_name": "Soup House",
    "business_phone_number": "+14155752700",
    "business_postal_code": "94116",
    "business_state": "CA",
    "inspection_date": "2016-09-07T00:00:00.000",
    "inspection_score": 96,
    "inspection_type": "Routine - Unscheduled",
    "risk_category": "Low Risk",
    "violation_description": "Unapproved or unmaintained equipment or utensils",
    "violation_id": "5794_20160907_103144",
  }

```

(continues on next page)

(continued from previous page)

```

    },
    # duplicate record
    {
        "inspection_id": "5794_20160907",
        "business_address": "2162 24th Ave",
        "business_city": "San Francisco",
        "business_id": "5794",
        "business_location": {"lon": -122.481299, "lat": 37.747228},
        "business_name": "Soup-or-Salad",
        "business_phone_number": "+14155752700",
        "business_postal_code": "94116",
        "business_state": "CA",
        "inspection_date": "2016-09-07T00:00:00.000",
        "inspection_score": 96,
        "inspection_type": "Routine - Unscheduled",
        "risk_category": "Low Risk",
        "violation_description": "Unapproved or unmaintained equipment or utensils",
        "violation_id": "5794_20160907_103144",
    },
]

```

2. Indexing (load)

Index documents (no Pandas)

```
[5]: # index documents w/o providing keys (_id is auto-generated)
```

```

wr.opensearch.index_documents(
    client,
    documents=sf_restaurants_inspections,
    index="sf_restaurants_inspections"
)

```

```
Indexing: 100% (6/6) |#####|Elapsed Time: 0:00:01
```

```
[5]: {'success': 6, 'errors': []}
```

```
[6]: # read all documents. There are total 6 documents
```

```

wr.opensearch.search(
    client,
    index="sf_restaurants_inspections",
    _source=["inspection_id", "business_name", "business_location"]
)

```

```
[6]:
```

	_id	business_name \
0	663dd72d-0da4-495b-b0ae-ed000105ae73	TIO CHILOS GRILL
1	ff2f50f6-5415-4706-9bcb-af7c5eb0afa3	Soup House
2	b9e8f6a2-8fd1-4660-b041-2997a1a80984	San Francisco Soup Company
3	56b352e6-102b-4eff-8296-7e1fb2459bab	Soup Unlimited
4	6fec5411-f79a-48e4-be7b-e0e44d5ebbab	San Francisco Restaurant
5	7ba4fb17-f9a9-49da-b90e-8b3553d6d97c	Soup-or-Salad

(continues on next page)

(continued from previous page)

	inspection_id	business_location.lon	business_location.lat
0	1797_20160705	-122.409752	37.752807
1	5794_20160907	-122.481299	37.747228
2	24936_20160609	-122.400152	37.793199
3	60354_20161123	-122.409061	37.783527
4	66198_20160527	-122.388478	37.750720
5	5794_20160907	-122.481299	37.747228

Index json file

```
[ ]: import pandas as pd
df = pd.DataFrame(sf_restaurants_inspections)
path = f"s3://{bucket}/json/sf_restaurants_inspections.json"
wr.s3.to_json(df, path, orient='records', lines=True)
```

```
[8]: # index json w/ providing keys
wr.opensearch.index_json(
    client,
    path=path, # path can be s3 or local
    index="sf_restaurants_inspections_dedup",
    id_keys=["inspection_id"] # can be multiple fields. arg applicable to all index_
    ↪ * functions
)
```

```
Indexing: 100% (6/6) |#####|Elapsed Time: 0:00:00
```

```
[8]: {'success': 6, 'errors': []}
```

```
[9]: # now there are no duplicates. There are total 5 documents
wr.opensearch.search(
    client,
    index="sf_restaurants_inspections_dedup",
    _source=["inspection_id", "business_name", "business_location"]
)
```

```
[9]:
```

	_id	business_name	inspection_id \
0	24936_20160609	San Francisco Soup Company	24936_20160609
1	66198_20160527	San Francisco Restaurant	66198_20160527
2	5794_20160907	Soup-or-Salad	5794_20160907
3	60354_20161123	Soup Unlimited	60354_20161123
4	1797_20160705	TIO CHILOS GRILL	1797_20160705

	business_location.lon	business_location.lat
0	-122.400152	37.793199
1	-122.388478	37.750720
2	-122.481299	37.747228
3	-122.409061	37.783527
4	-122.409752	37.752807

Index CSV

```
[11]: wr.opensearch.index_csv(
        client,
        index="nyc_restaurants_inspections_sample",
        path='https://data.cityofnewyork.us/api/views/43nn-pn8j/rows.csv?
        ↪accessType=DOWNLOAD', # index_csv supports local, s3 and url path
        id_keys=["CAMIS"],
        pandas_kwargs={'na_filter': True, 'nrows': 1000}, # pandas.read_csv() args ↪
        ↪https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html
        bulk_size=500 # modify based on your cluster size
    )
```

```
Indexing: 100% (1000/1000)|#####|Elapsed Time: 0:00:00
```

```
[11]: {'success': 1000, 'errors': []}
```

```
[12]: wr.opensearch.search(
        client,
        index="nyc_restaurants_inspections_sample",
        size=5
    )
```

```
[12]:
```

	_id	CAMIS	DBA	BORO	BUILDING	\
0	41610426	41610426	GLOW THAI RESTAURANT	Brooklyn	7107	
1	40811162	40811162	CARMINE'S	Manhattan	2450	
2	50012113	50012113	TANG	Queens	196-50	
3	50014618	50014618	TOTTO RAMEN	Manhattan	248	
4	50045782	50045782	OLLIE'S CHINESE RESTAURANT	Manhattan	2705	

	STREET	ZIPCODE	PHONE	CUISINE	DESCRIPTION	\
0	3 AVENUE	11209.0	7187481920		Thai	
1	BROADWAY	10024.0	2123622200		Italian	
2	NORTHERN BOULEVARD	11358.0	7182797080		Korean	
3	EAST 52 STREET	10022.0	2124210052		Japanese	
4	BROADWAY	10025.0	2129323300		Chinese	

	INSPECTION DATE	...	RECORD DATE	INSPECTION TYPE	\
0	02/26/2020	...	10/04/2021	Cycle Inspection / Re-inspection	
1	05/28/2019	...	10/04/2021	Cycle Inspection / Initial Inspection	
2	08/16/2018	...	10/04/2021	Cycle Inspection / Initial Inspection	
3	08/20/2018	...	10/04/2021	Cycle Inspection / Re-inspection	
4	10/21/2019	...	10/04/2021	Cycle Inspection / Re-inspection	

	Latitude	Longitude	Community	Board	Council	District	Census Tract	\
0	40.633865	-74.026798		310.0		43.0	6800.0	
1	40.791168	-73.974308		107.0		6.0	17900.0	
2	40.757850	-73.784593		411.0		19.0	145101.0	
3	40.756596	-73.968749		106.0		4.0	9800.0	
4	40.799318	-73.968440		107.0		6.0	19100.0	

	BIN	BBL	NTA
0	3146519.0	3.058910e+09	BK31
1	1033560.0	1.012380e+09	MN12

(continues on next page)

(continued from previous page)

```

2  4124565.0  4.055200e+09  QN48
3  1038490.0  1.013250e+09  MN19
4  1056562.0  1.018750e+09  MN12

```

```
[5 rows x 27 columns]
```

3. Search

3.1 Search by DSL

```

[13]: # add a search query. search all soup businesses
wr.opensearch.search(
    client,
    index="sf_restaurants_inspections",
    _source=["inspection_id", "business_name", "business_location"],
    filter_path=["hits.hits._id", "hits.hits._source"],
    search_body={
        "query": {
            "match": {
                "business_name": "soup"
            }
        }
    }
)

```

```

[13]:
      _id                                business_name \
0  ff2f50f6-5415-4706-9bcb-af7c5eb0afa3          Soup House
1  7ba4fb17-f9a9-49da-b90e-8b3553d6d97c      Soup-or-Salad
2  b9e8f6a2-8fd1-4660-b041-2997a1a80984  San Francisco Soup Company
3  56b352e6-102b-4eff-8296-7e1fb2459bab      Soup Unlimited

      inspection_id  business_location.lon  business_location.lat
0  5794_20160907          -122.481299          37.747228
1  5794_20160907          -122.481299          37.747228
2  24936_20160609          -122.400152          37.793199
3  60354_20161123          -122.409061          37.783527

```

3.1 Search by SQL

```

[14]: wr.opensearch.search_by_sql(
    client,
    sql_query="""SELECT business_name, inspection_score
FROM sf_restaurants_inspections_dedup
WHERE business_name LIKE '%soup%'
ORDER BY inspection_score DESC LIMIT 5"""
)

```

```

[14]:
      _index _type      _id _score \
0  sf_restaurants_inspections_dedup  _doc  5794_20160907      None

```

(continues on next page)

(continued from previous page)

```

1 sf_restaurants_inspections_dedup _doc 60354_20161123 None
2 sf_restaurants_inspections_dedup _doc 24936_20160609 None

           business_name  inspection_score
0           Soup-or-Salad                96
1           Soup Unlimited                95
2 San Francisco Soup Company                77

```

4. Delete Indices

```
[15]: wr.opensearch.delete_index(
        client=client,
        index="sf_restaurants_inspections"
    )
```

```
[15]: {'acknowledged': True}
```

5. Bonus - Prepare data and index from DataFrame

For this exercise we'll use [DOHMH New York City Restaurant Inspection Results](#) dataset

```
[16]: import pandas as pd
```

```
[17]: df = pd.read_csv('https://data.cityofnewyork.us/api/views/43nn-pn8j/rows.csv?
    ↪accessType=DOWNLOAD')
```

Prepare the data for indexing

```
[18]: # fields names underscore casing
df.columns = [col.lower().replace(' ', '_') for col in df.columns]

# convert lon/lat to OpenSearch geo_point
df['business_location'] = "POINT (" + df.longitude.fillna('0').astype(str) + " " + df.
    ↪latitude.fillna('0').astype(str) + ")"
```

Create index with mapping

```
[19]: # delete index if exists
wr.opensearch.delete_index(
    client=client,
    index="nyc_restaurants"
)

# use dynamic_template to map date fields
# define business_location as geo_point
```

(continues on next page)

(continued from previous page)

```

wr.opensearch.create_index(
    client=client,
    index="nyc_restaurants_inspections",
    mappings={
        "dynamic_templates" : [
            {
                "dates" : {
                    "match" : "*date",
                    "mapping" : {
                        "type" : "date",
                        "format" : 'MM/dd/yyyy'
                    }
                }
            }
        ],
        "properties": {
            "business_location": {
                "type": "geo_point"
            }
        }
    }
)

```

```

[19]: {'acknowledged': True,
      'shards_acknowledged': True,
      'index': 'nyc_restaurants_inspections'}

```

Index dataframe

```

[20]: wr.opensearch.index_df(
      client,
      df=df,
      index="nyc_restaurants_inspections",
      id_keys=["camis"],
      bulk_size=1000
    )

```

```
Indexing: 100% (382655/382655)|#####|Elapsed Time: 0:04:15
```

```

[20]: {'success': 382655, 'errors': []}

```

Execute geo query

Sort restaurants by distance from Times-Square

```

[21]: wr.opensearch.search(
      client,
      index="nyc_restaurants_inspections",
      filter_path=["hits.hits._source"],

```

(continues on next page)

(continued from previous page)

```

size=100,
search_body={
    "query": {
        "match_all": {}
    },
    "sort": [
        {
            "_geo_distance": {
                "business_location": { # Times-Square - https://geojson.io/#map=16/40.
↪7563/-73.9862
                "lat": 40.75613228383523,
                "lon": -73.9865791797638
            },
            "order": "asc"
        }
    ]
}
)

```

```

[21]:
      camis      dba      boro building      street \
0  41551304      THE COUNTER  Manhattan      7      TIMES SQUARE
1  50055665      ANN INC CAFE  Manhattan      7      TIMES SQUARE
2  50049552      ERNST AND YOUNG  Manhattan      5      TIMES SQ
3  50014078      RED LOBSTER  Manhattan      5      TIMES SQ
4  50015171  NEW AMSTERDAM THEATER  Manhattan    214  WEST  42 STREET
..      ...      ...      ...      ...      ...
95  41552060      PROSKAUER ROSE  Manhattan     11      TIMES SQUARE
96  41242148      GABBY O'HARA'S  Manhattan    123  WEST  39 STREET
97  50095860      THE TIMES EATERY  Manhattan    680      8 AVENUE
98  50072861      ITSU  Manhattan     530      7 AVENUE
99  50068109      LUKE'S LOBSTER  Manhattan    1407      BROADWAY

      zipcode      phone cuisine_description inspection_date \
0  10036.0  2129976801      American  12/22/2016
1  10036.0  2125413287      American  12/11/2019
2  10036.0  2127739994      Coffee/Tea  11/30/2018
3  10036.0  2127306706      Seafood  10/03/2017
4  10036.0  2125825472      American  06/26/2018
..      ...      ...      ...      ...
95  10036.0  2129695493      American  08/11/2017
96  10018.0  2122788984      Irish  07/30/2019
97  10036.0  6463867787      American  02/28/2020
98  10018.0  9176393645  Asian/Asian Fusion  09/10/2018
99  10018.0  9174759192      Seafood  09/06/2017

      action ... \
0  Violations were cited in the following area(s). ...
1  Violations were cited in the following area(s). ...
2  Violations were cited in the following area(s). ...
3  Violations were cited in the following area(s). ...
4  Violations were cited in the following area(s). ...

```

(continues on next page)

(continued from previous page)

```

..          ...
95 Violations were cited in the following area(s). ...
96 Violations were cited in the following area(s). ...
97 Violations were cited in the following area(s). ...
98 Violations were cited in the following area(s). ...
99 Violations were cited in the following area(s). ...

      inspection_type  latitude  longitude  \
0      Cycle Inspection / Initial Inspection  40.755908 -73.986681
1      Cycle Inspection / Initial Inspection  40.755908 -73.986681
2      Cycle Inspection / Initial Inspection  40.755702 -73.987208
3      Cycle Inspection / Initial Inspection  40.755702 -73.987208
4      Cycle Inspection / Re-inspection      40.756317 -73.987652
..          ...
95 Administrative Miscellaneous / Initial Inspection  40.756891 -73.990023
96      Cycle Inspection / Re-inspection      40.753405 -73.986602
97      Pre-permit (Operational) / Initial Inspection  40.757991 -73.989218
98      Pre-permit (Operational) / Initial Inspection  40.753844 -73.988551
99      Pre-permit (Operational) / Initial Inspection  40.753432 -73.987151

      community_board  council_district  census_tract      bin      bbl  \
0      105.0      3.0      11300.0  1086069.0  1.009940e+09
1      105.0      3.0      11300.0  1086069.0  1.009940e+09
2      105.0      3.0      11300.0  1024656.0  1.010130e+09
3      105.0      3.0      11300.0  1024656.0  1.010130e+09
4      105.0      3.0      11300.0  1024660.0  1.010130e+09
..          ...
95      105.0      3.0      11300.0  1087978.0  1.010138e+09
96      105.0      4.0      11300.0  1080611.0  1.008150e+09
97      105.0      3.0      11900.0  1024703.0  1.010150e+09
98      105.0      3.0      11300.0  1014485.0  1.007880e+09
99      105.0      3.0      11300.0  1015265.0  1.008140e+09

      nta      business_location
0  MN17 POINT (-73.986680953809 40.755907817312)
1  MN17 POINT (-73.986680953809 40.755907817312)
2  MN17 POINT (-73.987207980138 40.755702020307)
3  MN17 POINT (-73.987207980138 40.755702020307)
4  MN17 POINT (-73.987651832547 40.756316895053)
..          ...
95  MN17 POINT (-73.990023200823 40.756890780426)
96  MN17 POINT (-73.986602050292 40.753404587174)
97  MN17 POINT (-73.989218092096 40.757991356019)
98  MN17 POINT (-73.988551029682 40.753843959794)
99  MN17 POINT (-73.98715066791 40.753432097521)

[100 rows x 27 columns]

```



AWS SDK for pandas

1.3.32 32 - AWS Lake Formation - Glue Governed tables

This tutorial assumes that your IAM user/role has the required Lake Formation permissions to create and read AWS Glue Governed tables

Table of Contents

- *1. Read Governed table*
 - *1.1 Read PartiQL query*
 - * *1.1.1 Read within transaction*
 - * *1.1.2 Read within query as of time*
 - *1.2 Read full table*
- *2. Write Governed table*
 - *2.1 Create new Governed table*
 - * *2.1.1 CSV table*
 - * *2.1.2 Parquet table*
 - *2.2 Overwrite operations*
 - * *2.2.1 Overwrite*
 - * *2.2.2 Append*
 - * *2.2.3 Create partitioned Governed table*
 - * *2.2.4 Overwrite partitions*
- *3. Multiple read/write operations within a transaction*

1. Read Governed table

1.1 Read PartiQL query

```
[ ]: import awswrangler as wr

database = "gov_db" # Assumes a Glue database registered with Lake Formation exists in
↳ the account
table = "gov_table" # Assumes a Governed table exists in the account
catalog_id = "111111111111" # AWS Account Id

# Note 1: If a transaction_id is not specified, a new transaction is started
df = wr.lakeformation.read_sql_query(
    sql=f"SELECT * FROM {table};",
    database=database,
    catalog_id=catalog_id
)
```

1.1.1 Read within transaction

```
[ ]: transaction_id = wr.lakeformation.start_transaction(read_only=True)
df = wr.lakeformation.read_sql_query(
    sql=f"SELECT * FROM {table};",
    database=database,
    transaction_id=transaction_id
)
```

1.1.2 Read within query as of time

```
[ ]: import calendar
import time

query_as_of_time = query_as_of_time = calendar.timegm(time.gmtime())
df = wr.lakeformation.read_sql_query(
    sql=f"SELECT * FROM {table} WHERE id=:id; AND name=:name;",
    database=database,
    query_as_of_time=query_as_of_time,
    params={"id": 1, "name": "Ayoub"}
)
```

1.2 Read full table

```
[ ]: df = wr.lakeformation.read_sql_table(  
    table=table,  
    database=database  
)
```

2. Write Governed table

2.1 Create a new Governed table

Enter your bucket name:

```
[ ]: import getpass  
  
bucket = getpass.getpass()
```

If a governed table does not exist, it can be created by passing an S3 path argument. Make sure your IAM user/role has enough permissions in the Lake Formation database

2.1.1 CSV table

```
[ ]: import pandas as pd  
  
table = "gov_table_csv"  
  
df=pd.DataFrame({  
    "col": [1, 2, 3],  
    "col2": ["A", "A", "B"],  
    "col3": [None, "test", None]  
})  
  
# Note 1: If a transaction_id is not specified, a new transaction is started  
# Note 2: When creating a new Governed table, `table_type="GOVERNED"` must be specified.  
↳ Otherwise the default is to create an EXTERNAL_TABLE  
wr.s3.to_csv(  
    df=df,  
    path=f"s3://{bucket}/{database}/{table}/", # S3 path  
    dataset=True,  
    database=database,  
    table=table,  
    table_type="GOVERNED"  
)
```

2.1.2 Parquet table

```
[ ]: table = "gov_table_parquet"

df = pd.DataFrame({"c0": [0, None]}, dtype="Int64")
wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/{database}/{table}/",
    dataset=True,
    database=database,
    table=table,
    table_type="GOVERNED",
    description="c0",
    parameters={"num_cols": str(len(df.columns)), "num_rows": str(len(df.index))},
    columns_comments={"c0": "0"}
)
```

2.2 Overwrite operations

2.2.1 Overwrite

```
[ ]: df = pd.DataFrame({"c1": [None, 1, None]}, dtype="Int16")
wr.s3.to_parquet(
    df=df,
    dataset=True,
    mode="overwrite",
    database=database,
    table=table,
    description="c1",
    parameters={"num_cols": str(len(df.columns)), "num_rows": str(len(df.index))},
    columns_comments={"c1": "1"}
)
```

2.2.2 Append

```
[ ]: df = pd.DataFrame({"c1": [None, 2, None]}, dtype="Int8")
wr.s3.to_parquet(
    df=df,
    dataset=True,
    mode="append",
    database=database,
    table=table,
    description="c1",
    parameters={"num_cols": str(len(df.columns)), "num_rows": str(len(df.index) * 2)},
    columns_comments={"c1": "1"}
)
```

2.2.3 Create partitioned Governed table

```
[ ]: table = "gov_table_parquet_partitioned"

df = pd.DataFrame({"c0": ["foo", None], "c1": [0, 1]})
wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/{database}/{table}/",
    dataset=True,
    database=database,
    table=table,
    table_type="GOVERNED",
    partition_cols=["c1"],
    description="c0+c1",
    parameters={"num_cols": "2", "num_rows": "2"},
    columns_comments={"c0": "zero", "c1": "one"}
)
```

2.2.4 Overwrite partitions

```
[ ]: df = pd.DataFrame({"c0": [None, None], "c1": [0, 2]})
wr.s3.to_parquet(
    df=df,
    dataset=True,
    mode="overwrite_partitions",
    database=database,
    table=table,
    partition_cols=["c1"],
    description="c0+c1",
    parameters={"num_cols": "2", "num_rows": "3"},
    columns_comments={"c0": "zero", "c1": "one"}
)
```

3. Multiple read/write operations within a transaction

```
[ ]: read_table = "gov_table_parquet"
write_table = "gov_table_multi_parquet"

transaction_id = wr.lakeformation.start_transaction(read_only=False)

df = pd.DataFrame({"c0": [0, None]}, dtype="Int64")
wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/{database}/{write_table}_1",
    dataset=True,
    database=database,
    table=f"{write_table}_1",
    table_type="GOVERNED",
    transaction_id=transaction_id,
)
```

(continues on next page)

(continued from previous page)

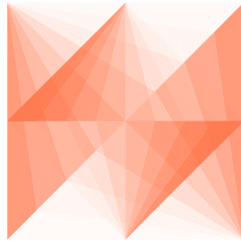
```

df2 = wr.lakeformation.read_sql_table(
    table=read_table,
    database=database,
    transaction_id=transaction_id,
    use_threads=True
)

df3 = pd.DataFrame({"c1": [None, 1, None]}, dtype="Int16")
wr.s3.to_parquet(
    df=df2,
    path=f"s3://{bucket}/{database}/{write_table}_2",
    dataset=True,
    mode="append",
    database=database,
    table=f"{write_table}_2",
    table_type="GOVERNED",
    transaction_id=transaction_id,
)

wr.lakeformation.commit_transaction(transaction_id=transaction_id)

```



AWS SDK for pandas

1.3.33 33 - Amazon Neptune

Note: to be able to use SPARQL you must either install SPARQLWrapper or install AWS SDK for pandas with sparql extra:

```
[ ]: !pip install awswrangler[sparql]
```

Initialize

The first step to using AWS SDK for pandas with Amazon Neptune is to import the library and create a client connection.

Note: Connecting to Amazon Neptune requires that the application you are running has access to the Private VPC where Neptune is located. Without this access you will not be able to connect using AWS SDK for pandas.

```
[ ]: import awswrangler as wr
import pandas as pd

url='<INSERT CLUSTER ENDPOINT>' # The Neptune Cluster endpoint
iam_enabled = False # Set to True/False based on the configuration of your cluster
neptune_port = 8182 # Set to the Neptune Cluster Port, Default is 8182
client = wr.neptune.connect(url, neptune_port, iam_enabled=iam_enabled)
```

Return the status of the cluster

```
[ ]: print(client.status())
```

Retrieve Data from Neptune using AWS SDK for pandas

AWS SDK for pandas supports querying Amazon Neptune using TinkerPop Gremlin and openCypher for property graph data or SPARQL for RDF data.

Gremlin

```
[ ]: query = "g.E().project('source', 'target').by(outV().id()).by(inV().id()).limit(5)"
df = wr.neptune.execute_gremlin(client, query)
display(df.head(5))
```

SPARQL

```
[ ]: query = """
PREFIX foaf: <https://xmlns.com/foaf/0.1/>
PREFIX ex: <https://www.example.com/>
SELECT ?firstName WHERE { ex:JaneDoe foaf:knows ?person . ?person foaf:firstName_
↪?firstName }"""
df = wr.neptune.execute_sparql(client, query)
display(df.head(5))
```

openCypher

```
[ ]: query = "MATCH (n)-[r]->(d) RETURN id(n) as source, id(d) as target LIMIT 5"
df = wr.neptune.execute_opencypher(client, query)
display(df.head(5))
```

Saving Data using AWS SDK for pandas

AWS SDK for pandas supports saving Pandas DataFrames into Amazon Neptune using either a property graph or RDF data model.

Property Graph

If writing to a property graph then DataFrames for vertices and edges must be written separately. DataFrames for vertices must have a `~label` column with the label and a `~id` column for the vertex id.

If the `~id` column does not exist, the specified id does not exist, or is empty then a new vertex will be added.

If no `~label` column exists then writing to the graph will be treated as an update of the element with the specified `~id` value.

DataFrames for edges must have a `~id`, `~label`, `~to`, and `~from` column. If the `~id` column does not exist the specified id does not exist, or is empty then a new edge will be added. If no `~label`, `~to`, or `~from` column exists an exception will be thrown.

Add Vertices/Nodes

```
[ ]: import uuid
import random
import string
def _create_dummy_vertex():
    data = dict()
    data["~id"] = uuid.uuid4()
    data["~label"] = "foo"
    data["int"] = random.randint(0, 1000)
    data["str"] = "".join(random.choice(string.ascii_lowercase) for i in range(10))
    data["list"] = [random.randint(0, 1000), random.randint(0, 1000)]
    return data

data = [_create_dummy_vertex(), _create_dummy_vertex(), _create_dummy_vertex()]
df = pd.DataFrame(data)
res = wr.neptune.to_property_graph(client, df)
query = f"MATCH (s) WHERE id(s)='{data[0]['~id']}' RETURN s"
df = wr.neptune.execute_opencypher(client, query)
display(df)
```

Add Edges

```
[ ]: import uuid
import random
import string
def _create_dummy_edge():
    data = dict()
    data["~id"] = uuid.uuid4()
    data["~label"] = "bar"
    data["~to"] = uuid.uuid4()
    data["~from"] = uuid.uuid4()
    data["int"] = random.randint(0, 1000)
    data["str"] = "".join(random.choice(string.ascii_lowercase) for i in range(10))
    return data

data = [_create_dummy_edge(), _create_dummy_edge(), _create_dummy_edge()]
df = pd.DataFrame(data)
res = wr.neptune.to_property_graph(client, df)
query = f"MATCH (s)-[r]->(d) WHERE id(r)='{data[0]['~id']}' RETURN r"
df = wr.neptune.execute_opencypher(client, query)
display(df)
```

Update Existing Nodes

```
[ ]: idval=uuid.uuid4()
wr.neptune.execute_gremlin(client, f"g.addV().property(T.id, '{str(idval)}')")
query = f"MATCH (s) WHERE id(s)='{idval}' RETURN s"
df = wr.neptune.execute_opencypher(client, query)
print("Before")
display(df)
data = [{"~id": idval, "age": 50}]
df = pd.DataFrame(data)
res = wr.neptune.to_property_graph(client, df)
df = wr.neptune.execute_opencypher(client, query)
print("After")
display(df)
```

Setting cardinality based on the header

If you would like to save data using single cardinality then you can postfix (single) to the column header and set `use_header_cardinality=True` (default). e.g. A column named `name(single)` will save the name property as single cardinality. You can disable this by setting `use_header_cardinality=False`.

```
[ ]: data = [_create_dummy_vertex()]
df = pd.DataFrame(data)
# Adding (single) to the column name in the DataFrame will cause it to write that_
# property as `single` cardinality
df.rename(columns={"int": "int(single)"}, inplace=True)
res = wr.neptune.to_property_graph(client, df, use_header_cardinality=True)
```

(continues on next page)

(continued from previous page)

```
# This can be disabled by setting `use_header_cardinality = False`
df.rename(columns={"int": "int(single)"}, inplace=True)
res = wr.neptune.to_property_graph(client, df, use_header_cardinality=False)
```

RDF

The DataFrame must consist of triples with column names for the subject, predicate, and object specified. If none are provided then s, p, and o are the default.

If you want to add data into a named graph then you will also need the graph column, default is g.

Write Triples

```
[ ]: def _create_dummy_triple():
    data = dict()
    data["s"] = "http://example.com/resources/foo"
    data["p"] = uuid.uuid4()
    data["o"] = random.randint(0, 1000)
    return data

data = [_create_dummy_triple(), _create_dummy_triple(), _create_dummy_triple()]
df = pd.DataFrame(data)
res = wr.neptune.to_rdf_graph(client, df)
query = """
    PREFIX foo: <http://example.com/resources/>
    SELECT ?o WHERE { <foo:foo> <" + str(data[0]['p']) + "> ?o .}"""
df = wr.neptune.execute_sparql(client, query)
display(df)
```

Write Quads

```
[ ]: def _create_dummy_quad():
    data = _create_dummy_triple()
    data["g"] = "bar"
    return data

data = [_create_dummy_quad(), _create_dummy_quad(), _create_dummy_quad()]
df = pd.DataFrame(data)
res = wr.neptune.to_rdf_graph(client, df)
query = """
    PREFIX foo: <http://example.com/resources/>
    SELECT ?o WHERE { <foo:foo> <" + str(data[0]['p']) + "> ?o .}"""
df = wr.neptune.execute_sparql(client, query)
display(df)
```

Flatten DataFrames

One of the complexities of working with a row/columns paradigm, such as Pandas, with graph results set is that it is very common for graph results to return complex and nested objects. To help simplify using the results returned from a graph within a more tabular format we have added a method to flatten the returned Pandas DataFrame.

Flattening the DataFrame

```
[ ]: client = wr.neptune.connect(url, 8182, iam_enabled=False)
      query = "MATCH (n) RETURN n LIMIT 1"
      df = wr.neptune.execute_opencypher(client, query)
      print("Original")
      display(df)
      df_new=wr.neptune.flatten_nested_df(df)
      print("Flattened")
      display(df_new)
```

Removing the prefixing of the parent column name

```
[ ]: df_new=wr.neptune.flatten_nested_df(df, include_prefix=False)
      display(df_new)
```

Specifying the column header seperator

```
[ ]: df_new=wr.neptune.flatten_nested_df(df, seperator='|')
      display(df_new)
```

Putting it into a workflow

```
[ ]: pip install igraph networkx
```

Running PageRank using NetworkX

```
[ ]: import networkx as nx

      # Retrieve Data from neptune
      client = wr.neptune.connect(url, 8182, iam_enabled=False)
      query = "MATCH (n)-[r]->(d) RETURN id(n) as source, id(d) as target LIMIT 100"
      df = wr.neptune.execute_opencypher(client, query)

      # Run PageRank
      G=nx.from_pandas_edgelist(df, edge_attr=True)
      pg = nx.pagerank(G)

      # Save values back into Neptune
```

(continues on next page)

(continued from previous page)

```

rows=[]
for k in pg.keys():
    rows.append({'~id': k, 'pageRank_nx(single)': pg[k]})
pg_df=pd.DataFrame(rows, columns=['~id','pageRank_nx(single)'])
res = wr.neptune.to_property_graph(client, pg_df, use_header_cardinality=True)

# Retrieve newly saved data
query = "MATCH (n:airport) WHERE n.pageRank_nx IS NOT NULL RETURN n.code, n.pageRank_nx_
↳ORDER BY n.pageRank_nx DESC LIMIT 5"
df = wr.neptune.execute_opencypher(client, query)
display(df)

```

Running PageRank using iGraph

```

[ ]: import igraph as ig

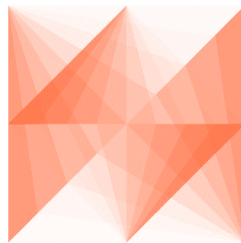
# Retrieve Data from neptune
client = wr.neptune.connect(url, 8182, iam_enabled=False)
query = "MATCH (n)-[r]->(d) RETURN id(n) as source, id(d) as target LIMIT 100"
df = wr.neptune.execute_opencypher(client, query)

# Run PageRank
g = ig.Graph.TupleList(df.itertuples(index=False), directed=True, weights=False)
pg = g.pagerank()

# Save values back into Neptune
rows=[]
for idx, v in enumerate(g.vs):
    rows.append({'~id': v['name'], 'pageRank_ig(single)': pg[idx]})
pg_df=pd.DataFrame(rows, columns=['~id','pageRank_ig(single)'])
res = wr.neptune.to_property_graph(client, pg_df, use_header_cardinality=True)

# Retrieve newly saved data
query = "MATCH (n:airport) WHERE n.pageRank_ig IS NOT NULL RETURN n.code, n.pageRank_ig_
↳ORDER BY n.pageRank_ig DESC LIMIT 5"
df = wr.neptune.execute_opencypher(client, query)
display(df)

```



AWS SDK for pandas

1.3.34 34 - Glue Data Quality

AWS Glue Data Quality helps you evaluate and monitor the quality of your data.

Create test data

First, let's start by creating test data, writing it to S3, and registering it in the Glue Data Catalog.

```
[ ]: import awswrangler as wr
import pandas as pd

glue_database = "aws_sdk_pandas"
glue_table = "my_glue_table"
path = "s3://BUCKET_NAME/my_glue_table/"

df = pd.DataFrame({"c0": [0, 1, 2], "c1": [0, 1, 2], "c2": [0, 0, 0]})
wr.s3.to_parquet(df, path, dataset=True, database=glue_database, table=glue_table,
partition_cols=["c2"])
```

Start with recommended data quality rules

AWS Glue Data Quality can recommend a set of data quality rules so you can get started quickly.

Note: Running Glue Data Quality recommendation and evaluation tasks requires an IAM role. This role must trust the Glue principal and allow permissions to various resources including the Glue table and the S3 bucket where your data is stored. Moreover, data quality IAM actions must be granted. To find out more, check [Authorization](#).

```
[7]: first_ruleset = "ruleset_1"
iam_role_arn = "arn:aws:iam:...:" # IAM role assumed by the Glue Data Quality job to
access resources

df_recommended_ruleset = wr.data_quality.create_recommendation_ruleset( # Creates a
recommended ruleset
name=first_ruleset,
database=glue_database,
table=glue_table,
iam_role_arn=iam_role_arn,
number_of_workers=2,
)

df_recommended_ruleset
```

```
[7]:
```

	rule_type	parameter	expression
0	RowCount	None	between 1 and 6
1	IsComplete	"c0"	None
2	Uniqueness	"c0"	> 0.95
3	ColumnValues	"c0"	<= 2
4	IsComplete	"c1"	None
5	Uniqueness	"c1"	> 0.95
6	ColumnValues	"c1"	<= 2
7	IsComplete	"c2"	None
8	ColumnValues	"c2"	in ["0"]

Update the recommended rules

Recommended rulesets are not perfect and you are likely to modify them or create your own.

```
[17]: # Append and update rules
df_updated_ruleset = df_recommended_ruleset.append(
    {"rule_type": "Uniqueness", "parameter": '"c2"', "expression": "> 0.95"}, ignore_
    ↪index=True
)

df_updated_ruleset.at[8, "expression"] = "in [0, 1, 2]"

# Update the existing ruleset (upsert)
wr.data_quality.update_ruleset(
    name=first_ruleset,
    df_rules=df_updated_ruleset,
    mode="upsert", # update existing or insert new rules to the ruleset
)

wr.data_quality.get_ruleset(name=first_ruleset)
```

```
[17]:
```

	rule_type	parameter	expression
0	RowCount	None	between 1 and 6
1	IsComplete	"c0"	None
2	Uniqueness	"c0"	> 0.95
3	ColumnValues	"c0"	<= 2
4	IsComplete	"c1"	None
5	Uniqueness	"c1"	> 0.95
6	ColumnValues	"c1"	<= 2
7	IsComplete	"c2"	None
8	ColumnValues	"c2"	in [0, 1, 2]
9	Uniqueness	"c2"	> 0.95

Run a data quality task

The ruleset can now be evaluated against the data. A cluster with 2 workers is used for the run. It returns a report with PASS/FAIL results for each rule.

```
[20]: wr.data_quality.evaluate_ruleset(
    name=first_ruleset,
    iam_role_arn=iam_role_arn,
    number_of_workers=2,
)
```

```
[20]:
```

	Name	Description	Result	\
0	Rule_1	RowCount between 1 and 6	PASS	
1	Rule_2	IsComplete "c0"	PASS	
2	Rule_3	Uniqueness "c0" > 0.95	PASS	
3	Rule_4	ColumnValues "c0" <= 2	PASS	
4	Rule_5	IsComplete "c1"	PASS	
5	Rule_6	Uniqueness "c1" > 0.95	PASS	
6	Rule_7	ColumnValues "c1" <= 2	PASS	
7	Rule_8	IsComplete "c2"	PASS	

(continues on next page)

(continued from previous page)

```

8 Rule_9 ColumnValues "c2" in [0,1,2] PASS
9 Rule_10 Uniqueness "c2" > 0.95 FAIL

                                ResultId \
0 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
1 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
2 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
3 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
4 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
5 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
6 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
7 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
8 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5
9 dqresult-be413b527c0e5520ad843323fec9cf2e2edbdd5

                                EvaluationMessage
0
1
2
3
4
5
6
7
8
9 Value: 0.0 does not meet the constraint requir...
```

Create ruleset from Data Quality Definition Language definition

The Data Quality Definition Language (DQDL) is a domain specific language that you can use to define Data Quality rules. For the full syntax reference, see [DQDL](#).

```
[21]: second_ruleset = "ruleset_2"

dqml_rules = (
    "Rules = ["
    "RowCount between 1 and 6,"
    'IsComplete "c0",'
    'Uniqueness "c0" > 0.95,'
    'ColumnValues "c0" <= 2,'
    'IsComplete "c1",'
    'Uniqueness "c1" > 0.95,'
    'ColumnValues "c1" <= 2,'
    'IsComplete "c2",'
    'ColumnValues "c2" <= 1'
    "]"
)

wr.data_quality.create_ruleset(
    name=second_ruleset,
    database=glue_database,
```

(continues on next page)

(continued from previous page)

```

    table=glue_table,
    dqdl_rules=dqdl_rules,
)

```

Create or update a ruleset from a data frame

AWS SDK for pandas also enables you to create or update a ruleset from a pandas data frame.

```

[24]: third_ruleset = "ruleset_3"

df_rules = pd.DataFrame({
    "rule_type": ["RowCount", "ColumnCorrelation", "Uniqueness"],
    "parameter": [None, '"c0" "c1"', '"c0"'],
    "expression": ["between 2 and 8", "> 0.8", "> 0.95"],
})

wr.data_quality.create_ruleset(
    name=third_ruleset,
    df_rules=df_rules,
    database=glue_database,
    table=glue_table,
)

wr.data_quality.get_ruleset(name=third_ruleset)

```

```

[24]:
   rule_type  parameter  expression
0      RowCount      None  between 2 and 8
1  ColumnCorrelation  "c0" "c1"      > 0.8
2      Uniqueness      "c0"      > 0.95

```

Get multiple rulesets

```

[25]: wr.data_quality.get_ruleset(name=[first_ruleset, second_ruleset, third_ruleset])

```

```

[25]:
   rule_type  parameter  expression  ruleset
0      RowCount      None  between 1 and 6  ruleset_1
1      IsComplete  "c0"      None  ruleset_1
2      Uniqueness  "c0"      > 0.95  ruleset_1
3      ColumnValues  "c0"      <= 2  ruleset_1
4      IsComplete  "c1"      None  ruleset_1
5      Uniqueness  "c1"      > 0.95  ruleset_1
6      ColumnValues  "c1"      <= 2  ruleset_1
7      IsComplete  "c2"      None  ruleset_1
8      ColumnValues  "c2"  in [0, 1, 2]  ruleset_1
9      Uniqueness  "c2"      > 0.95  ruleset_1
0      RowCount      None  between 1 and 6  ruleset_2
1      IsComplete  "c0"      None  ruleset_2
2      Uniqueness  "c0"      > 0.95  ruleset_2
3      ColumnValues  "c0"      <= 2  ruleset_2
4      IsComplete  "c1"      None  ruleset_2

```

(continues on next page)

(continued from previous page)

5	Uniqueness	"c1"	> 0.95	ruleset_2
6	ColumnValues	"c1"	<= 2	ruleset_2
7	IsComplete	"c2"	None	ruleset_2
8	ColumnValues	"c2"	<= 1	ruleset_2
0	RowCount	None	between 2 and 8	ruleset_3
1	ColumnCorrelation	"c0" "c1"	> 0.8	ruleset_3
2	Uniqueness	"c0"	> 0.95	ruleset_3

Evaluate Data Quality for a given partition

A data quality evaluation run can be limited to specific partition(s) by leveraging the pushDownPredicate expression in the additional_options argument

```
[26]: df = pd.DataFrame({"c0": [2, 0, 1], "c1": [1, 0, 2], "c2": [1, 1, 1]})
wr.s3.to_parquet(df, path, dataset=True, database=glue_database, table=glue_table,
partition_cols=["c2"])

wr.data_quality.evaluate_ruleset(
    name=third_ruleset,
    iam_role_arn=iam_role_arn,
    number_of_workers=2,
    additional_options={
        "pushDownPredicate": "(c2 == '1')",
    },
)
```

```
[26]:      Name      Description Result \
0 Rule_1      RowCount between 2 and 8 PASS
1 Rule_2  ColumnCorrelation "c0" "c1" > 0.8 FAIL
2 Rule_3      Uniqueness "c0" > 0.95 PASS

      ResultId \
0 dqresult-f676cfe0345aa93f492e3e3c3d6cf1ad99b84dc6
1 dqresult-f676cfe0345aa93f492e3e3c3d6cf1ad99b84dc6
2 dqresult-f676cfe0345aa93f492e3e3c3d6cf1ad99b84dc6

      EvaluationMessage
0 NaN
1 Value: 0.5 does not meet the constraint requir...
2 NaN
```



AWS SDK for pandas

1.3.35 35 - OpenSearch Serverless

Amazon OpenSearch Serverless is an on-demand serverless configuration for Amazon OpenSearch Service.

Create collection

A collection in Amazon OpenSearch Serverless is a logical grouping of one or more indexes that represent an analytics workload.

Collections must have an assigned encryption policy, network policy, and a matching data access policy that grants permission to its resources.

```
[1]: import awswrangler as wr

[8]: data_access_policy = [
    {
        "Rules": [
            {
                "ResourceType": "index",
                "Resource": [
                    "index/my-collection/*",
                ],
                "Permission": [
                    "aoss:*",
                ],
            },
            {
                "ResourceType": "collection",
                "Resource": [
                    "collection/my-collection",
                ],
                "Permission": [
                    "aoss:*",
                ],
            },
        ],
        "Principal": [
            wr.sts.get_current_identity_arn(),
        ],
    },
]
```

(continues on next page)

(continued from previous page)

```
}  
]
```

AWS SDK for pandas can create default network and encryption policies based on the user input.

By default, the network policy allows public access to the collection, and the encryption policy encrypts the collection using AWS-managed KMS key.

Create a collection, and a corresponding data, network, and access policies:

```
[10]: collection = wr.opensearch.create_collection(  
        name="my-collection",  
        data_policy=data_access_policy,  
    )  
  
collection_endpoint = collection["collectionEndpoint"]
```

The call will wait and exit when the collection and corresponding policies are created and active.

To create a collection encrypted with customer KMS key, and attached to a VPC, provide KMS Key ARN and / or VPC endpoints:

```
[ ]: kms_key_arn = "arn:aws:kms:..."  
    vpc_endpoint = "vpce-..."  
  
collection = wr.opensearch.create_collection(  
    name="my-secure-collection",  
    data_policy=data_access_policy,  
    kms_key_arn=kms_key_arn,  
    vpc_endpoints=[vpc_endpoint],  
)
```

Connect

Connect to the collection endpoint:

```
[12]: client = wr.opensearch.connect(host=collection_endpoint)
```

Create index

To create an index, run:

```
[13]: index="my-index-1"  
  
wr.opensearch.create_index(  
    client=client,  
    index=index,  
)  
  
[13]: {'acknowledged': True, 'shards_acknowledged': True, 'index': 'my-index-1'}
```

Index documents

To index documents:

```
[25]: wr.opensearch.index_documents(
        client,
        documents=[{"_id": "1", "name": "John"}, {"_id": "2", "name": "George"}, {"_id": "3",
        ↪ "name": "Julia"}],
        index=index,
    )
```

```
Indexing: 100% (3/3)|#####|Elapsed Time: 0:00:12
```

```
[25]: {'success': 3, 'errors': []}
```

It is also possible to index Pandas data frames:

```
[26]: import pandas as pd

df = pd.DataFrame(
    [{"_id": "1", "name": "John", "tags": ["foo", "bar"]}, {"_id": "2", "name": "George",
    ↪ "tags": ["foo"]}])
```

```
wr.opensearch.index_df(
    client,
    df=df,
    index="index-df",
)
```

```
Indexing: 100% (2/2)|#####|Elapsed Time: 0:00:12
```

```
[26]: {'success': 2, 'errors': []}
```

AWS SDK for pandas also supports indexing JSON and CSV documents.

For more examples, refer to the [031 - OpenSearch tutorial](#)

Search

Search using search DSL:

```
[27]: wr.opensearch.search(
        client,
        index=index,
        search_body={
            "query": {
                "match": {
                    "name": "Julia"
                }
            }
        }
    )
```

```
[27]:  _id  name
0     3  Julia
```

Delete index

To delete an index, run:

```
[ ]: wr.opensearch.delete_index(  
    client=client,  
    index=index  
)
```

1.4 API Reference

- *Amazon S3*
- *AWS Glue Catalog*
- *Amazon Athena*
- *AWS Lake Formation*
- *Amazon Redshift*
- *PostgreSQL*
- *MySQL*
- *Microsoft SQL Server*
- *Oracle*
- *Data API Redshift*
- *Data API RDS*
- *AWS Glue Data Quality*
- *OpenSearch*
- *Amazon Neptune*
- *DynamoDB*
- *Amazon Timestream*
- *Amazon EMR*
- *Amazon CloudWatch Logs*
- *Amazon QuickSight*
- *AWS STS*
- *AWS Secrets Manager*
- *Amazon Chime*
- *Global Configurations*

1.4.1 Amazon S3

<code>copy_objects(paths, source_path, target_path)</code>	Copy a list of S3 objects to another S3 directory.
<code>delete_objects(path[, use_threads, ...])</code>	Delete Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
<code>describe_objects(path[, version_id, ...])</code>	Describe Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
<code>does_object_exist(path[, ...])</code>	Check if object exists on S3.
<code>download(path, local_file[, version_id, ...])</code>	Download file from a received S3 path to local file.
<code>get_bucket_region(bucket[, boto3_session])</code>	Get bucket region name.
<code>list_buckets([boto3_session])</code>	List Amazon S3 buckets.
<code>list_directories(path[, chunked, ...])</code>	List Amazon S3 objects from a prefix.
<code>list_objects(path[, suffix, ignore_suffix, ...])</code>	List Amazon S3 objects from a prefix.
<code>merge_datasets(source_path, target_path[, ...])</code>	Merge a source dataset into a target dataset.
<code>merge_upsert_table(delta_df, database, ...)</code>	Perform Upsert (Update else Insert) onto an existing Glue table.
<code>read_csv(path[, path_suffix, ...])</code>	Read CSV file(s) from a received S3 prefix or list of S3 objects paths.
<code>read_deltalake([path, version, partitions, ...])</code>	Load a Deltalake table data from an S3 path.
<code>read_excel(path[, version_id, use_threads, ...])</code>	Read EXCEL file(s) from a received S3 path.
<code>read_fwf(path[, path_suffix, ...])</code>	Read fixed-width formatted file(s) from a received S3 prefix or list of S3 objects paths.
<code>read_json(path[, path_suffix, ...])</code>	Read JSON file(s) from a received S3 prefix or list of S3 objects paths.
<code>read_parquet(path[, path_root, path_suffix, ...])</code>	Read Apache Parquet file(s) from a received S3 prefix or list of S3 objects paths.
<code>read_parquet_metadata(path[, version_id, ...])</code>	Read Apache Parquet file(s) metadata from a received S3 prefix or list of S3 objects paths.
<code>read_parquet_table(table, database[, ...])</code>	Read Apache Parquet table registered on AWS Glue Catalog.
<code>select_query(sql, path, input_serialization, ...)</code>	Filter contents of an Amazon S3 object based on SQL statement.
<code>size_objects(path[, version_id, ...])</code>	Get the size (ContentLength) in bytes of Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
<code>store_parquet_metadata(path, database, table)</code>	Infer and store parquet metadata on AWS Glue Catalog.
<code>to_csv(df[, path, sep, index, columns, ...])</code>	Write CSV file or dataset on Amazon S3.
<code>to_excel(df, path[, boto3_session, ...])</code>	Write EXCEL file on Amazon S3.
<code>to_json(df[, path, index, columns, ...])</code>	Write JSON file on Amazon S3.
<code>to_parquet(df[, path, index, compression, ...])</code>	Write Parquet file or dataset on Amazon S3.
<code>upload(local_file, path[, use_threads, ...])</code>	Upload file from a local file to received S3 path.
<code>wait_objects_exist(paths[, delay, ...])</code>	Wait Amazon S3 objects exist.
<code>wait_objects_not_exist(paths[, delay, ...])</code>	Wait Amazon S3 objects not exist.

awswrangler.s3.copy_objects

`awswrangler.s3.copy_objects(paths: List[str], source_path: str, target_path: str, replace_filenames: Dict[str, str] | None = None, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None) → List[str]`

Copy a list of S3 objects to another S3 directory.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **paths** (`List[str]`) – List of S3 objects paths (e.g. `[s3://bucket/dir0/key0, s3://bucket/dir0/key1]`).
- **source_path** (`str`,) – S3 Path for the source directory.
- **target_path** (`str`,) – S3 Path for the target directory.
- **replace_filenames** (`Dict[str, str]`, *optional*) – e.g. `{“old_name.csv”: “new_name.csv”, “old_name2.csv”: “new_name2.csv”}`
- **use_threads** (`bool`, `int`) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (`boto3.Session()`, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (`Optional[Dict[str, Any]]`) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={ ‘ServerSideEncryption’: ‘aws:kms’, ‘SSEKMSKeyId’: ‘YOUR_KMS_KEY_ARN’}`

Returns

List of new objects paths.

Return type

List[str]

Examples

Copying

```
>>> import awswrangler as wr
>>> wr.s3.copy_objects(
...     paths=["s3://bucket0/dir0/key0", "s3://bucket0/dir0/key1"],
...     source_path="s3://bucket0/dir0/",
...     target_path="s3://bucket1/dir1/"
... )
["s3://bucket1/dir1/key0", "s3://bucket1/dir1/key1"]
```

Copying with a KMS key

```
>>> import awswrangler as wr
>>> wr.s3.copy_objects(
```

(continues on next page)

(continued from previous page)

```

...     paths=["s3://bucket0/dir0/key0", "s3://bucket0/dir0/key1"],
...     source_path="s3://bucket0/dir0/",
...     target_path="s3://bucket1/dir1/",
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
["s3://bucket1/dir1/key0", "s3://bucket1/dir1/key1"]

```

aws wrangler.s3.delete_objects

`aws wrangler.s3.delete_objects(path: str | List[str], use_threads: bool | int = True, last_modified_begin: datetime | None = None, last_modified_end: datetime | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, boto3_session: Session | None = None) → None`

Delete Amazon S3 objects from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: The filter by `last_modified begin last_modified end` is applied after list all S3 files

Parameters

- **path** (`Union[str, List[str]]`) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **use_threads** (`bool, int`) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (`datetime, optional`) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **s3_additional_kwargs** (`Optional[Dict[str, Any]]`) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'RequestPayer': 'requester'}`
- **boto3_session** (`boto3.Session(), optional`) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.delete_objects(['s3://bucket/key0', 's3://bucket/key1']) # Delete both
↪objects
>>> wr.s3.delete_objects('s3://bucket/prefix') # Delete all objects under the
↪received prefix
```

awswrangler.s3.describe_objects

`awswrangler.s3.describe_objects`(*path*: *str* | *List[str]*, *version_id*: *str* | *Dict[str, str]* | *None* = *None*, *use_threads*: *bool* | *int* = *True*, *last_modified_begin*: *datetime* | *None* = *None*, *last_modified_end*: *datetime* | *None* = *None*, *s3_additional_kwargs*: *Dict[str, Any]* | *None* = *None*, *boto3_session*: *Session* | *None* = *None*) → *Dict[str, Dict[str, Any]]*

Describe Amazon S3 objects from a received S3 prefix or list of S3 objects paths.

Fetch attributes like `ContentLength`, `DeleteMarker`, `last_modified`, `ContentType`, etc The full list of attributes can be explored under the boto3 `head_object` documentation: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.head_object

This function accepts Unix shell-style wildcards in the path argument. `*` (matches everything), `?` (matches any single character), `[seq]` (matches any character in seq), `[!seq]` (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (`*`, `?`, `[]`), you can use `glob.escape(path)` before passing the path to this function.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: The filter by `last_modified` begin `last_modified` end is applied after list all S3 files

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. `{ 's3://bucket/key0': '121212', 's3://bucket/key1': '343434' }`)
- **use_threads** (*bool, int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.

- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'RequestPayer': 'requester'}`
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.

Returns

Return a dictionary of objects returned from `head_objects` where the key is the object path. The response object can be explored here: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.head_object

Return type

`Dict[str, Dict[str, Any]]`

Examples

```
>>> import awswrangler as wr
>>> desc0 = wr.s3.describe_objects(['s3://bucket/key0', 's3://bucket/key1']) # Describe both objects
>>> desc1 = wr.s3.describe_objects('s3://bucket/prefix') # Describe all objects under the prefix
```

awswrangler.s3.does_object_exist

`awswrangler.s3.does_object_exist(path: str, s3_additional_kwargs: Dict[str, Any] | None = None, boto3_session: Session | None = None, version_id: str | None = None) → bool`

Check if object exists on S3.

Parameters

- **path** (*str*) – S3 path (e.g. `s3://bucket/key`).
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'RequestPayer': 'requester'}`
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.
- **version_id** (*str*, *optional*) – Specific version of the object that should exist.

Returns

True if exists, False otherwise.

Return type

`bool`

Examples

Using the default boto3 session

```
>>> import awswrangler as wr
>>> wr.s3.does_object_exist('s3://bucket/key_real')
True
>>> wr.s3.does_object_exist('s3://bucket/key_unreal')
False
```

Using a custom boto3 session

```
>>> import boto3
>>> import awswrangler as wr
>>> wr.s3.does_object_exist('s3://bucket/key_real', boto3_session=boto3.Session())
True
>>> wr.s3.does_object_exist('s3://bucket/key_unreal', boto3_session=boto3.Session())
False
```

awswrangler.s3.download

`awswrangler.s3.download(path: str, local_file: str | Any, version_id: str | None = None, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None) → None`

Download file from a received S3 path to local file.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **path** (*str*) – S3 path (e.g. `s3://bucket/key0`).
- **local_file** (*Union[str, Any]*) – A file-like object in binary mode or a path to local file (e.g. `./local/path/to/key0`).
- **version_id** (*Optional[str]*) – Version id of the object.
- **use_threads** (*bool, int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm”, “SSECustomerKey” and “RequestPayer” arguments will be considered.

Return type

None

Examples

Downloading a file using a path to local file

```
>>> import awswrangler as wr
>>> wr.s3.download(path='s3://bucket/key', local_file='./key')
```

Downloading a file using a file-like object

```
>>> import awswrangler as wr
>>> with open(file='./key', mode='wb') as local_f:
>>>     wr.s3.download(path='s3://bucket/key', local_file=local_f)
```

awswrangler.s3.get_bucket_region

`awswrangler.s3.get_bucket_region(bucket: str, boto3_session: Session | None = None) → str`

Get bucket region name.

Parameters

- **bucket** (*str*) – Bucket name.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Region code (e.g. 'us-east-1').

Return type

str

Examples

Using the default boto3 session

```
>>> import awswrangler as wr
>>> region = wr.s3.get_bucket_region('bucket-name')
```

Using a custom boto3 session

```
>>> import boto3
>>> import awswrangler as wr
>>> region = wr.s3.get_bucket_region('bucket-name', boto3_session=boto3.Session())
```

awswrangler.s3.list_buckets

`awswrangler.s3.list_buckets(boto3_session: Session | None = None) → List[str]`

List Amazon S3 buckets.

Parameters

boto3_session (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session to use, default to None.

Returns

List of bucket names.

Return type

List[str]

awswrangler.s3.list_directories

```
awswrangler.s3.list_directories(path: str, chunked: bool = False, s3_additional_kwargs: Dict[str, Any] |  
                                None = None, boto3_session: Session | None = None) → List[str] |  
                                Iterator[List[str]]
```

List Amazon S3 objects from a prefix.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Parameters

- **path** (str) – S3 path (e.g. s3://bucket/prefix).
- **chunked** (bool) – If True returns iterator, and a single list otherwise. False by default.
- **s3_additional_kwargs** (Optional[Dict[str, Any]]) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

List of objects paths.

Return type

Union[List[str], Iterator[List[str]]]

Examples

Using the default boto3 session

```
>>> import awswrangler as wr  
>>> wr.s3.list_directories('s3://bucket/prefix/')  
['s3://bucket/prefix/dir0/', 's3://bucket/prefix/dir1/', 's3://bucket/prefix/dir2/']
```

Using a custom boto3 session

```
>>> import boto3  
>>> import awswrangler as wr  
>>> wr.s3.list_directories('s3://bucket/prefix/', boto3_session=boto3.Session())  
['s3://bucket/prefix/dir0/', 's3://bucket/prefix/dir1/', 's3://bucket/prefix/dir2/']
```


awswrangler.s3.list_objects

```
awswrangler.s3.list_objects(path: str, suffix: str | List[str] | None = None, ignore_suffix: str | List[str] | None
                             = None, last_modified_begin: datetime | None = None, last_modified_end:
                             datetime | None = None, ignore_empty: bool = False, chunked: bool = False,
                             s3_additional_kwargs: Dict[str, Any] | None = None, boto3_session: Session |
                             None = None) → List[str] | Iterator[List[str]]
```

List Amazon S3 objects from a prefix.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

Parameters

- **path** (*str*) – S3 path (e.g. s3://bucket/prefix).
- **suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for filtering S3 keys.
- **ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **chunked** (*bool*) – If True returns iterator, and a single list otherwise. False by default.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

List of objects paths.

Return type

Union[List[str], Iterator[List[str]]]

Examples

Using the default boto3 session

```
>>> import awswrangler as wr
>>> wr.s3.list_objects('s3://bucket/prefix')
['s3://bucket/prefix0', 's3://bucket/prefix1', 's3://bucket/prefix2']
```

Using a custom boto3 session

```
>>> import boto3
>>> import awswrangler as wr
>>> wr.s3.list_objects('s3://bucket/prefix', boto3_session=boto3.Session())
['s3://bucket/prefix0', 's3://bucket/prefix1', 's3://bucket/prefix2']
```

awswrangler.s3.merge_datasets

`awswrangler.s3.merge_datasets(source_path: str, target_path: str, mode: str = 'append', ignore_empty: bool = False, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None) → List[str]`

Merge a source dataset into a target dataset.

This function accepts Unix shell-style wildcards in the `source_path` argument. `*` (matches everything), `?` (matches any single character), `[seq]` (matches any character in seq), `[!seq]` (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (`*`, `?`, `[]`), you can use `glob.escape(source_path)` before passing the path to this function.

Note: If you are merging tables (S3 datasets + Glue Catalog metadata), remember that you will also need to update your partitions metadata in some cases. (e.g. `wr.athena.repair_table(table='...', database='...')`)

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **source_path** (*str*,) – S3 Path for the source directory.
- **target_path** (*str*,) – S3 Path for the target directory.
- **mode** (*str*, optional) – append (Default), overwrite, overwrite_partitions.
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session()*, optional) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`

Returns

List of new objects paths.

Return type

List[str]

Examples

Merging

```
>>> import awswrangler as wr
>>> wr.s3.merge_datasets(
...     source_path="s3://bucket0/dir0/",
...     target_path="s3://bucket1/dir1/",
...     mode="append"
... )
["s3://bucket1/dir1/key0", "s3://bucket1/dir1/key1"]
```

Merging with a KMS key

```
>>> import awswrangler as wr
>>> wr.s3.merge_datasets(
...     source_path="s3://bucket0/dir0/",
...     target_path="s3://bucket1/dir1/",
...     mode="append",
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
["s3://bucket1/dir1/key0", "s3://bucket1/dir1/key1"]
```

awswrangler.s3.merge_upsert_table

`awswrangler.s3.merge_upsert_table(delta_df: DataFrame, database: str, table: str, primary_key: List[str], boto3_session: Session | None = None) → None`

Perform Upsert (Update else Insert) onto an existing Glue table.

Warning: This API is deprecated and will be removed in future AWS SDK for Pandas releases.

Parameters

- **delta_df** (*pandas.DataFrame*) – The delta dataframe has all the data which needs to be merged on the primary key
- **database** (*Str*) – An existing database name
- **table** (*Str*) – An existing table name
- **primary_key** (*List[str]*) – Pass the primary key as a List of string columns List['column_a', 'column_b']
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type

None

Examples

```
Reading all Parquet files under a prefix >>> import awswrangler as wr >>> import pandas as pd >>>
delta_df = pd.DataFrame({"id": [1], "cchar": ["foo"], "date": [datetime.date(2021, 1, 2)]}) >>> primary_key
= ["id", "cchar"] >>> wr.s3.merge_upsert_table(delta_df=delta_df, database='database', table='table', pri-
mary_key=primary_key)
```

awswrangler.s3.read_csv

```
awswrangler.s3.read_csv(path: str | List[str], path_suffix: str | List[str] | None = None, path_ignore_suffix: str
| List[str] | None = None, version_id: str | Dict[str, str] | None = None, ignore_empty:
bool = True, use_threads: bool | int = True, last_modified_begin: datetime | None =
None, last_modified_end: datetime | None = None, boto3_session: Session | None =
None, s3_additional_kwargs: Dict[str, Any] | None = None, chunksize: int | None =
None, dataset: bool = False, partition_filter: Callable[[Dict[str, str]], bool] | None =
None, **pandas_kwargs: Any) → DataFrame | Iterator[DataFrame]
```

Read CSV file(s) from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), [!seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: For partial and gradual reading use the argument `chunksize` instead of `iterator`.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: The filter by `last_modified` begin `last_modified` end is applied after list all S3 files

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. `[".csv"]`). If `None`, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. `["_SUCCESS"]`). If `None`, will try to read all files. (default)
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. `{ 's3://bucket/key0': '121212', 's3://bucket/key1': '343434' }`)
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.

- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If *True* read a CSV dataset instead of simple file(s) loading all the related partitions as columns.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (*Dict[str, str]*) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/023%20-%20Flexible%20Partitions%20Filter.html>
- **pandas_kwargs** – KEYWORD arguments forwarded to `pandas.read_csv()`. You can NOT pass *pandas_kwargs* explicitly, just add valid Pandas arguments in the function call and `awsrangler` will accept it. e.g. `wr.s3.read_csv('s3://bucket/prefix/', sep='|', na_values=['null', 'none'], skip_blank_lines=True)` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Returns

Pandas DataFrame or a Generator in case of *chunksize != None*.

Return type

`Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]`

Examples

Reading all CSV files under a prefix

```
>>> import awswrangler as wr
>>> df = wr.s3.read_csv(path='s3://bucket/prefix/')
```

Reading all CSV files under a prefix and using `pandas_kwargs`

```
>>> import awswrangler as wr
>>> df = wr.s3.read_csv('s3://bucket/prefix/', sep='|', na_values=['null', 'none'],
↳ skip_blank_lines=True)
```

Reading all CSV files from a list

```
>>> import awswrangler as wr
>>> df = wr.s3.read_csv(path=['s3://bucket/filename0.csv', 's3://bucket/filename1.
↳ csv'])
```

Reading in chunks of 100 lines

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_csv(path=['s3://bucket/filename0.csv', 's3://bucket/filename1.
↳ csv'], chunksize=100)
>>> for df in dfs:
>>>     print(df) # 100 lines Pandas DataFrame
```

Reading CSV Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_csv(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.read_deltalake

`awswrangler.s3.read_deltalake`(*path*: str | None = None, *version*: int | None = None, *partitions*: List[Tuple[str, str, Any]] | None = None, *columns*: List[str] | None = None, *without_files*: bool = False, *boto3_session*: Session | None = None, *s3_additional_kwargs*: Dict[str, str] | None = None, *pyarrow_additional_kwargs*: Dict[str, Any] | None = None) → DataFrame

Load a Deltalake table data from an S3 path.

This function requires the `deltalake` package. See the [How to load a Delta table](#) guide for loading instructions.

Parameters

- **path** (*Optional*[str]) – The path of the DeltaTable.
- **version** (*Optional*[int]) – The version of the DeltaTable.
- **partitions** (*Optional*[List[Tuple[str, str, Any]]) – A list of partition filters, see `help(DeltaTable.files_by_partitions)` for filter syntax.
- **columns** (*Optional*[List[str]]) – The columns to project. This can be a list of column names to include (order and duplicates are preserved).
- **without_files** (bool) – If True, load the table without tracking files (memory-friendly). Some append-only applications might not need to track files.
- **boto3_session** (*Optional*[`boto3.Session()`]) – Boto3 Session. If None, the default boto3 session is used.
- **s3_additional_kwargs** (*Optional*[Dict[str, str]]) – Forwarded to the Delta Table class for the storage options of the S3 backend.
- **pyarrow_additional_kwargs** (*Optional*[Dict[str, str]]) – Forwarded to the PyArrow `to_pandas` method.

Returns

df – DataFrame with the results.

Return type

pd.DataFrame

See also:

deltalake.DeltaTable

Create a DeltaTable instance with the deltalake library.

aws wrangler.s3.read_excel

`aws wrangler.s3.read_excel(path: str, version_id: str | None = None, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, **pandas_kwargs: Any) → DataFrame`

Read EXCEL file(s) from a received S3 path.

Note: This function accepts any Pandas's `read_excel()` argument. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Note: Depending on the file extension ('xlsx', 'xls', 'odf'...), an additional library might have to be installed first (e.g. xlrld).

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **path** (*str*) – S3 path (e.g. `s3://bucket/key.xlsx`).
- **version_id** (*Optional[str]*) – Version id of the object.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If given an int will use the given amount of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.
- **pandas_kwargs** – KEYWORD arguments forwarded to `pandas.read_excel()`. You can NOT pass `pandas_kwargs` explicit, just add valid Pandas arguments in the function call and `aws wrangler` will accept it. e.g. `wr.s3.read_excel("s3://bucket/key.xlsx", na_rep="", verbose=True)`
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Returns

Pandas DataFrame.

Return type

`pandas.DataFrame`

Examples

Reading an EXCEL file

```
>>> import awswrangler as wr
>>> df = wr.s3.read_excel('s3://bucket/key.xlsx')
```

awswrangler.s3.read_fwf

`awswrangler.s3.read_fwf(path: str | List[str], path_suffix: str | List[str] | None = None, path_ignore_suffix: str | List[str] | None = None, version_id: str | Dict[str, str] | None = None, ignore_empty: bool = True, use_threads: bool | int = True, last_modified_begin: datetime | None = None, last_modified_end: datetime | None = None, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, chunksize: int | None = None, dataset: bool = False, partition_filter: Callable[[Dict[str, str]], bool] | None = None, **pandas_kwargs: Any) → DataFrame | Iterator[DataFrame]`

Read fixed-width formatted file(s) from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: For partial and gradual reading use the argument `chunksize` instead of `iterator`.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: The filter by `last_modified` begin `last_modified` end is applied after list all S3 files

Parameters

- **path** (`Union[str, List[str]]`) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path_suffix** (`Union[str, List[str], None]`) – Suffix or List of suffixes to be read (e.g. `[“.txt”]`). If `None`, will try to read all files. (default)
- **path_ignore_suffix** (`Union[str, List[str], None]`) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. `[“_SUCCESS”]`). If `None`, will try to read all files. (default)
- **version_id** (`Optional[Union[str, Dict[str, str]]]`) – Version id of the object or mapping of object path to version id. (e.g. `{‘s3://bucket/key0’: ‘121212’, ‘s3://bucket/key1’: ‘343434’}`)
- **ignore_empty** (`bool`) – Ignore files with 0 bytes.
- **use_threads** (`Union[bool, int]`) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.

- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If *True* read a FWF dataset instead of simple file(s) loading all the related partitions as columns.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/023%20-%20Flexible%20Partitions%20Filter.html>
- **pandas_kwargs** – KEYWORD arguments forwarded to `pandas.read_fwf()`. You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and awswrangler will accept it. e.g. `wr.s3.read_fwf(path='s3://bucket/prefix/', widths=[1, 3], names=["c0", "c1"])` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_fwf.html

Returns

Pandas DataFrame or a Generator in case of *chunksize != None*.

Return type

Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]

Examples

Reading all fixed-width formatted (FWF) files under a prefix

```
>>> import awswrangler as wr
>>> df = wr.s3.read_fwf(path='s3://bucket/prefix/', widths=[1, 3], names=['c0', 'c1', 'c2'])
```

Reading all fixed-width formatted (FWF) files from a list

```
>>> import awswrangler as wr
>>> df = wr.s3.read_fwf(path=['s3://bucket/0.txt', 's3://bucket/1.txt'], widths=[1, 3], names=['c0', 'c1'])
```

Reading in chunks of 100 lines

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_fwf(
...     path=['s3://bucket/0.txt', 's3://bucket/1.txt'],
...     chunksize=100,
...     widths=[1, 3],
...     names=["c0", "c1"]
... )
>>> for df in dfs:
>>>     print(df) # 100 lines Pandas DataFrame
```

Reading FWF Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_fwf(path, dataset=True, partition_filter=my_filter, widths=[1,
↪3], names=["c0", "c1"])
```

awswrangler.s3.read_json

`awswrangler.s3.read_json(path: str | List[str], path_suffix: str | List[str] | None = None, path_ignore_suffix: str | List[str] | None = None, version_id: str | Dict[str, str] | None = None, ignore_empty: bool = True, orient: str = 'columns', use_threads: bool | int = True, last_modified_begin: datetime | None = None, last_modified_end: datetime | None = None, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, chunksize: int | None = None, dataset: bool = False, partition_filter: Callable[[Dict[str, str]], bool] | None = None, **pandas_kwargs: Any) → DataFrame | Iterator[DataFrame]`

Read JSON file(s) from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: For partial and gradual reading use the argument `chunksize` instead of `iterator`.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: The filter by `last_modified` begin `last_modified` end is applied after list all S3 files

Parameters

- **path** (`Union[str, List[str]]`) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path_suffix** (`Union[str, List[str], None]`) – Suffix or List of suffixes to be read (e.g. `[".json"]`). If `None`, will try to read all files. (default)

- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. ["_SUCCESS"]). If None, will try to read all files. (default)
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **orient** (*str*) – Same as Pandas: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If integer is provided, specified number is used.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime, optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.
- **chunksize** (*int, optional*) – If specified, return an generator where chunksize is the number of rows to include in each chunk.
- **dataset** (*bool*) – If *True* read a JSON dataset instead of simple file(s) loading all the related partitions as columns. If *True*, the *lines=True* will be assumed by default.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/023%20-%20Flexible%20Partitions%20Filter.html>
- **pandas_kwargs** – KEYWORD arguments forwarded to `pandas.read_json()`. You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and awswrangler will accept it. e.g. `wr.s3.read_json('s3://bucket/prefix/', lines=True, keep_default_dates=True)` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html

Returns

Pandas DataFrame or a Generator in case of *chunksize != None*.

Return type

`Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]`

Examples

Reading all JSON files under a prefix

```
>>> import awswrangler as wr
>>> df = wr.s3.read_json(path='s3://bucket/prefix/')
```

Reading all CSV files under a prefix and using pandas_kwargs

```
>>> import awswrangler as wr
>>> df = wr.s3.read_json('s3://bucket/prefix/', lines=True, keep_default_dates=True)
```

Reading all JSON files from a list

```
>>> import awswrangler as wr
>>> df = wr.s3.read_json(path=['s3://bucket/filename0.json', 's3://bucket/filename1.
↪ json'])
```

Reading in chunks of 100 lines

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_json(path=['s3://bucket/0.json', 's3://bucket/1.json'], ↪
↪ chunksize=100, lines=True)
>>> for df in dfs:
>>>     print(df) # 100 lines Pandas DataFrame
```

Reading JSON Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_json(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.read_parquet

`awswrangler.s3.read_parquet(path: str | List[str], path_root: str | None = None, path_suffix: str | List[str] | None = None, path_ignore_suffix: str | List[str] | None = None, version_id: str | Dict[str, str] | None = None, ignore_empty: bool = True, ignore_index: bool | None = None, partition_filter: Callable[[Dict[str, str]], bool] | None = None, columns: List[str] | None = None, validate_schema: bool = False, chunked: bool | int = False, dataset: bool = False, categories: List[str] | None = None, safe: bool = True, map_types: bool = True, use_threads: bool | int = True, last_modified_begin: datetime | None = None, last_modified_end: datetime | None = None, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, pyarrow_additional_kwargs: Dict[str, Any] | None = None) → DataFrame | Iterator[DataFrame]`

Read Apache Parquet file(s) from a received S3 prefix or list of S3 objects paths.

The concept of Dataset goes beyond the simple idea of files and enable more complex features like partitioning and catalog integration (AWS Glue Catalog).

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: Batching (*chunked* argument) (Memory Friendly):

Will enable the function to return a Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies on awswrangler:

- If **chunked=True**, a new DataFrame will be returned for each file in your path/dataset.
- If **chunked=INTEGER**, awswrangler will iterate on the data by number of rows equal the received INTEGER.

P.S. chunked=True is faster and uses less memory while *chunked=INTEGER* is more precise in number of rows for each DataFrame.

Note: In case of *use_threads=True* the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: The filter by last_modified begin last_modified end is applied after list all S3 files

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **path_root** (*Optional[str]*) – Root path of the table. If `dataset=True`, will be used as a starting point to load partition columns.
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. `[“.gz.parquet”, “.snappy.parquet”]`). If `None`, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored (e.g. `[“.csv”, “_SUCCESS”]`). If `None`, will try to read all files. (default)
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. `{‘s3://bucket/key0’: ‘121212’, ‘s3://bucket/key1’: ‘343434’}`)
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **ignore_index** (*Optional[bool]*) – Ignore index when combining multiple parquet files to one DataFrame.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (`Dict[str, str]`) where keys are partitions names and values are partitions values. Partitions values will be always strings extracted from S3. This function MUST return a bool, `True` to read the partition or `False` to ignore it. Ignored if `dataset=False`. E.g `lambda x: True if x[“year”] == “2020” and x[“month”] == “1” else False`
- **columns** (*List[str], optional*) – Names of columns to read from the file(s).
- **validate_schema** – Check that individual file schemas are all the same / compatible. Schemas within a folder prefix should all be the same. Disable if you have schemas that are different and want to disable this check.

- **chunked** (*Union[int, bool]*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* awswrangler iterates on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed awswrangler will iterate on the data by number of rows equal the received *INTEGER*.
- **dataset** (*bool*) – If *True* read a parquet dataset instead of simple file(s) loading all the related partitions as columns.
- **categories** (*Optional[List[str]]*, *optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **safe** (*bool*, *default True*) – For certain data types, a cast is needed in order to store the data in a pandas DataFrame or Series (e.g. timestamps are always stored as nanoseconds in pandas). This option controls whether it is a safe cast or not.
- **map_types** (*bool*, *default True*) – True to convert pyarrow DataTypes to pandas ExtensionDtypes. It is used to override the default pandas type for conversion of built-in pyarrow types or in absence of pandas_metadata in the Table schema.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If integer is provided, specified number is used.
- **last_modified_begin** – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **last_modified_end** (*datetime*, *optional*) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.
- **pyarrow_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to the ParquetFile class or converting an Arrow table to Pandas, currently only an “coerce_int96_timestamp_unit” or “timestamp_as_object” argument will be considered. If reading parquet files where you cannot convert a timestamp to pandas Timestamp[ns] consider setting timestamp_as_object=True, to allow for timestamp units larger than “ns”. If reading parquet data that still uses INT96 (like Athena outputs) you can use coerce_int96_timestamp_unit to specify what timestamp unit to encode INT96 to (by default this is “ns”, if you know the output parquet came from a system that encodes timestamp to a particular unit then set this to that same unit e.g. coerce_int96_timestamp_unit=“ms”).

Returns

Pandas DataFrame or a Generator in case of *chunked=True*.

Return type

Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]

Examples

Reading all Parquet files under a prefix

```
>>> import awswrangler as wr
>>> df = wr.s3.read_parquet(path='s3://bucket/prefix/')
```

Reading all Parquet files from a list

```
>>> import awswrangler as wr
>>> df = wr.s3.read_parquet(path=['s3://bucket/filename0.parquet', 's3://bucket/
↳ filename1.parquet'])
```

Reading in chunks (Chunk by file)

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_parquet(path=['s3://bucket/filename0.parquet', 's3://bucket/
↳ filename1.parquet'], chunked=True)
>>> for df in dfs:
>>>     print(df) # Smaller Pandas DataFrame
```

Reading in chunks (Chunk by 1MM rows)

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_parquet(
...     path=['s3://bucket/filename0.parquet', 's3://bucket/filename1.parquet'],
...     chunked=1_000_000
... )
>>> for df in dfs:
>>>     print(df) # 1MM Pandas DataFrame
```

Reading Parquet Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_parquet(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.read_parquet_metadata

`awswrangler.s3.read_parquet_metadata(path: str | List[str], version_id: str | Dict[str, str] | None = None, path_suffix: str | None = None, path_ignore_suffix: str | None = None, ignore_empty: bool = True, ignore_null: bool = False, dtype: Dict[str, str] | None = None, sampling: float = 1.0, dataset: bool = False, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, pyarrow_additional_kwargs: Dict[str, Any] | None = None) → Any`

Read Apache Parquet file(s) metadata from a received S3 prefix or list of S3 objects paths.

The concept of Dataset goes beyond the simple idea of files and enable more complex features like partitioning and catalog integration (AWS Glue Catalog).

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. `{ 's3://bucket/key0': '121212', 's3://bucket/key1': '343434' }`)
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. `[“.gz.parquet”, “.snappy.parquet”]`). If `None`, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. `[“.csv”, “_SUCCESS”]`). If `None`, will try to read all files. (default)
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **ignore_null** (*bool*) – Ignore columns with null type.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined data types as partitions columns. (e.g. `{ 'col name': 'bigint', 'col2 name': 'int' }`)
- **sampling** (*float*) – Random sample ratio of files that will have the metadata inspected. Must be $0.0 < sampling \leq 1.0$. The higher, the more accurate. The lower, the faster.
- **dataset** (*bool*) – If `True` read a parquet dataset instead of simple file(s) loading all the related partitions as columns.
- **use_threads** (*bool, int*) – `True` to enable concurrent requests, `False` to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.
- **pyarrow_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward kwargs to parquet reader currently only excepts “coerce_int96_timestamp_unit”. Which can be used to cast deprecated Parquet INT96 into a specified timestamp unit (e.g. “ms”).

Returns

`columns_types`: Dictionary with keys as column names and values as data types (e.g. `{ 'col0': 'bigint', 'col1': 'double' }`). / `partitions_types`: Dictionary with keys as partition names and values as data types (e.g. `{ 'col2': 'date' }`).

Return type

Tuple[Dict[str, str], Optional[Dict[str, str]]]

Examples

Reading all Parquet files (with partitions) metadata under a prefix

```
>>> import awswrangler as wr
>>> columns_types, partitions_types = wr.s3.read_parquet_metadata(path='s3://bucket/
↳ prefix/', dataset=True)
```

Reading all Parquet files metadata from a list

```
>>> import awswrangler as wr
>>> columns_types, partitions_types = wr.s3.read_parquet_metadata(path=[
...     's3://bucket/filename0.parquet',
...     's3://bucket/filename1.parquet'
... ])
```

awswrangler.s3.read_parquet_table

`awswrangler.s3.read_parquet_table(table: str, database: str, filename_suffix: str | List[str] | None = None, filename_ignore_suffix: str | List[str] | None = None, catalog_id: str | None = None, partition_filter: Callable[[Dict[str, str]], bool] | None = None, columns: List[str] | None = None, validate_schema: bool = True, categories: List[str] | None = None, safe: bool = True, map_types: bool = True, chunked: bool | int = False, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None) → Any`

Read Apache Parquet table registered on AWS Glue Catalog.

Note: Batching (*chunked* argument) (Memory Friendly):

Will enable the function to return an Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies on awswrangler:

- If **chunked=True**, a new DataFrame will be returned for each file in your path/dataset.
- If **chunked=INTEGER**, awswrangler will paginate through files slicing and concatenating to return DataFrames with the number of row equal the received INTEGER.

P.S. chunked=True if faster and uses less memory while *chunked=INTEGER* is more precise in number of rows for each Dataframe.

Note: In case of *use_threads=True* the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `catalog_id`

- database

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **table** (*str*) – AWS Glue Catalog table name.
- **database** (*str*) – AWS Glue Catalog database name.
- **filename_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be read (e.g. [“.gz.parquet”, “.snappy.parquet”]). If None, will try to read all files. (default)
- **filename_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. [“.csv”, “_SUCCESS”]). If None, will try to read all files. (default)
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **partition_filter** (*Optional[Callable[[Dict[str, str]], bool]]*) – Callback Function filters to apply on PARTITION columns (PUSH-DOWN filter). This function MUST receive a single argument (Dict[str, str]) where keys are partition names and values are partition values. Partition values will be always strings extracted from S3. This function MUST return a bool, True to read the partition or False to ignore it. Ignored if *dataset=False*. E.g `lambda x: True if x["year"] == "2020" and x["month"] == "1" else False` <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/023%20-%20Flexible%20Partitions%20Filter.html>
- **columns** (*List[str], optional*) – Names of columns to read from the file(s).
- **validate_schema** – Check that individual file schemas are all the same / compatible. Schemas within a folder prefix should all be the same. Disable if you have schemas that are different and want to disable this check.
- **categories** (*Optional[List[str]], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **safe** (*bool, default True*) – For certain data types, a cast is needed in order to store the data in a pandas DataFrame or Series (e.g. timestamps are always stored as nanoseconds in pandas). This option controls whether it is a safe cast or not.
- **map_types** (*bool, default True*) – True to convert pyarrow DataTypes to pandas ExtensionDtypes. It is used to override the default pandas type for conversion of built-in pyarrow types or in absence of pandas_metadata in the Table schema.
- **chunked** (*bool*) – If True will break the data in smaller DataFrames (Non-deterministic number of lines). Otherwise return a single DataFrame with the whole data.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.

Returns

Pandas DataFrame or a Generator in case of *chunked=True*.

Return type

Union[pandas.DataFrame, Generator[pandas.DataFrame, None, None]]

Examples

Reading Parquet Table

```
>>> import awswrangler as wr
>>> df = wr.s3.read_parquet_table(database='...', table='...')
```

Reading Parquet Table encrypted

```
>>> import awswrangler as wr
>>> df = wr.s3.read_parquet_table(
...     database='...',
...     table='...',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
```

Reading Parquet Table in chunks (Chunk by file)

```
>>> import awswrangler as wr
>>> dfs = wr.s3.read_parquet_table(database='...', table='...', chunked=True)
>>> for df in dfs:
>>>     print(df) # Smaller Pandas DataFrame
```

Reading Parquet Dataset with PUSH-DOWN filter over partitions

```
>>> import awswrangler as wr
>>> my_filter = lambda x: True if x["city"].startswith("new") else False
>>> df = wr.s3.read_parquet_table(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.select_query

awswrangler.s3.select_query(*sql: str, path: str, input_serialization: str, input_serialization_params: Dict[str, bool | str], compression: str | None = None, use_threads: bool | int = False, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None*) → DataFrame

Filter contents of an Amazon S3 object based on SQL statement.

Note: Scan ranges are only supported for uncompressed CSV/JSON, CSV (without quoted delimiters) and JSON objects (in LINES mode only). It means scanning cannot be split across threads if the latter conditions are not met, leading to lower performance.

Parameters

- **sql** (*str*) – SQL statement used to query the object.
- **path** (*str*) – S3 path to the object (e.g. `s3://bucket/key`).

- **input_serialization** (*str*,) – Format of the S3 object queried. Valid values: “CSV”, “JSON”, or “Parquet”. Case sensitive.
- **input_serialization_params** (*Dict[str, Union[bool, str]]*) – Dictionary describing the serialization of the S3 object.
- **compression** (*Optional[str]*) – Compression type of the S3 object. Valid values: None, “gzip”, or “bzip2”. gzip and bzip2 are only valid for CSV and JSON objects.
- **use_threads** (*Union[bool, int]*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` is used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session is used if none is provided.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. Valid values: “SSECustomerAlgorithm”, “SSECustomerKey”, “ExpectedBucketOwner”. e.g. `s3_additional_kwargs={'SSECustomerAlgorithm': 'md5'}`

Returns

Pandas DataFrame with results from query.

Return type

`pandas.DataFrame`

Examples

Reading a gzip compressed JSON document

```
>>> import awswrangler as wr
>>> df = wr.s3.select_query(
...     sql='SELECT * FROM s3object[*][*]',
...     path='s3://bucket/key.json.gzip',
...     input_serialization='JSON',
...     input_serialization_params={
...         'Type': 'Document',
...     },
...     compression="gzip",
... )
```

Reading an entire CSV object using threads

```
>>> import awswrangler as wr
>>> df = wr.s3.select_query(
...     sql='SELECT * FROM s3object',
...     path='s3://bucket/key.csv',
...     input_serialization='CSV',
...     input_serialization_params={
...         'FileHeaderInfo': 'Use',
...         'RecordDelimiter': '\r\n'
...     },
...     use_threads=True,
... )
```

Reading a single column from Parquet object with pushdown filter

```
>>> import awswrangler as wr
>>> df = wr.s3.select_query(
...     sql='SELECT s.\"id\" FROM s3object s where s.\"id\" = 1.0',
...     path='s3://bucket/key.snappy.parquet',
...     input_serialization='Parquet',
... )
```

awswrangler.s3.size_objects

awswrangler.s3.size_objects(*path*: str | List[str], *version_id*: str | Dict[str, str] | None = None, *use_threads*: bool | int = True, *s3_additional_kwargs*: Dict[str, Any] | None = None, *boto3_session*: Session | None = None) → Dict[str, int | None]

Get the size (ContentLength) in bytes of Amazon S3 objects from a received S3 prefix or list of S3 objects paths.

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **path** (*Union[str, List[str]]*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`) or list of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **version_id** (*Optional[Union[str, Dict[str, str]]]*) – Version id of the object or mapping of object path to version id. (e.g. `{'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'}`)
- **use_threads** (*bool, int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'RequestPayer': 'requester'}`
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

Dictionary where the key is the object path and the value is the object size.

Return type

Dict[str, Optional[int]]

Examples

```
>>> import awswrangler as wr
>>> sizes0 = wr.s3.size_objects(['s3://bucket/key0', 's3://bucket/key1']) # Get_
↳ the sizes of both objects
>>> sizes1 = wr.s3.size_objects('s3://bucket/prefix') # Get the sizes of all_
↳ objects under the received prefix
```

awswrangler.s3.store_parquet_metadata

`awswrangler.s3.store_parquet_metadata`(*path: str, database: str, table: str, catalog_id: str | None = None, path_suffix: str | None = None, path_ignore_suffix: str | None = None, ignore_empty: bool = True, dtype: Dict[str, str] | None = None, sampling: float = 1.0, dataset: bool = False, use_threads: bool | int = True, description: str | None = None, parameters: Dict[str, str] | None = None, columns_comments: Dict[str, str] | None = None, compression: str | None = None, mode: str = 'overwrite', catalog_versioning: bool = False, regular_partitions: bool = True, projection_enabled: bool = False, projection_types: Dict[str, str] | None = None, projection_ranges: Dict[str, str] | None = None, projection_values: Dict[str, str] | None = None, projection_intervals: Dict[str, str] | None = None, projection_digits: Dict[str, str] | None = None, projection_formats: Dict[str, str] | None = None, projection_storage_location_template: str | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, boto3_session: Session | None = None*) → Any

Infer and store parquet metadata on AWS Glue Catalog.

Infer Apache Parquet file(s) metadata from a received S3 prefix And then stores it on AWS Glue Catalog including all inferred partitions (No need for 'MSCK REPAIR TABLE')

The concept of Dataset goes beyond the simple idea of files and enables more complex features like partitioning and catalog integration (AWS Glue Catalog).

This function accepts Unix shell-style wildcards in the path argument. * (matches everything), ? (matches any single character), [seq] (matches any character in seq), ![seq] (matches any character not in seq). If you want to use a path which includes Unix shell-style wildcard characters (*, ?, []), you can use `glob.escape(path)` before passing the path to this function.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **path** (*str*) – S3 prefix (accepts Unix shell-style wildcards) (e.g. `s3://bucket/prefix`).
- **table** (*str*) – Glue/Athena catalog: Table name.
- **database** (*str*) – AWS Glue Catalog database name.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for filtering S3 keys.
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored.
- **ignore_empty** (*bool*) – Ignore files with 0 bytes.
- **dtype** (*Dict[str, str]*, *optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined data types as partitions columns. (e.g. `{'col name': 'bigint', 'col2 name': 'int'}`)
- **sampling** (*float*) – Random sample ratio of files that will have the metadata inspected. Must be $0.0 < sampling \leq 1.0$. The higher, the more accurate. The lower, the faster.
- **dataset** (*bool*) – If True read a parquet dataset instead of simple file(s) loading all the related partitions as columns.
- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **description** (*str*, *optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str]*, *optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str]*, *optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. `{'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}`).
- **compression** (*str*, *optional*) – Compression style (None, snappy, gzip, etc).
- **mode** (*str*) – 'overwrite' to recreate any possible existing table or 'append' to keep any possible existing table.
- **catalog_versioning** (*bool*) – If True and *mode*="overwrite", creates an archived version of the table catalog before updating it.
- **regular_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': 'enum', 'col2_name': 'integer'}`)
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/>

`partition-projection-supported-types.html` (e.g. `{'col_name': '0,10', 'col2_name': '-1,8675309'}`)

- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'}`)
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': '1', 'col2_name': '5'}`)
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': '1', 'col2_name': '2'}`)
- **projection_formats** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections formats. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_date': 'yyyy-MM-dd', 'col2_timestamp': 'yyyy-MM-dd HH:mm:ss'}`)
- **projection_storage_location_template** (*Optional[str]*) – Value which allows Athena to properly map partition values if the S3 file locations do not follow a typical `.../column=value/...` pattern. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-setting-up.html> (e.g. `s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/${c}/`)
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.

Returns

The metadata used to create the Glue Table. `columns_types`: Dictionary with keys as column names and values as data types (e.g. `{'col0': 'bigint', 'col1': 'double'}`). / `partitions_types`: Dictionary with keys as partition names and values as data types (e.g. `{'col2': 'date'}`). / `partitions_values`: Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. `{'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}`).

Return type

`Tuple[Dict[str, str], Optional[Dict[str, str]], Optional[Dict[str, List[str]]]`

Examples

Reading all Parquet files metadata under a prefix

```
>>> import awswrangler as wr
>>> columns_types, partitions_types, partitions_values = wr.s3.store_parquet_
↳ metadata(
...     path='s3://bucket/prefix/',
...     database='...',
...     table='...',
...     dataset=True
... )
```


aws wrangler.s3.to_csv

```
aws wrangler.s3.to_csv(df: DataFrame, path: str | None = None, sep: str = ',', index: bool = True, columns:
    List[str] | None = None, use_threads: bool | int = True, boto3_session: Session | None =
    None, s3_additional_kwargs: Dict[str, Any] | None = None, sanitize_columns: bool =
    False, dataset: bool = False, filename_prefix: str | None = None, partition_cols:
    List[str] | None = None, bucketing_info: Tuple[List[str], int] | None = None,
    concurrent_partitioning: bool = False, mode: str | None = None, catalog_versioning:
    bool = False, schema_evolution: bool = False, database: str | None = None, table: str |
    None = None, table_type: str | None = None, transaction_id: str | None = None, dtype:
    Dict[str, str] | None = None, description: str | None = None, parameters: Dict[str, str] |
    None = None, columns_comments: Dict[str, str] | None = None, regular_partitions:
    bool = True, projection_enabled: bool = False, projection_types: Dict[str, str] | None =
    None, projection_ranges: Dict[str, str] | None = None, projection_values: Dict[str, str] |
    None = None, projection_intervals: Dict[str, str] | None = None, projection_digits:
    Dict[str, str] | None = None, projection_formats: Dict[str, str] | None = None,
    projection_storage_location_template: str | None = None, catalog_id: str | None =
    None, **pandas_kwargs: Any) → Any
```

Write CSV file or dataset on Amazon S3.

The concept of Dataset goes beyond the simple idea of ordinary files and enable more complex features like partitioning and catalog integration (Amazon Athena/AWS Glue Catalog).

Note: If `database` and `table` arguments are passed, the table name and all column names will be automatically sanitized using `wr.catalog.sanitize_table_name` and `wr.catalog.sanitize_column_name`. Please, pass `sanitize_columns=True` to enforce this behaviour always.

Note: If `table` and `database` arguments are passed, `pandas_kwargs` will be ignored due restrictive quoting, `date_format`, `escapechar` and encoding required by Athena/Glue Catalog.

Note: Compression: The minimum acceptable version to achieve it is Pandas 1.2.0 that requires Python `>= 3.7.1`.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `concurrent_partitioning`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **path** (*str, optional*) – Amazon S3 path (e.g. `s3://bucket/prefix/filename.csv`) (for dataset e.g. `s3://bucket/prefix`). Required if `dataset=False` or when creating a new dataset
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **index** (*bool*) – Write row names (index).
- **columns** (*Optional[List[str]]*) – Columns to write.
- **use_threads** (*bool, int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **sanitize_columns** (*bool*) – True to sanitize columns names or False to keep it as is. True value is forced if `dataset=True`.
- **dataset** (*bool*) – If True store as a dataset instead of ordinary file(s) If True, enable all follow arguments: `partition_cols`, `mode`, `database`, `table`, `description`, `parameters`, `columns_comments`, `concurrent_partitioning`, `catalog_versioning`, `projection_enabled`, `projection_types`, `projection_ranges`, `projection_values`, `projection_intervals`, `projection_digits`, `catalog_id`, `schema_evolution`.
- **filename_prefix** (*str, optional*) – If `dataset=True`, add a filename prefix to the output files.
- **partition_cols** (*List[str], optional*) – List of column names that will be used to create partitions. Only takes effect if `dataset=True`.
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **concurrent_partitioning** (*bool*) – If True will increase the parallelism level during the partitions writing. It will decrease the writing time and increase the memory usage. <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html>
- **mode** (*str, optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if `dataset=True`. For details check the related tutorial: https://aws-sdk-pandas.readthedocs.io/en/2.20.0/stubs/awswrangler.s3.to_parquet.html#awswrangler.s3.to_parquet
- **catalog_versioning** (*bool*) – If True and `mode="overwrite"`, creates an archived version of the table catalog before updating it.
- **schema_evolution** (*bool*) – If True allows schema evolution (new or missing columns), otherwise a exception will be raised. (Only considered if `dataset=True` and `mode` in (“append”, “overwrite_partitions”). False by default. Related tutorial: <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/014%20-%20Schema%20Evolution.html>
- **database** (*str, optional*) – Glue/Athena catalog: Database name.

- **table** (*str*, *optional*) – Glue/Athena catalog: Table name.
- **table_type** (*str*, *optional*) – The type of the Glue Table. Set to `EXTERNAL_TABLE` if None
- **transaction_id** (*str*, *optional*) – The ID of the transaction when writing to a Governed Table.
- **dtype** (*Dict[str, str]*, *optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. `{'col name': 'bigint', 'col2 name': 'int'}`)
- **description** (*str*, *optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str]*, *optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str]*, *optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. `{'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}`).
- **regular_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: “enum”, “integer”, “date”, “injected” <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': 'enum', 'col2_name': 'integer'}`)
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': '0,10', 'col2_name': '-1,8675309'}`)
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'}`)
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': '1', 'col2_name': '5'}`)
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_name': '1', 'col2_name': '2'}`)
- **projection_formats** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections formats. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{'col_date': 'yyyy-MM-dd', 'col2_timestamp': 'yyyy-MM-dd HH:mm:ss'}`)
- **projection_storage_location_template** (*Optional[str]*) – Value which allows Athena to properly map partition values if the S3 file locations do not follow a typical `.../column=value/...` pattern. <https://>

docs.aws.amazon.com/athena/latest/ug/partition-projection-setting-up.html (e.g. `s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/${c}/`)

- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **pandas_kwargs** – KEYWORD arguments forwarded to `pandas.DataFrame.to_csv()`. You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and `aws wrangler` will accept it. e.g. `wr.s3.to_csv(df, path, sep='|', na_rep='NULL', decimal=',')` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html

Returns

Dictionary with: 'paths': List of all stored files paths on S3. 'partitions_values': Dictionary of partitions added with keys as S3 path locations and values as a list of partitions values as str.

Return type

`Dict[str, Union[List[str], Dict[str, List[str]]]]`

Examples

Writing single file

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/prefix/my_file.csv',
... )
{
  'paths': ['s3://bucket/prefix/my_file.csv'],
  'partitions_values': {}
}
```

Writing single file with `pandas_kwargs`

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/prefix/my_file.csv',
...     sep='|',
...     na_rep='NULL',
...     decimal=',',
... )
{
  'paths': ['s3://bucket/prefix/my_file.csv'],
  'partitions_values': {}
}
```

Writing single file encrypted with a KMS key

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
```

(continues on next page)

(continued from previous page)

```

...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/prefix/my_file.csv',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
{
    'paths': ['s3://bucket/prefix/my_file.csv'],
    'partitions_values': {}
}

```

Writing partitioned dataset

```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['col2']
... )
{
    'paths': ['s3://.../col2=A/x.csv', 's3://.../col2=B/y.csv'],
    'partitions_values': {
        's3://.../col2=A/': ['A'],
        's3://.../col2=B/': ['B']
    }
}

```

Writing bucketed dataset

```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     bucketing_info=(["col2"], 2)
... )
{
    'paths': ['s3://.../x_bucket-00000.csv', 's3://.../col2=B/x_bucket-00001.csv'],
    'partitions_values': {}
}

```

Writing dataset to S3 with metadata on Athena/Glue Catalog.

```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['col2'],
...     database='default', # Athena/Glue database
...     table='my_table' # Athena/Glue table
... )
{
  'paths': ['s3://.../col2=A/x.csv', 's3://.../col2=B/y.csv'],
  'partitions_values': {
    's3://.../col2=A/': ['A'],
    's3://.../col2=B/': ['B']
  }
}

```

Writing dataset to Glue governed table

```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B'],
...         'col3': [None, None, None]
...     }),
...     dataset=True,
...     mode='append',
...     database='default', # Athena/Glue database
...     table='my_table', # Athena/Glue table
...     table_type='GOVERNED',
...     transaction_id="xxx",
... )
{
  'paths': ['s3://.../x.csv'],
  'partitions_values': {}
}

```

Writing dataset casting empty column data type

```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_csv(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B'],
...         'col3': [None, None, None]
...     }),

```

(continues on next page)

(continued from previous page)

```

...     path='s3://bucket/prefix',
...     dataset=True,
...     database='default', # Athena/Glue database
...     table='my_table' # Athena/Glue table
...     dtype={'col3': 'date'}
... )
{
  'paths': ['s3://.../x.csv'],
  'partitions_values': {}
}

```

aws wrangler.s3.to_excel

`aws wrangler.s3.to_excel(df: DataFrame, path: str, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, use_threads: bool | int = True, **pandas_kwargs: Any) → str`

Write EXCEL file on Amazon S3.

Note: This function accepts any Pandas's `read_excel()` argument. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Note: Depending on the file extension ('xlsx', 'xls', 'odf...'), an additional library might have to be installed first (e.g. xlrld).

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **path** (*str*) – Amazon S3 path (e.g. `s3://bucket/filename.xlsx`).
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **pandas_kwargs** – KEYWORD arguments forwarded to `pandas.DataFrame.to_excel()`. You can NOT pass `pandas_kwargs` explicit, just add valid Pandas arguments in the function call and `aws wrangler` will accept it. e.g. `wr.s3.to_excel(df, path, na_rep="", index=False)` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_excel.html

Returns

Written S3 path.

Return type

str

Examples

Writing EXCEL file

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_excel(df, 's3://bucket/filename.xlsx')
```

awswrangler.s3.to_json

`awswrangler.s3.to_json(df: DataFrame, path: str | None = None, index: bool = True, columns: List[str] | None = None, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, sanitize_columns: bool = False, dataset: bool = False, filename_prefix: str | None = None, partition_cols: List[str] | None = None, bucketing_info: Tuple[List[str], int] | None = None, concurrent_partitioning: bool = False, mode: str | None = None, catalog_versioning: bool = False, schema_evolution: bool = True, database: str | None = None, table: str | None = None, table_type: str | None = None, transaction_id: str | None = None, dtype: Dict[str, str] | None = None, description: str | None = None, parameters: Dict[str, str] | None = None, columns_comments: Dict[str, str] | None = None, regular_partitions: bool = True, projection_enabled: bool = False, projection_types: Dict[str, str] | None = None, projection_ranges: Dict[str, str] | None = None, projection_values: Dict[str, str] | None = None, projection_intervals: Dict[str, str] | None = None, projection_digits: Dict[str, str] | None = None, projection_formats: Dict[str, str] | None = None, projection_storage_location_template: str | None = None, catalog_id: str | None = None, **pandas_kwargs: Any) → List[str] | Dict[str, List[str] | Dict[str, List[str]]]`

Write JSON file on Amazon S3.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: Compression: The minimum acceptable version to achieve it is Pandas 1.2.0 that requires Python `>= 3.7.1`.

Parameters

- **df** (`pandas.DataFrame`) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **path** (`str`) – Amazon S3 path (e.g. `s3://bucket/filename.json`).
- **index** (`bool`) – Write row names (index).
- **columns** (`Optional[List[str]]`) – Columns to write.

- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **sanitize_columns** (*bool*) – True to sanitize columns names or False to keep it as is. True value is forced if `dataset=True`.
- **dataset** (*bool*) – If True store as a dataset instead of ordinary file(s) If True, enable all follow arguments: `partition_cols`, `mode`, `database`, `table`, `description`, `parameters`, `columns_comments`, `concurrent_partitioning`, `catalog_versioning`, `projection_enabled`, `projection_types`, `projection_ranges`, `projection_values`, `projection_intervals`, `projection_digits`, `catalog_id`, `schema_evolution`.
- **filename_prefix** (*str*, *optional*) – If `dataset=True`, add a filename prefix to the output files.
- **partition_cols** (*List[str]*, *optional*) – List of column names that will be used to create partitions. Only takes effect if `dataset=True`.
- **bucketing_info** (*Tuple[List[str], int]*, *optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **concurrent_partitioning** (*bool*) – If True will increase the parallelism level during the partitions writing. It will decrease the writing time and increase the memory usage. <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html>
- **mode** (*str*, *optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if `dataset=True`. For details check the related tutorial: https://aws-sdk-pandas.readthedocs.io/en/2.20.0/stubs/awswrangler.s3.to_parquet.html#awswrangler.s3.to_parquet
- **catalog_versioning** (*bool*) – If True and `mode="overwrite"`, creates an archived version of the table catalog before updating it.
- **schema_evolution** (*bool*) – If True allows schema evolution (new or missing columns), otherwise a exception will be raised. (Only considered if `dataset=True` and `mode` in (“append”, “overwrite_partitions”)) Related tutorial: <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/014%20-%20Schema%20Evolution.html>
- **database** (*str*, *optional*) – Glue/Athena catalog: Database name.
- **table** (*str*, *optional*) – Glue/Athena catalog: Table name.
- **table_type** (*str*, *optional*) – The type of the Glue Table. Set to `EXTERNAL_TABLE` if None
- **transaction_id** (*str*, *optional*) – The ID of the transaction when writing to a Governed Table.
- **dtype** (*Dict[str, str]*, *optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. `{'col name': 'bigint', 'col2 name': 'int'}`)

- **description** (*str*, *optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str]*, *optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str]*, *optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **regular_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: “enum”, “integer”, “date”, “injected” <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': 'enum', 'col2_name': 'integer'})
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '0,10', 'col2_name': '-1,8675309'})
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '1', 'col2_name': '2'})
- **projection_formats** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections formats. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_date': 'yyyy-MM-dd', 'col2_timestamp': 'yyyy-MM-dd HH:mm:ss'})
- **projection_storage_location_template** (*Optional[str]*) – Value which allows Athena to properly map partition values if the S3 file locations do not follow a typical `.../column=value/...` pattern. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-setting-up.html> (e.g. `s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/${c}/`)
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **pandas_kwargs** – KEYWORD arguments forwarded to `pandas.DataFrame.to_json()`. You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and `aws wrangler` will accept it. e.g. `wr.s3.to_json(df, path, lines=True, date_format='iso')` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html

Returns

List of written files.

Return type

List[str]

Examples

Writing JSON file

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_json(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/filename.json',
... )
```

Writing JSON file using pandas_kwargs

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_json(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/filename.json',
...     lines=True,
...     date_format='iso'
... )
```

Writing CSV file encrypted with a KMS key

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_json(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/filename.json',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
```

awswrangler.s3.to_parquet

```
awswrangler.s3.to_parquet(df: DataFrame, path: str | None = None, index: bool = False, compression: str | None = 'snappy', pyarrow_additional_kwargs: Dict[str, Any] | None = None, max_rows_by_file: int | None = None, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, sanitize_columns: bool = False, dataset: bool = False, filename_prefix: str | None = None, partition_cols: List[str] | None = None, bucketing_info: Tuple[List[str], int] | None = None, concurrent_partitioning: bool = False, mode: str | None = None, catalog_versioning: bool = False, schema_evolution: bool = True, database: str | None = None, table: str | None = None, table_type: str | None = None, transaction_id: str | None = None, dtype: Dict[str, str] | None = None, description: str | None = None, parameters: Dict[str, str] | None = None, columns_comments: Dict[str, str] | None = None, regular_partitions: bool = True, projection_enabled: bool = False, projection_types: Dict[str, str] | None = None, projection_ranges: Dict[str, str] | None = None, projection_values: Dict[str, str] | None = None, projection_intervals: Dict[str, str] | None = None, projection_digits: Dict[str, str] | None = None, projection_formats: Dict[str, str] | None = None, projection_storage_location_template: str | None = None, catalog_id: str | None = None) -> Any
```

Write Parquet file or dataset on Amazon S3.

The concept of Dataset goes beyond the simple idea of ordinary files and enable more complex features like partitioning and catalog integration (Amazon Athena/AWS Glue Catalog).

Note: This operation may mutate the original pandas dataframe in-place. To avoid this behaviour please pass in a deep copy instead (i.e. `df.copy()`)

Note: If `database` and `table` arguments are passed, the table name and all column names will be automatically sanitized using `wr.catalog.sanitize_table_name` and `wr.catalog.sanitize_column_name`. Please, pass `sanitize_columns=True` to enforce this behaviour always.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `concurrent_partitioning`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **df** (`pandas.DataFrame`) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>

- **path** (*str*, *optional*) – S3 path (for file e.g. `s3://bucket/prefix/filename.parquet`) (for dataset e.g. `s3://bucket/prefix`). Required if `dataset=False` or when `dataset=True` and creating a new dataset
- **index** (*bool*) – True to store the DataFrame index in file, otherwise False to ignore it.
- **compression** (*str*, *optional*) – Compression style (None, snappy, gzip, zstd).
- **pyarrow_additional_kwargs** (*Optional[Dict[str, Any]]*) – Additional parameters forwarded to pyarrow. e.g. `pyarrow_additional_kwargs={'coerce_timestamps': 'ns', 'use_deprecated_int96_timestamps': False, 'allow_truncated_timestamps': False}`
- **max_rows_by_file** (*int*) – Max number of rows in each file. Default is None i.e. don't split the files. (e.g. 33554432, 268435456)
- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **sanitize_columns** (*bool*) – True to sanitize columns names (using `wr.catalog.sanitize_table_name` and `wr.catalog.sanitize_column_name`) or False to keep it as is. True value behaviour is enforced if `database` and `table` arguments are passed.
- **dataset** (*bool*) – If True store a parquet dataset instead of a ordinary file(s) If True, enable all follow arguments: `partition_cols`, `mode`, `database`, `table`, `description`, `parameters`, `columns_comments`, `concurrent_partitioning`, `catalog_versioning`, `projection_enabled`, `projection_types`, `projection_ranges`, `projection_values`, `projection_intervals`, `projection_digits`, `catalog_id`, `schema_evolution`.
- **filename_prefix** (*str*, *optional*) – If `dataset=True`, add a filename prefix to the output files.
- **partition_cols** (*List[str]*, *optional*) – List of column names that will be used to create partitions. Only takes effect if `dataset=True`.
- **bucketing_info** (*Tuple[List[str], int]*, *optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **concurrent_partitioning** (*bool*) – If True will increase the parallelism level during the partitions writing. It will decrease the writing time and increase the memory usage. <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html>
- **mode** (*str*, *optional*) – `append` (Default), `overwrite`, `overwrite_partitions`. Only takes effect if `dataset=True`. For details check the related tutorial: <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/004%20-%20Parquet%20Datasets.html>
- **catalog_versioning** (*bool*) – If True and `mode="overwrite"`, creates an archived version of the table catalog before updating it.
- **schema_evolution** (*bool*) – If True allows schema evolution (new or missing columns), otherwise a exception will be raised. True by default. (Only considered if `dataset=True` and `mode` in ("append", "overwrite_partitions")) Related tutorial: <https://aws-sdk-pandas.readthedocs.io/en/2.20.0/tutorials/014%20-%20Schema%20Evolution.html>

- **database** (*str, optional*) – Glue/Athena catalog: Database name.
- **table** (*str, optional*) – Glue/Athena catalog: Table name.
- **table_type** (*str, optional*) – The type of the Glue Table. Set to `EXTERNAL_TABLE` if None.
- **transaction_id** (*str, optional*) – The ID of the transaction when writing to a Governed Table.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. `{‘col name’: ‘bigint’, ‘col2 name’: ‘int’}`)
- **description** (*str, optional*) – Glue/Athena catalog: Table description
- **parameters** (*Dict[str, str], optional*) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str], optional*) – Glue/Athena catalog: Columns names and the related comments (e.g. `{‘col0’: ‘Column 0.’, ‘col1’: ‘Column 1.’, ‘col2’: ‘Partition.’}`).
- **regular_partitions** (*bool*) – Create regular partitions (Non projected partitions) on Glue Catalog. Disable when you will work only with Partition Projection. Keep enabled even when working with projections is useful to keep Redshift Spectrum working with the regular partitions.
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: “enum”, “integer”, “date”, “injected” <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{‘col_name’: ‘enum’, ‘col2_name’: ‘integer’}`)
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{‘col_name’: ‘0,10’, ‘col2_name’: ‘-1,8675309’}`)
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{‘col_name’: ‘A,B,Unknown’, ‘col2_name’: ‘foo,boo,bar’}`)
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{‘col_name’: ‘1’, ‘col2_name’: ‘5’}`)
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{‘col_name’: ‘1’, ‘col2_name’: ‘2’}`)
- **projection_formats** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections formats. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{‘col_date’: ‘yyyy-MM-dd’, ‘col2_timestamp’: ‘yyyy-MM-dd HH:mm:ss’}`)
- **projection_storage_location_template** (*Optional[str]*) – Value which allows Athena to properly map partition values if the S3 file locations do not follow a typical `.../column=value/...` pattern. <https://>

docs.aws.amazon.com/athena/latest/ug/partition-projection-setting-up.html (e.g.
 s3://bucket/table_root/a=\${a}/\${b}/some_static_subdirectory/\${c}/)

- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns

Dictionary with: 'paths': List of all stored files paths on S3. 'partitions_values': Dictionary of partitions added with keys as S3 path locations and values as a list of partitions values as str.

Return type

Dict[str, Union[List[str], Dict[str, List[str]]]]

Examples

Writing single file

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/prefix/my_file.parquet',
... )
{
  'paths': ['s3://bucket/prefix/my_file.parquet'],
  'partitions_values': {}
}
```

Writing single file encrypted with a KMS key

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/prefix/my_file.parquet',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
{
  'paths': ['s3://bucket/prefix/my_file.parquet'],
  'partitions_values': {}
}
```

Writing partitioned dataset

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
```

(continues on next page)

(continued from previous page)

```

...     dataset=True,
...     partition_cols=['col2']
... )
{
  'paths': ['s3://.../col2=A/x.parquet', 's3://.../col2=B/y.parquet'],
  'partitions_values': {
    's3://.../col2=A/': ['A'],
    's3://.../col2=B/': ['B']
  }
}

```

Writing bucketed dataset

```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     bucketing_info=(["col2"], 2)
... )
{
  'paths': ['s3://.../x_bucket-00000.csv', 's3://.../col2=B/x_bucket-00001.csv'],
  'partitions_values': {}
}

```

Writing dataset to S3 with metadata on Athena/Glue Catalog.

```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['col2'],
...     database='default', # Athena/Glue database
...     table='my_table' # Athena/Glue table
... )
{
  'paths': ['s3://.../col2=A/x.parquet', 's3://.../col2=B/y.parquet'],
  'partitions_values': {
    's3://.../col2=A/': ['A'],
    's3://.../col2=B/': ['B']
  }
}

```

Writing dataset to Glue governed table


```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B'],
...         'col3': [None, None, None]
...     }),
...     dataset=True,
...     mode='append',
...     database='default', # Athena/Glue database
...     table='my_table', # Athena/Glue table
...     table_type='GOVERNED',
...     transaction_id="xxx",
... )
{
  'paths': ['s3://.../x.parquet'],
  'partitions_values': {}
}

```

Writing dataset casting empty column data type

```

>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B'],
...         'col3': [None, None, None]
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     database='default', # Athena/Glue database
...     table='my_table' # Athena/Glue table
...     dtype={'col3': 'date'}
... )
{
  'paths': ['s3://.../x.parquet'],
  'partitions_values': {}
}

```

awswrangler.s3.upload

`awswrangler.s3.upload(local_file: str | Any, path: str, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, Any] | None = None) → None`

Upload file from a local file to received S3 path.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **local_file** (*Union[str, Any]*) – A file-like object in binary mode or a path to local file (e.g. `./local/path/to/key0`).
- **path** (*str*) – S3 path (e.g. `s3://bucket/key0`).
- **use_threads** (*bool, int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.

Return type

None

Examples

Uploading a file using a path to local file

```
>>> import awswrangler as wr
>>> wr.s3.upload(local_file='./key', path='s3://bucket/key')
```

Uploading a file using a file-like object

```
>>> import awswrangler as wr
>>> with open(file='./key', mode='wb') as local_f:
>>>     wr.s3.upload(local_file=local_f, path='s3://bucket/key')
```

awswrangler.s3.wait_objects_exist

`awswrangler.s3.wait_objects_exist(paths: List[str], delay: float | None = None, max_attempts: int | None = None, use_threads: bool | int = True, boto3_session: Session | None = None) → None`

Wait Amazon S3 objects exist.

Polls `S3.Client.head_object()` every 5 seconds (default) until a successful state is reached. An error is returned after 20 (default) failed checks. <https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Waiter.ObjectExists>

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **paths** (*List[str]*) – List of S3 objects paths (e.g. `[s3://bucket/key0, s3://bucket/key1]`).
- **delay** (*Union[int, float], optional*) – The amount of time in seconds to wait between attempts. Default: 5
- **max_attempts** (*int, optional*) – The maximum number of attempts to be made. Default: 20

- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.wait_objects_exist(['s3://bucket/key0', 's3://bucket/key1']) # wait both_
↪ objects
```

awswrangler.s3.wait_objects_not_exist

`awswrangler.s3.wait_objects_not_exist` (*paths: List[str]*, *delay: float | None = None*, *max_attempts: int | None = None*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*) → None

Wait Amazon S3 objects not exist.

Polls `S3.Client.head_object()` every 5 seconds (default) until a successful state is reached. An error is returned after 20 (default) failed checks. <https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Waiter.ObjectNotExists>

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **paths** (*List[str]*) – List of S3 objects paths (e.g. `s3://bucket/key0`, `s3://bucket/key1`).
- **delay** (*Union[int, float]*, *optional*) – The amount of time in seconds to wait between attempts. Default: 5
- **max_attempts** (*int*, *optional*) – The maximum number of attempts to be made. Default: 20
- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.wait_objects_not_exist(['s3://bucket/key0', 's3://bucket/key1']) # wait_
↳ both objects not exist
```

1.4.2 AWS Glue Catalog

<code>add_column(database, table, column_name[, ...])</code>	Add a column in a AWS Glue Catalog table.
<code>add_csv_partitions(database, table, ...[, ...])</code>	Add partitions (metadata) to a CSV Table in the AWS Glue Catalog.
<code>add_parquet_partitions(database, table, ...)</code>	Add partitions (metadata) to a Parquet Table in the AWS Glue Catalog.
<code>create_csv_table(database, table, path, ...)</code>	Create a CSV Table (Metadata Only) in the AWS Glue Catalog.
<code>create_database(name[, description, ...])</code>	Create a database in AWS Glue Catalog.
<code>create_json_table(database, table, path, ...)</code>	Create a JSON Table (Metadata Only) in the AWS Glue Catalog.
<code>create_parquet_table(database, table, path, ...)</code>	Create a Parquet Table (Metadata Only) in the AWS Glue Catalog.
<code>databases([limit, catalog_id, boto3_session])</code>	Get a Pandas DataFrame with all listed databases.
<code>delete_column(database, table, column_name)</code>	Delete a column in a AWS Glue Catalog table.
<code>delete_database(name[, catalog_id, ...])</code>	Delete a database in AWS Glue Catalog.
<code>delete_partitions(table, database, ...[, ...])</code>	Delete specified partitions in a AWS Glue Catalog table.
<code>delete_all_partitions(table, database[, ...])</code>	Delete all partitions in a AWS Glue Catalog table.
<code>delete_table_if_exists(database, table[, ...])</code>	Delete Glue table if exists.
<code>does_table_exist(database, table[, ...])</code>	Check if the table exists.
<code>drop_duplicated_columns(df)</code>	Drop all repeated columns (duplicated names).
<code>extract_athena_types(df[, index, ...])</code>	Extract columns and partitions types (Amazon Athena) from Pandas DataFrame.
<code>get_columns_comments(database, table[, ...])</code>	Get all columns comments.
<code>get_csv_partitions(database, table[, ...])</code>	Get all partitions from a Table in the AWS Glue Catalog.
<code>get_databases([catalog_id, boto3_session])</code>	Get an iterator of databases.
<code>get_parquet_partitions(database, table[, ...])</code>	Get all partitions from a Table in the AWS Glue Catalog.
<code>get_partitions(database, table[, ...])</code>	Get all partitions from a Table in the AWS Glue Catalog.
<code>get_table_description(database, table[, ...])</code>	Get table description.
<code>get_table_location(database, table[, ...])</code>	Get table's location on Glue catalog.
<code>get_table_number_of_versions(database, table)</code>	Get total number of versions.
<code>get_table_parameters(database, table[, ...])</code>	Get all parameters.
<code>get_table_types(database, table[, ...])</code>	Get all columns and types from a table.
<code>get_table_versions(database, table[, ...])</code>	Get all versions.
<code>get_tables([catalog_id, database, ...])</code>	Get an iterator of tables.
<code>overwrite_table_parameters(parameters, ...)</code>	Overwrite all existing parameters.
<code>sanitize_column_name(column)</code>	Convert the column name to be compatible with Amazon Athena and the AWS Glue Catalog.
<code>sanitize_dataframe_columns_names(df[, ...])</code>	Normalize all columns names to be compatible with Amazon Athena.
<code>sanitize_table_name(table)</code>	Convert the table name to be compatible with Amazon Athena and the AWS Glue Catalog.
<code>search_tables(text[, catalog_id, boto3_session])</code>	Get Pandas DataFrame of tables filtered by a search string.

continues on next page

Table 2 – continued from previous page

<code>table(database, table[, transaction_id, ...])</code>	Get table details as Pandas DataFrame.
<code>tables([limit, catalog_id, database, ...])</code>	Get a DataFrame with tables filtered by a search term, prefix, suffix.
<code>upsert_table_parameters(parameters, ...[, ...])</code>	Insert or Update the received parameters.

aws wrangler.catalog.add_column

`aws wrangler.catalog.add_column(database: str, table: str, column_name: str, column_type: str = 'string', column_comment: str | None = None, transaction_id: str | None = None, boto3_session: Session | None = None, catalog_id: str | None = None) → Any`

Add a column in a AWS Glue Catalog table.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **column_name** (*str*) – Column name
- **column_type** (*str*) – Column type.
- **column_comment** (*str*) – Column Comment
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns

None

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.add_column(
...     database='my_db',
...     table='my_table',
...     column_name='my_col',
...     column_type='int'
... )
```

awswrangler.catalog.add_csv_partitions

`awswrangler.catalog.add_csv_partitions(database: str, table: str, partitions_values: Dict[str, List[str]], bucketing_info: Tuple[List[str], int] | None = None, catalog_id: str | None = None, compression: str | None = None, sep: str = ',', serde_library: str | None = None, serde_parameters: Dict[str, str] | None = None, boto3_session: Session | None = None, columns_types: Dict[str, str] | None = None, partitions_parameters: Dict[str, str] | None = None) → Any`

Add partitions (metadata) to a CSV Table in the AWS Glue Catalog.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **partitions_values** (*Dict[str, List[str]]*) – Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. { 's3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **compression** (*str, optional*) – Compression style (None, gzip, etc).
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **serde_library** (*Optional[str]*) – Specifies the SerDe Serialization library which will be used. You need to provide the Class library name as a string. If no library is provided the default is `org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe`.
- **serde_parameters** (*Optional[str]*) – Dictionary of initialization parameters for the SerDe. The default is {`"field.delim": sep`, `"escape.delim": "\\"`}.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **columns_types** (*Optional[Dict[str, str]]*) – Only required for Hive compatibility. Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). P.S. Only materialized columns please, not partition columns.
- **partitions_parameters** (*Optional[Dict[str, str]]*) – Dictionary with key-value pairs defining partition parameters.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.add_csv_partitions(
...     database='default',
...     table='my_table',
...     partitions_values={
...         's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
...         's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
...         's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
...     }
... )
```

awswrangler.catalog.add_parquet_partitions

`awswrangler.catalog.add_parquet_partitions(database: str, table: str, partitions_values: Dict[str, List[str]], bucketing_info: Tuple[List[str], int] | None = None, catalog_id: str | None = None, compression: str | None = None, boto3_session: Session | None = None, columns_types: Dict[str, str] | None = None, partitions_parameters: Dict[str, str] | None = None) → Any`

Add partitions (metadata) to a Parquet Table in the AWS Glue Catalog.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.

- **partitions_values** (*Dict[str, List[str]]*) – Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).
- **bucketing_info** (*Tuple[List[str], int, optional]*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **compression** (*str, optional*) – Compression style (None, snappy, gzip, etc).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **columns_types** (*Optional[Dict[str, str]]*) – Only required for Hive compability. Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). P.S. Only materialized columns please, not partition columns.
- **partitions_parameters** (*Optional[Dict[str, str]]*) – Dictionary with key-value pairs defining partition parameters.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.add_parquet_partitions(
...     database='default',
...     table='my_table',
...     partitions_values={
...         's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
...         's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
...         's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
...     }
... )
```

awswrangler.catalog.create_csv_table


```

aws wrangler.catalog.create_csv_table(
    database: str, table: str, path: str, columns_types: Dict[str, str],
    table_type: str | None = None, partitions_types: Dict[str, str] | None
    = None, bucketing_info: Tuple[List[str], int] | None = None,
    compression: str | None = None, description: str | None = None,
    parameters: Dict[str, str] | None = None, columns_comments:
    Dict[str, str] | None = None, mode: str = 'overwrite',
    catalog_versioning: bool = False, schema_evolution: bool = False,
    sep: str = ',', skip_header_line_count: int | None = None,
    serde_library: str | None = None, serde_parameters: Dict[str, str] |
    None = None, transaction_id: str | None = None, boto3_session:
    Session | None = None, projection_enabled: bool = False,
    projection_types: Dict[str, str] | None = None, projection_ranges:
    Dict[str, str] | None = None, projection_values: Dict[str, str] | None
    = None, projection_intervals: Dict[str, str] | None = None,
    projection_digits: Dict[str, str] | None = None, projection_formats:
    Dict[str, str] | None = None, projection_storage_location_template:
    str | None = None, catalog_id: str | None = None) -> Any

```

Create a CSV Table (Metadata Only) in the AWS Glue Catalog.

<https://docs.aws.amazon.com/athena/latest/ug/data-types.html>

Note: Athena requires the columns in the underlying CSV files in S3 to be in the same order as the columns in the Glue data catalog.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. `s3://bucket/prefix/`).
- **columns_types** (*Dict[str, str]*) – Dictionary with keys as column names and values as data types (e.g. `{‘col0’: ‘bigint’, ‘col1’: ‘double’}`).
- **table_type** (*str, optional*) – The type of the Glue Table (`EXTERNAL_TABLE`, `GOVERNED...`). Set to `EXTERNAL_TABLE` if `None`
- **partitions_types** (*Dict[str, str], optional*) – Dictionary with keys as partition names and values as data types (e.g. `{‘col2’: ‘date’}`).
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **compression** (*str, optional*) – Compression style (`None`, `gzip`, etc).
- **description** (*str, optional*) – Table description

- **parameters** (*Dict[str, str]*, *optional*) – Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str]*, *optional*) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **mode** (*str*) – 'overwrite' to recreate any possible axisting table or 'append' to keep any possible axisting table.
- **catalog_versioning** (*bool*) – If True and *mode*="overwrite", creates an archived version of the table catalog before updating it.
- **schema_evolution** (*bool*) – If True allows schema evolution (new or missing columns), otherwise a exception will be raised. (Only considered if *dataset*=True and *mode* in ("append", "overwrite_partitions")) Related tutorial: <https://aws-sdk-pandas.readthedocs.io/en/2.11.0/tutorials/014%20-%20Schema%20Evolution.html>
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **skip_header_line_count** (*Optional[int]*) – Number of Lines to skip regarding to the header.
- **serde_library** (*Optional[str]*) – Specifies the SerDe Serialization library which will be used. You need to provide the Class library name as a string. If no library is provided the default is *org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe*.
- **serde_parameters** (*Optional[str]*) – Dictionary of initialization parameters for the SerDe. The default is {'field.delim': *sep*, "escape.delim": "\\\\"}.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': 'enum', 'col2_name': 'integer'})
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '0,10', 'col2_name': '-1,8675309'})
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '1', 'col2_name': '2'})
- **projection_formats** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections formats. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_date': 'yyyy-MM-dd', 'col2_timestamp': 'yyyy-MM-dd HH:mm:ss'})

- **projection_storage_location_template** (*Optional[str]*) – Value which allows Athena to properly map partition values if the S3 file locations do not follow a typical `.../column=value/...` pattern. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-setting-up.html> (e.g. `s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/${c}/`)
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.create_csv_table(
...     database='default',
...     table='my_table',
...     path='s3://bucket/prefix/',
...     columns_types={'col0': 'bigint', 'col1': 'double'},
...     partitions_types={'col2': 'date'},
...     compression='gzip',
...     description='My own table!',
...     parameters={'source': 'postgresql'},
...     columns_comments={'col0': 'Column 0.', 'col1': 'Column 1.', 'col2':
→ 'Partition.'}
... )
```

awswrangler.catalog.create_database

`awswrangler.catalog.create_database`(*name: str, description: str | None = None, catalog_id: str | None = None, exist_ok: bool = False, boto3_session: Session | None = None*)
→ Any

Create a database in AWS Glue Catalog.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **name** (*str*) – Database name.
- **description** (*str*, *optional*) – A Description for the Database.

- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **exist_ok** (*bool*) – If set to True will not raise an Exception if a Database with the same already exists. In this case the description will be updated if it is different from the current one.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.create_database(
...     name='awswrangler_test'
... )
```

awswrangler.catalog.create_json_table

`awswrangler.catalog.create_json_table(database: str, table: str, path: str, columns_types: Dict[str, str], table_type: str | None = None, partitions_types: Dict[str, str] | None = None, bucketing_info: Tuple[List[str], int] | None = None, compression: str | None = None, description: str | None = None, parameters: Dict[str, str] | None = None, columns_comments: Dict[str, str] | None = None, mode: str = 'overwrite', catalog_versioning: bool = False, schema_evolution: bool = False, serde_library: str | None = None, serde_parameters: Dict[str, str] | None = None, transaction_id: str | None = None, boto3_session: Session | None = None, projection_enabled: bool = False, projection_types: Dict[str, str] | None = None, projection_ranges: Dict[str, str] | None = None, projection_values: Dict[str, str] | None = None, projection_intervals: Dict[str, str] | None = None, projection_digits: Dict[str, str] | None = None, projection_formats: Dict[str, str] | None = None, projection_storage_location_template: str | None = None, catalog_id: str | None = None) → Any`

Create a JSON Table (Metadata Only) in the AWS Glue Catalog.

[‘https://docs.aws.amazon.com/athena/latest/ug/data-types.html’](https://docs.aws.amazon.com/athena/latest/ug/data-types.html)

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- catalog_id
- database

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. `s3://bucket/prefix/`).
- **columns_types** (*Dict[str, str]*) – Dictionary with keys as column names and values as data types (e.g. `{‘col0’: ‘bigint’, ‘col1’: ‘double’}`).
- **table_type** (*str, optional*) – The type of the Glue Table (`EXTERNAL_TABLE`, `GOVERNED...`). Set to `EXTERNAL_TABLE` if `None`
- **partitions_types** (*Dict[str, str], optional*) – Dictionary with keys as partition names and values as data types (e.g. `{‘col2’: ‘date’}`).
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **compression** (*str, optional*) – Compression style (`None`, `gzip`, etc).
- **description** (*str, optional*) – Table description
- **parameters** (*Dict[str, str], optional*) – Key/value pairs to tag the table.
- **columns_comments** (*Dict[str, str], optional*) – Columns names and the related comments (e.g. `{‘col0’: ‘Column 0.’, ‘col1’: ‘Column 1.’, ‘col2’: ‘Partition.’}`).
- **mode** (*str*) – ‘`overwrite`’ to recreate any possible axisting table or ‘`append`’ to keep any possible axisting table.
- **catalog_versioning** (*bool*) – If `True` and *mode*=“`overwrite`”, creates an archived version of the table catalog before updating it.
- **schema_evolution** (*bool*) – If `True` allows schema evolution (new or missing columns), otherwise a exception will be raised. (Only considered if *dataset*=`True` and *mode* in (“`append`”, “`overwrite_partitions`”)) Related tutorial: <https://aws-sdk-pandas.readthedocs.io/en/2.11.0/tutorials/014%20-%20Schema%20Evolution.html>
- **serde_library** (*Optional[str]*) – Specifies the SerDe Serialization library which will be used. You need to provide the Class library name as a string. If no library is provided the default is `org.openx.data.jsonserde.JsonSerDe`.
- **serde_parameters** (*Optional[str]*) – Dictionary of initialization parameters for the SerDe. The default is `{“field.delim”: sep, “escape.delim”: “\”}`.
- **transaction_id** (*str, optional*) – The ID of the transaction (i.e. used with `GOVERNED` tables).
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: “`enum`”, “`integer`”, “`date`”, “`injected`” <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{‘col_name’: ‘enum’, ‘col2_name’: ‘integer’}`)
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. `{‘col_name’: ‘0,10’, ‘col2_name’: ‘-1,8675309’}`)

- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '1', 'col2_name': '2'})
- **projection_formats** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections formats. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_date': 'yyyy-MM-dd', 'col2_timestamp': 'yyyy-MM-dd HH:mm:ss'})
- **projection_storage_location_template** (*Optional[str]*) – Value which allows Athena to properly map partition values if the S3 file locations do not follow a typical `.../column=value/...` pattern. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-setting-up.html> (e.g. `s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/${c}/`)
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.create_json_table(
...     database='default',
...     table='my_table',
...     path='s3://bucket/prefix/',
...     columns_types={'col0': 'bigint', 'col1': 'double'},
...     partitions_types={'col2': 'date'},
...     description='My very own JSON table!',
...     parameters={'source': 'postgresql'},
...     columns_comments={'col0': 'Column 0.', 'col1': 'Column 1.', 'col2':
↪ 'Partition.'}
... )
```

awswrangler.catalog.create_parquet_table

```
awswrangler.catalog.create_parquet_table(database: str, table: str, path: str, columns_types: Dict[str,
                                             str], table_type: str | None = None, partitions_types: Dict[str,
                                             str] | None = None, bucketing_info: Tuple[List[str], int] | None
                                             = None, catalog_id: str | None = None, compression: str |
                                             None = None, description: str | None = None, parameters:
                                             Dict[str, str] | None = None, columns_comments: Dict[str, str]
                                             | None = None, mode: str = 'overwrite', catalog_versioning:
                                             bool = False, transaction_id: str | None = None,
                                             projection_enabled: bool = False, projection_types: Dict[str,
                                             str] | None = None, projection_ranges: Dict[str, str] | None =
                                             None, projection_values: Dict[str, str] | None = None,
                                             projection_intervals: Dict[str, str] | None = None,
                                             projection_digits: Dict[str, str] | None = None,
                                             projection_formats: Dict[str, str] | None = None,
                                             projection_storage_location_template: str | None = None,
                                             boto3_session: Session | None = None) → Any
```

Create a Parquet Table (Metadata Only) in the AWS Glue Catalog.

[‘https://docs.aws.amazon.com/athena/latest/ug/data-types.html’](https://docs.aws.amazon.com/athena/latest/ug/data-types.html)

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **path** (*str*) – Amazon S3 path (e.g. `s3://bucket/prefix/`).
- **columns_types** (*Dict[str, str]*) – Dictionary with keys as column names and values as data types (e.g. `{‘col0’: ‘bigint’, ‘col1’: ‘double’}`).
- **table_type** (*str, optional*) – The type of the Glue Table (`EXTERNAL_TABLE`, `GOVERNED...`). Set to `EXTERNAL_TABLE` if `None`
- **partitions_types** (*Dict[str, str], optional*) – Dictionary with keys as partition names and values as data types (e.g. `{‘col2’: ‘date’}`).
- **bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **compression** (*str, optional*) – Compression style (`None`, `snappy`, `gzip`, etc).
- **description** (*str, optional*) – Table description
- **parameters** (*Dict[str, str], optional*) – Key/value pairs to tag the table.

- **columns_comments** (*Dict[str, str]*, *optional*) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **mode** (*str*) – 'overwrite' to recreate any possible existing table or 'append' to keep any possible existing table.
- **catalog_versioning** (*bool*) – If True and *mode*="overwrite", creates an archived version of the table catalog before updating it.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **projection_enabled** (*bool*) – Enable Partition Projection on Athena (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>)
- **projection_types** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections types. Valid types: "enum", "integer", "date", "injected" <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': 'enum', 'col2_name': 'integer'})
- **projection_ranges** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections ranges. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '0,10', 'col2_name': '-1,8675309'})
- **projection_values** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections values. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': 'A,B,Unknown', 'col2_name': 'foo,boo,bar'})
- **projection_intervals** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections intervals. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '1', 'col2_name': '5'})
- **projection_digits** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections digits. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_name': '1', 'col2_name': '2'})
- **projection_formats** (*Optional[Dict[str, str]]*) – Dictionary of partitions names and Athena projections formats. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-supported-types.html> (e.g. {'col_date': 'yyyy-MM-dd', 'col2_timestamp': 'yyyy-MM-dd HH:mm:ss'})
- **projection_storage_location_template** (*Optional[str]*) – Value which is allows Athena to properly map partition values if the S3 file locations do not follow a typical `.../column=value/...` pattern. <https://docs.aws.amazon.com/athena/latest/ug/partition-projection-setting-up.html> (e.g. `s3://bucket/table_root/a=${a}/${b}/some_static_subdirectory/${c}/`)
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.create_parquet_table(
...     database='default',
...     table='my_table',
...     path='s3://bucket/prefix/',
...     columns_types={'col0': 'bigint', 'col1': 'double'},
...     partitions_types={'col2': 'date'},
...     compression='snappy',
...     description='My own table!',
...     parameters={'source': 'postgresql'},
...     columns_comments={'col0': 'Column 0.', 'col1': 'Column 1.', 'col2':
↪ 'Partition.'}
... )
```

awswrangler.catalog.databases

`awswrangler.catalog.databases(limit: int = 100, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Get a Pandas DataFrame with all listed databases.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **limit** (*int, optional*) – Max number of tables to be returned.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Pandas DataFrame filled by formatted infos.

Return type

`pandas.DataFrame`

Examples

```
>>> import awswrangler as wr
>>> df_dbs = wr.catalog.databases()
```

awswrangler.catalog.delete_column

`awswrangler.catalog.delete_column(database: str, table: str, column_name: str, transaction_id: str | None = None, boto3_session: Session | None = None, catalog_id: str | None = None) → Any`

Delete a column in a AWS Glue Catalog table.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **column_name** (*str*) – Column name
- **transaction_id** (*str, optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Returns

None

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_column(
...     database='my_db',
...     table='my_table',
...     column_name='my_col',
... )
```

awswrangler.catalog.delete_database

awswrangler.catalog.delete_database(*name: str, catalog_id: str | None = None, boto3_session: Session | None = None*) → Any

Delete a database in AWS Glue Catalog.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `catalog_id`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **name** (*str*) – Database name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_database(
...     name='awswrangler_test'
... )
```

awswrangler.catalog.delete_partitions

awswrangler.catalog.delete_partitions(*table: str, database: str, partitions_values: List[List[str]], catalog_id: str | None = None, boto3_session: Session | None = None*) → Any

Delete specified partitions in a AWS Glue Catalog table.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – Table name.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **partitions_values** (*List[List[str]]*) – List of lists of partitions values as strings. (e.g. [['2020', '10', '25'], ['2020', '11', '16'], ['2020', '12', '19']]).
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_partitions(
...     table='my_table',
...     database='awswrangler_test',
...     partitions_values=[['2020', '10', '25'], ['2020', '11', '16'], ['2020', '12
→', '19']]
... )
```

awswrangler.catalog.delete_all_partitions

awswrangler.catalog.delete_all_partitions(*table: str, database: str, catalog_id: str | None = None, boto3_session: Session | None = None*) → Any

Delete all partitions in a AWS Glue Catalog table.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- **catalog_id**
- **database**

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – Table name.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Partitions values.

Return type

List[List[str]]

Examples

```
>>> import awswrangler as wr
>>> partitions = wr.catalog.delete_all_partitions(
...     table='my_table',
...     database='awswrangler_test',
... )
```

awswrangler.catalog.delete_table_if_exists

`awswrangler.catalog.delete_table_if_exists(database: str, table: str, transaction_id: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Delete Glue table if exists.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **transaction_id** (*str, optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

True if deleted, otherwise False.

Return type

bool

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.delete_table_if_exists(database='default', table='my_table') #_
↳deleted
True
>>> wr.catalog.delete_table_if_exists(database='default', table='my_table') #_
↳Nothing to be deleted
False
```

awswrangler.catalog.does_table_exist

`awswrangler.catalog.does_table_exist(database: str, table: str, boto3_session: Session | None = None, catalog_id: str | None = None, transaction_id: str | None = None) → Any`

Check if the table exists.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).

Returns

True if exists, otherwise False.

Return type

bool

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.does_table_exist(database='default', table='my_table')
```

awswrangler.catalog.drop_duplicated_columns

`awswrangler.catalog.drop_duplicated_columns(df: DataFrame) → DataFrame`

Drop all repeated columns (duplicated names).

Note: This transformation will run *inplace* and will make changes in the original DataFrame.

Note: It is different from Panda's `drop_duplicates()` function which considers the column values. `wr.catalog.drop_duplicated_columns()` will deduplicate by column name.

Parameters

df (`pandas.DataFrame`) – Original Pandas DataFrame.

Returns

Pandas DataFrame without duplicated columns.

Return type

`pandas.DataFrame`

Examples

```
>>> import awswrangler as wr
>>> df = pd.DataFrame({"A": [1, 2], "B": [3, 4]})
>>> df.columns = ["A", "A"]
>>> wr.catalog.drop_duplicated_columns(df=df)
  A
0  1
1  2
```

awswrangler.catalog.extract_athena_types

`awswrangler.catalog.extract_athena_types(df: DataFrame, index: bool = False, partition_cols: List[str] | None = None, dtype: Dict[str, str] | None = None, file_format: str = 'parquet') → Tuple[Dict[str, str], Dict[str, str]]`

Extract columns and partitions types (Amazon Athena) from Pandas DataFrame.

<https://docs.aws.amazon.com/athena/latest/ug/data-types.html>

Parameters

- **df** (`pandas.DataFrame`) – Pandas DataFrame.
- **index** (`bool`) – Should consider the DataFrame index as a column?.
- **partition_cols** (`List[str]`, *optional*) – List of partitions names.

- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **file_format** (*str, optional*) – File format to be considered to place the index column: “parquet” | “csv”.

Returns

columns_types: Dictionary with keys as column names and values as data types (e.g. {'col0': 'bigint', 'col1': 'double'}). / partitions_types: Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).

Return type

Tuple[Dict[str, str], Dict[str, str]]

Examples

```
>>> import awswrangler as wr
>>> columns_types, partitions_types = wr.catalog.extract_athena_types(
...     df=df, index=False, partition_cols=["par0", "par1"], file_format="csv"
... )
```

awswrangler.catalog.get_columns_comments

awswrangler.catalog.get_columns_comments(*database: str, table: str, transaction_id: str | None = None, query_as_of_time: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None*) → Any

Get all columns comments.

Note: If reading from a governed table, pass only one of *transaction_id* or *query_as_of_time*.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- *catalog_id*
- *database*

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **transaction_id** (*str, optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **query_as_of_time** (*str, optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside *transaction_id*.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Columns comments. e.g. {"col1": "foo boo bar"}.

Return type

Dict[str, str]

Examples

```
>>> import awswrangler as wr
>>> pars = wr.catalog.get_columns_comments(database="...", table="...")
```

awswrangler.catalog.get_csv_partitions

`awswrangler.catalog.get_csv_partitions(database: str, table: str, expression: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Get all partitions from a Table in the AWS Glue Catalog.

Expression argument instructions: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- catalog_id
- database

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **expression** (*str*, *optional*) – An expression that filters the partitions to be returned.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

partitions_values: Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. {'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}).

Return type

Dict[str, List[str]]

Examples

Fetch all partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_csv_partitions(
...     database='default',
...     table='my_table',
... )
{
  's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
  's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
  's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
}
```

Filtering partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_csv_partitions(
...     database='default',
...     table='my_table',
...     expression='m=10'
... )
{
  's3://bucket/prefix/y=2020/m=10/': ['2020', '10']
}
```

awswrangler.catalog.get_databases

`awswrangler.catalog.get_databases(catalog_id: str | None = None, boto3_session: Session | None = None)`
→ `Iterator[Dict[str, Any]]`

Get an iterator of databases.

Parameters

- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Iterator of Databases.

Return type

`Iterator[Dict[str, Any]]`

Examples

```
>>> import awswrangler as wr
>>> dbs = wr.catalog.get_databases()
```

awswrangler.catalog.get_parquet_partitions

`awswrangler.catalog.get_parquet_partitions(database: str, table: str, expression: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Get all partitions from a Table in the AWS Glue Catalog.

Expression argument instructions: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **expression** (*str*, *optional*) – An expression that filters the partitions to be returned.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

`partitions_values`: Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. `{ 's3://bucket/prefix/y=2020/m=10/': ['2020', '10'] }`).

Return type

`Dict[str, List[str]]`

Examples

Fetch all partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_parquet_partitions(
...     database='default',
...     table='my_table',
... )
{
```

(continues on next page)

(continued from previous page)

```

's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
}

```

Filtering partitions

```

>>> import awswrangler as wr
>>> wr.catalog.get_parquet_partitions(
...     database='default',
...     table='my_table',
...     expression='m=10'
... )
{
  's3://bucket/prefix/y=2020/m=10/': ['2020', '10']
}

```

awswrangler.catalog.get_partitions

`awswrangler.catalog.get_partitions(database: str, table: str, expression: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Get all partitions from a Table in the AWS Glue Catalog.

Expression argument instructions: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **expression** (*str*, *optional*) – An expression that filters the partitions to be returned.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.

Returns

`partitions_values`: Dictionary with keys as S3 path locations and values as a list of partitions values as str (e.g. `{'s3://bucket/prefix/y=2020/m=10/': ['2020', '10']}`).

Return type

`Dict[str, List[str]]`

Examples

Fetch all partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_partitions(
...     database='default',
...     table='my_table',
... )
{
  's3://bucket/prefix/y=2020/m=10/': ['2020', '10'],
  's3://bucket/prefix/y=2020/m=11/': ['2020', '11'],
  's3://bucket/prefix/y=2020/m=12/': ['2020', '12']
}
```

Filtering partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_partitions(
...     database='default',
...     table='my_table',
...     expression='m=10'
... )
{
  's3://bucket/prefix/y=2020/m=10/': ['2020', '10']
}
```

awswrangler.catalog.get_table_description

`awswrangler.catalog.get_table_description(database: str, table: str, transaction_id: str | None = None, query_as_of_time: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None) → str | None`

Get table description.

Note: If reading from a governed table, pass only one of *transaction_id* or *query_as_of_time*.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **query_as_of_time** (*str*, *optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside *transaction_id*.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if *boto3_session* receive None.

Returns

Description if exists.

Return type

Optional[str]

Examples

```
>>> import awswrangler as wr
>>> desc = wr.catalog.get_table_description(database="...", table="...")
```

awswrangler.catalog.get_table_location

`awswrangler.catalog.get_table_location(database: str, table: str, transaction_id: str | None = None, query_as_of_time: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Get table's location on Glue catalog.

Note: If reading from a governed table, pass only one of *transaction_id* or *query_as_of_time*.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **query_as_of_time** (*str*, *optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside *transaction_id*.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if *boto3_session* receive None.

Returns

Table's location.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.get_table_location(database='default', table='my_table')
's3://bucket/prefix/'
```

awswrangler.catalog.get_table_number_of_versions

`awswrangler.catalog.get_table_number_of_versions(database: str, table: str, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Get total number of versions.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

Total number of versions.

Return type

int

Examples

```
>>> import awswrangler as wr
>>> num = wr.catalog.get_table_number_of_versions(database="...", table="...")
```

awswrangler.catalog.get_table_parameters

```
awswrangler.catalog.get_table_parameters(database: str, table: str, transaction_id: str | None = None,
                                         query_as_of_time: str | None = None, catalog_id: str | None =
                                         None, boto3_session: Session | None = None) → Dict[str, str]
```

Get all parameters.

Note: If reading from a governed table, pass only one of *transaction_id* or *query_as_of_time*.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **query_as_of_time** (*str*, *optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside *transaction_id*.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if *boto3_session* receive None.

Returns

Dictionary of parameters.

Return type

Dict[str, str]

Examples

```
>>> import awswrangler as wr
>>> pars = wr.catalog.get_table_parameters(database="...", table="...")
```

awswrangler.catalog.get_table_types

```
awswrangler.catalog.get_table_types(database: str, table: str, transaction_id: str | None = None,
                                     query_as_of_time: str | None = None, catalog_id: str | None = None,
                                     boto3_session: Session | None = None) → Any
```

Get all columns and types from a table.

Note: If reading from a governed table, pass only one of *transaction_id* or *query_as_of_time*.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- *catalog_id*
- *database*

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **query_as_of_time** (*str*, *optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside `transaction_id`.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

If table exists, a dictionary like {'col name': 'col data type'}. Otherwise None.

Return type

Optional[Dict[str, str]]

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.get_table_types(database='default', table='my_table')
{'col0': 'int', 'col1': 'double'}
```

awswrangler.catalog.get_table_versions

`awswrangler.catalog.get_table_versions(database: str, table: str, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Get all versions.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

List of table inputs: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_table_versions

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> tables_versions = wr.catalog.get_table_versions(database="...", table="...")
```

awswrangler.catalog.get_tables

awswrangler.catalog.get_tables(*catalog_id: str | None = None, database: str | None = None, transaction_id: str | None = None, name_contains: str | None = None, name_prefix: str | None = None, name_suffix: str | None = None, boto3_session: Session | None = None*) → Any

Get an iterator of tables.

Note: Please, do not filter using name_contains and name_prefix/name_suffix at the same time. Only name_prefix and name_suffix can be combined together.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- catalog_id
- database

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **database** (*str, optional*) – Database name.
- **transaction_id** (*str, optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **name_contains** (*str, optional*) – Select by a specific string on table name
- **name_prefix** (*str, optional*) – Select by a specific prefix on table name
- **name_suffix** (*str, optional*) – Select by a specific suffix on table name
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Iterator of tables.

Return type

Iterator[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> tables = wr.catalog.get_tables()
```

awswrangler.catalog.override_table_parameters

`awswrangler.catalog.override_table_parameters(parameters: Dict[str, str], database: str, table: str, transaction_id: str | None = None, catalog_versioning: bool = False, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Override all existing parameters.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **parameters** (*Dict[str, str]*) – e.g. {"source": "mysql", "destination": "datalake"}
- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **transaction_id** (*str, optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

All parameters after the overwrite (The same received).

Return type

Dict[str, str]

Examples

```
>>> import awswrangler as wr
>>> pars = wr.catalog.overwrite_table_parameters(
...     parameters={"source": "mysql", "destination": "datalake"},
...     database="...",
...     table="...")
```

awswrangler.catalog.sanitize_column_name

awswrangler.catalog.sanitize_column_name(column: str) → str

Convert the column name to be compatible with Amazon Athena and the AWS Glue Catalog.

<https://docs.aws.amazon.com/athena/latest/ug/tables-databases-columns-names.html>

Possible transformations: - Strip accents - Remove non alphanumeric characters

Parameters

column (str) – Column name.

Returns

Normalized column name.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.sanitize_column_name('MyNewColumn')
'mynewcolumn'
```

awswrangler.catalog.sanitize_dataframe_columns_names

awswrangler.catalog.sanitize_dataframe_columns_names(df: DataFrame, handle_duplicate_columns: str | None = 'warn') → DataFrame

Normalize all columns names to be compatible with Amazon Athena.

<https://docs.aws.amazon.com/athena/latest/ug/tables-databases-columns-names.html>

Possible transformations: - Strip accents - Remove non alphanumeric characters

Note: After transformation, some column names might not be unique anymore. Example: the columns ["A", "a"] will be sanitized to ["a", "a"]

Parameters

- **df** (pandas.DataFrame) – Original Pandas DataFrame.
- **handle_duplicate_columns** (str, optional) – How to handle duplicate columns. Can be “warn” or “drop” or “rename”. “drop” will drop all but the first duplicated column. “rename” will rename all duplicated columns with an incremental number. Defaults to “warn”.

Returns

Original Pandas DataFrame with columns names normalized.

Return type

pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df_normalized = wr.catalog.sanitize_dataframe_columns_names(df=pd.DataFrame({"A": [1, 2]}))
>>> df_normalized_drop = wr.catalog.sanitize_dataframe_columns_names(
    df=pd.DataFrame({"A": [1, 2], "a": [3, 4]}), handle_duplicate_columns="drop"
)
>>> df_normalized_rename = wr.catalog.sanitize_dataframe_columns_names(
    df=pd.DataFrame({"A": [1, 2], "a": [3, 4], "a_1": [4, 6]}), handle_
duplicate_columns="rename"
)
```

awswrangler.catalog.sanitize_table_name

awswrangler.catalog.sanitize_table_name(table: str) → str

Convert the table name to be compatible with Amazon Athena and the AWS Glue Catalog.

<https://docs.aws.amazon.com/athena/latest/ug/tables-databases-columns-names.html>

Possible transformations: - Strip accents - Remove non alphanumeric characters

Parameters

table (str) – Table name.

Returns

Normalized table name.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> wr.catalog.sanitize_table_name('MyNewTable')
'mynewtable'
```

awswrangler.catalog.search_tables

`awswrangler.catalog.search_tables(text: str, catalog_id: str | None = None, boto3_session: Session | None = None) → Iterator[Dict[str, Any]]`

Get Pandas DataFrame of tables filtered by a search string.

Note: Search feature is not supported for Governed tables.

Parameters

- **text** (*str*, *optional*) – Select only tables with the given string in table's properties.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Iterator of tables.

Return type

Iterator[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> df_tables = wr.catalog.search_tables(text='my_property')
```

awswrangler.catalog.table

`awswrangler.catalog.table(database: str, table: str, transaction_id: str | None = None, query_as_of_time: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None) → Any`

Get table details as Pandas DataFrame.

Note: If reading from a governed table, pass only one of *transaction_id* or *query_as_of_time*.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.

- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **query_as_of_time** (*str*, *optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside `transaction_id`.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.

Returns

Pandas DataFrame filled by formatted infos.

Return type

`pandas.DataFrame`

Examples

```
>>> import awswrangler as wr
>>> df_table = wr.catalog.table(database='default', table='my_table')
```

awswrangler.catalog.tables

`awswrangler.catalog.tables`(*limit*: *int* = 100, *catalog_id*: *str* | *None* = *None*, *database*: *str* | *None* = *None*, *transaction_id*: *str* | *None* = *None*, *search_text*: *str* | *None* = *None*, *name_contains*: *str* | *None* = *None*, *name_prefix*: *str* | *None* = *None*, *name_suffix*: *str* | *None* = *None*, *boto3_session*: *Session* | *None* = *None*) → Any

Get a DataFrame with tables filtered by a search term, prefix, suffix.

Note: Search feature is not supported for Governed tables.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **limit** (*int*, *optional*) – Max number of tables to be returned.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **database** (*str*, *optional*) – Database name.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).

- **search_text** (*str*, *optional*) – Select only tables with the given string in table’s properties.
- **name_contains** (*str*, *optional*) – Select by a specific string on table name
- **name_prefix** (*str*, *optional*) – Select by a specific prefix on table name
- **name_suffix** (*str*, *optional*) – Select by a specific suffix on table name
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Pandas Dataframe filled by formatted infos.

Return type

pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df_tables = wr.catalog.tables()
```

awswrangler.catalog.upsert_table_parameters

awswrangler.catalog.upsert_table_parameters(*parameters: Dict[str, str]*, *database: str*, *table: str*, *transaction_id: str | None = None*, *catalog_versioning: bool = False*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → Any

Insert or Update the received parameters.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- *catalog_id*
- *database*

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **parameters** (*Dict[str, str]*) – e.g. {"source": "mysql", "destination": "datalake"}
- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **transaction_id** (*str*, *optional*) – The ID of the transaction (i.e. used with GOVERNED tables).
- **catalog_versioning** (*bool*) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

All parameters after the upsert.

Return type

Dict[str, str]

Examples

```
>>> import awswrangler as wr
>>> pars = wr.catalog.upsert_table_parameters(
...     parameters={"source": "mysql", "destination": "datalake"},
...     database="...",
...     table="...")
```

1.4.3 Amazon Athena

<code>create_athena_bucket</code> ([boto3_session])	Create the default Athena bucket if it doesn't exist.
<code>create_ctas_table</code> (sql[, database, ...])	Create a new table populated with the results of a SELECT query.
<code>generate_create_query</code> (table[, database, ...])	Generate the query that created a table(EXTERNAL_TABLE) or a view(VIRTUAL_TABLE).
<code>get_query_columns_types</code> (query_execution_id)	Get the data type of all columns queried.
<code>get_query_execution</code> (query_execution_id[, ...])	Fetch query execution details.
<code>get_query_executions</code> (query_execution_ids[, ...])	From specified query execution IDs, return a DataFrame of query execution details.
<code>get_query_results</code> (query_execution_id[, ...])	Get AWS Athena SQL query results as a Pandas DataFrame.
<code>get_named_query_statement</code> (named_query_id[, ...])	Get the named query statement string from a query ID.
<code>get_work_group</code> (workgroup[, boto3_session])	Return information about the workgroup with the specified name.
<code>list_query_executions</code> ([workgroup, boto3_session])	Fetch list query execution IDs ran in specified workgroup or primary work group if not specified.
<code>read_sql_query</code> (sql, database[, ...])	Execute any SQL query on AWS Athena and return the results as a Pandas DataFrame.
<code>read_sql_table</code> (table, database[, ...])	Extract the full table AWS Athena and return the results as a Pandas DataFrame.
<code>repair_table</code> (table[, database, data_source, ...])	Run the Hive's metastore consistency check: 'MSCK REPAIR TABLE table;'.
<code>start_query_execution</code> (sql[, database, ...])	Start a SQL Query against AWS Athena.
<code>stop_query_execution</code> (query_execution_id[, ...])	Stop a query execution.
<code>unload</code> (sql, path, database[, file_format, ...])	Write query results from a SELECT statement to the specified data format using UNLOAD.
<code>wait_query</code> (query_execution_id[, ...])	Wait for the query end.

awswrangler.athena.create_athena_bucket

awswrangler.athena.create_athena_bucket(boto3_session: Session | None = None) → str

Create the default Athena bucket if it doesn't exist.

Parameters

boto3_session (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Bucket s3 path (E.g. s3://aws-athena-query-results-ACCOUNT-REGION/)

Return type

str

Examples

```
>>> import awswrangler as wr
>>> wr.athena.create_athena_bucket()
's3://aws-athena-query-results-ACCOUNT-REGION/'
```

awswrangler.athena.create_ctas_table

awswrangler.athena.create_ctas_table(sql: str, database: str | None = None, ctas_table: str | None = None, ctas_database: str | None = None, s3_output: str | None = None, storage_format: str | None = None, write_compression: str | None = None, partitioning_info: List[str] | None = None, bucketing_info: Tuple[List[str], int] | None = None, field_delimiter: str | None = None, schema_only: bool = False, workgroup: str | None = None, data_source: str | None = None, encryption: str | None = None, kms_key: str | None = None, categories: List[str] | None = None, wait: bool = False, athena_query_wait_polling_delay: float = 0.25, boto3_session: Session | None = None) → Any

Create a new table populated with the results of a SELECT query.

<https://docs.aws.amazon.com/athena/latest/ug/create-table-as.html>

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- database
- athena_query_wait_polling_delay
- workgroup

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **sql** (str) – SELECT SQL query.
- **database** (Optional[str], optional) – The name of the database where the original table is stored.

- **ctas_table** (*Optional[str], optional*) – The name of the CTAS table. If None, a name with a random string is used.
- **ctas_database** (*Optional[str], optional*) – The name of the alternative database where the CTAS table should be stored. If None, *database* is used, that is the CTAS table is stored in the same database as the original table.
- **s3_output** (*Optional[str], optional*) – The output Amazon S3 path. If None, either the Athena workgroup or client-side location setting is used. If a workgroup enforces a query results location, then it overrides this argument.
- **storage_format** (*Optional[str], optional*) – The storage format for the CTAS query results, such as ORC, PARQUET, AVRO, JSON, or TEXTFILE. PARQUET by default.
- **write_compression** (*Optional[str], optional*) – The compression type to use for any storage format that allows compression to be specified.
- **partitioning_info** (*Optional[List[str]], optional*) – A list of columns by which the CTAS table will be partitioned.
- **bucketing_info** (*Optional[Tuple[List[str], int]], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **field_delimiter** (*Optional[str], optional*) – The single-character field delimiter for files in CSV, TSV, and text files.
- **schema_only** (*bool, optional*) – *_description_*, by default False
- **workgroup** (*Optional[str], optional*) – Athena workgroup.
- **data_source** (*Optional[str], optional*) – Data Source / Catalog name. If None, 'AwsDataCatalog' is used.
- **encryption** (*str, optional*) – Valid values: [None, 'SSE_S3', 'SSE_KMS']. Note: 'CSE_KMS' is not supported.
- **kms_key** (*str, optional*) – For SSE-KMS, this is the KMS key ARN or ID.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **wait** (*bool, default False*) – Whether to wait for the query to finish and return a dictionary with the Query metadata.
- **athena_query_wait_polling_delay** (*float, default: 0.25 seconds*) – Interval in seconds for how often the function will check if the Athena query has completed.
- **boto3_session** (*Optional[boto3.Session], optional*) – Boto3 Session. The default boto3 session is used if boto3_session is None.

Returns

A dictionary with the the CTAS database and table names. If *wait* is *False*, the query ID is included, otherwise a Query metadata object is added instead.

Return type

Dict[str, Union[str, _QueryMetadata]]

Examples

Select all into a new table and encrypt the results

```
>>> import awswrangler as wr
>>> wr.athena.create_ctas_table(
...     sql="select * from table",
...     database="default",
...     encryption="SSE_KMS",
...     kms_key="1234abcd-12ab-34cd-56ef-1234567890ab",
... )
{'ctas_database': 'default', 'ctas_table': 'temp_table_5669340090094....', 'ctas_
↪query_id': 'cc7dfa81-831d-...'}

```

Create a table with schema only

```
>>> wr.athena.create_ctas_table(
...     sql="select col1, col2 from table",
...     database="default",
...     ctas_table="my_ctas_table",
...     schema_only=True,
...     wait=True,
... )

```

Partition data and save to alternative CTAS database

```
>>> wr.athena.create_ctas_table(
...     sql="select * from table",
...     database="default",
...     ctas_database="my_ctas_db",
...     storage_format="avro",
...     write_compression="snappy",
...     partitioning_info=["par0", "par1"],
...     wait=True,
... )

```

awswrangler.athena.generate_create_query

`awswrangler.athena.generate_create_query`(*table: str, database: str | None = None, transaction_id: str | None = None, query_as_of_time: str | None = None, catalog_id: str | None = None, boto3_session: Session | None = None*) → Any

Generate the query that created a table(EXTERNAL_TABLE) or a view(VIRTUAL_TABLE).

Analyzes an existing table named `table_name` to generate the query that created it.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – Database name.
- **transaction_id** (*str*, *optional*) – The ID of the transaction.
- **query_as_of_time** (*str*, *optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside `transaction_id`.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

The query that created the table or view.

Return type

`str`

Examples

```
>>> import awswrangler as wr
>>> view_create_query: str = wr.athena.generate_create_query(table='my_view',
↳ database='default')
```

awswrangler.athena.get_query_columns_types

`awswrangler.athena.get_query_columns_types(query_execution_id: str, boto3_session: Session | None = None) → Dict[str, str]`

Get the data type of all columns queried.

<https://docs.aws.amazon.com/athena/latest/ug/data-types.html>

Parameters

- **query_execution_id** (*str*) – Athena query execution ID.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. If none, the default boto3 session is used.

Returns

Dictionary with all data types.

Return type

`Dict[str, str]`

Examples

```
>>> import awswrangler as wr
>>> wr.athena.get_query_columns_types('query-execution-id')
{'col0': 'int', 'col1': 'double'}
```

awswrangler.athena.get_query_execution

`awswrangler.athena.get_query_execution(query_execution_id: str, boto3_session: Session | None = None)`
→ Dict[str, Any]

Fetch query execution details.

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.get_query_execution

Parameters

- **query_execution_id** (*str*) – Athena query execution ID.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Dictionary with the get_query_execution response.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.get_query_execution(query_execution_id='query-execution-id')
```

awswrangler.athena.get_query_executions

`awswrangler.athena.get_query_executions(query_execution_ids: List[str], return_unprocessed: bool = False, boto3_session: Session | None = None)` → Tuple[DataFrame, DataFrame] | DataFrame

From specified query execution IDs, return a DataFrame of query execution details.

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.batch_get_query_execution

Parameters

- **query_execution_ids** (*List[str]*) – Athena query execution IDs.
- **return_unprocessed** (*bool*.) – True to also return query executions id that are unable to be processed. False to only return DataFrame of query execution details. Default is False
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

- *DataFrame* – DataFrame contain information about query execution details.

- *DataFrame* – DataFrames contain information about unprocessed query execution ids.

Examples

```
>>> import awswrangler as wr
>>> query_executions_df, unprocessed_query_executions_df = wr.athena.get_query_
↳ executions(
    query_execution_ids=['query-execution-id', 'query-execution-id1']
)
```

awswrangler.athena.get_query_results

`awswrangler.athena.get_query_results(query_execution_id: str, use_threads: bool | int = True, boto3_session: Session | None = None, categories: List[str] | None = None, chunksize: int | bool | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, pyarrow_additional_kwargs: Dict[str, Any] | None = None, athena_query_wait_polling_delay: float = 0.25) → Any`

Get AWS Athena SQL query results as a Pandas DataFrame.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `athena_query_wait_polling_delay`
- `chunksize`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **query_execution_id** (*str*) – SQL query’s execution_id on AWS Athena.
- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **categories** (*List[str]*, *optional*) – List of columns names that should be returned as `pandas.Categorical`. Recommended for memory restricted environments.
- **chunksize** (*Union[int, bool]*, *optional*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* awswrangler iterates on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed awswrangler will iterate on the data by number of rows equal the received *INTEGER*.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'RequestPayer': 'requester'}`
- **pyarrow_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to the `ParquetFile` class or converting an Arrow table to Pandas, currently only an “`coerce_int96_timestamp_unit`” or “`timestamp_as_object`” argument will be considered. If

reading parquet files where you cannot convert a timestamp to pandas Timestamp[ns] consider setting `timestamp_as_object=True`, to allow for timestamp units larger than “ns”. If reading parquet data that still uses INT96 (like Athena outputs) you can use `coerce_int96_timestamp_unit` to specify what timestamp unit to encode INT96 to (by default this is “ns”, if you know the output parquet came from a system that encodes timestamp to a particular unit then set this to that same unit e.g. `coerce_int96_timestamp_unit="ms"`).

- **athena_query_wait_polling_delay** (*float, default: 0.25 seconds*) – Interval in seconds for how often the function will check if the Athena query has completed.

Returns

Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

Return type

Union[pd.DataFrame, Iterator[pd.DataFrame]]

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.get_query_results(
...     query_execution_id="cbae5b41-8103-4709-95bb-887f88edd4f2"
... )
```

awswrangler.athena.get_named_query_statement

`awswrangler.athena.get_named_query_statement(named_query_id: str, boto3_session: Session | None = None) → str`

Get the named query statement string from a query ID.

Parameters

- **named_query_id** (*str*) – The unique ID of the query. Used to get the query statement from a saved query. Requires access to the workgroup where the query is saved.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. If none, the default boto3 session is used.

Returns

The named query statement string

Return type

str

awswrangler.athena.get_work_group

`awswrangler.athena.get_work_group(workgroup: str, boto3_session: Session | None = None) → Dict[str, Any]`

Return information about the workgroup with the specified name.

Parameters

- **workgroup** (*str*) – Work Group name.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.get_work_group

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.get_work_group(workgroup='workgroup_name')
```

awswrangler.athena.list_query_executions

`awswrangler.athena.list_query_executions(workgroup: str | None = None, boto3_session: Session | None = None) → List[str]`

Fetch list query execution IDs ran in specified workgroup or primary work group if not specified.

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/athena.html#Athena.Client.list_query_executions

Parameters

- **workgroup** (*str*) – The name of the workgroup from which the query_id are being returned. If not specified, a list of available query execution IDs for the queries in the primary work-group is returned.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

List of query execution IDs.

Return type

List[str]

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.list_query_executions(workgroup='workgroup-name')
```

awswrangler.athena.read_sql_query

```
aws wrangler.athena.read_sql_query(sql: str, database: str, ctas_approach: bool = True, unload_approach: bool = False, unload_parameters: Dict[str, Any] | None = None, categories: List[str] | None = None, chunksize: int | bool | None = None, s3_output: str | None = None, workgroup: str | None = None, encryption: str | None = None, kms_key: str | None = None, keep_files: bool = True, ctas_database_name: str | None = None, ctas_temp_table_name: str | None = None, ctas_bucketing_info: Tuple[List[str], int] | None = None, ctas_write_compression: str | None = None, use_threads: bool | int = True, boto3_session: Session | None = None, max_cache_seconds: int = 0, max_cache_query_inspections: int = 50, max_remote_cache_entries: int = 50, max_local_cache_entries: int = 100, data_source: str | None = None, athena_query_wait_polling_delay: float = 0.25, params: Dict[str, Any] | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, pyarrow_additional_kwargs: Dict[str, Any] | None = None) → Any
```

Execute any SQL query on AWS Athena and return the results as a Pandas DataFrame.

Related tutorial:

- [Amazon Athena](#)
- [Athena Cache](#)
- [Global Configurations](#)

There are three approaches available through `ctas_approach` and `unload_approach` parameters:

1 - `ctas_approach=True` (Default):

Wrap the query with a CTAS and then reads the table data as parquet directly from s3.

PROS:

- Faster for mid and big result sizes.
- Can handle some level of nested types.

CONS:

- Requires create/delete table permissions on Glue.
- Does not support timestamp with time zone
- Does not support columns with repeated names.
- Does not support columns with undefined data types.
- A temporary table will be created and then deleted immediately.
- Does not support custom `data_source/catalog_id`.

2 - `unload_approach=True` and `ctas_approach=False`:

Does an UNLOAD query on Athena and parse the Parquet result on s3.

PROS:

- Faster for mid and big result sizes.
- Can handle some level of nested types.
- Does not modify Glue Data Catalog

CONS:

- Output S3 path must be empty.

- Does not support timestamp with time zone.
- Does not support columns with repeated names.
- Does not support columns with undefined data types.

3 - `ctas_approach=False`:

Does a regular query on Athena and parse the regular CSV result on s3.

PROS:

- Faster for small result sizes (less latency).
- Does not require create/delete table permissions on Glue
- Supports timestamp with time zone.
- Support custom `data_source/catalog_id`.

CONS:

- Slower for big results (But stills faster than other libraries that uses the regular Athena's API)
- Does not handle nested types at all.

Note: The resulting DataFrame (or every DataFrame in the returned Iterator for chunked queries) have a `query_metadata` attribute, which brings the query result metadata returned by [Boto3/Athena](#) .

For a practical example check out the [related tutorial](#)!

Note: Valid encryption modes: [None, 'SSE_S3', 'SSE_KMS'].

P.S. 'CSE_KMS' is not supported.

Note: Create the default Athena bucket if it doesn't exist and `s3_output` is None.

(E.g. `s3://aws-athena-query-results-ACCOUNT-REGION/`)

Note: `chunksize` argument (Memory Friendly) (i.e batching):

Return an Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies:

- If **`chunksize=True`**, a new DataFrame will be returned for each file in the query result.
- If **`chunksize=INTEGER`**, `awswrangler` will iterate on the data by number of rows igual the received `INTEGER`.

P.S. `chunksize=True` is faster and uses less memory while `chunksize=INTEGER` is more precise in number of rows for each DataFrame.

P.P.S. If `ctas_approach=False` and `chunksize=True`, you will always receive an iterator with a single DataFrame because regular Athena queries only produces a single output file.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `ctas_approach`
- `database`
- `max_cache_query_inspections`
- `max_cache_seconds`
- `max_remote_cache_entries`
- `max_local_cache_entries`
- `athena_query_wait_polling_delay`
- `workgroup`
- `chunksize`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **sql** (*str*) – SQL query.
- **database** (*str*) – AWS Glue/Athena database name - It is only the origin database from where the query will be launched. You can still using and mixing several databases writing the full table name within the sql (e.g. `database.table`).
- **ctas_approach** (*bool*) – Wraps the query using a CTAS, and read the resulted parquet data on S3. If false, read the regular CSV on S3.
- **unload_approach** (*bool*) – Wraps the query using UNLOAD, and read the results from S3. Only PARQUET format is supported.
- **unload_parameters** (*Optional[Dict[str, Any]]*) – Params of the UNLOAD such as format, compression, field_delimiter, and partitioned_by.
- **categories** (*List[str], optional*) – List of columns names that should be returned as `pandas.Categorical`. Recommended for memory restricted environments.
- **chunksize** (*Union[int, bool], optional*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If `True` awswrangler iterates on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed awswrangler will iterate on the data by number of rows equal the received *INTEGER*.
- **s3_output** (*str, optional*) – Amazon S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.
- **encryption** (*str, optional*) – Valid values: `[None, 'SSE_S3', 'SSE_KMS']`. Notice: 'CSE_KMS' is not supported.
- **kms_key** (*str, optional*) – For SSE-KMS, this is the KMS key ARN or ID.

- **keep_files** (*bool*) – Whether staging files produced by Athena are retained. ‘True’ by default.
- **ctas_database_name** (*str, optional*) – The name of the alternative database where the CTAS temporary table is stored. If None, the default *database* is used.
- **ctas_temp_table_name** (*str, optional*) – The name of the temporary table and also the directory name on S3 where the CTAS result is stored. If None, it will use the follow random pattern: *f"temp_table_{uuid.uuid4().hex()}"*. On S3 this directory will be under under the pattern: *f"{s3_output}/{ctas_temp_table_name}/"*.
- **ctas_bucketing_info** (*Tuple[List[str], int], optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **ctas_write_compression** (*str, optional*) – Write compression for the temporary table where the CTAS result is stored. Corresponds to the *write_compression* parameters for CREATE TABLE AS statement in Athena.
- **use_threads** (*bool, int*) – True to enable concurrent requests, False to disable multiple threads. If enabled *os.cpu_count()* will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if *boto3_session* receive None.
- **max_cache_seconds** (*int*) – awswrangler can look up in Athena’s history if this query has been run before. If so, and its completion time is less than *max_cache_seconds* before now, awswrangler skips query execution and just returns the same results as last time. If cached results are valid, awswrangler ignores the *ctas_approach*, *s3_output*, *encryption*, *kms_key*, *keep_files* and *ctas_temp_table_name* params. If reading cached data fails for any reason, execution falls back to the usual query run path.
- **max_cache_query_inspections** (*int*) – Max number of queries that will be inspected from the history to try to find some result to reuse. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if *max_cache_seconds* > 0.
- **max_remote_cache_entries** (*int*) – Max number of queries that will be retrieved from AWS for cache inspection. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if *max_cache_seconds* > 0 and default value is 50.
- **max_local_cache_entries** (*int*) – Max number of queries for which metadata will be cached locally. This will reduce the latency and also enables keeping more than *max_remote_cache_entries* available for the cache. This value should not be smaller than *max_remote_cache_entries*. Only takes effect if *max_cache_seconds* > 0 and default value is 100.
- **data_source** (*str, optional*) – Data Source/Catalog name. If None, ‘AwsDataCatalog’ will be used by default.
- **athena_query_wait_polling_delay** (*float, default: 0.25 seconds*) – Interval in seconds for how often the function will check if the Athena query has completed.
- **params** (*Dict[str, any], optional*) – Dict of parameters that will be used for constructing the SQL query. Only named parameters are supported. The dict needs to contain the information in the form {‘name’: ‘value’} and the SQL query needs to contain *:name;*. Note that for varchar columns and similar, you must surround the value in single quotes.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. *s3_additional_kwargs*={'RequestPayer': 'requester'}

- **pyarrow_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to the ParquetFile class or converting an Arrow table to Pandas, currently only an “coerce_int96_timestamp_unit” or “timestamp_as_object” argument will be considered. If reading parquet files where you cannot convert a timestamp to pandas Timestamp[ns] consider setting timestamp_as_object=True, to allow for timestamp units larger than “ns”. If reading parquet data that still uses INT96 (like Athena outputs) you can use coerce_int96_timestamp_unit to specify what timestamp unit to encode INT96 to (by default this is “ns”, if you know the output parquet came from a system that encodes timestamp to a particular unit then set this to that same unit e.g. coerce_int96_timestamp_unit=“ms”).

Returns

Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

Return type

Union[pd.DataFrame, Iterator[pd.DataFrame]]

Examples

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_query(sql="...", database="...")
>>> scanned_bytes = df.query_metadata["Statistics"]["DataScannedInBytes"]
```

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_query(
...     sql="SELECT * FROM my_table WHERE name=:name; AND city=:city;",
...     params={"name": "'filtered_name'", "city": "'filtered_city'"}
... )
```

awswrangler.athena.read_sql_table

awswrangler.athena.read_sql_table(*table: str, database: str, ctas_approach: bool = True, unload_approach: bool = False, unload_parameters: Dict[str, Any] | None = None, categories: List[str] | None = None, chunksize: int | bool | None = None, s3_output: str | None = None, workgroup: str | None = None, encryption: str | None = None, kms_key: str | None = None, keep_files: bool = True, ctas_database_name: str | None = None, ctas_temp_table_name: str | None = None, ctas_bucketing_info: Tuple[List[str], int] | None = None, ctas_write_compression: str | None = None, use_threads: bool | int = True, boto3_session: Session | None = None, max_cache_seconds: int = 0, max_cache_query_inspections: int = 50, max_remote_cache_entries: int = 50, max_local_cache_entries: int = 100, data_source: str | None = None, s3_additional_kwargs: Dict[str, Any] | None = None, pyarrow_additional_kwargs: Dict[str, Any] | None = None) → Any*

Extract the full table AWS Athena and return the results as a Pandas DataFrame.

Related tutorial:

- [Amazon Athena](#)
- [Athena Cache](#)
- [Global Configurations](#)

There are two approaches to be defined through `ctas_approach` parameter:

1 - `ctas_approach=True` (Default):

Wrap the query with a CTAS and then reads the table data as parquet directly from s3.

PROS:

- Faster for mid and big result sizes.
- Can handle some level of nested types.

CONS:

- Requires create/delete table permissions on Glue.
- Does not support timestamp with time zone
- Does not support columns with repeated names.
- Does not support columns with undefined data types.
- A temporary table will be created and then deleted immediately.

2 - `ctas_approach=False`:

Does a regular query on Athena and parse the regular CSV result on s3.

PROS:

- Faster for small result sizes (less latency).
- Does not require create/delete table permissions on Glue
- Supports timestamp with time zone.

CONS:

- Slower for big results (But stills faster than other libraries that uses the regular Athena's API)
- Does not handle nested types at all.

Note: The resulting DataFrame (or every DataFrame in the returned Iterator for chunked queries) have a `query_metadata` attribute, which brings the query result metadata returned by [Boto3/Athena](#) .

For a practical example check out the [related tutorial](#)!

Note: Valid encryption modes: [None, 'SSE_S3', 'SSE_KMS'].

P.S. 'CSE_KMS' is not supported.

Note: Create the default Athena bucket if it doesn't exist and `s3_output` is None.

(E.g. `s3://aws-athena-query-results-ACCOUNT-REGION/`)

Note: `chunksize` argument (Memory Friendly) (i.e batching):

Return an Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies:

- If **`chunksize=True`**, a new DataFrame will be returned for each file in the query result.

- If **chunksize=INTEGER**, awswrangler will iterate on the data by number of rows equal the received INTEGER.

P.S. chunksize=True is faster and uses less memory while *chunksize=INTEGER* is more precise in number of rows for each DataFrame.

P.P.S. If ctas_approach=False and chunksize=True, you will always receive an iterator with a single DataFrame because regular Athena queries only produces a single output file.

Note: In case of *use_threads=True* the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `ctas_approach`
- `database`
- `max_cache_query_inspections`
- `max_cache_seconds`
- `max_remote_cache_entries`
- `max_local_cache_entries`
- `workgroup`
- `chunksize`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str*) – AWS Glue/Athena database name.
- **ctas_approach** (*bool*) – Wraps the query using a CTAS, and read the resulted parquet data on S3. If false, read the regular CSV on S3.
- **unload_approach** (*bool*) – Wraps the query using UNLOAD, and read the results from S3. Only PARQUET format is supported.
- **unload_parameters** (*Optional[Dict[str, Any]]*) – Params of the UNLOAD such as format, compression, field_delimiter, and partitioned_by.
- **categories** (*List[str], optional*) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **chunksize** (*Union[int, bool], optional*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* awswrangler iterates on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed awswrangler will iterate on the data by number of rows equal the received INTEGER.
- **s3_output** (*str, optional*) – AWS S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.

- **encryption** (*str*, *optional*) – Valid values: [None, 'SSE_S3', 'SSE_KMS']. Notice: 'CSE_KMS' is not supported.
- **kms_key** (*str*, *optional*) – For SSE-KMS, this is the KMS key ARN or ID.
- **keep_files** (*bool*) – Should awswrangler delete or keep the staging files produced by Athena?
- **ctas_database_name** (*str*, *optional*) – The name of the alternative database where the CTAS temporary table is stored. If None, the default *database* is used.
- **ctas_temp_table_name** (*str*, *optional*) – The name of the temporary table and also the directory name on S3 where the CTAS result is stored. If None, it will use the follow random pattern: *f"temp_table_{uuid.uuid4().hex}"*. On S3 this directory will be under under the pattern: *f"{s3_output}/{ctas_temp_table_name}/"*.
- **ctas_bucketing_info** (*Tuple[List[str], int]*, *optional*) – Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element. Only *str*, *int* and *bool* are supported as column data types for bucketing.
- **ctas_write_compression** (*str*, *optional*) – Write compression for the temporary table where the CTAS result is stored. Corresponds to the *write_compression* parameters for CREATE TABLE AS statement in Athena.
- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled *os.cpu_count()* will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if *boto3_session* receive None.
- **max_cache_seconds** (*int*) – awswrangler can look up in Athena's history if this table has been read before. If so, and its completion time is less than *max_cache_seconds* before now, awswrangler skips query execution and just returns the same results as last time. If cached results are valid, awswrangler ignores the *ctas_approach*, *s3_output*, *encryption*, *kms_key*, *keep_files* and *ctas_temp_table_name* params. If reading cached data fails for any reason, execution falls back to the usual query run path.
- **max_cache_query_inspections** (*int*) – Max number of queries that will be inspected from the history to try to find some result to reuse. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if *max_cache_seconds* > 0.
- **max_remote_cache_entries** (*int*) – Max number of queries that will be retrieved from AWS for cache inspection. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if *max_cache_seconds* > 0 and default value is 50.
- **max_local_cache_entries** (*int*) – Max number of queries for which metadata will be cached locally. This will reduce the latency and also enables keeping more than *max_remote_cache_entries* available for the cache. This value should not be smaller than *max_remote_cache_entries*. Only takes effect if *max_cache_seconds* > 0 and default value is 100.
- **data_source** (*str*, *optional*) – Data Source/Catalog name. If None, 'AwsDataCatalog' will be used by default.
- **s3_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to botocore requests. e.g. *s3_additional_kwargs*={ 'RequestPayer': 'requester' }
- **pyarrow_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forward to the ParquetFile class or converting an Arrow table to Pandas, currently only an "co-

ercoerce_int96_timestamp_unit" or "timestamp_as_object" argument will be considered. If reading parquet files where you cannot convert a timestamp to pandas Timestamp[ns] consider setting timestamp_as_object=True, to allow for timestamp units > NS. If reading parquet data that still uses INT96 (like Athena outputs) you can use coerce_int96_timestamp_unit to specify what timestamp unit to encode INT96 to (by default this is "ns", if you know the output parquet came from a system that encodes timestamp to a particular unit then set this to that same unit e.g. coerce_int96_timestamp_unit="ms").

Returns

Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

Return type

Union[pd.DataFrame, Iterator[pd.DataFrame]]

Examples

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_table(table="...", database="...")
>>> scanned_bytes = df.query_metadata["Statistics"]["DataScannedInBytes"]
```

awswrangler.athena.repair_table

`awswrangler.athena.repair_table`(*table: str, database: str | None = None, data_source: str | None = None, s3_output: str | None = None, workgroup: str | None = None, encryption: str | None = None, kms_key: str | None = None, athena_query_wait_polling_delay: float = 0.25, boto3_session: Session | None = None*) → Any

Run the Hive's metastore consistency check: 'MSCK REPAIR TABLE table;'.

Recovers partitions and data associated with partitions. Use this statement when you add partitions to the catalog. It is possible it will take some time to add all partitions. If this operation times out, it will be in an incomplete state where only a few partitions are added to the catalog.

Note: Create the default Athena bucket if it doesn't exist and s3_output is None. (E.g. s3://aws-athena-query-results-ACCOUNT-REGION/)

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- database
- athena_query_wait_polling_delay
- workgroup

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **table** (*str*) – Table name.
- **database** (*str, optional*) – AWS Glue/Athena database name.

- **data_source** (*Optional[str], optional*) – Data Source / Catalog name. If None, 'AwsDataCatalog' is used.
- **s3_output** (*str, optional*) – AWS S3 path.
- **workgroup** (*str, optional*) – Athena workgroup.
- **encryption** (*str, optional*) – None, 'SSE_S3', 'SSE_KMS', 'CSE_KMS'.
- **kms_key** (*str, optional*) – For SSE-KMS and CSE-KMS, this is the KMS key ARN or ID.
- **athena_query_wait_polling_delay** (*float, default: 0.25 seconds*) – Interval in seconds for how often the function will check if the Athena query has completed.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Query final state ('SUCCEEDED', 'FAILED', 'CANCELLED').

Return type

str

Examples

```
>>> import awswrangler as wr
>>> query_final_state = wr.athena.repair_table(table='...', database='...')
```

awswrangler.athena.start_query_execution

awswrangler.athena.start_query_execution(*sql: str, database: str | None = None, s3_output: str | None = None, workgroup: str | None = None, encryption: str | None = None, kms_key: str | None = None, params: Dict[str, Any] | None = None, boto3_session: Session | None = None, max_cache_seconds: int = 0, max_cache_query_inspections: int = 50, max_remote_cache_entries: int = 50, max_local_cache_entries: int = 100, athena_query_wait_polling_delay: float = 0.25, data_source: str | None = None, wait: bool = False*) → Any

Start a SQL Query against AWS Athena.

Note: Create the default Athena bucket if it doesn't exist and s3_output is None. (E.g. s3://aws-athena-query-results-ACCOUNT-REGION/)

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- database
- max_cache_query_inspections
- max_cache_seconds
- max_remote_cache_entries

- `max_local_cache_entries`
- `athena_query_wait_polling_delay`
- `workgroup`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **`sql`** (*str*) – SQL query.
- **`database`** (*str*, *optional*) – AWS Glue/Athena database name.
- **`s3_output`** (*str*, *optional*) – AWS S3 path.
- **`workgroup`** (*str*, *optional*) – Athena workgroup.
- **`encryption`** (*str*, *optional*) – None, 'SSE_S3', 'SSE_KMS', 'CSE_KMS'.
- **`kms_key`** (*str*, *optional*) – For SSE-KMS and CSE-KMS, this is the KMS key ARN or ID.
- **`params`** (*Dict[str, any]*, *optional*) – Dict of parameters that will be used for constructing the SQL query. Only named parameters are supported. The dict needs to contain the information in the form {'name': 'value'} and the SQL query needs to contain `:name;`. Note that for varchar columns and similar, you must surround the value in single quotes.
- **`boto3_session`** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **`max_cache_seconds`** (*int*) – awswrangler can look up in Athena's history if this query has been run before. If so, and its completion time is less than `max_cache_seconds` before now, awswrangler skips query execution and just returns the same results as last time. If cached results are valid, awswrangler ignores the `s3_output`, `encryption` and `kms_key` params. If reading cached data fails for any reason, execution falls back to the usual query run path.
- **`max_cache_query_inspections`** (*int*) – Max number of queries that will be inspected from the history to try to find some result to reuse. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if `max_cache_seconds` > 0.
- **`max_remote_cache_entries`** (*int*) – Max number of queries that will be retrieved from AWS for cache inspection. The bigger the number of inspection, the bigger will be the latency for not cached queries. Only takes effect if `max_cache_seconds` > 0 and default value is 50.
- **`max_local_cache_entries`** (*int*) – Max number of queries for which metadata will be cached locally. This will reduce the latency and also enables keeping more than `max_remote_cache_entries` available for the cache. This value should not be smaller than `max_remote_cache_entries`. Only takes effect if `max_cache_seconds` > 0 and default value is 100.
- **`athena_query_wait_polling_delay`** (*float*, *default: 0.25 seconds*) – Interval in seconds for how often the function will check if the Athena query has completed.
- **`data_source`** (*str*, *optional*) – Data Source/Catalog name. If None, 'AwsDataCatalog' will be used by default.
- **`wait`** (*bool*, *default False*) – Indicates whether to wait for the query to finish and return a dictionary with the query execution response.

Returns

Query execution ID if *wait* is set to *False*, dictionary with the `get_query_execution` response otherwise.

Return type

Union[str, Dict[str, Any]]

Examples

Querying into the default data source (Amazon s3 - 'AwsDataCatalog')

```
>>> import awswrangler as wr
>>> query_exec_id = wr.athena.start_query_execution(sql='...', database='...')
```

Querying into another data source (PostgreSQL, Redshift, etc)

```
>>> import awswrangler as wr
>>> query_exec_id = wr.athena.start_query_execution(sql='...', database='...', data_
↳ source='...')
```

awswrangler.athena.stop_query_execution

`awswrangler.athena.stop_query_execution(query_execution_id: str, boto3_session: Session | None = None) → None`

Stop a query execution.

Requires you to have access to the workgroup in which the query ran.

Parameters

- **query_execution_id** (*str*) – Athena query execution ID.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.

Returns

`None`.

Return type

`None`

Examples

```
>>> import awswrangler as wr
>>> wr.athena.stop_query_execution(query_execution_id='query-execution-id')
```

awswrangler.athena.unload

```
awswrangler.athena.unload(sql: str, path: str, database: str, file_format: str = 'PARQUET', compression: str |
    None = None, field_delimiter: str | None = None, partitioned_by: List[str] | None
    = None, workgroup: str | None = None, encryption: str | None = None, kms_key:
    str | None = None, boto3_session: Session | None = None, data_source: str | None
    = None, params: Dict[str, Any] | None = None, athena_query_wait_polling_delay:
    float = 0.25) → Any
```

Write query results from a SELECT statement to the specified data format using UNLOAD.

<https://docs.aws.amazon.com/athena/latest/ug/unload.html>

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `database`
- `athena_query_wait_polling_delay`
- `workgroup`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **sql** (*str*) – SQL query.
- **path** (*str*, *optional*) – Amazon S3 path.
- **database** (*str*) – AWS Glue/Athena database name - It is only the origin database from where the query will be launched. You can still using and mixing several databases writing the full table name within the sql (e.g. *database.table*).
- **file_format** (*str*) – File format of the output. Possible values are ORC, PARQUET, AVRO, JSON, or TEXTFILE
- **compression** (*Optional[str]*) – This option is specific to the ORC and Parquet formats. For ORC, possible values are lz4, snappy, zlib, or zstd. For Parquet, possible values are gzip or snappy. For ORC, the default is zlib, and for Parquet, the default is gzip.
- **field_delimiter** (*str*) – A single-character field delimiter for files in CSV, TSV, and other text formats.
- **partitioned_by** (*Optional[List[str]]*) – An array list of columns by which the output is partitioned.
- **workgroup** (*str*, *optional*) – Athena workgroup.
- **encryption** (*str*, *optional*) – Valid values: [None, 'SSE_S3', 'SSE_KMS']. Notice: 'CSE_KMS' is not supported.
- **kms_key** (*str*, *optional*) – For SSE-KMS, this is the KMS key ARN or ID.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **data_source** (*str*, *optional*) – Data Source/Catalog name. If None, 'AwsDataCatalog' will be used by default.
- **params** (*Dict[str, any]*, *optional*) – Dict of parameters that will be used for constructing the SQL query. Only named parameters are supported. The dict needs to contain

the information in the form `{'name': 'value'}` and the SQL query needs to contain `:name`. Note that for varchar columns and similar, you must surround the value in single quotes.

- **athena_query_wait_polling_delay** (*float, default: 0.25 seconds*) – Interval in seconds for how often the function will check if the Athena query has completed.

Returns

Query metadata including query execution id, dtypes, manifest & output location.

Return type

`_QueryMetadata`

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.unload(
...     sql="SELECT * FROM my_table WHERE name=:name; AND city=:city;",
...     params={"name": "'filtered_name'", "city": "'filtered_city'"}
... )
```

awswrangler.athena.wait_query

`awswrangler.athena.wait_query(query_execution_id: str, boto3_session: Session | None = None, athena_query_wait_polling_delay: float = 0.25) → Any`

Wait for the query end.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `athena_query_wait_polling_delay`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **query_execution_id** (*str*) – Athena query execution ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.
- **athena_query_wait_polling_delay** (*float, default: 0.25 seconds*) – Interval in seconds for how often the function will check if the Athena query has completed.

Returns

Dictionary with the `get_query_execution` response.

Return type

`Dict[str, Any]`

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.wait_query(query_execution_id='query-execution-id')
```

1.4.4 AWS Lake Formation

<code>read_sql_query(sql, database[, ...])</code>	Execute PartiQL query on AWS Glue Table (Transaction ID or time travel timestamp).
<code>read_sql_table(table, database[, ...])</code>	Extract all rows from AWS Glue Table (Transaction ID or time travel timestamp).
<code>cancel_transaction(transaction_id[, ...])</code>	Cancel the specified transaction.
<code>commit_transaction(transaction_id[, ...])</code>	Commit the specified transaction.
<code>describe_transaction(transaction_id[, ...])</code>	Return the status of a single transaction.
<code>extend_transaction(transaction_id[, ...])</code>	Indicate to the service that the specified transaction is still active and should not be canceled.
<code>start_transaction([read_only, time_out, ...])</code>	Start a new transaction and returns its transaction ID.
<code>wait_query(query_id[, boto3_session, ...])</code>	Wait for the query to end.

awswrangler.lakeformation.read_sql_query

`awswrangler.lakeformation.read_sql_query(sql: str, database: str, transaction_id: str | None = None, query_as_of_time: str | None = None, catalog_id: str | None = None, categories: List[str] | None = None, safe: bool = True, map_types: bool = True, use_threads: bool = True, boto3_session: Session | None = None, params: Dict[str, Any] | None = None) → Any`

Execute PartiQL query on AWS Glue Table (Transaction ID or time travel timestamp). Return Pandas DataFrame.

Note: ORDER BY operations are not honoured. i.e. `sql="SELECT * FROM my_table ORDER BY my_column"` is NOT valid

Note: The database must NOT be explicitly defined in the PartiQL statement. i.e. `sql="SELECT * FROM my_table"` is valid but `sql="SELECT * FROM my_db.my_table"` is NOT valid

Note: Pass one of `transaction_id` or `query_as_of_time`, not both.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **sql** (*str*) – partiQL query.
- **database** (*str*) – AWS Glue database name
- **transaction_id** (*str*, *optional*) – The ID of the transaction at which to read the table contents. Cannot be specified alongside `query_as_of_time`
- **query_as_of_time** (*str*, *optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside `transaction_id`
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **categories** (*Optional[List[str]]*, *optional*) – List of columns names that should be returned as `pandas.Categorical`. Recommended for memory restricted environments.
- **safe** (*bool*, *default True*) – For certain data types, a cast is needed in order to store the data in a pandas DataFrame or Series (e.g. timestamps are always stored as nanoseconds in pandas). This option controls whether it is a safe cast or not.
- **map_types** (*bool*, *default True*) – True to convert pyarrow DataTypes to pandas ExtensionDtypes. It is used to override the default pandas type for conversion of built-in pyarrow types or in absence of `pandas_metadata` in the Table schema.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. When enabled, `os.cpu_count()` is used as the max number of threads.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session is used if `boto3_session` receives None.
- **params** (*Dict[str, any]*, *optional*) – Dict of parameters used to format the partiQL query. Only named parameters are supported. The dict must contain the information in the form `{“name”: “value”}` and the SQL query must contain `:name`.

Returns

Pandas DataFrame.

Return type

`pd.DataFrame`

Examples

```
>>> import awswrangler as wr
>>> df = wr.lakeformation.read_sql_query(
...     sql="SELECT * FROM my_table;",
...     database="my_db",
...     catalog_id="111111111111"
... )
```

```
>>> import awswrangler as wr
>>> df = wr.lakeformation.read_sql_query(
...     sql="SELECT * FROM my_table LIMIT 10;",
...     database="my_db",
...     transaction_id=
↪ "1b62811fa3e02c4e5fdbaa642b752030379c4a8a70da1f8732ce6ccca47afdc9"
... )
```

```
>>> import awswrangler as wr
>>> df = wr.lakeformation.read_sql_query(
...     sql="SELECT * FROM my_table WHERE name=:name; AND city=:city;",
...     database="my_db",
...     query_as_of_time="1611142914",
...     params={"name": "'filtered_name'", "city": "'filtered_city'"}
... )
```

awswrangler.lakeformation.read_sql_table

`awswrangler.lakeformation.read_sql_table`(*table: str, database: str, transaction_id: str | None = None, query_as_of_time: str | None = None, catalog_id: str | None = None, categories: List[str] | None = None, safe: bool = True, map_types: bool = True, use_threads: bool = True, boto3_session: Session | None = None*) → Any

Extract all rows from AWS Glue Table (Transaction ID or time travel timestamp). Return Pandas DataFrame.

Note: ORDER BY operations are not honoured. i.e. `sql="SELECT * FROM my_table ORDER BY my_column"` is NOT valid

Note: Pass one of *transaction_id* or *query_as_of_time*, not both.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **table** (*str*) – AWS Glue table name.
- **database** (*str*) – AWS Glue database name
- **transaction_id** (*str, optional*) – The ID of the transaction at which to read the table contents. Cannot be specified alongside `query_as_of_time`
- **query_as_of_time** (*str, optional*) – The time as of when to read the table contents. Must be a valid Unix epoch timestamp. Cannot be specified alongside `transaction_id`
- **catalog_id** (*str, optional*) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **categories** (*Optional[List[str]], optional*) – List of columns names that should be returned as `pandas.Categorical`. Recommended for memory restricted environments.
- **safe** (*bool, default True*) – For certain data types, a cast is needed in order to store the data in a pandas DataFrame or Series (e.g. timestamps are always stored as nanoseconds in pandas). This option controls whether it is a safe cast or not.

- **map_types** (*bool*, *default True*) – True to convert pyarrow DataTypes to pandas ExtensionDtypes. It is used to override the default pandas type for conversion of built-in pyarrow types or in absence of pandas_metadata in the Table schema.
- **use_threads** (*bool*) – True to enable concurrent requests, False to disable multiple threads. When enabled, `os.cpu_count()` is used as the max number of threads.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session is used if `boto3_session` receives None.

Returns

Pandas DataFrame.

Return type

pd.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df = wr.lakeformation.read_sql_table(
...     table="my_table",
...     database="my_db",
...     catalog_id="111111111111",
... )
```

```
>>> import awswrangler as wr
>>> df = wr.lakeformation.read_sql_table(
...     table="my_table",
...     database="my_db",
...     transaction_id=
↳ "1b62811fa3e02c4e5fdbaa642b752030379c4a8a70da1f8732ce6ccca47afdc9",
... )
```

```
>>> import awswrangler as wr
>>> df = wr.lakeformation.read_sql_table(
...     table="my_table",
...     database="my_db",
...     query_as_of_time="1611142914",
...     use_threads=True,
... )
```

awswrangler.lakeformation.cancel_transaction

`awswrangler.lakeformation.cancel_transaction(transaction_id: str, boto3_session: Session | None = None) → None`

Cancel the specified transaction. Returns exception if the transaction was previously committed.

Parameters

- **transaction_id** (*str*) – The ID of the transaction.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` received None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.lakeformation.cancel_transaction(transaction_id="...")
```

awswrangler.lakeformation.commit_transaction

`awswrangler.lakeformation.commit_transaction(transaction_id: str, boto3_session: Session | None = None) → None`

Commit the specified transaction. Returns exception if the transaction was previously canceled.

Parameters

- **transaction_id** (*str*) – The ID of the transaction.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session received None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.lakeformation.commit_transaction(transaction_id="...")
```

awswrangler.lakeformation.describe_transaction

`awswrangler.lakeformation.describe_transaction(transaction_id: str, boto3_session: Session | None = None) → str`

Return the status of a single transaction.

Parameters

- **transaction_id** (*str*) – The ID of the transaction.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session received None.

Returns

Transaction status (i.e. active|committed|aborted).

Return type

str

Examples

```
>>> import awswrangler as wr
>>> status = wr.lakeformation.describe_transaction(transaction_id="...")
```

awswrangler.lakeformation.extend_transaction

`awswrangler.lakeformation.extend_transaction(transaction_id: str, boto3_session: Session | None = None) → None`

Indicate to the service that the specified transaction is still active and should not be canceled.

Parameters

- **transaction_id** (*str*) – The ID of the transaction.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session received None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.lakeformation.extend_transaction(transaction_id="...")
```

awswrangler.lakeformation.start_transaction

`awswrangler.lakeformation.start_transaction(read_only: bool | None = False, time_out: float | None = inf, boto3_session: Session | None = None) → str`

Start a new transaction and returns its transaction ID.

The transaction is periodically extended until it's committed, canceled or the defined time-out is reached.

Parameters

- **read_only** (*bool*, *optional*) – Indicates that that this transaction should be read only. Writes made using a read-only transaction ID will be rejected. Read-only transactions do not need to be committed.
- **time_out** (*float*, *optional*) – Maximum duration over which a transaction is extended.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session received None.

Returns

An opaque identifier for the transaction.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> transaction_id = wr.lakeformation.start_transaction(read_only=False)
```

awswrangler.lakeformation.wait_query

awswrangler.lakeformation.**wait_query**(*query_id: str, boto3_session: Session | None = None, lakeformation_query_wait_polling_delay: float = 2*) → Any

Wait for the query to end.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `lakeformation_query_wait_polling_delay`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **query_id** (*str*) – Lake Formation query execution ID.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session received None.
- **lakeformation_query_wait_polling_delay** (*float, default: 2 seconds*) – Interval in seconds for how often the function will check if the LakeFormation query has completed.

Returns

Dictionary with the `get_query_state` response.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> res = wr.lakeformation.wait_query(query_id='query-id')
```

1.4.5 Amazon Redshift

<code>connect([connection, secret_id, catalog_id, ...])</code>	Return a <code>redshift_connector</code> connection from a Glue Catalog or Secret Manager.
<code>connect_temp(cluster_identifier, user[, ...])</code>	Return a <code>redshift_connector</code> temporary connection (No password required).
<code>copy(df, path, con, table, schema[, ...])</code>	Load Pandas DataFrame as a Table on Amazon Redshift using parquet files on S3 as stage.
<code>copy_from_files(path, con, table, schema[, ...])</code>	Load Parquet files from S3 to a Table on Amazon Redshift (Through COPY command).
<code>read_sql_query(sql, con[, index_col, ...])</code>	Return a DataFrame corresponding to the result set of the query string.
<code>read_sql_table(table, con[, schema, ...])</code>	Return a DataFrame corresponding the table.
<code>to_sql(df, con, table, schema[, mode, ...])</code>	Write records stored in a DataFrame into Redshift.
<code>unload(sql, path, con[, iam_role, ...])</code>	Load Pandas DataFrame from a Amazon Redshift query result using Parquet files on s3 as stage.
<code>unload_to_files(sql, path, con[, iam_role, ...])</code>	Unload Parquet files on s3 from a Redshift query result (Through the UNLOAD command).

awswrangler.redshift.connect

`awswrangler.redshift.connect(connection: str | None = None, secret_id: str | None = None, catalog_id: str | None = None, dbname: str | None = None, boto3_session: Session | None = None, ssl: bool = True, timeout: int | None = None, max_prepared_statements: int = 1000, tcp_keepalive: bool = True, **kwargs: Any) → Connection`

Return a `redshift_connector` connection from a Glue Catalog or Secret Manager.

Note: You MUST pass a `connection` OR `secret_id`. Here is an example of the secret structure in Secrets Manager: { "host": "my-host.us-east-1.redshift.amazonaws.com", "username": "test", "password": "test", "engine": "redshift", "port": "5439", "dbname": "mydb" }

<https://github.com/aws/amazon-redshift-python-driver>

Parameters

- **connection** (*str*, *optional*) – Glue Catalog Connection name.
- **secret_id** (*Optional[str]*) – Specifies the secret containing the connection details that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **catalog_id** (*str*, *optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*str*, *optional*) – Optional database name to overwrite the stored one.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **ssl** (*bool*) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to `redshift_connector`. <https://github.com/aws/amazon-redshift-python-driver>
- **timeout** (*int*, *optional*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forward to

redshift_connector. <https://github.com/aws/amazon-redshift-python-driver>

- **max_prepared_statements** (*int*) – This parameter is forward to redshift_connector. <https://github.com/aws/amazon-redshift-python-driver>
- **tcp_keepalive** (*bool*) – If True then use TCP keepalive. The default is True. This parameter is forward to redshift_connector. <https://github.com/aws/amazon-redshift-python-driver>
- ****kwargs** (*Any*) – Forwarded to redshift_connector.connect. e.g. `is_serverless=True`, `serverless_acct_id='...'`, `serverless_work_group='...'`

Returns

redshift_connector connection.

Return type

redshift_connector.Connection

Examples

Fetching Redshift connection from Glue Catalog

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

Fetching Redshift connection from Secrets Manager

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect(secret_id="MY_SECRET")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.redshift.connect_temp

`awswrangler.redshift.connect_temp`(*cluster_identifier: str, user: str, database: str | None = None, duration: int = 900, auto_create: bool = True, db_groups: List[str] | None = None, boto3_session: Session | None = None, ssl: bool = True, timeout: int | None = None, max_prepared_statements: int = 1000, tcp_keepalive: bool = True, **kwargs: Any*) → Connection

Return a redshift_connector temporary connection (No password required).

<https://github.com/aws/amazon-redshift-python-driver>

Parameters

- **cluster_identifier** (*str*) – The unique identifier of a cluster. This parameter is case sensitive.
- **user** (*str, optional*) – The name of a database user.
- **database** (*str, optional*) – Database name. If None, the default Database is used.

- **duration** (*int*, *optional*) – The number of seconds until the returned temporary password expires. Constraint: minimum 900, maximum 3600. Default: 900
- **auto_create** (*bool*) – Create a database user with the name specified for the user named in user if one does not exist.
- **db_groups** (*List[str]*, *optional*) – A list of the names of existing database groups that the user named in user will join for the current session, in addition to any group memberships for an existing user. If not specified, a new user is added only to PUBLIC.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **ssl** (*bool*) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to redshift_connector. <https://github.com/aws/amazon-redshift-python-driver>
- **timeout** (*int*, *optional*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forward to redshift_connector. <https://github.com/aws/amazon-redshift-python-driver>
- **max_prepared_statements** (*int*) – This parameter is forward to redshift_connector. <https://github.com/aws/amazon-redshift-python-driver>
- **tcp_keepalive** (*bool*) – If True then use TCP keepalive. The default is True. This parameter is forward to redshift_connector. <https://github.com/aws/amazon-redshift-python-driver>
- ****kwargs** (*Any*) – Forwarded to redshift_connector.connect. e.g. `is_serverless=True`, `serverless_acct_id='...'`, `serverless_work_group='...'`

Returns

redshift_connector connection.

Return type

redshift_connector.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect_temp(cluster_identifier="my-cluster", user="test")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.redshift.copy

```
aws wrangler.redshift.copy(df: DataFrame, path: str, con: Connection, table: str, schema: str, iam_role: str | None = None, aws_access_key_id: str | None = None, aws_secret_access_key: str | None = None, aws_session_token: str | None = None, index: bool = False, dtype: Dict[str, str] | None = None, mode: str = 'append', overwrite_method: str = 'drop', diststyle: str = 'AUTO', distkey: str | None = None, sortstyle: str = 'COMPOUND', sortkey: List[str] | None = None, primary_keys: List[str] | None = None, varchar_lengths_default: int = 256, varchar_lengths: Dict[str, int] | None = None, serialize_to_json: bool = False, keep_files: bool = False, use_threads: bool | int = True, lock: bool = False, commit_transaction: bool = True, sql_copy_extra_params: List[str] | None = None, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, str] | None = None, max_rows_by_file: int | None = 10000000, precombine_key: str | None = None, use_column_names: bool = False) → None
```

Load Pandas DataFrame as a Table on Amazon Redshift using parquet files on S3 as stage.

This is a **HIGH** latency and **HIGH** throughput alternative to `wr.redshift.to_sql()` to load large DataFrames into Amazon Redshift through the **** SQL COPY command****.

This strategy has more overhead and requires more IAM privileges than the regular `wr.redshift.to_sql()` function, so it is only recommended to inserting +1K rows at once.

https://docs.aws.amazon.com/redshift/latest/dg/r_COPY.html

Note: If the table does not exist yet, it will be automatically created for you using the Parquet metadata to infer the columns data types.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **df** (`pandas.DataFrame`) – Pandas DataFrame.
- **path** (`str`) – S3 path to write stage files (e.g. `s3://bucket_name/any_name/`). Note: This path must be empty.
- **con** (`redshift_connector.Connection`) – Use `redshift_connector.connect()` to use “credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **table** (`str`) – Table name
- **schema** (`str`) – Schema name
- **iam_role** (`str`, *optional*) – AWS IAM role with the related permissions.
- **aws_access_key_id** (`str`, *optional*) – The access key for your AWS account.
- **aws_secret_access_key** (`str`, *optional*) – The secret key for your AWS account.
- **aws_session_token** (`str`, *optional*) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **index** (`bool`) – True to store the DataFrame index in file, otherwise False to ignore it.
- **dtype** (`Dict[str, str]`, *optional*) – Dictionary of columns names and Athena/Glue types to be casted. Useful when you have columns with undetermined or mixed data types. Only takes effect if `dataset=True`. (e.g. `{‘col name’: ‘bigint’, ‘col2 name’: ‘int’}`)

- **mode** (*str*) – Append, overwrite or upsert.
- **overwrite_method** (*str*) – Drop, cascade, truncate, or delete. Only applicable in overwrite mode.
 "drop" - DROP ... RESTRICT - drops the table. Fails if there are any views that depend on it.
 "cascade" - DROP ... CASCADE - drops the table, and all views that depend on it. "truncate"
 - TRUNCATE ... - truncates the table, but immediately commits current transaction & starts
 a new one, hence the overwrite happens in two transactions and is not atomic. "delete" -
 DELETE FROM ... - deletes all rows from the table. Slow relative to the other methods.
- **diststyle** (*str*) – Redshift distribution styles. Must be in ["AUTO", "EVEN", "ALL",
 "KEY"]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html
- **distkey** (*str*, *optional*) – Specifies a column name or positional number for the distri-
 bution key.
- **sortstyle** (*str*) – Sorting can be "COMPOUND" or "INTERLEAVED". https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html
- **sortkey** (*List[str]*, *optional*) – List of columns to be sorted.
- **primary_keys** (*List[str]*, *optional*) – Primary keys.
- **varchar_lengths_default** (*int*) – The size that will be set for all VARCHAR columns
 not specified with varchar_lengths.
- **varchar_lengths** (*Dict[str, int]*, *optional*) – Dict of VARCHAR length by
 columns. (e.g. {"col1": 10, "col5": 200}).
- **keep_files** (*bool*) – Should keep stage files?
- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple
 threads. If enabled os.cpu_count() will be used as the max number of threads. If integer is
 provided, specified number is used.
- **lock** (*bool*) – True to execute LOCK command inside the transaction to force serializable
 isolation.
- **commit_transaction** (*bool*) – Whether to commit the transaction. True by default.
- **sql_copy_extra_params** (*Optional[List[str]]*) – Additional copy parameters to
 pass to the command. For example: ["STATUPDATE ON"]
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3
 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*Dict[str, str]*, *optional*) – Forwarded to botocore re-
 quests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId':
 'YOUR_KMS_KEY_ARN'}
- **max_rows_by_file** (*int*) – Max number of rows in each file. Default is None i.e. dont
 split the files. (e.g. 33554432, 268435456)
- **precombine_key** (*str*, *optional*) – When there is a primary_key match during upsert,
 this column will change the upsert method, comparing the values of the specified column
 from source and target, and keeping the larger of the two. Will only work when mode =
 upsert.
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame
 for generating the INSERT SQL Query. E.g. If the DataFrame has two columns *col1* and
col3 and *use_column_names* is True, data will only be inserted into the database columns
col1 and *col3*.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.redshift.copy(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path="s3://bucket/my_parquet_files/",
...     con=con,
...     table="my_table",
...     schema="public",
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

awswrangler.redshift.copy_from_files

`awswrangler.redshift.copy_from_files`(*path: str, con: Connection, table: str, schema: str, iam_role: str | None = None, aws_access_key_id: str | None = None, aws_secret_access_key: str | None = None, aws_session_token: str | None = None, parquet_infer_sampling: float = 1.0, mode: str = 'append', overwrite_method: str = 'drop', diststyle: str = 'AUTO', distkey: str | None = None, sortstyle: str = 'COMPOUND', sortkey: List[str] | None = None, primary_keys: List[str] | None = None, varchar_lengths_default: int = 256, varchar_lengths: Dict[str, int] | None = None, serialize_to_json: bool = False, path_suffix: str | None = None, path_ignore_suffix: str | None = None, use_threads: bool | int = True, lock: bool = False, commit_transaction: bool = True, manifest: bool | None = False, sql_copy_extra_params: List[str] | None = None, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, str] | None = None, precombine_key: str | None = None, column_names: List[str] | None = None*) → None

Load Parquet files from S3 to a Table on Amazon Redshift (Through COPY command).

https://docs.aws.amazon.com/redshift/latest/dg/r_COPY.html

Note: If the table does not exist yet, it will be automatically created for you using the Parquet metadata to infer the columns data types.

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **path** (*str*) – S3 prefix (e.g. `s3://bucket/prefix/`)
- **con** (*redshift_connector.Connection*) – Use `redshift_connector.connect()` to use "credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **iam_role** (*str, optional*) – AWS IAM role with the related permissions.
- **aws_access_key_id** (*str, optional*) – The access key for your AWS account.
- **aws_secret_access_key** (*str, optional*) – The secret key for your AWS account.
- **aws_session_token** (*str, optional*) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **parquet_infer_sampling** (*float*) – Random sample ratio of files that will have the meta-data inspected. Must be $0.0 < sampling \leq 1.0$. The higher, the more accurate. The lower, the faster.
- **mode** (*str*) – Append, overwrite or upsert.
- **overwrite_method** (*str*) – Drop, cascade, truncate, or delete. Only applicable in overwrite mode.
 "drop" - DROP ... RESTRICT - drops the table. Fails if there are any views that depend on it.
 "cascade" - DROP ... CASCADE - drops the table, and all views that depend on it. "truncate"
 - TRUNCATE ... - truncates the table, but immediately commits current transaction & starts a new one, hence the overwrite happens in two transactions and is not atomic. "delete" - DELETE FROM ... - deletes all rows from the table. Slow relative to the other methods.
- **diststyle** (*str*) – Redshift distribution styles. Must be in ["AUTO", "EVEN", "ALL", "KEY"]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html
- **distkey** (*str, optional*) – Specifies a column name or positional number for the distribution key.
- **sortstyle** (*str*) – Sorting can be "COMPOUND" or "INTERLEAVED". https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html
- **sortkey** (*List[str], optional*) – List of columns to be sorted.
- **primary_keys** (*List[str], optional*) – Primary keys.
- **varchar_lengths_default** (*int*) – The size that will be set for all VARCHAR columns not specified with `varchar_lengths`.
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **serialize_to_json** (*bool*) – Should awswrangler add SERIALIZETOJSON parameter into the COPY command? SERIALIZETOJSON is necessary to load nested data https://docs.aws.amazon.com/redshift/latest/dg/ingest-super.html#copy_json
- **path_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes to be scanned on s3 for the schema extraction (e.g. [".gz.parquet", ".snappy.parquet"]). Only has effect during the table creation. If None, will try to read all files. (default)
- **path_ignore_suffix** (*Union[str, List[str], None]*) – Suffix or List of suffixes for S3 keys to be ignored during the schema extraction. (e.g. [".csv", "_SUCCESS"]). Only has effect during the table creation. If None, will try to read all files. (default)

- **use_threads** (*bool*, *int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **lock** (*bool*) – True to execute LOCK command inside the transaction to force serializable isolation.
- **commit_transaction** (*bool*) – Whether to commit the transaction. True by default.
- **manifest** (*bool*) – If set to true path argument accepts a S3 uri to a manifest file.
- **sql_copy_extra_params** (*Optional[List[str]]*) – Additional copy parameters to pass to the command. For example: ["STATUPDATE ON"]
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Dict[str, str]*, *optional*) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}`
- **precombine_key** (*str*, *optional*) – When there is a `primary_key` match during upsert, this column will change the upsert method, comparing the values of the specified column from source and target, and keeping the larger of the two. Will only work when `mode = upsert`.
- **column_names** (*List[str]*, *optional*) – List of column names to map source data fields to the target columns.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.redshift.copy_from_files(
...     path="s3://bucket/my_parquet_files/",
...     con=con,
...     table="my_table",
...     schema="public",
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

awswrangler.redshift.read_sql_query

awswrangler.redshift.read_sql_query(*sql*: str, *con*: Connection, *index_col*: str | List[str] | None = None, *params*: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, *chunksize*: int | None = None, *dtype*: Dict[str, DataType] | None = None, *safe*: bool = True, *timestamp_as_object*: bool = False) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding to the result set of the query string.

Note: For large extractions (1K+ rows) consider the function **wr.redshift.unload()**.

Parameters

- **sql** (str) – SQL query.
- **con** (redshift_connector.Connection) – Use redshift_connector.connect() to use "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **index_col** (Union[str, List[str]], optional) – Column(s) to set as index(MultiIndex).
- **params** (Union[List, Tuple, Dict], optional) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (int, optional) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (Dict[str, pyarrow.DataType], optional) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (bool) – Check for overflows or other unsafe data type conversions.
- **timestamp_as_object** (bool) – Cast non-nanosecond timestamps (np.datetime64) to objects.

Returns

Result as Pandas DataFrame(s).

Return type

Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.redshift.read_sql_query(
...     sql="SELECT * FROM public.my_table",
...     con=con
... )
>>> con.close()
```

awswrangler.redshift.read_sql_table

```
awswrangler.redshift.read_sql_table(table: str, con: Connection, schema: str | None = None, index_col: str | List[str] | None = None, params: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, chunksize: int | None = None, dtype: Dict[str, DataType] | None = None, safe: bool = True, timestamp_as_object: bool = False) → DataFrame | Iterator[DataFrame]
```

Return a DataFrame corresponding the table.

Note: For large extractions (1K+ rows) consider the function **wr.redshift.unload()**.

Parameters

- **table** (*str*) – Table name.
- **con** (*redshift_connector.Connection*) – Use `redshift_connector.connect()` to use “credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **schema** (*str*, *optional*) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if *None* (default).
- **index_col** (*Union[str, List[str]]*, *optional*) – Column(s) to set as index(*MultiIndex*).
- **params** (*Union[List, Tuple, Dict]*, *optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s *paramstyle*, is supported.
- **chunksize** (*int*, *optional*) – If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType]*, *optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the *PyArrow* types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.
- **timestamp_as_object** (*bool*) – Cast non-nanosecond timestamps (*np.datetime64*) to objects.

Returns

Result as *Pandas DataFrame(s)*.

Return type

Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.redshift.read_sql_table(
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

awswrangler.redshift.to_sql

awswrangler.redshift.to_sql(*df: DataFrame, con: Connection, table: str, schema: str, mode: str = 'append', overwrite_method: str = 'drop', index: bool = False, dtype: Dict[str, str] | None = None, diststyle: str = 'AUTO', distkey: str | None = None, sortstyle: str = 'COMPOUND', sortkey: List[str] | None = None, primary_keys: List[str] | None = None, varchar_lengths_default: int = 256, varchar_lengths: Dict[str, int] | None = None, use_column_names: bool = False, lock: bool = False, chunksize: int = 200, commit_transaction: bool = True, precombine_key: str | None = None*) → Any

Write records stored in a DataFrame into Redshift.

Note: For large DataFrames (1K+ rows) consider the function **wr.redshift.copy()**.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- chunksize

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **con** (*redshift_connector.Connection*) – Use *redshift_connector.connect()* to use "credentials directly or *wr.redshift.connect()* to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) – Append, overwrite or upsert.
- **overwrite_method** (*str*) – Drop, cascade, truncate, or delete. Only applicable in overwrite mode.
 - ”drop” - DROP ... RESTRICT - drops the table. Fails if there are any views that depend on it.
 - ”cascade” - DROP ... CASCADE - drops the table, and all views that depend on it.
 - ”truncate” - TRUNCATE ... - truncates the table, but immediately commits current transaction & starts

a new one, hence the overwrite happens in two transactions and is not atomic. “delete” - DELETE FROM ... - deletes all rows from the table. Slow relative to the other methods.

- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Redshift types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'VARCHAR(10)', 'col2 name': 'FLOAT'}) *diststyle* : str Redshift distribution styles. Must be in [“AUTO”, “EVEN”, “ALL”, “KEY”]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html
- **distkey** (*str, optional*) – Specifies a column name or positional number for the distribution key.
- **sortstyle** (*str*) – Sorting can be “COMPOUND” or “INTERLEAVED”. https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html
- **sortkey** (*List[str], optional*) – List of columns to be sorted.
- **primary_keys** (*List[str], optional*) – Primary keys.
- **varchar_lengths_default** (*int*) – The size that will be set for all VARCHAR columns not specified with *varchar_lengths*.
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns *col1* and *col3* and *use_column_names* is True, data will only be inserted into the database columns *col1* and *col3*.
- **lock** (*bool*) – True to execute LOCK command inside the transaction to force serializable isolation.
- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.
- **commit_transaction** (*bool*) – Whether to commit the transaction. True by default.
- **precombine_key** (*str, optional*) – When there is a primary_key match during upsert, this column will change the upsert method, comparing the values of the specified column from source and target, and keeping the larger of the two. Will only work when mode = upsert.

Returns

None.

Return type

None

Examples

Writing to Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.redshift.to_sql(
...     df=df,
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

awswrangler.redshift.unload

`awswrangler.redshift.unload(sql: str, path: str, con: Connection, iam_role: str | None = None, aws_access_key_id: str | None = None, aws_secret_access_key: str | None = None, aws_session_token: str | None = None, region: str | None = None, max_file_size: float | None = None, kms_key_id: str | None = None, categories: List[str] | None = None, chunked: bool | int = False, keep_files: bool = False, use_threads: bool | int = True, boto3_session: Session | None = None, s3_additional_kwargs: Dict[str, str] | None = None) → DataFrame | Iterator[DataFrame]`

Load Pandas DataFrame from a Amazon Redshift query result using Parquet files on s3 as stage.

This is a **HIGH** latency and **HIGH** throughput alternative to `wr.redshift.read_sql_query()/wr.redshift.read_sql_table()` to extract large Amazon Redshift data into a Pandas DataFrames through the **UNLOAD command**.

This strategy has more overhead and requires more IAM privileges than the regular `wr.redshift.read_sql_query()/wr.redshift.read_sql_table()` function, so it is only recommended to fetch 1k+ rows at once.

https://docs.aws.amazon.com/redshift/latest/dg/r_UNLOAD.html

Note: *Batching* (*chunked* argument) (Memory Friendly):

Will enable the function to return an Iterable of DataFrames instead of a regular DataFrame.

There are two batching strategies on awswrangler:

- If **chunked=True**, a new DataFrame will be returned for each file in your path/dataset.
- If **chunked=INTEGER**, awswrangler will iterate on the data by number of rows (equal to the received INTEGER).

P.S. chunked=True is faster and uses less memory while *chunked=INTEGER* is more precise in the number of rows for each Dataframe.

Note: In case of *use_threads=True* the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **sql** (*str*) – SQL query.
- **path** (*Union[str, List[str]]*) – S3 path to write stage files (e.g. `s3://bucket_name/any_name/`)
- **con** (*redshift_connector.Connection*) – Use `redshift_connector.connect()` to use “credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **iam_role** (*str, optional*) – AWS IAM role with the related permissions.
- **aws_access_key_id** (*str, optional*) – The access key for your AWS account.
- **aws_secret_access_key** (*str, optional*) – The secret key for your AWS account.
- **aws_session_token** (*str, optional*) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **region** (*str, optional*) – Specifies the AWS Region where the target Amazon S3 bucket is located. REGION is required for UNLOAD to an Amazon S3 bucket that isn’t in the same AWS Region as the Amazon Redshift cluster. By default, UNLOAD assumes that the target Amazon S3 bucket is located in the same AWS Region as the Amazon Redshift cluster.
- **max_file_size** (*float, optional*) – Specifies the maximum size (MB) of files that UNLOAD creates in Amazon S3. Specify a decimal value between 5.0 MB and 6200.0 MB. If None, the default maximum file size is 6200.0 MB.
- **kms_key_id** (*str, optional*) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3.
- **categories** (*List[str], optional*) – List of columns names that should be returned as `pandas.Categorical`. Recommended for memory restricted environments.
- **keep_files** (*bool*) – Should keep stage files?
- **chunked** (*Union[int, bool]*) – If passed will split the data in a Iterable of DataFrames (Memory friendly). If *True* `aws wrangler` iterates on the data by files in the most efficient way without guarantee of chunksize. If an *INTEGER* is passed `aws wrangler` will iterate on the data by number of rows equal the received *INTEGER*.
- **use_threads** (*bool, int*) – True to enable concurrent requests, False to disable multiple threads. If enabled `os.cpu_count()` will be used as the max number of threads. If integer is provided, specified number is used.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.
- **s3_additional_kwargs** (*Dict[str, str], optional*) – Forward to botocore requests, only “SSECustomerAlgorithm” and “SSECustomerKey” arguments will be considered.

Returns

Result as `Pandas DataFrame(s)`.

Return type

`Union[pandas.DataFrame, Iterator[pandas.DataFrame]]`

Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> df = wr.redshift.unload(
...     sql="SELECT * FROM public.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     con=con,
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

awswrangler.redshift.unload_to_files

`awswrangler.redshift.unload_to_files(sql: str, path: str, con: Connection, iam_role: str | None = None, aws_access_key_id: str | None = None, aws_secret_access_key: str | None = None, aws_session_token: str | None = None, region: str | None = None, unload_format: str | None = None, max_file_size: float | None = None, kms_key_id: str | None = None, manifest: bool = False, partition_cols: List[str] | None = None, boto3_session: Session | None = None) → None`

Unload Parquet files on s3 from a Redshift query result (Through the UNLOAD command).

https://docs.aws.amazon.com/redshift/latest/dg/r_UNLOAD.html

Note: In case of `use_threads=True` the number of threads that will be spawned will be gotten from `os.cpu_count()`.

Parameters

- **sql** (*str*) – SQL query.
- **path** (*Union[str, List[str]]*) – S3 path to write stage files (e.g. `s3://bucket_name/any_name/`)
- **con** (*redshift_connector.Connection*) – Use `redshift_connector.connect()` to use "credentials directly or `wr.redshift.connect()` to fetch it from the Glue Catalog.
- **iam_role** (*str, optional*) – AWS IAM role with the related permissions.
- **aws_access_key_id** (*str, optional*) – The access key for your AWS account.
- **aws_secret_access_key** (*str, optional*) – The secret key for your AWS account.
- **aws_session_token** (*str, optional*) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **region** (*str, optional*) – Specifies the AWS Region where the target Amazon S3 bucket is located. REGION is required for UNLOAD to an Amazon S3 bucket that isn't in the same AWS Region as the Amazon Redshift cluster. By default, UNLOAD assumes that the target Amazon S3 bucket is located in the same AWS Region as the Amazon Redshift cluster.
- **unload_format** (*str, optional*) – Format of the unloaded S3 objects from the query. Valid values: "CSV", "PARQUET". Case sensitive. Defaults to PARQUET.

- **max_file_size** (*float, optional*) – Specifies the maximum size (MB) of files that UNLOAD creates in Amazon S3. Specify a decimal value between 5.0 MB and 6200.0 MB. If None, the default maximum file size is 6200.0 MB.
- **kms_key_id** (*str, optional*) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3.
- **manifest** (*bool*) – Unload a manifest file on S3.
- **partition_cols** (*List[str], optional*) – Specifies the partition keys for the unload operation.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> con = wr.redshift.connect("MY_GLUE_CONNECTION")
>>> wr.redshift.unload_to_files(
...     sql="SELECT * FROM public.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     con=con,
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
>>> con.close()
```

1.4.6 PostgreSQL

<code>connect([connection, secret_id, catalog_id, ...])</code>	Return a pg8000 connection from a Glue Catalog Connection.
<code>read_sql_query(sql, con[, index_col, ...])</code>	Return a DataFrame corresponding to the result set of the query string.
<code>read_sql_table(table, con[, schema, ...])</code>	Return a DataFrame corresponding the table.
<code>to_sql(df, con, table, schema[, mode, ...])</code>	Write records stored in a DataFrame into PostgreSQL.

awswrangler.postgresql.connect

`awswrangler.postgresql.connect(connection: str | None = None, secret_id: str | None = None, catalog_id: str | None = None, dbname: str | None = None, boto3_session: Session | None = None, ssl_context: bool | SSLContext | None = None, timeout: int | None = None, tcp_keepalive: bool = True) → Connection`

Return a pg8000 connection from a Glue Catalog Connection.

<https://github.com/tlocke/pg8000>

Note: You MUST pass a *connection* OR *secret_id*. Here is an example of the secret structure in Secrets Manager: { "host": "postgresql-instance-wrangler.dr8vkeyrb9ml.us-east-1.rds.amazonaws.com", "username": "test",

```
"password": "test", "engine": "postgresql", "port": "3306", "dbname": "mydb" # Optional }
```

Parameters

- **connection** (*Optional[str]*) – Glue Catalog Connection name.
- **secret_id** (*Optional[str]*) – Specifies the secret containing the connection details that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*Optional[str]*) – Optional database name to overwrite the stored one.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **ssl_context** (*Optional[Union[bool, SSLContext]]*) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to pg8000. <https://github.com/tlocke/pg8000#functions>
- **timeout** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forward to pg8000. <https://github.com/tlocke/pg8000#functions>
- **tcp_keepalive** (*bool*) – If True then use TCP keepalive. The default is True. This parameter is forward to pg8000. <https://github.com/tlocke/pg8000#functions>

Returns

pg8000 connection.

Return type

pg8000.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.postgresql.read_sql_query

awswrangler.postgresql.read_sql_query(*sql: str, con: Connection, index_col: str | List[str] | None = None, params: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, chunksize: int | None = None, dtype: Dict[str, DataType] | None = None, safe: bool = True, timestamp_as_object: bool = False*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (*str*) – SQL query.

- **con** (*pg8000.Connection*) – Use `pg8000.connect()` to use credentials directly or `wr.postgresql.connect()` to fetch it from the Glue Catalog.
- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(`MultiIndex`).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s `paramstyle`, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where `chunksize` is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.
- **timestamp_as_object** (*bool*) – Cast non-nanosecond timestamps (`np.datetime64`) to objects.

Returns

Result as Pandas `DataFrame(s)`.

Return type

`Union[pandas.DataFrame, Iterator[pandas.DataFrame]]`

Examples

Reading from PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> df = wr.postgresql.read_sql_query(
...     sql="SELECT * FROM public.my_table",
...     con=con
... )
>>> con.close()
```

awswrangler.postgresql.read_sql_table

`awswrangler.postgresql.read_sql_table`(*table: str, con: Connection, schema: str | None = None, index_col: str | List[str] | None = None, params: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, chunksize: int | None = None, dtype: Dict[str, DataType] | None = None, safe: bool = True, timestamp_as_object: bool = False*) → `DataFrame | Iterator[DataFrame]`

Return a `DataFrame` corresponding the table.

Parameters

- **table** (*str*) – Table name.
- **con** (*pg8000.Connection*) – Use `pg8000.connect()` to use credentials directly or `wr.postgresql.connect()` to fetch it from the Glue Catalog.

- **schema** (*str*, *optional*) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index_col** (*Union[str, List[str]]*, *optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict]*, *optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s paramstyle, is supported.
- **chunksize** (*int*, *optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType]*, *optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.
- **timestamp_as_object** (*bool*) – Cast non-nanosecond timestamps (np.datetime64) to objects.

Returns

Result as Pandas DataFrame(s).

Return type

Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> df = wr.postgresql.read_sql_table(
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

awswrangler.postgresql.to_sql

awswrangler.postgresql.to_sql(*df: DataFrame*, *con: Connection*, *table: str*, *schema: str*, *mode: str = 'append'*, *index: bool = False*, *dtype: Dict[str, str] | None = None*, *varchar_lengths: Dict[str, int] | None = None*, *use_column_names: bool = False*, *chunksize: int = 200*, *upsert_conflict_columns: List[str] | None = None*, *insert_conflict_columns: List[str] | None = None*) → Any

Write records stored in a DataFrame into PostgreSQL.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- chunksize

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **con** (*pg8000.Connection*) – Use `pg8000.connect()` to use credentials directly or `wr.postgresql.connect()` to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) –

Append, overwrite or upsert.

`append`: Inserts new records into table. `overwrite`: Drops table and recreates. `upsert`: Perform an upsert which checks for conflicts on columns given by `upsert_conflict_columns` and sets the new values on conflicts. Note that `upsert_conflict_columns` is required for this mode.

- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and PostgreSQL types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. `{‘col name’: ‘TEXT’, ‘col2 name’: ‘FLOAT’}`)
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. `{“col1”: 10, “col5”: 200}`).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns `col1` and `col3` and `use_column_names` is True, data will only be inserted into the database columns `col1` and `col3`.
- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.
- **upsert_conflict_columns** (*List[str], optional*) – This parameter is only supported if `mode` is set top `upsert`. In this case conflicts for the given columns are checked for evaluating the upsert.
- **insert_conflict_columns** (*List[str], optional*) – This parameter is only supported if `mode` is set top `append`. In this case conflicts for the given columns are checked for evaluating the insert ‘ON CONFLICT DO NOTHING’.

Returns

None.

Return type

None

Examples

Writing to PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.postgresql.connect("MY_GLUE_CONNECTION")
>>> wr.postgresql.to_sql(
...     df=df,
...     table="my_table",
...     schema="public",
...     con=con
... )
>>> con.close()
```

1.4.7 MySQL

<code>connect([connection, secret_id, catalog_id, ...])</code>	Return a pymysql connection from a Glue Catalog Connection or Secrets Manager.
<code>read_sql_query(sql, con[, index_col, ...])</code>	Return a DataFrame corresponding to the result set of the query string.
<code>read_sql_table(table, con[, schema, ...])</code>	Return a DataFrame corresponding the table.
<code>to_sql(df, con, table, schema[, mode, ...])</code>	Write records stored in a DataFrame into MySQL.

awswrangler.mysql.connect

`awswrangler.mysql.connect`(*connection*: str | None = None, *secret_id*: str | None = None, *catalog_id*: str | None = None, *dbname*: str | None = None, *boto3_session*: ~boto3.session.Session | None = None, *read_timeout*: int | None = None, *write_timeout*: int | None = None, *connect_timeout*: int = 10, *cursorclass*: ~typing.Type[~pymysql.cursors.Cursor] = <class 'pymysql.cursors.Cursor'>) → pymysql.connections.Connection[Any]

Return a pymysql connection from a Glue Catalog Connection or Secrets Manager.

<https://pymysql.readthedocs.io>

Note: You MUST pass a *connection* OR *secret_id*. Here is an example of the secret structure in Secrets Manager: { "host": "mysql-instance-wrangler.dr8vkeyrb9m1.us-east-1.rds.amazonaws.com", "username": "test", "password": "test", "engine": "mysql", "port": "3306", "dbname": "mydb" # Optional }

Note: It is only possible to configure SSL using Glue Catalog Connection. More at: <https://docs.aws.amazon.com/glue/latest/dg/connection-defining.html>

Note: Consider using SSCursor *cursorclass* for queries that return a lot of data. More at: <https://pymysql.readthedocs.io/en/latest/modules/cursors.html#pymysql.cursors.SSCursor>

Parameters

- **connection** (str) – Glue Catalog Connection name.

- **secret_id** (*Optional[str]*) – Specifies the secret containing the connection details that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*Optional[str]*) – Optional database name to overwrite the stored one.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **read_timeout** (*Optional[int]*) – The timeout for reading from the connection in seconds (default: None - no timeout). This parameter is forward to pymysql. <https://pymysql.readthedocs.io/en/latest/modules/connections.html>
- **write_timeout** (*Optional[int]*) – The timeout for writing to the connection in seconds (default: None - no timeout) This parameter is forward to pymysql. <https://pymysql.readthedocs.io/en/latest/modules/connections.html>
- **connect_timeout** (*int*) – Timeout before throwing an exception when connecting. (default: 10, min: 1, max: 31536000) This parameter is forward to pymysql. <https://pymysql.readthedocs.io/en/latest/modules/connections.html>
- **cursorclass** (*Cursor*) – Cursor class to use, e.g. SSCursor; defaults to pymysql.cursors.Cursor <https://pymysql.readthedocs.io/en/latest/modules/cursors.html>

Returns

pymysql connection.

Return type

pymysql.connections.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.mysql.read_sql_query

awswrangler.mysql.read_sql_query(*sql: str, con: pymysql.connections.Connection[Any], index_col: str | List[str] | None = None, params: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, chunksize: int | None = None, dtype: Dict[str, DataType] | None = None, safe: bool = True, timestamp_as_object: bool = False*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (*str*) – SQL query.
- **con** (*pymysql.connections.Connection*) – Use pymysql.connect() to use credentials directly or wr.mysql.connect() to fetch it from the Glue Catalog.

- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.
- **timestamp_as_object** (*bool*) – Cast non-nanosecond timestamps (np.datetime64) to objects.

Returns

Result as Pandas DataFrame(s).

Return type

Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> df = wr.mysql.read_sql_query(
...     sql="SELECT * FROM test.my_table",
...     con=con
... )
>>> con.close()
```

awswrangler.mysql.read_sql_table

awswrangler.mysql.read_sql_table(*table: str, con: pymysql.connections.Connection[Any], schema: str | None = None, index_col: str | List[str] | None = None, params: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, chunksize: int | None = None, dtype: Dict[str, DataType] | None = None, safe: bool = True, timestamp_as_object: bool = False*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding the table.

Parameters

- **table** (*str*) – Table name.
- **con** (*pymysql.connections.Connection*) – Use pymysql.connect() to use credentials directly or wr.mysql.connect() to fetch it from the Glue Catalog.
- **schema** (*str, optional*) – Name of SQL schema in database to query. Uses default schema if None.

- **index_col** (*Union[str, List[str]], optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict], optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249’s paramstyle, is supported.
- **chunksize** (*int, optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType], optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.
- **timestamp_as_object** (*bool*) – Cast non-nanosecond timestamps (np.datetime64) to objects.

Returns

Result as Pandas DataFrame(s).

Return type

Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Reading from MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> df = wr.mysql.read_sql_table(
...     table="my_table",
...     schema="test",
...     con=con
... )
>>> con.close()
```

awswrangler.mysql.to_sql

`awswrangler.mysql.to_sql(df: ~pandas.core.frame.DataFrame, con: pymysql.connections.Connection[Any], table: str, schema: str, mode: str = 'append', index: bool = False, dtype: ~typing.Dict[str, str] | None = None, varchar_lengths: ~typing.Dict[str, int] | None = None, use_column_names: bool = False, chunksize: int = 200, cursorclass: ~typing.Type[~pymysql.cursors.Cursor] = <class 'pymysql.cursors.Cursor'>) → Any`

Write records stored in a DataFrame into MySQL.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- chunksize

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **con** (*pymysql.connections.Connection*) – Use `pymysql.connect()` to use credentials directly or `wr.mysql.connect()` to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) –

Append, overwrite, upsert_duplicate_key, upsert_replace_into, upsert_distinct, ignore.

append: Inserts new records into table. overwrite: Drops table and recreates. upsert_duplicate_key: Performs an upsert using *ON DUPLICATE KEY* clause. Requires table schema to have defined keys, otherwise duplicate records will be inserted. upsert_replace_into: Performs upsert using *REPLACE INTO* clause. Less efficient and still requires the table schema to have keys or else duplicate records will be inserted. upsert_distinct: Inserts new records, including duplicates, then recreates the table and inserts *DISTINCT* records from old table. This is the least efficient approach but handles scenarios where there are no keys on table. ignore: Inserts new records into table using *INSERT IGNORE* clause.

- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and MySQL types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'TEXT', 'col2 name': 'FLOAT'})
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns *col1* and *col3* and *use_column_names* is True, data will only be inserted into the database columns *col1* and *col3*.
- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.
- **cursorclass** (*Cursor*) – Cursor class to use, e.g. `SSCursor`; defaults to `pymysql.cursors.Cursor` <https://pymysql.readthedocs.io/en/latest/modules/cursors.html>

Returns

None.

Return type

None

Examples

Writing to MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.mysql.connect("MY_GLUE_CONNECTION")
>>> wr.mysql.to_sql(
...     df=df,
...     table="my_table",
...     schema="test",
...     con=con
... )
>>> con.close()
```

Microsoft SQL Server

<code>connect([connection, secret_id, catalog_id, ...])</code>	Return a pyodbc connection from a Glue Catalog Connection.
<code>read_sql_query(sql, con[, index_col, ...])</code>	Return a DataFrame corresponding to the result set of the query string.
<code>read_sql_table(table, con[, schema, ...])</code>	Return a DataFrame corresponding the table.
<code>to_sql(df, con, table, schema[, mode, ...])</code>	Write records stored in a DataFrame into Microsoft SQL Server.

awswrangler.sqlserver.connect

`awswrangler.sqlserver.connect(connection: str | None = None, secret_id: str | None = None, catalog_id: str | None = None, dbname: str | None = None, odbc_driver_version: int = 17, boto3_session: Session | None = None, timeout: int | None = 0) → Any`

Return a pyodbc connection from a Glue Catalog Connection.

<https://github.com/mkleehammer/pyodbc>

Note: You MUST pass a *connection* OR *secret_id*. Here is an example of the secret structure in Secrets Manager: { "host": "sqlserver-instance-wrangler.dr8vkeyrb9m1.us-east-1.rds.amazonaws.com", "username": "test", "password": "test", "engine": "sqlserver", "port": "1433", "dbname": "mydb" # Optional }

Parameters

- **connection** (*Optional[str]*) – Glue Catalog Connection name.
- **secret_id** (*Optional[str]*) – Specifies the secret containing the connection details that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*Optional[str]*) – Optional database name to overwrite the stored one.
- **odbc_driver_version** (*int*) – Major version of the ODBC Driver version that is installed and should be used.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **timeout** (*Optional[int]*) – This is the time in seconds before the connection to the server will time out. The default is None which means no timeout. This parameter is forwarded to pyodbc. <https://github.com/mkleehammer/pyodbc/wiki/The-pyodbc-Module#connect>

Returns

pyodbc connection.

Return type

pyodbc.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↳ version=17)
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.sqlserver.read_sql_query

awswrangler.sqlserver.read_sql_query(*sql: str, con: pyodbc.Connection, index_col: str | List[str] | None = None, params: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, chunksize: int | None = None, dtype: Dict[str, DataType] | None = None, safe: bool = True, timestamp_as_object: bool = False*) → Any

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (*str*) – SQL query.
- **con** (*pyodbc.Connection*) – Use pyodbc.connect() to use credentials directly or wr.sqlserver.connect() to fetch it from the Glue Catalog.
- **index_col** (*Union[str, List[str]]*, *optional*) – Column(s) to set as index(MultiIndex).
- **params** (*Union[List, Tuple, Dict]*, *optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (*int*, *optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType]*, *optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

- **timestamp_as_object** (*bool*) – Cast non-nanosecond timestamps (*np.datetime64*) to objects.

Returns

Result as Pandas *DataFrame*(s).

Return type

Union[*pandas.DataFrame*, *Iterator*[*pandas.DataFrame*]]

Examples

Reading from Microsoft SQL Server using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↳ version=17)
>>> df = wr.sqlserver.read_sql_query(
...     sql="SELECT * FROM dbo.my_table",
...     con=con
... )
>>> con.close()
```

awswrangler.sqlserver.read_sql_table

awswrangler.sqlserver.read_sql_table(*table: str*, *con: pyodbc.Connection*, *schema: str | None = None*, *index_col: str | List[str] | None = None*, *params: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None*, *chunksize: int | None = None*, *dtype: Dict[str, DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*) → *Any*

Return a *DataFrame* corresponding the table.

Parameters

- **table** (*str*) – Table name.
- **con** (*pyodbc.Connection*) – Use *pyodbc.connect()* to use credentials directly or *wr.sqlserver.connect()* to fetch it from the Glue Catalog.
- **schema** (*str*, *optional*) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if *None* (default).
- **index_col** (*Union[str, List[str]]*, *optional*) – Column(s) to set as index(*MultiIndex*).
- **params** (*Union[List, Tuple, Dict]*, *optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's *paramstyle*, is supported.
- **chunksize** (*int*, *optional*) – If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType]*, *optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.

- **timestamp_as_object** (*bool*) – Cast non-nanosecond timestamps (*np.datetime64*) to objects.

Returns

Result as Pandas *DataFrame*(s).

Return type

Union[pandas.*DataFrame*, Iterator[pandas.*DataFrame*]]

Examples

Reading from Microsoft SQL Server using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↳ version=17)
>>> df = wr.sqlserver.read_sql_table(
...     table="my_table",
...     schema="dbo",
...     con=con
... )
>>> con.close()
```

awswrangler.sqlserver.to_sql

awswrangler.sqlserver.to_sql(*df: DataFrame*, *con: pyodbc.Connection*, *table: str*, *schema: str*, *mode: str = 'append'*, *index: bool = False*, *dtype: Dict[str, str] | None = None*, *varchar_lengths: Dict[str, int] | None = None*, *use_column_names: bool = False*, *chunksize: int = 200*, *fast_executemany: bool = False*) → Any

Write records stored in a *DataFrame* into Microsoft SQL Server.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- *chunksize*

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas *DataFrame* <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **con** (*pyodbc.Connection*) – Use *pyodbc.connect()* to use credentials directly or *wr.sqlserver.connect()* to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) – Append or overwrite.
- **index** (*bool*) – True to store the *DataFrame* index as a column in the table, otherwise False to ignore it.

- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Microsoft SQL Server types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. {'col name': 'TEXT', 'col2 name': 'FLOAT'})
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. {"col1": 10, "col5": 200}).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns *col1* and *col3* and *use_column_names* is True, data will only be inserted into the database columns *col1* and *col3*.
- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.
- **fast_executemany** (*bool*) – Mode of execution which greatly reduces round trips for a DBAPI executemany() call when using Microsoft ODBC drivers, for limited size batches that fit in memory. *False* by default.

https://github.com/mkleehammer/pyodbc/wiki/Cursor#executemany-sql-params-with-fast_executemanytrue

Note: when using this mode, pyodbc converts the Python parameter values to their ODBC "C" equivalents, based on the target column types in the database which may lead to subtle data type conversion differences depending on whether *fast_executemany* is True or False.

Returns

None.

Return type

None

Examples

Writing to Microsoft SQL Server using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↳ version=17)
>>> wr.sqlserver.to_sql(
...     df=df,
...     table="table",
...     schema="dbo",
...     con=con
... )
>>> con.close()
```

Oracle

<code>connect([connection, secret_id, catalog_id, ...])</code>	Return a oracledb connection from a Glue Catalog Connection.
<code>read_sql_query(sql, con[, index_col, ...])</code>	Return a DataFrame corresponding to the result set of the query string.
<code>read_sql_table(table, con[, schema, ...])</code>	Return a DataFrame corresponding the table.
<code>to_sql(df, con, table, schema[, mode, ...])</code>	Write records stored in a DataFrame into Oracle Database.

awswrangler.oracle.connect

`awswrangler.oracle.connect(connection: str | None = None, secret_id: str | None = None, catalog_id: str | None = None, dbname: str | None = None, boto3_session: Session | None = None, call_timeout: int | None = 0) → Any`

Return a oracledb connection from a Glue Catalog Connection.

<https://github.com/oracle/python-oracledb>

Note: You MUST pass a *connection* OR *secret_id*. Here is an example of the secret structure in Secrets Manager: { "host": "oracle-instance-wrangler.cr4trrvge8rz.us-east-1.rds.amazonaws.com", "username": "test", "password": "test", "engine": "oracle", "port": "1521", "dbname": "mydb" # Optional }

Parameters

- **connection** (*Optional[str]*) – Glue Catalog Connection name.
- **secret_id** (*Optional[str]*) – Specifies the secret containing the connection details that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **catalog_id** (*str, optional*) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (*Optional[str]*) – Optional database name to overwrite the stored one.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **call_timeout** (*Optional[int]*) – This is the time in milliseconds that a single round-trip to the database may take before a timeout will occur. The default is None which means no timeout. This parameter is forwarded to oracledb. https://cx-oracle.readthedocs.io/en/latest/api_manual/connection.html#Connection.call_timeout

Returns

oracledb connection.

Return type

oracledb.Connection

Examples

```
>>> import awswrangler as wr
>>> con = wr.oracle.connect(connection="MY_GLUE_CONNECTION")
>>> with con.cursor() as cursor:
>>>     cursor.execute("SELECT 1 FROM DUAL")
>>>     print(cursor.fetchall())
>>> con.close()
```

awswrangler.oracle.read_sql_query

`awswrangler.oracle.read_sql_query`(*sql*: str, *con*: *oracledb.Connection*, *index_col*: str | List[str] | None = None, *params*: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, *chunksize*: int | None = None, *dtype*: Dict[str, *DataType*] | None = None, *safe*: bool = True, *timestamp_as_object*: bool = False) → Any

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (str) – SQL query.
- **con** (*oracledb.Connection*) – Use `oracledb.connect()` to use credentials directly or `wr.oracle.connect()` to fetch it from the Glue Catalog.
- **index_col** (*Union[str, List[str]]*, optional) – Column(s) to set as index(*MultiIndex*).
- **params** (*Union[List, Tuple, Dict]*, optional) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (int, optional) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType]*, optional) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (bool) – Check for overflows or other unsafe data type conversions.
- **timestamp_as_object** (bool) – Cast non-nanosecond timestamps (`np.datetime64`) to objects.

Returns

Result as Pandas DataFrame(s).

Return type

`Union[pandas.DataFrame, Iterator[pandas.DataFrame]]`

Examples

Reading from Oracle Database using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.oracle.connect(connection="MY_GLUE_CONNECTION")
>>> df = wr.oracle.read_sql_query(
...     sql="SELECT * FROM test.my_table",
...     con=con
... )
>>> con.close()
```

awswrangler.oracle.read_sql_table

`awswrangler.oracle.read_sql_table(table: str, con: oracledb.Connection, schema: str | None = None, index_col: str | List[str] | None = None, params: List[Any] | Tuple[Any, ...] | Dict[Any, Any] | None = None, chunksize: int | None = None, dtype: Dict[str, DataType] | None = None, safe: bool = True, timestamp_as_object: bool = False) → Any`

Return a DataFrame corresponding the table.

Parameters

- **table** (*str*) – Table name.
- **con** (*oracledb.Connection*) – Use `oracledb.connect()` to use credentials directly or `wr.oracle.connect()` to fetch it from the Glue Catalog.
- **schema** (*str*, *optional*) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if `None` (default).
- **index_col** (*Union[str, List[str]]*, *optional*) – Column(s) to set as index(`MultiIndex`).
- **params** (*Union[List, Tuple, Dict]*, *optional*) – List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported.
- **chunksize** (*int*, *optional*) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (*Dict[str, pyarrow.DataType]*, *optional*) – Specifying the datatype for columns. The keys should be the column names and the values should be the PyArrow types.
- **safe** (*bool*) – Check for overflows or other unsafe data type conversions.
- **timestamp_as_object** (*bool*) – Cast non-nanosecond timestamps (`np.datetime64`) to objects.

Returns

Result as Pandas DataFrame(s).

Return type

`Union[pandas.DataFrame, Iterator[pandas.DataFrame]]`

Examples

Reading from Oracle Database using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.oracle.connect(connection="MY_GLUE_CONNECTION")
>>> df = wr.oracle.read_sql_table(
...     table="my_table",
...     schema="test",
...     con=con
... )
>>> con.close()
```

awswrangler.oracle.to_sql

`awswrangler.oracle.to_sql(df: DataFrame, con: oracledb.Connection, table: str, schema: str, mode: str = 'append', index: bool = False, dtype: Dict[str, str] | None = None, varchar_lengths: Dict[str, int] | None = None, use_column_names: bool = False, chunksize: int = 200) → Any`

Write records stored in a DataFrame into Oracle Database.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `chunksize`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **con** (*oracledb.Connection*) – Use `oracledb.connect()` to use credentials directly or `wr.oracle.connect()` to fetch it from the Glue Catalog.
- **table** (*str*) – Table name
- **schema** (*str*) – Schema name
- **mode** (*str*) – Append or overwrite.
- **index** (*bool*) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (*Dict[str, str], optional*) – Dictionary of columns names and Oracle types to be casted. Useful when you have columns with undetermined or mixed data types. (e.g. `{'col name': 'TEXT', 'col2 name': 'FLOAT'}`)
- **varchar_lengths** (*Dict[str, int], optional*) – Dict of VARCHAR length by columns. (e.g. `{'col1': 10, 'col5': 200}`).
- **use_column_names** (*bool*) – If set to True, will use the column names of the DataFrame for generating the INSERT SQL Query. E.g. If the DataFrame has two columns `col1` and `col3` and `use_column_names` is True, data will only be inserted into the database columns `col1` and `col3`.

- **chunksize** (*int*) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.

Returns

None.

Return type

None

Examples

Writing to Oracle Database using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> con = wr.oracle.connect(connection="MY_GLUE_CONNECTION")
>>> wr.oracle.to_sql(
...     df=df,
...     table="table",
...     schema="ORCL",
...     con=con
... )
>>> con.close()
```

1.4.8 Data API Redshift

<code>RedshiftDataApi</code> ([cluster_id, database, ...])	Provides access to a Redshift cluster via the Data API.
<code>connect</code> ([cluster_id, database, ...])	Create a Redshift Data API connection.
<code>read_sql_query</code> (sql, con[, database])	Run an SQL query on a RedshiftDataApi connection and return the result as a dataframe.

awswrangler.data_api.redshift.RedshiftDataApi

```
class awswrangler.data_api.redshift.RedshiftDataApi(cluster_id: str = "", database: str = "",
workgroup_name: str = "", secret_arn: str = "",
db_user: str = "", sleep: float = 0.25, backoff:
float = 1.5, retries: int = 15, boto3_session:
Session | None = None)
```

Provides access to a Redshift cluster via the Data API.

Note: When connecting to a standard Redshift cluster, *cluster_id* is used. When connecting to Redshift Serverless, *workgroup_name* is used. These two arguments are mutually exclusive.

Parameters

- **cluster_id** (*str*) – Id for the target Redshift cluster - only required if *workgroup_name* not provided.
- **database** (*str*) – Target database name.
- **workgroup_name** (*str*) – Name for the target serverless Redshift workgroup - only required if *cluster_id* not provided.

- **secret_arn** (*str*) – The ARN for the secret to be used for authentication - only required if *db_user* not provided.
- **db_user** (*str*) – The database user to generate temporary credentials for - only required if *secret_arn* not provided.
- **sleep** (*float*) – Number of seconds to sleep between result fetch attempts - defaults to 0.25.
- **backoff** (*float*) – Factor by which to increase the sleep between result fetch attempts - defaults to 1.5.
- **retries** (*int*) – Maximum number of result fetch attempts - defaults to 15.
- **boto3_session** (*boto3.Session()*, *optional*) – The boto3 session. If *None*, the default boto3 session is used.

__init__(*cluster_id: str = "", database: str = "", workgroup_name: str = "", secret_arn: str = "", db_user: str = "", sleep: float = 0.25, backoff: float = 1.5, retries: int = 15, boto3_session: Session | None = None*) → *None*

Methods

__init__(*[cluster_id, database, ...]*)

execute(*sql[, database]*)

Execute SQL statement against a Data API Service.

awsrangler.data_api.redshift.connect

awsrangler.data_api.redshift.connect(*cluster_id: str = "", database: str = "", workgroup_name: str = "", secret_arn: str = "", db_user: str = "", boto3_session: Session | None = None, **kwargs: Any*) → *RedshiftDataApi*

Create a Redshift Data API connection.

Note: When connecting to a standard Redshift cluster, *cluster_id* is used. When connecting to Redshift Serverless, *workgroup_name* is used. These two arguments are mutually exclusive.

Parameters

- **cluster_id** (*str*) – Id for the target Redshift cluster - only required if *workgroup_name* not provided.
- **database** (*str*) – Target database name.
- **workgroup_name** (*str*) – Name for the target serverless Redshift workgroup - only required if *cluster_id* not provided.
- **secret_arn** (*str*) – The ARN for the secret to be used for authentication - only required if *db_user* not provided.
- **db_user** (*str*) – The database user to generate temporary credentials for - only required if *secret_arn* not provided.
- **boto3_session** (*boto3.Session()*, *optional*) – The boto3 session. If *None*, the default boto3 session is used.

- ****kwargs** – Any additional kwargs are passed to the underlying RedshiftDataApi class.

Return type

A RedshiftDataApi connection instance that can be used with `wr.redshift.data_api.read_sql_query`.

awswrangler.data_api.redshift.read_sql_query

`awswrangler.data_api.redshift.read_sql_query(sql: str, con: RedshiftDataApi, database: str | None = None) → DataFrame`

Run an SQL query on a RedshiftDataApi connection and return the result as a dataframe.

Parameters

- **sql** (*str*) – SQL query to run.
- **con** (*RedshiftDataApi*) – A RedshiftDataApi connection instance
- **database** (*str*) – Database to run query on - defaults to the database specified by *con*.

Return type

A Pandas dataframe containing the query results.

1.4.9 Data API RDS

<code>RdsDataApi(resource_arn, database[, ...])</code>	Provides access to the RDS Data API.
<code>connect(resource_arn, database[, ...])</code>	Create a RDS Data API connection.
<code>read_sql_query(sql, con[, database])</code>	Run an SQL query on an RdsDataApi connection and return the result as a dataframe.

awswrangler.data_api.rds.RdsDataApi

`class awswrangler.data_api.rds.RdsDataApi(resource_arn: str, database: str, secret_arn: str = "", sleep: float = 0.5, backoff: float = 1.0, retries: int = 30, boto3_session: Session | None = None)`

Provides access to the RDS Data API.

Parameters

- **resource_arn** (*str*) – ARN for the RDS resource.
- **database** (*str*) – Target database name.
- **secret_arn** (*str*) – The ARN for the secret to be used for authentication.
- **sleep** (*float*) – Number of seconds to sleep between connection attempts to paused clusters - defaults to 0.5.
- **backoff** (*float*) – Factor by which to increase the sleep between connection attempts to paused clusters - defaults to 1.0.
- **retries** (*int*) – Maximum number of connection attempts to paused clusters - defaults to 10.
- **boto3_session** (*boto3.Session()*, *optional*) – The boto3 session. If *None*, the default boto3 session is used.

```
__init__(resource_arn: str, database: str, secret_arn: str = "", sleep: float = 0.5, backoff: float = 1.0,
         retries: int = 30, boto3_session: Session | None = None) → None
```

Methods

```
__init__(resource_arn, database[, ...])
```

```
execute(sql[, database])
```

Execute SQL statement against a Data API Service.

`awswrangler.data_api.rds.connect`

```
awswrangler.data_api.rds.connect(resource_arn: str, database: str, secret_arn: str = "", boto3_session:
                                Session | None = None, **kwargs: Any) → RdsDataApi
```

Create a RDS Data API connection.

Parameters

- **resource_arn** (*str*) – ARN for the RDS resource.
- **database** (*str*) – Target database name.
- **secret_arn** (*str*) – The ARN for the secret to be used for authentication.
- **boto3_session** (*boto3.Session()*, *optional*) – The boto3 session. If *None*, the default boto3 session is used.
- ****kwargs** – Any additional kwargs are passed to the underlying `RdsDataApi` class.

Return type

A `RdsDataApi` connection instance that can be used with `wr.rds.data_api.read_sql_query`.

`awswrangler.data_api.rds.read_sql_query`

```
awswrangler.data_api.rds.read_sql_query(sql: str, con: RdsDataApi, database: str | None = None) →
                                         DataFrame
```

Run an SQL query on an `RdsDataApi` connection and return the result as a dataframe.

Parameters

- **sql** (*str*) – SQL query to run.
- **con** ([RdsDataApi](#)) – A `RdsDataApi` connection instance
- **database** (*str*) – Database to run query on - defaults to the database specified by *con*.

Return type

A Pandas dataframe containing the query results.

1.4.10 AWS Glue Data Quality

<code>create_recommendation_ruleset(database, ...)</code>	Create recommendation Data Quality ruleset.
<code>create_ruleset(name, database, table[, ...])</code>	Create Data Quality ruleset.
<code>evaluate_ruleset(name, iam_role_arn[, ...])</code>	Evaluate Data Quality ruleset.
<code>get_ruleset(name[, boto3_session])</code>	Get a Data Quality ruleset.
<code>update_ruleset(name[, updated_name, mode, ...])</code>	Update Data Quality ruleset.

`awswrangler.data_quality.create_recommendation_ruleset`

`awswrangler.data_quality.create_recommendation_ruleset(database: str, table: str, iam_role_arn: str, name: str | None = None, catalog_id: str | None = None, connection_name: str | None = None, additional_options: Dict[str, Any] | None = None, number_of_workers: int = 5, timeout: int = 2880, client_token: str | None = None, boto3_session: Session | None = None) → Any`

Create recommendation Data Quality ruleset.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **database** (*str*) – Glue database name.
- **table** (*str*) – Glue table name.
- **iam_role_arn** (*str*) – IAM Role ARN.
- **name** (*str*, *optional*) – Ruleset name.
- **catalog_id** (*str*, *optional*) – Glue Catalog id.
- **connection_name** (*str*, *optional*) – Glue connection name.
- **additional_options** (*dict*, *optional*) – Additional options for the table. Supported keys: `pushDownPredicate`: to filter on partitions without having to list and read all the files in your dataset. `catalogPartitionPredicate`: to use server-side partition pruning using partition indexes in the Glue Data Catalog.
- **number_of_workers** (*int*, *optional*) – The number of G.1X workers to be used in the run. The default is 5.
- **timeout** (*int*, *optional*) – The timeout for a run in minutes. The default is 2880 (48 hours).
- **client_token** (*str*, *optional*) – Random id used for idempotency. Is automatically generated if not provided.

- **boto3_session** (*boto3.Session*, *optional*) – Boto3 Session. If none, the default boto3 session is used.

Returns

Data frame with recommended ruleset details.

Return type

pd.DataFrame

Examples

```
>>> import awswrangler as wr

>>> df_recommended_ruleset = wr.data_quality.create_recommendation_ruleset(
>>>     database="database",
>>>     table="table",
>>>     iam_role_arn="arn:...",
>>> )
```

awswrangler.data_quality.create_ruleset

awswrangler.data_quality.**create_ruleset**(*name: str*, *database: str*, *table: str*, *df_rules: DataFrame | None = None*, *dqdl_rules: str | None = None*, *description: str = ""*, *client_token: str | None = None*, *boto3_session: Session | None = None*) → Any

Create Data Quality ruleset.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- database

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **name** (*str*) – Ruleset name.
- **database** (*str*) – Glue database name.
- **table** (*str*) – Glue table name.
- **df_rules** (*str*, *optional*) – Data frame with *rule_type*, *parameter*, and *expression* columns.
- **dqdl_rules** (*str*, *optional*) – Data Quality Definition Language definition.
- **description** (*str*) – Ruleset description.
- **client_token** (*str*, *optional*) – Random id used for idempotency. Is automatically generated if not provided.
- **boto3_session** (*boto3.Session*, *optional*) – Boto3 Session. If none, the default boto3 session is used.

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>>
>>> df = pd.DataFrame({"c0": [0, 1, 2], "c1": [0, 1, 2], "c2": [0, 0, 1]})
>>> wr.s3.to_parquet(df, path, dataset=True, database="database", table="table")
>>> wr.data_quality.create_ruleset(
>>>     name="ruleset",
>>>     database="database",
>>>     table="table",
>>>     dqdl_rules="Rules = [ RowCount between 1 and 3 ]",
>>> )
```

```
>>> import awswrangler as wr
>>> import pandas as pd
>>>
>>> df = pd.DataFrame({"c0": [0, 1, 2], "c1": [0, 1, 2], "c2": [0, 0, 1]})
>>> df_rules = pd.DataFrame({
>>>     "rule_type": ["RowCount", "IsComplete", "Uniqueness"],
>>>     "parameter": [None, '"c0"', '"c0"'],
>>>     "expression": ["between 1 and 6", None, "> 0.95"],
>>> })
>>> wr.s3.to_parquet(df, path, dataset=True, database="database", table="table")
>>> wr.data_quality.create_ruleset(
>>>     name="ruleset",
>>>     database="database",
>>>     table="table",
>>>     df_rules=df_rules,
>>> )
```

awswrangler.data_quality.evaluate_ruleset

`awswrangler.data_quality.evaluate_ruleset`(*name: str | List[str], iam_role_arn: str, number_of_workers: int = 5, timeout: int = 2880, database: str | None = None, table: str | None = None, catalog_id: str | None = None, connection_name: str | None = None, additional_options: Dict[str, str] | None = None, additional_run_options: Dict[str, bool | str] | None = None, client_token: str | None = None, boto3_session: Session | None = None*) → Any

Evaluate Data Quality ruleset.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

- `catalog_id`
- `database`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **name** (*str* or *list[str]*) – Ruleset name or list of names.
- **iam_role_arn** (*str*) – IAM Role ARN.
- **number_of_workers** (*int*, *optional*) – The number of G.IX workers to be used in the run. The default is 5.
- **timeout** (*int*, *optional*) – The timeout for a run in minutes. The default is 2880 (48 hours).
- **database** (*str*, *optional*) – Glue database name. Database associated with the ruleset will be used if not provided.
- **table** (*str*, *optional*) – Glue table name. Table associated with the ruleset will be used if not provided.
- **catalog_id** (*str*, *optional*) – Glue Catalog id.
- **connection_name** (*str*, *optional*) – Glue connection name.
- **additional_options** (*dict*, *optional*) – Additional options for the table. Supported keys: *pushDownPredicate*: to filter on partitions without having to list and read all the files in your dataset. *catalogPartitionPredicate*: to use server-side partition pruning using partition indexes in the Glue Data Catalog.
- **additional_run_options** (*Dict[str, Union[str, bool]]*, *optional*) – Additional run options. Supported keys: *CloudWatchMetricsEnabled*: whether to enable CloudWatch metrics. *ResultsS3Prefix*: prefix for Amazon S3 to store results.
- **client_token** (*str*, *optional*) – Random id used for idempotency. Will be automatically generated if not provided.
- **boto3_session** (*boto3.Session*, *optional*) – Boto3 Session. If none, the default boto3 session is used.

Returns

Data frame with ruleset evaluation results.

Return type

pd.DataFrame

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>>
>>> df = pd.DataFrame({"c0": [0, 1, 2], "c1": [0, 1, 2], "c2": [0, 0, 1]})
>>> wr.s3.to_parquet(df, path, dataset=True, database="database", table="table")
>>> wr.data_quality.create_ruleset(
>>>     name="ruleset",
>>>     database="database",
>>>     table="table",
>>>     dqdl_rules="Rules = [ RowCount between 1 and 3 ]",
>>>)
>>> df_ruleset_results = wr.data_quality.evaluate_ruleset(
>>>     name="ruleset",
>>>     iam_role_arn=glue_data_quality_role,
>>> )
```


aws wrangler.data_quality.get_ruleset

`aws wrangler.data_quality.get_ruleset(name: str | List[str], boto3_session: Session | None = None) → DataFrame`

Get a Data Quality ruleset.

Parameters

- **name** (*str* or *list[str]*) – Ruleset name or list of names.
- **boto3_session** (*boto3.Session*, *optional*) – Boto3 Session. If none, the default boto3 session is used.

Returns

Data frame with ruleset(s) details.

Return type

`pd.DataFrame`

Examples

Get single ruleset >>> import aws wrangler as wr

```
>>> df_ruleset = wr.data_quality.get_ruleset(name="my_ruleset")
```

Get multiple rulesets. A column with the ruleset name is added to the data frame >>> df_rulesets = wr.data_quality.get_ruleset(name=["ruleset_1", "ruleset_2"])

aws wrangler.data_quality.update_ruleset

`aws wrangler.data_quality.update_ruleset(name: str, updated_name: str | None = None, mode: str = 'overwrite', df_rules: DataFrame | None = None, dqdl_rules: str | None = None, description: str = "", boto3_session: Session | None = None) → Any`

Update Data Quality ruleset.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **name** (*str*) – Ruleset name.
- **updated_name** (*str*) – New ruleset name if renaming an existing ruleset.
- **mode** (*str*) – overwrite (default) or upsert.
- **df_rules** (*str*, *optional*) – Data frame with *rule_type*, *parameter*, and *expression* columns.
- **dqdl_rules** (*str*, *optional*) – Data Quality Definition Language definition.
- **description** (*str*) – Ruleset description.
- **boto3_session** (*boto3.Session*, *optional*) – Boto3 Session. If none, the default boto3 session is used.

Examples

Overwrite rules in the existing ruleset. `>>> wr.data_quality.update_ruleset(>>> name="ruleset", >>> new_name="my_ruleset", >>> dqdl_rules="Rules = [RowCount between 1 and 3]", >>>)`

Update or insert rules in the existing ruleset. `>>> wr.data_quality.update_ruleset(>>> name="ruleset", >>> mode="insert", >>> dqdl_rules="Rules = [RowCount between 1 and 3]", >>>)`

1.4.11 OpenSearch

<code>connect(host[, port, boto3_session, region, ...])</code>	Create a secure connection to the specified Amazon OpenSearch domain.
<code>create_collection(name[, collection_type, ...])</code>	Create Amazon OpenSearch Serverless collection.
<code>create_index(client, index[, doc_type, ...])</code>	Create an index.
<code>delete_index(client, index)</code>	Delete an index.
<code>index_csv(client, path, index[, doc_type, ...])</code>	Index all documents from a CSV file to OpenSearch index.
<code>index_documents(client, documents, index[, ...])</code>	Index all documents to OpenSearch index.
<code>index_df(client, df, index[, doc_type])</code>	Index all documents from a DataFrame to OpenSearch index.
<code>index_json(client, path, index[, doc_type, ...])</code>	Index all documents from JSON file to OpenSearch index.
<code>search(client[, index, search_body, ...])</code>	Return results matching query DSL as pandas dataframe.
<code>search_by_sql(client, sql_query, **kwargs)</code>	Return results matching SQL query as pandas dataframe.

awswrangler.opensearch.connect

`awswrangler.opensearch.connect(host: str, port: int | None = 443, boto3_session: Session | None = None, region: str | None = None, username: str | None = None, password: str | None = None, service: str | None = None, timeout: int = 30, max_retries: int = 5, retry_on_timeout: bool = True, retry_on_status: Sequence[int] | None = None) → OpenSearch`

Create a secure connection to the specified Amazon OpenSearch domain.

Note: We use [opensearch-py](#), an OpenSearch python client.

The username and password are mandatory if the OS Cluster uses [Fine Grained Access Control](#). If fine grained access control is disabled, session access key and secret keys are used.

Parameters

- **host** (*str*) – Amazon OpenSearch domain, for example: `my-test-domain.us-east-1.es.amazonaws.com`.
- **port** (*int*) – OpenSearch Service only accepts connections over port 80 (HTTP) or 443 (HTTPS)
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **region** (*str*, *optional*) – AWS region of the Amazon OS domain. If not provided will be extracted from boto3_session.

- **username** (*str*, *optional*) – Fine-grained access control username. Mandatory if OS Cluster uses Fine Grained Access Control.
- **password** (*str*, *optional*) – Fine-grained access control password. Mandatory if OS Cluster uses Fine Grained Access Control.
- **service** (*str*, *optional*) – Service id. Supported values are *es*, corresponding to opensearch cluster, and *aoss* for serverless opensearch. By default, service will be parsed from the host URI.
- **timeout** (*int*) – Operation timeout. *30* by default.
- **max_retries** (*int*) – Maximum number of retries before an exception is propagated. *10* by default.
- **retry_on_timeout** (*bool*) – Should timeout trigger a retry on different node. *True* by default.
- **retry_on_status** (*List[int]*, *optional*) – Set of HTTP status codes on which we should retry on a different node. Defaults to [500, 502, 503, 504].

Returns

OpenSearch low-level client. https://github.com/opensearch-project/opensearch-py/blob/main/opensearchpy/client/__init__.py

Return type

opensearchpy.OpenSearch

awswrangler.opensearch.create_collection

```
awswrangler.opensearch.create_collection(name: str, collection_type: str = 'SEARCH', description: str =
    "", encryption_policy: Dict[str, Any] | List[Dict[str, Any]] |
    None = None, kms_key_arn: str | None = None,
    network_policy: Dict[str, Any] | List[Dict[str, Any]] | None =
    None, vpc_endpoints: List[str] | None = None, data_policy:
    List[Dict[str, Any]] | None = None, boto3_session: Session |
    None = None) → Dict[str, Any]
```

Create Amazon OpenSearch Serverless collection.

Creates Amazon OpenSearch Serverless collection, corresponding encryption and network policies, and data policy, if *data_policy* provided.

More in [Amazon OpenSearch Serverless (preview)] (<https://docs.aws.amazon.com/opensearch-service/latest/developerguide/serverless.html>)

Warning: This API is experimental and may change in future AWS SDK for Pandas releases.

Parameters

- **name** (*str*) – Collection name.
- **collection_type** (*str*) – Collection type. Allowed values are *SEARCH*, and *TIME-SERIES*.
- **description** (*str*) – Collection description.
- **encryption_policy** (*Union[Dict[str, Any], List[Dict[str, Any]]]*, *optional*) – Encryption policy of a form: { "Rules": [...] }

If not provided, default policy using AWS-managed KMS key will be created. To use user-defined key, provide *kms_key_arn*.

- **kms_key_arn** (*str*, *optional*) – Encryption key.
- **network_policy** (*Union[Dict[str, Any], List[Dict[str, Any]]], optional*) – Network policy of a form: [{ “Rules”: [...] }]

If not provided, default network policy allowing public access to the collection will be created. To create the collection in the VPC, provide *vpc_endpoints*.

- **vpc_endpoints** (*List[str], optional*) – List of VPC endpoints for access to non-public collection.
- **data_policy** (*Union[Dict[str, Any], List[Dict[str, Any]]], optional*) – Data policy of a form: [{ “Rules”: [...] }]
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if *boto3_session* receive None.

Returns

Collection details – Collection details

Return type

Dict[str, Any]

awswrangler.opensearch.create_index

`awswrangler.opensearch.create_index(client: OpenSearch, index: str, doc_type: str | None = None, settings: Dict[str, Any] | None = None, mappings: Dict[str, Any] | None = None) → Dict[str, Any]`

Create an index.

Parameters

- **client** (*OpenSearch*) – instance of *opensearchpy.OpenSearch* to use.
- **index** (*str*) – Name of the index.
- **doc_type** (*str, optional*) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **settings** (*Dict[str, Any], optional*) – Index settings <https://opensearch.org/docs/opensearch/rest-api/create-index/#index-settings>
- **mappings** (*Dict[str, Any], optional*) – Index mappings <https://opensearch.org/docs/opensearch/rest-api/create-index/#mappings>

Returns

OpenSearch rest api response <https://opensearch.org/docs/opensearch/rest-api/create-index/#response>.

Return type

Dict[str, Any]

Examples

Creating an index.

```
>>> import awswrangler as wr
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> response = wr.opensearch.create_index(
...     client=client,
...     index="sample-index1",
...     mappings={
...         "properties": {
...             "age": { "type" : "integer" }
...         }
...     },
...     settings={
...         "index": {
...             "number_of_shards": 2,
...             "number_of_replicas": 1
...         }
...     }
... )
```

awswrangler.opensearch.delete_index

`awswrangler.opensearch.delete_index(client: OpenSearch, index: str) → Dict[str, Any]`

Delete an index.

Parameters

- **client** (*OpenSearch*) – instance of `opensearchpy.OpenSearch` to use.
- **index** (*str*) – Name of the index.

Returns

OpenSearch rest api response

Return type

Dict[str, Any]

Examples

Deleting an index.

```
>>> import awswrangler as wr
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> response = wr.opensearch.delete_index(
...     client=client,
...     index="sample-index1"
... )
```

awswrangler.opensearch.index_csv

`awswrangler.opensearch.index_csv(client: OpenSearch, path: str, index: str, doc_type: str | None = None, pandas_kwargs: Dict[str, Any] | None = None, **kwargs: Any) → Dict[str, Any]`

Index all documents from a CSV file to OpenSearch index.

Parameters

- **client** (*OpenSearch*) – instance of `opensearchpy.OpenSearch` to use.
- **path** (*str*) – s3 or local path to the CSV file which contains the documents.
- **index** (*str*) – Name of the index.
- **doc_type** (*str, optional*) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **pandas_kwargs** (*Dict[str, Any], optional*) – Dictionary of arguments forwarded to `pandas.read_csv()`. e.g. `pandas_kwargs={'sep': '|', 'na_values': ['null', 'none']}` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html Note: these params values are enforced: `skip_blank_lines=True`
- ****kwargs** – KEYWORD arguments forwarded to `index_documents()` which is used to execute the operation

Returns

Response payload <https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#response>.

Return type

`Dict[str, Any]`

Examples

Writing contents of CSV file

```
>>> import awswrangler as wr
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> wr.opensearch.index_csv(
...     client=client,
...     path='docs.csv',
...     index='sample-index1'
... )
```

Writing contents of CSV file using `pandas_kwargs`

```
>>> import awswrangler as wr
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> wr.opensearch.index_csv(
...     client=client,
...     path='docs.csv',
...     index='sample-index1',
...     pandas_kwargs={'sep': '|', 'na_values': ['null', 'none']}
... )
```

aws wrangler.opensearch.index_documents

```
aws wrangler.opensearch.index_documents(client: OpenSearch, documents: Iterable[Mapping[str, Any]],
                                         index: str, doc_type: str | None = None, keys_to_write: List[str] |
                                         None = None, id_keys: List[str] | None = None, ignore_status:
                                         List[Any] | Tuple[Any] | None = None, bulk_size: int = 1000,
                                         chunk_size: int | None = 500, max_chunk_bytes: int | None =
                                         104857600, max_retries: int | None = 5, initial_backoff: int |
                                         None = 2, max_backoff: int | None = 600, **kwargs: Any) →
                                         Dict[str, Any]
```

Index all documents to OpenSearch index.

Note: Some of the args are referenced from opensearch-py client library (bulk helpers) <https://opensearch-py.readthedocs.io/en/latest/helpers.html#opensearchpy.helpers.bulk> https://opensearch-py.readthedocs.io/en/latest/helpers.html#opensearchpy.helpers.streaming_bulk

If you receive *Error 429 (Too Many Requests)* / *_bulk* please to to decrease *bulk_size* value. Please also consider modifying the cluster size and instance type - Read more here: <https://aws.amazon.com/premiumsupport/knowledge-center/resolve-429-error-es/>

Parameters

- **client** (*OpenSearch*) – instance of *opensearchpy.OpenSearch* to use.
- **documents** (*Iterable[Mapping[str, Any]]*) – List which contains the documents that will be inserted.
- **index** (*str*) – Name of the index.
- **doc_type** (*str, optional*) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **keys_to_write** (*List[str], optional*) – list of keys to index. If not provided all keys will be indexed
- **id_keys** (*List[str], optional*) – list of keys that compound document unique id. If not provided will use *_id* key if exists, otherwise will generate unique identifier for each document.
- **ignore_status** (*Union[List[Any], Tuple[Any]], optional*) – list of HTTP status codes that you want to ignore (not raising an exception)
- **bulk_size** (*int,*) – number of docs in each *_bulk* request (default: 1000)
- **chunk_size** (*int, optional*) – number of docs in one chunk sent to es (default: 500)
- **max_chunk_bytes** (*int, optional*) – the maximum size of the request in bytes (default: 100MB)
- **max_retries** (*int, optional*) – maximum number of times a document will be retried when 429 is received, set to 0 (default) for no retries on 429 (default: 2)
- **initial_backoff** (*int, optional*) – number of seconds we should wait before the first retry. Any subsequent retries will be powers of *initial_backoff*2**retry_number* (default: 2)
- **max_backoff** (*int, optional*) – maximum number of seconds a retry will wait (default: 600)

- ****kwargs** – KEYWORD arguments forwarded to bulk operation elasticsearch >= 7.10.2 / opensearch: <https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#url-parameters> elasticsearch < 7.10.2: <https://opendistro.github.io/for-elasticsearch-docs/docs/elasticsearch/rest-api-reference/#url-parameters>

Returns

Response payload <https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#response>.

Return type

Dict[str, Any]

Examples

Writing documents

```
>>> import awswrangler as wr
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> wr.opensearch.index_documents(
...     documents=[{'_id': '1', 'value': 'foo'}, {'_id': '2', 'value': 'bar'}],
...     index='sample-index1'
... )
```

awswrangler.opensearch.index_df

`awswrangler.opensearch.index_df(client: OpenSearch, df: DataFrame, index: str, doc_type: str | None = None, **kwargs: Any) → Dict[str, Any]`

Index all documents from a DataFrame to OpenSearch index.

Parameters

- **client** (*OpenSearch*) – instance of `opensearchpy.OpenSearch` to use.
- **df** (*pd.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **index** (*str*) – Name of the index.
- **doc_type** (*str, optional*) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- ****kwargs** – KEYWORD arguments forwarded to `index_documents()` which is used to execute the operation

Returns

Response payload <https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#response>.

Return type

Dict[str, Any]

Examples

Writing rows of DataFrame

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> wr.opensearch.index_df(
...     client=client,
...     df=pd.DataFrame([{'_id': '1'}, {'_id': '2'}, {'_id': '3'}]),
...     index='sample-index1'
... )
```

awswrangler.opensearch.index_json

`awswrangler.opensearch.index_json(client: OpenSearch, path: str, index: str, doc_type: str | None = None, boto3_session: Session | None = Session(region_name=None), json_path: str | None = None, **kwargs: Any) → Dict[str, Any]`

Index all documents from JSON file to OpenSearch index.

The JSON file should be in a JSON-Lines text format (newline-delimited JSON) - <https://jsonlines.org/> OR if the is a single large JSON please provide `json_path`.

Parameters

- **client** (*OpenSearch*) – instance of `opensearchpy.OpenSearch` to use.
- **path** (*str*) – s3 or local path to the JSON file which contains the documents.
- **index** (*str*) – Name of the index.
- **doc_type** (*str, optional*) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **json_path** (*str, optional*) – JsonPath expression to specify explicit path to a single name element in a JSON hierarchical data structure. Read more about [JsonPath](#)
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session to be used to access s3 if s3 path is provided. The default boto3 Session will be used if `boto3_session` receive `None`.
- ****kwargs** – KEYWORD arguments forwarded to `index_documents()` which is used to execute the operation

Returns

Response payload <https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#response>.

Return type

Dict[str, Any]

Examples

Writing contents of JSON file

```
>>> import awswrangler as wr
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> wr.opensearch.index_json(
...     client=client,
...     path='docs.json',
...     index='sample-index1'
... )
```

awswrangler.opensearch.search

`awswrangler.opensearch.search`(*client: OpenSearch, index: str | None = '_all', search_body: Dict[str, Any] | None = None, doc_type: str | None = None, is_scroll: bool | None = False, filter_path: str | Collection[str] | None = None, **kwargs: Any*) → `DataFrame`

Return results matching query DSL as pandas dataframe.

Parameters

- **client** (*OpenSearch*) – instance of `opensearchpy.OpenSearch` to use.
- **index** (*str, optional*) – A comma-separated list of index names to search. use `_all` or empty string to perform the operation on all indices.
- **search_body** (*Dict[str, Any], optional*) – The search definition using the [Query DSL](#).
- **doc_type** (*str, optional*) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **is_scroll** (*bool, optional*) – Allows to retrieve a large numbers of results from a single search request using [scroll](#) for example, for machine learning jobs. Because scroll search contexts consume a lot of memory, we suggest you don't use the scroll operation for frequent user queries.
- **filter_path** (*Union[str, Collection[str]], optional*) – Use the `filter_path` parameter to reduce the size of the OpenSearch Service response (default: `['hits.hits._id','hits.hits._source']`)
- ****kwargs** – KEYWORD arguments forwarded to `opensearchpy.OpenSearch.search` and also to `opensearchpy.helpers.scan`
if `is_scroll=True`

Returns

Results as Pandas DataFrame

Return type

`Union[pandas.DataFrame, Iterator[pandas.DataFrame]]`

Examples

Searching an index using query DSL

```
>>> import awswrangler as wr
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> df = wr.opensearch.search(
...     client=client,
...     index='movies',
...     search_body={
...         "query": {
...             "match": {
...                 "title": "wind"
...             }
...         }
...     }
... )
```

awswrangler.opensearch.search_by_sql

`awswrangler.opensearch.search_by_sql(client: OpenSearch, sql_query: str, **kwargs: Any) → DataFrame`

Return results matching [SQL query](#) as pandas dataframe.

Parameters

- **client** (*OpenSearch*) – instance of `opensearchpy.OpenSearch` to use.
- **sql_query** (*str*) – SQL query
- ****kwargs** – KEYWORD arguments forwarded to request url (e.g.: `filter_path`, etc.)

Returns

Results as Pandas DataFrame

Return type

`Union[pandas.DataFrame, Iterator[pandas.DataFrame]]`

Examples

Searching an index using SQL query

```
>>> import awswrangler as wr
>>> client = wr.opensearch.connect(host='DOMAIN-ENDPOINT')
>>> df = wr.opensearch.search_by_sql(
>>>     client=client,
>>>     sql_query='SELECT * FROM my-index LIMIT 50'
>>> )
```

1.4.12 Amazon Neptune

<code>connect(host, port[, iam_enabled])</code>	Create a connection to a Neptune cluster.
<code>execute_gremlin(client, query)</code>	Return results of a Gremlin traversal as pandas dataframe.
<code>execute_opencypher(client, query)</code>	Return results of a openCypher traversal as pandas dataframe.
<code>execute_sparql(client, query)</code>	Return results of a SPARQL query as pandas dataframe.
<code>flatten_nested_df(df[, include_prefix, ...])</code>	Flatten the lists and dictionaries of the input data frame.
<code>to_property_graph(client, df[, batch_size, ...])</code>	Write records stored in a DataFrame into Amazon Neptune.
<code>to_rdf_graph(client, df[, batch_size, ...])</code>	Write records stored in a DataFrame into Amazon Neptune.

awswrangler.neptune.connect

`awswrangler.neptune.connect(host: str, port: int, iam_enabled: bool = False, **kwargs: Any) → NeptuneClient`

Create a connection to a Neptune cluster.

Parameters

- **host** (*str*) – The host endpoint to connect to
- **port** (*int*) – The port endpoint to connect to
- **iam_enabled** (*bool*, *optional*) – True if IAM is enabled on the cluster. Defaults to False.

Returns

[description]

Return type

NeptuneClient

awswrangler.neptune.execute_gremlin

`awswrangler.neptune.execute_gremlin(client: NeptuneClient, query: str) → DataFrame`

Return results of a Gremlin traversal as pandas dataframe.

Parameters

- **client** (*neptune.Client*) – instance of the neptune client to use
- **traversal** (*str*) – The gremlin traversal to execute

Returns

Results as Pandas DataFrame

Return type

Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Run a Gremlin Query

```
>>> import awswrangler as wr
>>> client = wr.neptune.connect(neptune_endpoint, neptune_port, iam_
    ↪enabled=False)
>>> df = wr.neptune.execute_gremlin(client, "g.V().limit(1)")
```

awswrangler.neptune.execute_opencypher

`awswrangler.neptune.execute_opencypher(client: NeptuneClient, query: str) → DataFrame`

Return results of a openCypher traversal as pandas dataframe.

Parameters

- **client** (*NeptuneClient*) – instance of the neptune client to use
- **query** (*str*) – The openCypher query to execute

Returns

Results as Pandas DataFrame

Return type

Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Run an openCypher query

```
>>> import awswrangler as wr
>>> client = wr.neptune.connect(neptune_endpoint, neptune_port, iam_enabled=False)
>>> resp = wr.neptune.execute_opencypher(client, "MATCH (n) RETURN n LIMIT 1")
```

awswrangler.neptune.execute_sparql

`awswrangler.neptune.execute_sparql(client: NeptuneClient, query: str) → Any`

Return results of a SPARQL query as pandas dataframe.

Parameters

- **client** (*NeptuneClient*) – instance of the neptune client to use
- **query** (*str*) – The SPARQL traversal to execute

Returns

Results as Pandas DataFrame

Return type

Union[pandas.DataFrame, Iterator[pandas.DataFrame]]

Examples

Run a SPARQL query

```
>>> import awswrangler as wr
>>> client = wr.neptune.connect(neptune_endpoint, neptune_port, iam_enabled=False)
>>> df = wr.neptune.execute_sparql(client, "PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
WHERE {
    ?person foaf:name ?name .
```

awswrangler.neptune.flatten_nested_df

`awswrangler.neptune.flatten_nested_df(df: DataFrame, include_prefix: bool = True, separator: str = '_', recursive: bool = True) → DataFrame`

Flatten the lists and dictionaries of the input data frame.

Parameters

- **df** (*pd.DataFrame*) – The input data frame
- **include_prefix** (*bool, optional*) – If True, then it will prefix the new column name with the original column name. Defaults to True.
- **separator** (*str, optional*) – The separator to use between field names when a dictionary is exploded. Defaults to “_”.
- **recursive** (*bool, optional*) – If True, then this will recurse the fields in the data frame. Defaults to True.

Returns

pd.DataFrame

Return type

The flattened data frame

awswrangler.neptune.to_property_graph

`awswrangler.neptune.to_property_graph(client: NeptuneClient, df: DataFrame, batch_size: int = 50, use_header_cardinality: bool = True) → bool`

Write records stored in a DataFrame into Amazon Neptune.

If writing to a property graph then DataFrames for vertices and edges must be written separately. DataFrames for vertices must have a `~label` column with the label and a `~id` column for the vertex id. If the `~id` column does not exist, the specified id does not exist, or is empty then a new vertex will be added. If no `~label` column exists an exception will be thrown. DataFrames for edges must have a `~id`, `~label`, `~to`, and `~from` column. If the `~id` column does not exist the specified id does not exist, or is empty then a new edge will be added. If no `~label`, `~to`, or `~from` column exists an exception will be thrown.

If you would like to save data using *single* cardinality then you can postfix (single) to the column header and set `use_header_cardinality=True` (default). e.g. A column named `name(single)` will save the `name` property as single cardinality. You can disable this by setting by setting `use_header_cardinality=False`.

Parameters

- **client** (*NeptuneClient*) – instance of the neptune client to use

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **batch_size** (*int*) – The number of rows to save at a time. Default 50
- **use_header_cardinality** (*bool*) – If True, then the header cardinality will be used to save the data. Default True

Returns

True if records were written

Return type

bool

Examples

Writing to Amazon Neptune

```
>>> import awswrangler as wr
>>> client = wr.neptune.connect(neptune_endpoint, neptune_port, iam_enabled=False)
>>> wr.neptune.gremlin.to_property_graph(
...     df=df
... )
```

awswrangler.neptune.to_rdf_graph

awswrangler.neptune.to_rdf_graph(*client: NeptuneClient, df: DataFrame, batch_size: int = 50, subject_column: str = 's', predicate_column: str = 'p', object_column: str = 'o', graph_column: str = 'g'*) → Any

Write records stored in a DataFrame into Amazon Neptune.

The DataFrame must consist of triples with column names for the subject, predicate, and object specified. If you want to add data into a named graph then you will also need the graph column.

Parameters

- **(NeptuneClient)** (*client*) – instance of the neptune client to use
- **(pandas.DataFrame)** (*df*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **(str (graph_column))** – The column name in the dataframe for the subject. Defaults to 's'
- **optional** – The column name in the dataframe for the subject. Defaults to 's'
- **(str** – The column name in the dataframe for the predicate. Defaults to 'p'
- **optional** – The column name in the dataframe for the predicate. Defaults to 'p'
- **(str** – The column name in the dataframe for the object. Defaults to 'o'
- **optional** – The column name in the dataframe for the object. Defaults to 'o'
- **(str** – The column name in the dataframe for the graph if sending across quads. Defaults to 'g'
- **optional** – The column name in the dataframe for the graph if sending across quads. Defaults to 'g'

Returns

True if records were written

Return type

bool

Examples

Writing to Amazon Neptune

```
>>> import awswrangler as wr
>>> client = wr.neptune.connect(neptune_endpoint, neptune_port, iam_enabled=False)
>>> wr.neptune.gremlin.to_rdf_graph(
...     df=df
... )
```

1.4.13 DynamoDB

<code>delete_items</code> (items, table_name[, boto3_session])	Delete all items in the specified DynamoDB table.
<code>execute_statement</code> (statement[, parameters, ...])	Run a PartiQL statement against a DynamoDB table.
<code>get_table</code> (table_name[, boto3_session])	Get DynamoDB table object for specified table name.
<code>put_csv</code> (path, table_name[, boto3_session])	Write all items from a CSV file to a DynamoDB.
<code>put_df</code> (df, table_name[, boto3_session])	Write all items from a DataFrame to a DynamoDB.
<code>put_items</code> (items, table_name[, boto3_session])	Insert all items to the specified DynamoDB table.
<code>put_json</code> (path, table_name[, boto3_session])	Write all items from JSON file to a DynamoDB.
<code>read_items</code> (table_name[, index_name, ...])	Read items from given DynamoDB table.
<code>read_partiql_query</code> (query[, parameters, ...])	Read data from a DynamoDB table via a PartiQL query.

awswrangler.dynamodb.delete_items

`awswrangler.dynamodb.delete_items`(items: List[Dict[str, Any]], table_name: str, boto3_session: Session | None = None) → Any

Delete all items in the specified DynamoDB table.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

Check out the [Global Configurations Tutorial](#) for details.**Parameters**

- **items** (List[Dict[str, Any]]) – List which contains the items that will be deleted.
- **table_name** (str) – Name of the Amazon DynamoDB table.
- **boto3_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

Writing rows of DataFrame

```
>>> import awswrangler as wr
>>> wr.dynamodb.delete_items(
...     items=[{'key': 1}, {'key': 2, 'value': 'Hello'}],
...     table_name='table'
... )
```

awswrangler.dynamodb.execute_statement

`awswrangler.dynamodb.execute_statement`(*statement: str, parameters: List[Any] | None = None, consistent_read: bool = False, boto3_session: Session | None = None*) → `Iterator[Dict[str, Any]] | None`

Run a PartiQL statement against a DynamoDB table.

Parameters

- **statement** (*str*) – The PartiQL statement.
- **parameters** (*Optional[List[Any]]*) – The list of PartiQL parameters. These are applied to the statement in the order they are listed.
- **consistent_read** (*bool*) – The consistency of a read operation. If *True*, then a strongly consistent read is used. *False* by default.
- **boto3_session** (*Optional[boto3.Session]*) – Boto3 Session. If *None*, the default boto3 Session is used.

Returns

An iterator of the items from the statement response, if any.

Return type`Optional[Iterator[Dict[str, Any]]]`**Examples**

Insert an item

```
>>> import awswrangler as wr
>>> wr.dynamodb.execute_statement(
...     statement="INSERT INTO movies VALUE {'title': ?, 'year': ?, 'info': ?}",
...     parameters=[title, year, {"plot": plot, "rating": rating}],
... )
```

Select items

```
>>> wr.dynamodb.execute_statement(
...     statement="SELECT * FROM movies WHERE title=? AND year=?",
...     parameters=[title, year],
... )
```

Update items

```
>>> wr.dynamodb.execute_statement(  
...     statement="UPDATE movies SET info.rating=? WHERE title=? AND year=?",  
...     parameters=[rating, title, year],  
... )
```

Delete items

```
>>> wr.dynamodb.execute_statement(  
...     statement="DELETE FROM movies WHERE title=? AND year=?",  
...     parameters=[title, year],  
... )
```

aws wrangler.dynamodb.get_table

`aws wrangler.dynamodb.get_table(table_name: str, boto3_session: Session | None = None) → Any`

Get DynamoDB table object for specified table name.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **table_name** (*str*) – Name of the Amazon DynamoDB table.
- **boto3_session** (*Optional[boto3.Session()]*) – Boto3 Session. If None, the default boto3 Session is used.

Returns

dynamodb_table – Boto3 `DynamoDB.Table` object. <https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/dynamodb.html#DynamoDB.Table>

Return type

`boto3.resources.dynamodb.Table`

aws wrangler.dynamodb.put_csv

`aws wrangler.dynamodb.put_csv(path: str | Path, table_name: str, boto3_session: Session | None = None, **pandas_kwargs: Any) → Any`

Write all items from a CSV file to a DynamoDB.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **path** (*Union[str, Path]*) – Path as str or Path object to the CSV file which contains the items.

- **table_name** (*str*) – Name of the Amazon DynamoDB table.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **pandas_kwargs** – KEYWORD arguments forwarded to `pandas.read_csv()`. You can NOT pass *pandas_kwargs* explicit, just add valid Pandas arguments in the function call and awswrangler will accept it. e.g. `wr.dynamodb.put_csv('items.csv', 'my_table', sep='|', na_values=['null', 'none'], skip_blank_lines=True)` https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Returns

None.

Return type

None

Examples

Writing contents of CSV file

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_csv(
...     path='items.csv',
...     table_name='table'
... )
```

Writing contents of CSV file using pandas_kwargs

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_csv(
...     path='items.csv',
...     table_name='table',
...     sep='|',
...     na_values=['null', 'none']
... )
```

awswrangler.dynamodb.put_df

`awswrangler.dynamodb.put_df(df: DataFrame, table_name: str, boto3_session: Session | None = None) → Any`

Write all items from a DataFrame to a DynamoDB.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

Check out the [Global Configurations Tutorial](#) for details.**Parameters**

- **df** (*pd.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **table_name** (*str*) – Name of the Amazon DynamoDB table.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

Writing rows of DataFrame

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.dynamodb.put_df(
...     df=pd.DataFrame({'key': [1, 2, 3]}),
...     table_name='table'
... )
```

awswrangler.dynamodb.put_items

`awswrangler.dynamodb.put_items(items: List[Dict[str, Any]] | List[Mapping[str, Any]], table_name: str, boto3_session: Session | None = None) → Any`

Insert all items to the specified DynamoDB table.

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **items** (*Union[List[Dict[str, Any]], List[Mapping[str, Any]]*) – List which contains the items that will be inserted.
- **table_name** (*str*) – Name of the Amazon DynamoDB table.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

Writing items

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_items(
...     items=[{'key': 1}, {'key': 2, 'value': 'Hello'}],
...     table_name='table'
... )
```

awswrangler.dynamodb.put_json

`awswrangler.dynamodb.put_json(path: str | Path, table_name: str, boto3_session: Session | None = None) → Any`

Write all items from JSON file to a DynamoDB.

The JSON file can either contain a single item which will be inserted in the DynamoDB or an array of items which all be inserted.

Note: This function has arguments which can be configured globally through `wr.config` or environment variables:

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **path** (`Union[str, Path]`) – Path as str or Path object to the JSON file which contains the items.
- **table_name** (`str`) – Name of the Amazon DynamoDB table.
- **boto3_session** (`boto3.Session()`, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

Writing contents of JSON file

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_json(
...     path='items.json',
...     table_name='table'
... )
```

aws wrangler.dynamodb.read_items

```
aws wrangler.dynamodb.read_items(table_name: str, index_name: str | None = None, partition_values:
    Sequence[Any] | None = None, sort_values: Sequence[Any] | None =
    None, filter_expression: ConditionBase | str | None = None,
    key_condition_expression: ConditionBase | str | None = None,
    expression_attribute_names: Dict[str, str] | None = None,
    expression_attribute_values: Dict[str, Any] | None = None, consistent:
    bool = False, columns: Sequence[str] | None = None, allow_full_scan:
    bool = False, max_items_evaluated: int | None = None, as_dataframe:
    bool = True, boto3_session: Session | None = None) → DataFrame |
    List[Dict[str, Any]]
```

Read items from given DynamoDB table.

This function aims to gracefully handle (some of) the complexity of read actions available in Boto3 towards a DynamoDB table, abstracting it away while providing a single, unified endpoint.

Under the hood, it wraps all the four available read actions: `get_item`, `batch_get_item`, `query` and `scan`.

Parameters

- **table_name** (*str*) – DynamoDB table name.
- **index_name** (*str*, *optional*) – Name of the secondary global or local index on the table. Defaults to `None`.
- **partition_values** (*Sequence[Any]*, *optional*) – Partition key values to retrieve. Defaults to `None`.
- **sort_values** (*Sequence[Any]*, *optional*) – Sort key values to retrieve. Defaults to `None`.
- **filter_expression** (*Union[ConditionBase, str]*, *optional*) – Filter expression as string or combinations of `boto3.dynamodb.conditions.Attr` conditions. Defaults to `None`.
- **key_condition_expression** (*Union[ConditionBase, str]*, *optional*) – Key condition expression as string or combinations of `boto3.dynamodb.conditions.Key` conditions. Defaults to `None`.
- **expression_attribute_names** (*Mapping[str, str]*, *optional*) – Mapping of placeholder and target attributes. Defaults to `None`.
- **expression_attribute_values** (*Mapping[str, Any]*, *optional*) – Mapping of placeholder and target values. Defaults to `None`.
- **consistent** (*bool*) – If `True`, ensure that the performed read operation is strongly consistent, otherwise eventually consistent. Defaults to `False`.
- **columns** (*Sequence[str]*, *optional*) – Attributes to retain in the returned items. Defaults to `None` (all attributes).
- **allow_full_scan** (*bool*) – If `True`, allow full table scan without any filtering. Defaults to `False`.
- **max_items_evaluated** (*int*, *optional*) – Limit the number of items evaluated in case of query or scan operations. Defaults to `None` (all matching items).
- **as_dataframe** (*bool*) – If `True`, return items as `pd.DataFrame`, otherwise as `list/dict`. Defaults to `True`.
- **boto3_session** (*boto3.Session*, *optional*) – Boto3 Session. Defaults to `None` (the default boto3 Session will be used).

Raises

- **exceptions.InvalidArgumentType** – When the specified table has also a sort key but only the partition values are specified.
- **exceptions.InvalidArgumentCombination** – When both partition and sort values sequences are specified but they have different lengths, or when provided parameters are not enough informative to proceed with a read operation.

Returns

A Data frame containing the retrieved items, or a dictionary of returned items.

Return type

Union[pd.DataFrame, List[Mapping[str, Any]]]

Examples

Reading 5 random items from a table

```
>>> import awswrangler as wr
>>> df = wr.dynamodb.read_items(table_name='my-table', max_items_evaluated=5)
```

Strongly-consistent reading of a given partition value from a table

```
>>> import awswrangler as wr
>>> df = wr.dynamodb.read_items(table_name='my-table', partition_values=['my-value', '↪'], consistent=True)
```

Reading items pairwise-identified by partition and sort values, from a table with a composite primary key

```
>>> import awswrangler as wr
>>> df = wr.dynamodb.read_items(
...     table_name='my-table',
...     partition_values=['pv_1', 'pv_2'],
...     sort_values=['sv_1', 'sv_2']
... )
```

Reading items while retaining only specified attributes, automatically handling possible collision with DynamoDB reserved keywords

```
>>> import awswrangler as wr
>>> df = wr.dynamodb.read_items(
...     table_name='my-table',
...     partition_values=['my-value'],
...     columns=['connection', 'other_col'] # connection is a reserved keyword, ↪
...     ↪managed under the hood!
... )
```

Reading all items from a table explicitly allowing full scan

```
>>> import awswrangler as wr
>>> df = wr.dynamodb.read_items(table_name='my-table', allow_full_scan=True)
```

Reading items matching a KeyConditionExpression expressed with boto3.dynamodb.conditions.Key

```
>>> import awswrangler as wr
>>> from boto3.dynamodb.conditions import Key
>>> df = wr.dynamodb.read_items(
...     table_name='my-table',
...     key_condition_expression=(Key('key_1').eq('val_1') & Key('key_2').eq('val_2
... ))
... )
```

Same as above, but with KeyConditionExpression as string

```
>>> import awswrangler as wr
>>> df = wr.dynamodb.read_items(
...     table_name='my-table',
...     key_condition_expression='key_1 = :v1 and key_2 = :v2',
...     expression_attribute_values={':v1': 'val_1', ':v2': 'val_2'},
... )
```

Reading items matching a FilterExpression expressed with boto3.dynamodb.conditions.Attr

```
>>> import awswrangler as wr
>>> from boto3.dynamodb.conditions import Attr
>>> df = wr.dynamodb.read_items(
...     table_name='my-table',
...     filter_expression=Attr('my_attr').eq('this-value')
... )
```

Same as above, but with FilterExpression as string

```
>>> import awswrangler as wr
>>> df = wr.dynamodb.read_items(
...     table_name='my-table',
...     filter_expression='my_attr = :v',
...     expression_attribute_values={':v': 'this-value'}
... )
```

Reading items involving an attribute which collides with DynamoDB reserved keywords

```
>>> import awswrangler as wr
>>> df = wr.dynamodb.read_items(
...     table_name='my-table',
...     filter_expression='#operator = :v',
...     expression_attribute_names={'#operator': 'operator'},
...     expression_attribute_values={':v': 'this-value'}
... )
```


awswrangler.dynamodb.read_partiql_query

`awswrangler.dynamodb.read_partiql_query(query: str, parameters: List[Any] | None = None, chunked: bool = False, boto3_session: Session | None = None) → DataFrame | Iterator[DataFrame]`

Read data from a DynamoDB table via a PartiQL query.

Parameters

- **query** (*str*) – The PartiQL statement.
- **parameters** (*Optional[List[Any]]*) – The list of PartiQL parameters. These are applied to the statement in the order they are listed.
- **chunked** (*bool*) – If *True* an iterable of DataFrames is returned. False by default.
- **boto3_session** (*Optional[boto3.Session]*) – Boto3 Session. If *None*, the default boto3 Session is used.

Returns

Result as Pandas DataFrame.

Return type

Union[pd.DataFrame, Iterator[pd.DataFrame]]

Examples

Select all contents from a table

```
>>> import awswrangler as wr
>>> wr.dynamodb.read_partiql_query(
...     query="SELECT * FROM my_table WHERE title=? AND year=?",
...     parameters=[title, year],
... )
```

Select specific columns from a table

```
>>> wr.dynamodb.read_partiql_query(
...     query="SELECT id FROM table"
... )
```

1.4.14 Amazon Timestream

<code>create_database(database[, kms_key_id, ...])</code>	Create a new Timestream database.
<code>create_table(database, table, ...[, tags, ...])</code>	Create a new Timestream database.
<code>delete_database(database[, boto3_session])</code>	Delete a given Timestream database.
<code>delete_table(database, table[, boto3_session])</code>	Delete a given Timestream table.
<code>query(sql[, chunked, pagination_config, ...])</code>	Run a query and retrieve the result as a Pandas DataFrame.
<code>write(df, database, table, time_col, ...[, ...])</code>	Store a Pandas DataFrame into a Amazon Timestream table.

awsrangler.timestream.create_database

```
awsrangler.timestream.create_database(database: str, kms_key_id: str | None = None, tags: Dict[str, str] | None = None, boto3_session: Session | None = None) → str
```

Create a new Timestream database.

Note: If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.

Parameters

- **database** (*str*) – Database name.
- **kms_key_id** (*Optional[str]*) – The KMS key for the database. If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.
- **tags** (*Optional[Dict[str, str]]*) – Key/Value dict to put on the database. Tags enable you to categorize databases and/or tables, for example, by purpose, owner, or environment. e.g. {"foo": "boo", "bar": "xoo"}
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

The Amazon Resource Name that uniquely identifies this database. (ARN)

Return type

str

Examples

Creating a database.

```
>>> import awswrangler as wr
>>> arn = wr.timestream.create_database("MyDatabase")
```

awsrangler.timestream.create_table

```
awsrangler.timestream.create_table(database: str, table: str, memory_retention_hours: int,
                                   magnetic_retention_days: int, tags: Dict[str, str] | None = None,
                                   timestream_additional_kwargs: Dict[str, Any] | None = None,
                                   boto3_session: Session | None = None) → str
```

Create a new Timestream database.

Note: If the KMS key is not specified, the database will be encrypted with a Timestream managed KMS key located in your account.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.

- **memory_retention_hours** (*int*) – The duration for which data must be stored in the memory store.
- **magnetic_retention_days** (*int*) – The duration for which data must be stored in the magnetic store.
- **tags** (*Optional[Dict[str, str]]*) – Key/Value dict to put on the table. Tags enable you to categorize databases and/or tables, for example, by purpose, owner, or environment. e.g. {"foo": "boo", "bar": "xoo"}
- **timestream_additional_kwargs** (*Optional[Dict[str, Any]]*) – Forwarded to boto3 requests. e.g. timestream_additional_kwargs={'MagneticStoreWriteProperties': {'EnableMagneticStoreWrites': True}}
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

The Amazon Resource Name that uniquely identifies this database. (ARN)

Return type

str

Examples

Creating a table.

```
>>> import awswrangler as wr
>>> arn = wr.timestream.create_table(
...     database="MyDatabase",
...     table="MyTable",
...     memory_retention_hours=3,
...     magnetic_retention_days=7
... )
```

awswrangler.timestream.delete_database

awswrangler.timestream.delete_database(*database: str, boto3_session: Session | None = None*) → None

Delete a given Timestream database. This is an irreversible operation.

After a database is deleted, the time series data from its tables cannot be recovered.

All tables in the database must be deleted first, or a ValidationException error will be thrown.

Due to the nature of distributed retries, the operation can return either success or a ResourceNotFoundException. Clients should consider them equivalent.

Parameters

- **database** (*str*) – Database name.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

Deleting a database

```
>>> import awswrangler as wr
>>> arn = wr.timestream.delete_database("MyDatabase")
```

awswrangler.timestream.delete_table

`awswrangler.timestream.delete_table(database: str, table: str, boto3_session: Session | None = None) → None`

Delete a given Timestream table.

This is an irreversible operation.

After a Timestream database table is deleted, the time series data stored in the table cannot be recovered.

Due to the nature of distributed retries, the operation can return either success or a `ResourceNotFoundException`. Clients should consider them equivalent.

Parameters

- **database** (*str*) – Database name.
- **table** (*str*) – Table name.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

Deleting a table

```
>>> import awswrangler as wr
>>> arn = wr.timestream.delete_table("MyDatabase", "MyTable")
```

awswrangler.timestream.query

`awswrangler.timestream.query(sql: str, chunked: bool = False, pagination_config: Dict[str, Any] | None = None, boto3_session: Session | None = None) → DataFrame | Iterator[DataFrame]`

Run a query and retrieve the result as a Pandas DataFrame.

Parameters

- **sql** (*str*) – SQL query.
- **chunked** (*bool*) – If True returns dataframe iterator, and a single dataframe otherwise. False by default.

- **pagination_config** (*Dict[str, Any]*, *optional*) – Pagination configuration dictionary of a form { 'MaxItems': 10, 'PageSize': 10, 'StartingToken': '...' }
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>

Return type

Union[pd.DataFrame, Iterator[pd.DataFrame]]

Examples

Run a query and return the result as a Pandas DataFrame or an iterable.

```
>>> import awswrangler as wr
>>> df = wr.timestream.query('SELECT * FROM "sampleDB"."sampleTable" ORDER BY time_
↳DESC LIMIT 10')
```

awswrangler.timestream.write

awswrangler.timestream.write(*df: DataFrame, database: str, table: str, time_col: str, measure_col: str | List[str], dimensions_cols: List[str], version: int = 1, num_threads: int = 32, measure_name: str | None = None, boto3_session: Session | None = None*) → List[Dict[str, str]]

Store a Pandas DataFrame into a Amazon Timestream table.

If the Timestream service rejects a record(s), this function will not throw a Python exception. Instead it will return the rejection information.

Parameters

- **df** (*pandas.DataFrame*) – Pandas DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- **database** (*str*) – Amazon Timestream database name.
- **table** (*str*) – Amazon Timestream table name.
- **time_col** (*str*) – DataFrame column name to be used as time. MUST be a timestamp column.
- **measure_col** (*Union[str, List[str]]*) – DataFrame column name(s) to be used as measure.
- **dimensions_cols** (*List[str]*) – List of DataFrame column names to be used as dimensions.
- **version** (*int*) – Version number used for upserts. Documentation https://docs.aws.amazon.com/timestream/latest/developerguide/API_WriteRecords.html.
- **measure_name** (*Optional[str]*) – Name that represents the data attribute of the time series. Overrides measure_col if specified.
- **num_threads** (*str*) – Number of thread to be used for concurrent writing.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.

Returns

Rejected records. Possible reasons for rejection are described here: https://docs.aws.amazon.com/timestream/latest/developerguide/API_RejectedRecord.html

Return type

List[Dict[str, str]]

Examples

Store a Pandas DataFrame into a Amazon Timestream table.

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> df = pd.DataFrame(
>>>     {
>>>         "time": [datetime.now(), datetime.now(), datetime.now()],
>>>         "dim0": ["foo", "boo", "bar"],
>>>         "dim1": [1, 2, 3],
>>>         "measure": [1.0, 1.1, 1.2],
>>>     }
>>> )
>>> rejected_records = wr.timestream.write(
>>>     df=df,
>>>     database="sampleDB",
>>>     table="sampleTable",
>>>     time_col="time",
>>>     measure_col="measure",
>>>     dimensions_cols=["dim0", "dim1"],
>>> )
>>> assert len(rejected_records) == 0
```

Return value if some records are rejected.

```
>>> [
>>>     {
>>>         'ExistingVersion': 2,
>>>         'Reason': 'The record version 1 is lower than the existing version 2. A
→         'higher version is required to update the measure value.',
>>>         'RecordIndex': 0
>>>     }
>>> ]
```

1.4.15 Amazon EMR

<code>build_spark_step(path[, args, deploy_mode, ...])</code>	Build the Step structure (dictionary).
<code>build_step(command[, name, ...])</code>	Build the Step structure (dictionary).
<code>create_cluster(subnet_id[, cluster_name, ...])</code>	Create a EMR cluster with instance fleets configuration.
<code>get_cluster_state(cluster_id[, boto3_session])</code>	Get the EMR cluster state.
<code>get_step_state(cluster_id, step_id[, ...])</code>	Get EMR step state.
<code>submit_ecr_credentials_refresh(cluster_id, path)</code>	Update internal ECR credentials.
<code>submit_spark_step(cluster_id, path[, args, ...])</code>	Submit Spark Step.
<code>submit_step(cluster_id, command[, name, ...])</code>	Submit new job in the EMR Cluster.
<code>submit_steps(cluster_id, steps[, boto3_session])</code>	Submit a list of steps.
<code>terminate_cluster(cluster_id[, boto3_session])</code>	Terminate EMR cluster.

awswrangler.emr.build_spark_step

`awswrangler.emr.build_spark_step(path: str, args: List[str] | None = None, deploy_mode: str = 'cluster', docker_image: str | None = None, name: str = 'my-step', action_on_failure: str = 'CONTINUE', region: str | None = None, boto3_session: Session | None = None) → Dict[str, Any]`

Build the Step structure (dictionary).

Parameters

- **path** (*str*) – Script path. (e.g. `s3://bucket/app.py`)
- **args** (*List[str]*, *optional*) – CLI args to use with script
- **deploy_mode** (*str*) – “cluster” | “client”
- **docker_image** (*str*, *optional*) – e.g. “{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{IMAGE_NAME}”
- **name** (*str*, *optional*) – Step name.
- **action_on_failure** (*str*) – ‘TERMINATE_JOB_FLOW’, ‘TERMINATE_CLUSTER’, ‘CANCEL_AND_WAIT’, ‘CONTINUE’
- **region** (*str*, *optional*) – Region name to not get it from `boto3.Session`. (e.g. `us-east-1`)
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.

Returns

Step structure.

Return type

`Dict[str, Any]`

Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_steps(
>>>     cluster_id="cluster-id",
>>>     steps=[
>>>         wr.emr.build_spark_step(path="s3://bucket/app.py")
>>>     ]
>>> )
```

awswrangler.emr.build_step

`awswrangler.emr.build_step`(*command*: str, *name*: str = 'my-step', *action_on_failure*: str = 'CONTINUE', *script*: bool = False, *region*: str | None = None, *boto3_session*: Session | None = None) → Dict[str, Any]

Build the Step structure (dictionary).

Parameters

- **command** (str) – e.g. 'echo "Hello!"' e.g. for script 's3://.../script.sh arg1 arg2'
- **name** (str, optional) – Step name.
- **action_on_failure** (str) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **script** (bool) – False for raw command or True for script runner. <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-commandrunner.html>
- **region** (str, optional) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Step structure.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> steps = []
>>> for cmd in ['echo "Hello"', 'ls -la']:
...     steps.append(wr.emr.build_step(name=cmd, command=cmd))
>>> wr.emr.submit_steps(cluster_id="cluster-id", steps=steps)
```


aws wrangler.emr.create_cluster

```
aws wrangler.emr.create_cluster(subnet_id: str, cluster_name: str = 'my-emr-cluster', logging_s3_path: str |
    None = None, emr_release: str = 'emr-6.7.0', emr_ec2_role: str =
    'EMR_EC2_DefaultRole', emr_role: str = 'EMR_DefaultRole',
    instance_type_master: str = 'r5.xlarge', instance_type_core: str =
    'r5.xlarge', instance_type_task: str = 'r5.xlarge', instance_ebs_size_master:
    int = 64, instance_ebs_size_core: int = 64, instance_ebs_size_task: int =
    64, instance_num_on_demand_master: int = 1,
    instance_num_on_demand_core: int = 0, instance_num_on_demand_task:
    int = 0, instance_num_spot_master: int = 0, instance_num_spot_core: int =
    0, instance_num_spot_task: int = 0,
    spot_bid_percentage_of_on_demand_master: int = 100,
    spot_bid_percentage_of_on_demand_core: int = 100,
    spot_bid_percentage_of_on_demand_task: int = 100,
    spot_provisioning_timeout_master: int = 5,
    spot_provisioning_timeout_core: int = 5, spot_provisioning_timeout_task:
    int = 5, spot_timeout_to_on_demand_master: bool = True,
    spot_timeout_to_on_demand_core: bool = True,
    spot_timeout_to_on_demand_task: bool = True, python3: bool = True,
    spark_glue_catalog: bool = True, hive_glue_catalog: bool = True,
    presto_glue_catalog: bool = True, consistent_view: bool = False,
    consistent_view_retry_seconds: int = 10, consistent_view_retry_count: int
    = 5, consistent_view_table_name: str = 'EmrFSMetadata',
    bootstraps_paths: List[str] | None = None, debugging: bool = True,
    applications: List[str] | None = None, visible_to_all_users: bool = True,
    key_pair_name: str | None = None, security_group_master: str | None =
    None, security_groups_master_additional: List[str] | None = None,
    security_group_slave: str | None = None, security_groups_slave_additional:
    List[str] | None = None, security_group_service_access: str | None = None,
    security_configuration: str | None = None, docker: bool = False,
    extra_public_registries: List[str] | None = None, spark_log_level: str =
    'WARN', spark_jars_path: List[str] | None = None, spark_defaults: Dict[str,
    str] | None = None, spark_pyarrow: bool = False, custom_classifications:
    List[Dict[str, Any]] | None = None, maximize_resource_allocation: bool =
    False, steps: List[Dict[str, Any]] | None = None, custom_ami_id: str | None
    = None, step_concurrency_level: int = 1,
    keep_cluster_alive_when_no_steps: bool = True, termination_protected:
    bool = False, auto_termination_policy: Dict[str, int] | None = None, tags:
    Dict[str, str] | None = None, boto3_session: Session | None = None,
    configurations: List[Dict[str, Any]] | None = None) → str
```

Create a EMR cluster with instance fleets configuration.

<https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-instance-fleet.html>

Parameters

- **subnet_id** (*str*) – VPC subnet ID.
- **cluster_name** (*str*) – Cluster name.
- **logging_s3_path** (*str, optional*) – Logging s3 path (e.g. s3://BUCKET_NAME/DIRECTORY_NAME/). If None, the default is s3://aws-logs-{AccountId}-{RegionId}/elasticmapreduce/
- **emr_release** (*str*) – EMR release (e.g. emr-5.28.0).

- **emr_ec2_role** (*str*) – IAM role name.
- **emr_role** (*str*) – IAM role name.
- **instance_type_master** (*str*) – EC2 instance type.
- **instance_type_core** (*str*) – EC2 instance type.
- **instance_type_task** (*str*) – EC2 instance type.
- **instance_ebs_size_master** (*int*) – Size of EBS in GB.
- **instance_ebs_size_core** (*int*) – Size of EBS in GB.
- **instance_ebs_size_task** (*int*) – Size of EBS in GB.
- **instance_num_on_demand_master** (*int*) – Number of on demand instances.
- **instance_num_on_demand_core** (*int*) – Number of on demand instances.
- **instance_num_on_demand_task** (*int*) – Number of on demand instances.
- **instance_num_spot_master** (*int*) – Number of spot instances.
- **instance_num_spot_core** (*int*) – Number of spot instances.
- **instance_num_spot_task** (*int*) – Number of spot instances.
- **spot_bid_percentage_of_on_demand_master** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot_bid_percentage_of_on_demand_core** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot_bid_percentage_of_on_demand_task** (*int*) – The bid price, as a percentage of On-Demand price.
- **spot_provisioning_timeout_master** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the `TimeOutAction` is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot_provisioning_timeout_core** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the `TimeOutAction` is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot_provisioning_timeout_task** (*int*) – The spot provisioning timeout period in minutes. If Spot instances are not provisioned within this time period, the `TimeOutAction` is taken. Minimum value is 5 and maximum value is 1440. The timeout applies only during initial provisioning, when the cluster is first created.
- **spot_timeout_to_on_demand_master** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?
- **spot_timeout_to_on_demand_core** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?
- **spot_timeout_to_on_demand_task** (*bool*) – After a provisioning timeout should the cluster switch to on demand or shutdown?
- **python3** (*bool*) – Python 3 Enabled?
- **spark_glue_catalog** (*bool*) – Spark integration with Glue Catalog?
- **hive_glue_catalog** (*bool*) – Hive integration with Glue Catalog?

- **presto_glue_catalog** (*bool*) – Presto integration with Glue Catalog?
- **consistent_view** (*bool*) – Consistent view allows EMR clusters to check for list and read-after-write consistency for Amazon S3 objects written by or synced with EMRFS. <https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-consistent-view.html>
- **consistent_view_retry_seconds** (*int*) – Delay between the tries (seconds).
- **consistent_view_retry_count** (*int*) – Number of tries.
- **consistent_view_table_name** (*str*) – Name of the DynamoDB table to store the consistent view data.
- **bootstraps_paths** (*List[str], optional*) – Bootstraps paths (e.g. ["s3://BUCKET_NAME/script.sh"]).
- **debugging** (*bool*) – Debugging enabled?
- **applications** (*List[str], optional*) – List of applications (e.g. ["Hadoop", "Spark", "Ganglia", "Hive"]). If None, ["Spark"] will be considered.
- **visible_to_all_users** (*bool*) – True or False.
- **key_pair_name** (*str, optional*) – Key pair name.
- **security_group_master** (*str, optional*) – The identifier of the Amazon EC2 security group for the master node.
- **security_groups_master_additional** (*str, optional*) – A list of additional Amazon EC2 security group IDs for the master node.
- **security_group_slave** (*str, optional*) – The identifier of the Amazon EC2 security group for the core and task nodes.
- **security_groups_slave_additional** (*str, optional*) – A list of additional Amazon EC2 security group IDs for the core and task nodes.
- **security_group_service_access** (*str, optional*) – The identifier of the Amazon EC2 security group for the Amazon EMR service to access clusters in VPC private subnets.
- **security_configuration** (*str, optional*) – The name of a security configuration to apply to the cluster.
- **docker** (*bool*) – Enable Docker Hub and ECR registries access.
- **extra_public_registries** (*List[str], optional*) – Additional docker registries.
- **spark_log_level** (*str*) – log4j.rootCategory log level (ALL, DEBUG, INFO, WARN, ERROR, FATAL, OFF, TRACE).
- **spark_jars_path** (*List[str], optional*) – spark.jars e.g. [s3://.../foo.jar, s3://.../boo.jar] <https://spark.apache.org/docs/latest/configuration.html>
- **spark_defaults** (*Dict[str, str], optional*) – <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html#spark-defaults>
- **spark_pyarrow** (*bool*) – Enable PySpark to use PyArrow behind the scenes. P.S. You must install pyarrow by your self via bootstrap
- **custom_classifications** (*List[Dict[str, Any]], optional*) – Extra classifications.
- **maximize_resource_allocation** (*bool*) – Configure your executors to utilize the maximum resources possible <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html#emr-spark-maximizeresourceallocation>

- **custom_ami_id** (*Optional[str]*) – The custom AMI ID to use for the provisioned instance group
- **steps** (*List[Dict[str, Any]]*, *optional*) – Steps definitions (Obs : str Use `EMR.build_step()` to build it)
- **keep_cluster_alive_when_no_steps** (*bool*) – Specifies whether the cluster should remain available after completing all steps
- **termination_protected** (*bool*) – Specifies whether the Amazon EC2 instances in the cluster are protected from termination by API calls, user intervention, or in the event of a job-flow error.
- **auto_termination_policy** (*Optional[Dict[str, int]]*) – Specifies the auto-termination policy that is attached to an Amazon EMR cluster eg. `auto_termination_policy = {'IdleTimeout': 123}` IdleTimeout specifies the amount of idle time in seconds after which the cluster automatically terminates. You can specify a minimum of 60 seconds and a maximum of 604800 seconds (seven days).
- **tags** (*Dict[str, str]*, *optional*) – Key/Value collection to put on the Cluster. e.g. `{“foo”: “boo”, “bar”: “xoo”}`
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **configurations** (*List[Dict[str, Any]]*, *optional*) – The list of configurations supplied for an EMR cluster instance group.

By default, adds log4j config as follows: `{“Classification”: “spark-log4j”, “Properties”: {“log4j.rootCategory”: “f”{pars[‘spark_log_level’]}, console”}}`

Returns

Cluster ID.

Return type

str

Examples

Minimal Example

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster("SUBNET_ID")
```

Minimal Example With Custom Classification

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster(
>>>     subnet_id="SUBNET_ID",
>>>     custom_classifications=[
>>>         {
>>>             "Classification": "livy-conf",
>>>             "Properties": {
>>>                 "livy.spark.master": "yarn",
>>>                 "livy.spark.deploy-mode": "cluster",
>>>                 "livy.server.session.timeout": "16h",
>>>             },
>>>         },
>>>     ]
>>> )
```

(continues on next page)

(continued from previous page)

```
>>> ],
>>> )
```

Full Example

```
>>> import awswrangler as wr
>>> cluster_id = wr.emr.create_cluster(
...     cluster_name="wrangler_cluster",
...     logging_s3_path=f"s3://BUCKET_NAME/emr-logs/",
...     emr_release="emr-5.28.0",
...     subnet_id="SUBNET_ID",
...     emr_ec2_role="EMR_EC2_DefaultRole",
...     emr_role="EMR_DefaultRole",
...     instance_type_master="m5.xlarge",
...     instance_type_core="m5.xlarge",
...     instance_type_task="m5.xlarge",
...     instance_ebs_size_master=50,
...     instance_ebs_size_core=50,
...     instance_ebs_size_task=50,
...     instance_num_on_demand_master=1,
...     instance_num_on_demand_core=1,
...     instance_num_on_demand_task=1,
...     instance_num_spot_master=0,
...     instance_num_spot_core=1,
...     instance_num_spot_task=1,
...     spot_bid_percentage_of_on_demand_master=100,
...     spot_bid_percentage_of_on_demand_core=100,
...     spot_bid_percentage_of_on_demand_task=100,
...     spot_provisioning_timeout_master=5,
...     spot_provisioning_timeout_core=5,
...     spot_provisioning_timeout_task=5,
...     spot_timeout_to_on_demand_master=True,
...     spot_timeout_to_on_demand_core=True,
...     spot_timeout_to_on_demand_task=True,
...     python3=True,
...     spark_glue_catalog=True,
...     hive_glue_catalog=True,
...     presto_glue_catalog=True,
...     bootstraps_paths=None,
...     debugging=True,
...     applications=["Hadoop", "Spark", "Ganglia", "Hive"],
...     visible_to_all_users=True,
...     key_pair_name=None,
...     spark_jars_path=[f"s3://...jar"],
...     maximize_resource_allocation=True,
...     keep_cluster_alive_when_no_steps=True,
...     termination_protected=False,
...     spark_pyarrow=True,
...     tags={
...         "foo": "boo"
...     })
```

aws wrangler.emr.get_cluster_state

`aws wrangler.emr.get_cluster_state(cluster_id: str, boto3_session: Session | None = None) → str`

Get the EMR cluster state.

Possible states: 'STARTING', 'BOOTSTRAPPING', 'RUNNING', 'WAITING', 'TERMINATING', 'TERMINATED', 'TERMINATED_WITH_ERRORS'

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

State.

Return type

str

Examples

```
>>> import aws wrangler as wr
>>> state = wr.emr.get_cluster_state("cluster-id")
```

aws wrangler.emr.get_step_state

`aws wrangler.emr.get_step_state(cluster_id: str, step_id: str, boto3_session: Session | None = None) → str`

Get EMR step state.

Possible states: 'PENDING', 'CANCEL_PENDING', 'RUNNING', 'COMPLETED', 'CANCELLED', 'FAILED', 'INTERRUPTED'

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **step_id** (*str*) – Step ID.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

State.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> state = wr.emr.get_step_state("cluster-id", "step-id")
```

awswrangler.emr.submit_ecr_credentials_refresh

```
awswrangler.emr.submit_ecr_credentials_refresh(cluster_id: str, path: str, action_on_failure: str =
                                                'CONTINUE', boto3_session: Session | None = None)
                                                → str
```

Update internal ECR credentials.

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **path** (*str*) – Amazon S3 path where awswrangler will stage the script `ecr_credentials_refresh.py` (e.g. `s3://bucket/emr/`)
- **action_on_failure** (*str*) – ‘TERMINATE_JOB_FLOW’, ‘TERMINATE_CLUSTER’, ‘CANCEL_AND_WAIT’, ‘CONTINUE’
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive `None`.

Returns

Step ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_ecr_credentials_refresh("cluster-id", "s3://bucket/emr/
↳")
```

awswrangler.emr.submit_spark_step

```
awswrangler.emr.submit_spark_step(cluster_id: str, path: str, args: List[str] | None = None, deploy_mode:
                                   str = 'cluster', docker_image: str | None = None, name: str = 'my-step',
                                   action_on_failure: str = 'CONTINUE', region: str | None = None,
                                   boto3_session: Session | None = None) → str
```

Submit Spark Step.

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **path** (*str*) – Script path. (e.g. `s3://bucket/app.py`)
- **args** (*List[str]*, *optional*) – CLI args to use with script eg. `args = ["--name", "hello-world"]`
- **deploy_mode** (*str*) – “cluster” | “client”

- **docker_image**(*str*, *optional*) – e.g. “{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{IMAGE_NAME}”
- **name**(*str*, *optional*) – Step name.
- **action_on_failure**(*str*) – ‘TERMINATE_JOB_FLOW’, ‘TERMINATE_CLUSTER’, ‘CANCEL_AND_WAIT’, ‘CONTINUE’
- **region**(*str*, *optional*) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3_session**(*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Step ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_spark_step(
>>>     cluster_id="cluster-id",
>>>     path="s3://bucket/emr/app.py"
>>> )
```

awswrangler.emr.submit_step

awswrangler.emr.submit_step(*cluster_id: str*, *command: str*, *name: str = 'my-step'*, *action_on_failure: str = 'CONTINUE'*, *script: bool = False*, *boto3_session: Session | None = None*) → str

Submit new job in the EMR Cluster.

Parameters

- **cluster_id**(*str*) – Cluster ID.
- **command**(*str*) – e.g. ‘echo “Hello!”’ e.g. for script ‘s3://.../script.sh arg1 arg2’
- **name**(*str*, *optional*) – Step name.
- **action_on_failure**(*str*) – ‘TERMINATE_JOB_FLOW’, ‘TERMINATE_CLUSTER’, ‘CANCEL_AND_WAIT’, ‘CONTINUE’
- **script**(*bool*) – True for raw command or False for script runner. <https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-commandrunner.html>
- **boto3_session**(*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Step ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> step_id = wr.emr.submit_step(
...     cluster_id=cluster_id,
...     name="step_test",
...     command="s3://...script.sh arg1 arg2",
...     script=True)
```

awswrangler.emr.submit_steps

`awswrangler.emr.submit_steps(cluster_id: str, steps: List[Dict[str, Any]], boto3_session: Session | None = None) → List[str]`

Submit a list of steps.

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **steps** (*List[Dict[str, Any]]*) – Steps definitions (Obs: Use EMR.build_step() to build it).
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

List of step IDs.

Return type

List[str]

Examples

```
>>> import awswrangler as wr
>>> for cmd in ['echo "Hello"', 'ls -la']:
...     steps.append(wr.emr.build_step(name=cmd, command=cmd))
>>> wr.emr.submit_steps(cluster_id="cluster-id", steps=steps)
```

awswrangler.emr.terminate_cluster

`awswrangler.emr.terminate_cluster(cluster_id: str, boto3_session: Session | None = None) → None`

Terminate EMR cluster.

Parameters

- **cluster_id** (*str*) – Cluster ID.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.emr.terminate_cluster("cluster-id")
```

1.4.16 Amazon CloudWatch Logs

<code>read_logs(query, log_group_names[, ...])</code>	Run a query against AWS CloudWatchLogs Insights and convert the results to Pandas DataFrame.
<code>run_query(query, log_group_names[, ...])</code>	Run a query against AWS CloudWatchLogs Insights and wait the results.
<code>start_query(query, log_group_names[, ...])</code>	Run a query against AWS CloudWatchLogs Insights.
<code>wait_query(query_id[, boto3_session, ...])</code>	Wait query ends.
<code>describe_log_streams(log_group_name[, ...])</code>	List the log streams for the specified log group, return results as a Pandas DataFrame.
<code>filter_log_events(log_group_name[, ...])</code>	List log events from the specified log group.

awswrangler.cloudwatch.read_logs

```
awswrangler.cloudwatch.read_logs(query: str, log_group_names: List[str], start_time: datetime =
    datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc),
    end_time: datetime = datetime.datetime(2023, 3, 3, 12, 23, 44, 318160),
    limit: int | None = None, boto3_session: Session | None = None) →
    DataFrame
```

Run a query against AWS CloudWatchLogs Insights and convert the results to Pandas DataFrame.

https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html

Parameters

- **query** (*str*) – The query string.
- **log_group_names** (*str*) – The list of log groups to be queried. You can include up to 20 log groups.
- **start_time** (*datetime.datetime*) – The beginning of the time range to query.
- **end_time** (*datetime.datetime*) – The end of the time range to query.
- **limit** (*Optional[int]*) – The maximum number of log events to return in the query.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Result as a Pandas DataFrame.

Return type

pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df = wr.cloudwatch.read_logs(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
```

awswrangler.cloudwatch.run_query

```
awswrangler.cloudwatch.run_query(query: str, log_group_names: List[str], start_time: datetime =
                                datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc),
                                end_time: datetime = datetime.datetime(2023, 3, 3, 12, 23, 44, 318117),
                                limit: int | None = None, boto3_session: Session | None = None) →
                                List[List[Dict[str, str]]]
```

Run a query against AWS CloudWatch Logs Insights and wait the results.

https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html

Parameters

- **query** (*str*) – The query string.
- **log_group_names** (*str*) – The list of log groups to be queried. You can include up to 20 log groups.
- **start_time** (*datetime.datetime*) – The beginning of the time range to query.
- **end_time** (*datetime.datetime*) – The end of the time range to query.
- **limit** (*Optional[int]*) – The maximum number of log events to return in the query.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Result.

Return type

List[List[Dict[str, str]]]

Examples

```
>>> import awswrangler as wr
>>> result = wr.cloudwatch.run_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
```

awsrangler.cloudwatch.start_query

```
awsrangler.cloudwatch.start_query(query: str, log_group_names: List[str], start_time: datetime =
    datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc),
    end_time: datetime = datetime.datetime(2023, 3, 3, 12, 23, 44,
    318067), limit: int | None = None, boto3_session: Session | None =
    None) → str
```

Run a query against AWS CloudWatchLogs Insights.

https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html

Parameters

- **query** (*str*) – The query string.
- **log_group_names** (*str*) – The list of log groups to be queried. You can include up to 20 log groups.
- **start_time** (*datetime.datetime*) – The beginning of the time range to query.
- **end_time** (*datetime.datetime*) – The end of the time range to query.
- **limit** (*Optional[int]*) – The maximum number of log events to return in the query.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Query ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> query_id = wr.cloudwatch.start_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
```

awsrangler.cloudwatch.wait_query

```
awsrangler.cloudwatch.wait_query(query_id: str, boto3_session: Session | None = None,
    cloudwatch_query_wait_polling_delay: float = 0.2) → Any
```

Wait query ends.

https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html

Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

- `cloudwatch_query_wait_polling_delay`

Check out the [Global Configurations Tutorial](#) for details.

Parameters

- **query_id** (*str*) – Query ID.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **cloudwatch_query_wait_polling_delay** (*float*, *default: 0.2 seconds*) – Interval in seconds for how often the function will check if the CloudWatch query has completed.

Returns

Query result payload.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> query_id = wr.cloudwatch.start_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
... response = wr.cloudwatch.wait_query(query_id=query_id)
```

awswrangler.cloudwatch.describe_log_streams

awswrangler.cloudwatch.describe_log_streams(*log_group_name: str*, *log_stream_name_prefix: str | None = None*, *order_by: str | None = 'LogStreamName'*, *descending: bool | None = False*, *limit: int | None = 50*, *boto3_session: Session | None = None*) → DataFrame

List the log streams for the specified log group, return results as a Pandas DataFrame.

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/logs.html#CloudWatchLogs.Client.describe_log_streams

Parameters

- **log_group_name** (*str*) – The name of the log group.
- **log_stream_name_prefix** (*str*) – The prefix to match log streams' name
- **order_by** (*str*) – If the value is LogStreamName , the results are ordered by log stream name. If the value is LastEventTime , the results are ordered by the event time. The default value is LogStreamName .
- **descending** (*bool*) – If the value is True, results are returned in descending order. If the value is to False, results are returned in ascending order. The default value is False.
- **limit** (*Optional[int]*) – The maximum number of items returned. The default is up to 50 items.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Result as a Pandas DataFrame.

Return type

pandas.DataFrame

Examples

```
>>> import awswrangler as wr
>>> df = wr.cloudwatch.describe_log_streams(
...     log_group_name="aws_sdk_pandas_log_group",
...     log_stream_name_prefix="aws_sdk_pandas_log_stream",
... )
```

awswrangler.cloudwatch.filter_log_events

`awswrangler.cloudwatch.filter_log_events(log_group_name: str, log_stream_name_prefix: str | None = None, log_stream_names: List[str] | None = None, filter_pattern: str | None = None, start_time: datetime | None = None, end_time: datetime | None = None, boto3_session: Session | None = None) → DataFrame`

List log events from the specified log group. The results are returned as Pandas DataFrame.

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/logs.html#CloudWatchLogs.Client.filter_log_events

Note: Cannot call `filter_log_events` with both `log_stream_names` and `log_stream_name_prefix`.

Parameters

- **log_group_name** (*str*) – The name of the log group.
- **log_stream_name_prefix** (*str*) – Filters the results to include only events from log streams that have names starting with this prefix.
- **log_stream_names** (*List[str]*) – Filters the results to only logs from the log streams in this list.
- **filter_pattern** (*str*) – The filter pattern to use. If not provided, all the events are matched.
- **start_time** (*datetime.datetime*) – Events with a timestamp before this time are not returned.
- **end_time** (*datetime.datetime*) – Events with a timestamp later than this time are not returned.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

Result as a Pandas DataFrame.

Return type

pandas.DataFrame

Examples

Get all log events from log group 'aws_sdk_pandas_log_group' that have log stream prefix 'aws_sdk_pandas_log_stream'

```
>>> import awswrangler as wr
>>> df = wr.cloudwatch.filter_log_events(
...     log_group_name="aws_sdk_pandas_log_group",
...     log_stream_name_prefix="aws_sdk_pandas_log_stream",
... )
```

Get all log events contains 'REPORT' from log stream 'aws_sdk_pandas_log_stream_one' and 'aws_sdk_pandas_log_stream_two' from log group 'aws_sdk_pandas_log_group'

```
>>> import awswrangler as wr
>>> df = wr.cloudwatch.filter_log_events(
...     log_group_name="aws_sdk_pandas_log_group",
...     log_stream_names=["aws_sdk_pandas_log_stream_one", "aws_sdk_pandas_log_
↪ stream_two"],
...     filter_pattern='REPORT',
... )
```

1.4.17 Amazon QuickSight

<code>cancel_ingestion(ingestion_id[, ...])</code>	Cancel an ongoing ingestion of data into SPICE.
<code>create_athena_data_source(name[, workgroup, ...])</code>	Create a QuickSight data source pointing to an Athena/Workgroup.
<code>create_athena_dataset(name[, database, ...])</code>	Create a QuickSight dataset.
<code>create_ingestion([dataset_name, dataset_id, ...])</code>	Create and starts a new SPICE ingestion on a dataset.
<code>delete_all_dashboards([account_id, ...])</code>	Delete all dashboards.
<code>delete_all_data_sources([account_id, ...])</code>	Delete all data sources.
<code>delete_all_datasets([account_id, ...])</code>	Delete all datasets.
<code>delete_all_templates([account_id, ...])</code>	Delete all templates.
<code>delete_dashboard([name, dashboard_id, ...])</code>	Delete a dashboard.
<code>delete_data_source([name, data_source_id, ...])</code>	Delete a data source.
<code>delete_dataset([name, dataset_id, ...])</code>	Delete a dataset.
<code>delete_template([name, template_id, ...])</code>	Delete a template.
<code>describe_dashboard([name, dashboard_id, ...])</code>	Describe a QuickSight dashboard by name or ID.
<code>describe_data_source([name, data_source_id, ...])</code>	Describe a QuickSight data source by name or ID.
<code>describe_data_source_permissions([name, ...])</code>	Describe a QuickSight data source permissions by name or ID.
<code>describe_dataset([name, dataset_id, ...])</code>	Describe a QuickSight dataset by name or ID.
<code>describe_ingestion(ingestion_id[, ...])</code>	Describe a QuickSight ingestion by ID.
<code>get_dashboard_id(name[, account_id, ...])</code>	Get QuickSight dashboard ID given a name and fails if there is more than 1 ID associated with this name.
<code>get_dashboard_ids(name[, account_id, ...])</code>	Get QuickSight dashboard IDs given a name.
<code>get_data_source_arn(name[, account_id, ...])</code>	Get QuickSight data source ARN given a name and fails if there is more than 1 ARN associated with this name.
<code>get_data_source_arns(name[, account_id, ...])</code>	Get QuickSight Data source ARNs given a name.
<code>get_data_source_id(name[, account_id, ...])</code>	Get QuickSight data source ID given a name and fails if there is more than 1 ID associated with this name.

continues on next page

Table 3 – continued from previous page

<code>get_data_source_ids(name[, account_id, ...])</code>	Get QuickSight data source IDs given a name.
<code>get_dataset_id(name[, account_id, boto3_session])</code>	Get QuickSight Dataset ID given a name and fails if there is more than 1 ID associated with this name.
<code>get_dataset_ids(name[, account_id, ...])</code>	Get QuickSight dataset IDs given a name.
<code>get_template_id(name[, account_id, ...])</code>	Get QuickSight template ID given a name and fails if there is more than 1 ID associated with this name.
<code>get_template_ids(name[, account_id, ...])</code>	Get QuickSight template IDs given a name.
<code>list_dashboards([account_id, boto3_session])</code>	List dashboards in an AWS account.
<code>list_data_sources([account_id, boto3_session])</code>	List all QuickSight Data sources summaries.
<code>list_datasets([account_id, boto3_session])</code>	List all QuickSight datasets summaries.
<code>list_groups([namespace, account_id, ...])</code>	List all QuickSight Groups.
<code>list_group_memberships(group_name[, ...])</code>	List all QuickSight Group memberships.
<code>list_iam_policy_assignments([status, ...])</code>	List IAM policy assignments in the current Amazon QuickSight account.
<code>list_iam_policy_assignments_for_user(user_name[, ...])</code>	List all the IAM policy assignments.
<code>list_ingestions([dataset_name, dataset_id, ...])</code>	List the history of SPICE ingestions for a dataset.
<code>list_templates([account_id, boto3_session])</code>	List all QuickSight templates.
<code>list_users([namespace, account_id, ...])</code>	Return a list of all of the Amazon QuickSight users belonging to this account.
<code>list_user_groups(user_name[, namespace, ...])</code>	List the Amazon QuickSight groups that an Amazon QuickSight user is a member of.

awswrangler.quicksight.cancel_ingestion

`awswrangler.quicksight.cancel_ingestion(ingestion_id: str, dataset_name: str | None = None, dataset_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None) → None`

Cancel an ongoing ingestion of data into SPICE.

Note: You must pass a not None value for `dataset_name` or `dataset_id` argument.

Parameters

- **ingestion_id** (*str*) – Ingestion ID.
- **dataset_name** (*str*, *optional*) – Dataset name.
- **dataset_id** (*str*, *optional*) – Dataset ID.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.cancel_ingestion(ingestion_id="...", dataset_name="...")
```

awswrangler.quicksight.create_athena_data_source

```
awswrangler.quicksight.create_athena_data_source(name: str, workgroup: str = 'primary',
                                                  allowed_to_use: List[str] | None = None,
                                                  allowed_to_manage: List[str] | None = None, tags:
                                                  Dict[str, str] | None = None, account_id: str | None
                                                  = None, boto3_session: Session | None = None,
                                                  namespace: str = 'default') → None
```

Create a QuickSight data source pointing to an Athena/Workgroup.

Note: You will not be able to see the the data source in the console if you not pass your user to one of the `allowed_*` arguments.

Parameters

- **name** (*str*) – Data source name.
- **workgroup** (*str*) – Athena workgroup.
- **tags** (*Dict[str, str]*, *optional*) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"}
- **allowed_to_use** (*optional*) – List of principals that will be allowed to see and use the data source. e.g. ["John"]
- **allowed_to_manage** (*optional*) – List of principals that will be allowed to see, use, update and delete the data source. e.g. ["Mary"]
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **namespace** (*str*) – The namespace. Currently, you should set this to default.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.create_athena_data_source(
...     name="...",
...     allowed_to_manage=["john"]
... )
```

awswrangler.quicksight.create_athena_dataset

```
awswrangler.quicksight.create_athena_dataset(name: str, database: str | None = None, table: str | None
= None, sql: str | None = None, sql_name: str =
'CustomSQL', data_source_name: str | None = None,
data_source_arn: str | None = None, import_mode: str =
'DIRECT_QUERY', allowed_to_use: List[str] | None =
None, allowed_to_manage: List[str] | None = None,
logical_table_alias: str = 'LogicalTable',
rename_columns: Dict[str, str] | None = None,
cast_columns_types: Dict[str, str] | None = None,
tag_columns: Dict[str, List[Dict[str, Any]]] | None =
None, tags: Dict[str, str] | None = None, account_id: str |
None = None, boto3_session: Session | None = None,
namespace: str = 'default') → str
```

Create a QuickSight dataset.

Note: You will not be able to see the the dataset in the console if you not pass your username to one of the `allowed_*` arguments.

Note: You must pass database/table OR sql argument.

Note: You must pass `data_source_name` OR `data_source_arn` argument.

Parameters

- **name** (*str*) – Dataset name.
- **database** (*str*) – Athena's database name.
- **table** (*str*) – Athena's table name.
- **sql** (*str*) – Use a SQL query to define your table.
- **sql_name** (*str*) – Query name.
- **data_source_name** (*str*, *optional*) – QuickSight data source name.
- **data_source_arn** (*str*, *optional*) – QuickSight data source ARN.
- **import_mode** (*str*) – Indicates whether you want to import the data into SPICE. 'SPICE'|'DIRECT_QUERY'

- **tags** (*Dict[str, str]*, *optional*) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"}
- **allowed_to_use** (*optional*) – List of usernames that will be allowed to see and use the data source. e.g. ["john", "Mary"]
- **allowed_to_manage** (*optional*) – List of usernames that will be allowed to see, use, update and delete the data source. e.g. ["Mary"]
- **logical_table_alias** (*str*) – A display name for the logical table.
- **rename_columns** (*Dict[str, str]*, *optional*) – Dictionary to map column renames. e.g. {"old_name": "new_name", "old_name2": "new_name2"}
- **cast_columns_types** (*Dict[str, str]*, *optional*) – Dictionary to map column casts. e.g. {"col_name": "STRING", "col_name2": "DECIMAL"} Valid types: 'STRING'|'INTEGER'|'DECIMAL'|'DATETIME'
- **tag_columns** (*Dict[str, List[Dict[str, Any]]]*, *optional*) – Dictionary to map column tags. e.g. {"col_name": [{ "ColumnGeographicRole": "CITY" }],
"col_name2": [{ "ColumnDescription": { "Text": "description" }]}]
Valid geospatial roles: 'COUNTRY'|'STATE'|'COUNTY'|'CITY'|'POSTCODE'|'LONGITUDE'|'LATITUDE'
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **namespace** (*str*) – The namespace. Currently, you should set this to default.

Returns

Dataset ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> dataset_id = wr.quicksight.create_athena_dataset(
...     name="...",
...     database="..."
...     table="..."
...     data_source_name="..."
...     allowed_to_manage=["Mary"]
... )
```

awswrangler.quicksight.create_ingestion

`awswrangler.quicksight.create_ingestion(dataset_name: str | None = None, dataset_id: str | None = None, ingestion_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None) → str`

Create and starts a new SPICE ingestion on a dataset.

Note: You must pass `dataset_name` OR `dataset_id` argument.

Parameters

- **dataset_name** (*str, optional*) – Dataset name.
- **dataset_id** (*str, optional*) – Dataset ID.
- **ingestion_id** (*str, optional*) – Ingestion ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

Ingestion ID

Return type

str

Examples

```
>>> import awswrangler as wr
>>> status = wr.quicksight.create_ingestion("my_dataset")
```

awswrangler.quicksight.delete_all_dashboards

`awswrangler.quicksight.delete_all_dashboards(account_id: str | None = None, regex_filter: str | None = None, boto3_session: Session | None = None) → None`

Delete all dashboards.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **regex_filter** (*str, optional*) – Regex `regex_filter` that will delete all dashboards with a match in their Name
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_dashboards()
```

awswrangler.quicksight.delete_all_data_sources

`awswrangler.quicksight.delete_all_data_sources(account_id: str | None = None, regex_filter: str | None = None, boto3_session: Session | None = None) → None`

Delete all data sources.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **regex_filter** (*str, optional*) – Regex `regex_filter` that will delete all data sources with a match in their Name
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_data_sources()
```

awswrangler.quicksight.delete_all_datasets

`awswrangler.quicksight.delete_all_datasets(account_id: str | None = None, regex_filter: str | None = None, boto3_session: Session | None = None) → None`

Delete all datasets.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **regex_filter** (*str, optional*) – Regex `regex_filter` that will delete all datasets with a match in their Name
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_datasets()
```

awswrangler.quicksight.delete_all_templates

`awswrangler.quicksight.delete_all_templates(account_id: str | None = None, regex_filter: str | None = None, boto3_session: Session | None = None) → None`

Delete all templates.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **regex_filter** (*str, optional*) – Regex `regex_filter` that will delete all templates with a match in their Name
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_all_templates()
```

awswrangler.quicksight.delete_dashboard

`awswrangler.quicksight.delete_dashboard(name: str | None = None, dashboard_id: str | None = None, version_number: int | None = None, account_id: str | None = None, boto3_session: Session | None = None) → None`

Delete a dashboard.

Note: You must pass a not None name or `dashboard_id` argument.

Parameters

- **name** (*str, optional*) – Dashboard name.
- **dashboard_id** (*str, optional*) – The ID for the dashboard.
- **version_number** (*int, optional*) – The version number of the dashboard. If the version number property is provided, only the specified version of the dashboard is deleted.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_dashboard(name="...")
```

awswrangler.quicksight.delete_data_source

awswrangler.quicksight.delete_data_source(*name: str | None = None, data_source_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None*) → None

Delete a data source.

Note: You must pass a not None name or data_source_id argument.

Parameters

- **name** (*str*, *optional*) – Dashboard name.
- **data_source_id** (*str*, *optional*) – The ID for the data source.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_data_source(name="...")
```

awswrangler.quicksight.delete_dataset

awswrangler.quicksight.delete_dataset(*name: str | None = None, dataset_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None*) → None

Delete a dataset.

Note: You must pass a not None **name** or **dataset_id** argument.

Parameters

- **name** (*str, optional*) – Dashboard name.
- **dataset_id** (*str, optional*) – The ID for the dataset.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if **boto3_session** receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_dataset(name="...")
```

awswrangler.quicksight.delete_template

awswrangler.quicksight.delete_template(*name: str | None = None, template_id: str | None = None, version_number: int | None = None, account_id: str | None = None, boto3_session: Session | None = None*) → None

Delete a template.

Note: You must pass a not None **name** or **template_id** argument.

Parameters

- **name** (*str, optional*) – Dashboard name.
- **template_id** (*str, optional*) – The ID for the dashboard.
- **version_number** (*int, optional*) – Specifies the version of the template that you want to delete. If you don't provide a version number, it deletes all versions of the template.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.quicksight.delete_template(name="...")
```

awswrangler.quicksight.describe_dashboard

`awswrangler.quicksight.describe_dashboard`(*name: str | None = None, dashboard_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None*) → Dict[str, Any]

Describe a QuickSight dashboard by name or ID.

Note: You must pass a not None name or dashboard_id argument.

Parameters

- **name** (*str, optional*) – Dashboard name.
- **dashboard_id** (*str, optional*) – Dashboard ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Dashboard Description.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dashboard(name="my-dashboard")
```

awswrangler.quicksight.describe_data_source

`awswrangler.quicksight.describe_data_source(name: str | None = None, data_source_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None) → Dict[str, Any]`

Describe a QuickSight data source by name or ID.

Note: You must pass a not None name or data_source_id argument.

Parameters

- **name** (*str, optional*) – Data source name.
- **data_source_id** (*str, optional*) – Data source ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Data source Description.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_data_source("...")
```

awswrangler.quicksight.describe_data_source_permissions

`awswrangler.quicksight.describe_data_source_permissions(name: str | None = None, data_source_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None) → Dict[str, Any]`

Describe a QuickSight data source permissions by name or ID.

Note: You must pass a not None name or data_source_id argument.

Parameters

- **name** (*str, optional*) – Data source name.
- **data_source_id** (*str, optional*) – Data source ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Data source Permissions Description.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_data_source_permissions("my-data-source")
```

awswrangler.quicksight.describe_dataset

`awswrangler.quicksight.describe_dataset`(*name: str | None = None, dataset_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None*) → Dict[str, Any]

Describe a QuickSight dataset by name or ID.

Note: You must pass a not None name or dataset_id argument.

Parameters

- **name** (*str, optional*) – Dataset name.
- **dataset_id** (*str, optional*) – Dataset ID.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Dataset Description.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dataset("my-dataset")
```

awswrangler.quicksight.describe_ingestion

`awswrangler.quicksight.describe_ingestion(ingestion_id: str, dataset_name: str | None = None, dataset_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None) → Dict[str, Any]`

Describe a QuickSight ingestion by ID.

Note: You must pass a not None value for `dataset_name` or `dataset_id` argument.

Parameters

- **ingestion_id** (*str*) – Ingestion ID.
- **dataset_name** (*str*, *optional*) – Dataset name.
- **dataset_id** (*str*, *optional*) – Dataset ID.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

Ingestion Description.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> description = wr.quicksight.describe_dataset(ingestion_id="...", dataset_name="..."
↪ "...")
```

awswrangler.quicksight.get_dashboard_id

`awswrangler.quicksight.get_dashboard_id(name: str, account_id: str | None = None, boto3_session: Session | None = None) → str`

Get QuickSight dashboard ID given a name and fails if there is more than 1 ID associated with this name.

Parameters

- **name** (*str*) – Dashboard name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if `boto3_session` receive None.

Returns

Dashboad ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_dashboard_id(name="...")
```

awswrangler.quicksight.get_dashboard_ids

`awswrangler.quicksight.get_dashboard_ids(name: str, account_id: str | None = None, boto3_session: Session | None = None) → List[str]`

Get QuickSight dashboard IDs given a name.

Note: This function returns a list of ID because QuickSight accepts duplicated dashboard names, so you may have more than 1 ID for a given name.

Parameters

- **name** (*str*) – Dashboard name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Dashboard IDs.

Return type

List[str]

Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_dashboard_ids(name="...")
```

awswrangler.quicksight.get_data_source_arn

`awswrangler.quicksight.get_data_source_arn(name: str, account_id: str | None = None, boto3_session: Session | None = None) → str`

Get QuickSight data source ARN given a name and fails if there is more than 1 ARN associated with this name.

Note: This function returns a list of ARNs because QuickSight accepts duplicated data source names, so you may have more than 1 ARN for a given name.

Parameters

- **name** (*str*) – Data source name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Data source ARN.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> arn = wr.quicksight.get_data_source_arn("...")
```

awswrangler.quicksight.get_data_source_arns

awswrangler.quicksight.get_data_source_arns(*name: str, account_id: str | None = None, boto3_session: Session | None = None*) → List[str]

Get QuickSight Data source ARNs given a name.

Note: This function returns a list of ARNs because Quicksight accepts duplicated data source names, so you may have more than 1 ARN for a given name.

Parameters

- **name** (*str*) – Data source name.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Data source ARNs.

Return type

List[str]

Examples

```
>>> import awswrangler as wr
>>> arns = wr.quicksight.get_data_source_arns(name="...")
```

awswrangler.quicksight.get_data_source_id

`awswrangler.quicksight.get_data_source_id(name: str, account_id: str | None = None, boto3_session: Session | None = None) → str`

Get QuickSight data source ID given a name and fails if there is more than 1 ID associated with this name.

Parameters

- **name** (*str*) – Data source name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Dataset ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_data_source_id(name="...")
```

awswrangler.quicksight.get_data_source_ids

`awswrangler.quicksight.get_data_source_ids(name: str, account_id: str | None = None, boto3_session: Session | None = None) → List[str]`

Get QuickSight data source IDs given a name.

Note: This function returns a list of ID because Quicksight accepts duplicated data source names, so you may have more than 1 ID for a given name.

Parameters

- **name** (*str*) – Data source name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Data source IDs.

Return type

List[str]

Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_data_source_ids(name="...")
```

awswrangler.quicksight.get_dataset_id

`awswrangler.quicksight.get_dataset_id(name: str, account_id: str | None = None, boto3_session: Session | None = None) → str`

Get QuickSight Dataset ID given a name and fails if there is more than 1 ID associated with this name.

Parameters

- **name** (*str*) – Dataset name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Dataset ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_dataset_id(name="...")
```

awswrangler.quicksight.get_dataset_ids

`awswrangler.quicksight.get_dataset_ids(name: str, account_id: str | None = None, boto3_session: Session | None = None) → List[str]`

Get QuickSight dataset IDs given a name.

Note: This function returns a list of ID because Quicksight accepts duplicated datasets names, so you may have more than 1 ID for a given name.

Parameters

- **name** (*str*) – Dataset name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Datasets IDs.

Return type

List[str]

Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_dataset_ids(name="...")
```

awswrangler.quicksight.get_template_id

`awswrangler.quicksight.get_template_id(name: str, account_id: str | None = None, boto3_session: Session | None = None) → str`

Get QuickSight template ID given a name and fails if there is more than 1 ID associated with this name.

Parameters

- **name** (*str*) – Template name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Template ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> my_id = wr.quicksight.get_template_id(name="...")
```

awswrangler.quicksight.get_template_ids

`awswrangler.quicksight.get_template_ids(name: str, account_id: str | None = None, boto3_session: Session | None = None) → List[str]`

Get QuickSight template IDs given a name.

Note: This function returns a list of ID because Quicksight accepts duplicated templates names, so you may have more than 1 ID for a given name.

Parameters

- **name** (*str*) – Template name.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.

- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Template IDs.

Return type

List[str]

Examples

```
>>> import awswrangler as wr
>>> ids = wr.quicksight.get_template_ids(name="...")
```

awswrangler.quicksight.list_dashboards

awswrangler.quicksight.list_dashboards(*account_id: str | None = None*, *boto3_session: Session | None = None*) → List[Dict[str, Any]]

List dashboards in an AWS account.

Parameters

- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Dashboards.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> dashboards = wr.quicksight.list_dashboards()
```

awswrangler.quicksight.list_data_sources

awswrangler.quicksight.list_data_sources(*account_id: str | None = None*, *boto3_session: Session | None = None*) → List[Dict[str, Any]]

List all QuickSight Data sources summaries.

Parameters

- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Data sources summaries.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> sources = wr.quicksight.list_data_sources()
```

awswrangler.quicksight.list_datasets

awswrangler.quicksight.list_datasets(*account_id: str | None = None, boto3_session: Session | None = None*) → List[Dict[str, Any]]

List all QuickSight datasets summaries.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Datasets summaries.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> datasets = wr.quicksight.list_datasets()
```

awswrangler.quicksight.list_groups

awswrangler.quicksight.list_groups(*namespace: str = 'default', account_id: str | None = None, boto3_session: Session | None = None*) → List[Dict[str, Any]]

List all QuickSight Groups.

Parameters

- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Groups.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> groups = wr.quicksight.list_groups()
```

awswrangler.quicksight.list_group_memberships

```
awswrangler.quicksight.list_group_memberships(group_name: str, namespace: str = 'default',
                                              account_id: str | None = None, boto3_session: Session |
                                              None = None) → List[Dict[str, Any]]
```

List all QuickSight Group memberships.

Parameters

- **group_name** (*str*) – The name of the group that you want to see a membership list of.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Group memberships.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> memberships = wr.quicksight.list_group_memberships()
```

awswrangler.quicksight.list_iam_policy_assignments

```
awswrangler.quicksight.list_iam_policy_assignments(status: str | None = None, namespace: str =
                                                    'default', account_id: str | None = None,
                                                    boto3_session: Session | None = None) →
List[Dict[str, Any]]
```

List IAM policy assignments in the current Amazon QuickSight account.

Parameters

- **status** (*str, optional*) – The status of the assignments. 'ENABLED'|'DRAFT'|'DISABLED'
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

IAM policy assignments.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> assigns = wr.quicksight.list_iam_policy_assignments()
```

awswrangler.quicksight.list_iam_policy_assignments_for_user

```
awswrangler.quicksight.list_iam_policy_assignments_for_user(user_name: str, namespace: str =
                                                             'default', account_id: str | None =
                                                             None, boto3_session: Session | None
                                                             = None) → List[Dict[str, Any]]
```

List all the IAM policy assignments.

Including the Amazon Resource Names (ARNs) for the IAM policies assigned to the specified user and group or groups that the user belongs to.

Parameters

- **user_name** (*str*) – The name of the user.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

IAM policy assignments.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> assigns = wr.quicksight.list_iam_policy_assignments_for_user()
```

awswrangler.quicksight.list_ingestions

`awswrangler.quicksight.list_ingestions(dataset_name: str | None = None, dataset_id: str | None = None, account_id: str | None = None, boto3_session: Session | None = None) → List[Dict[str, Any]]`

List the history of SPICE ingestions for a dataset.

Parameters

- **dataset_name** (*str, optional*) – Dataset name.
- **dataset_id** (*str, optional*) – The ID of the dataset used in the ingestion.
- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

IAM policy assignments.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> ingestions = wr.quicksight.list_ingestions()
```

awswrangler.quicksight.list_templates

`awswrangler.quicksight.list_templates(account_id: str | None = None, boto3_session: Session | None = None) → List[Dict[str, Any]]`

List all QuickSight templates.

Parameters

- **account_id** (*str, optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session(), optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Templates summaries.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> templates = wr.quicksight.list_templates()
```

awswrangler.quicksight.list_users

`awswrangler.quicksight.list_users(namespace: str = 'default', account_id: str | None = None, boto3_session: Session | None = None) → List[Dict[str, Any]]`

Return a list of all of the Amazon QuickSight users belonging to this account.

Parameters

- **namespace** (*str*) – The namespace. Currently, you should set this to default.
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Groups.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> users = wr.quicksight.list_users()
```

awswrangler.quicksight.list_user_groups

`awswrangler.quicksight.list_user_groups(user_name: str, namespace: str = 'default', account_id: str | None = None, boto3_session: Session | None = None) → List[Dict[str, Any]]`

List the Amazon QuickSight groups that an Amazon QuickSight user is a member of.

Parameters

- **user_name** (*str*;) – The Amazon QuickSight user name that you want to list group memberships for.
- **namespace** (*str*) – The namespace. Currently, you should set this to default .
- **account_id** (*str*, *optional*) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (*boto3.Session()*, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Groups.

Return type

List[Dict[str, Any]]

Examples

```
>>> import awswrangler as wr
>>> groups = wr.quicksight.list_user_groups()
```

1.4.18 AWS STS

<code>get_account_id([boto3_session])</code>	Get Account ID.
<code>get_current_identity_arn([boto3_session])</code>	Get current user/role ARN.
<code>get_current_identity_name([boto3_session])</code>	Get current user/role name.

awswrangler.sts.get_account_id`awswrangler.sts.get_account_id(boto3_session: Session | None = None) → str`

Get Account ID.

Parameters

boto3_session (`boto3.Session()`, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Account ID.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> account_id = wr.sts.get_account_id()
```

awswrangler.sts.get_current_identity_arn`awswrangler.sts.get_current_identity_arn(boto3_session: Session | None = None) → str`

Get current user/role ARN.

Parameters

boto3_session (`boto3.Session()`, *optional*) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

User/role ARN.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> arn = wr.sts.get_current_identity_arn()
```

awswrangler.sts.get_current_identity_name

awswrangler.sts.get_current_identity_name(boto3_session: Session | None = None) → str

Get current user/role name.

Parameters

boto3_session (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

User/role name.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> name = wr.sts.get_current_identity_name()
```

1.4.19 AWS Secrets Manager

<code>get_secret(name[, boto3_session])</code>	Get secret value.
<code>get_secret_json(name[, boto3_session])</code>	Get JSON secret value.

awswrangler.secretsmanager.get_secret

awswrangler.secretsmanager.get_secret(name: str, boto3_session: Session | None = None) → str | bytes

Get secret value.

Parameters

- **name** (str:) – Specifies the secret containing the version that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **boto3_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Secret value.

Return type

Union[str, bytes]

Examples

```
>>> import awswrangler as wr
>>> value = wr.secretsmanager.get_secret("my-secret")
```

awswrangler.secretsmanager.get_secret_json

awswrangler.secretsmanager.get_secret_json(name: str, boto3_session: Session | None = None) → Dict[str, Any]

Get JSON secret value.

Parameters

- **name** (str) – Specifies the secret containing the version that you want to retrieve. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.
- **boto3_session** (boto3.Session(), optional) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Returns

Secret JSON value parsed as a dictionary.

Return type

Dict[str, Any]

Examples

```
>>> import awswrangler as wr
>>> value = wr.secretsmanager.get_secret_json("my-secret-with-json-content")
```

1.4.20 Amazon Chime

`post_message(webhook, message)`

Send message on an existing Chime Chat rooms.

awswrangler.chime.post_message

awswrangler.chime.post_message(webhook: str, message: str) → Any | None

Send message on an existing Chime Chat rooms.

:param : Webhook: This contains all the authentication information to send the message :type : param webhook
: param : The actual message which needs to be posted on Slack channel :type : param message :
message

Returns

Represents the response from Chime

Return type

dict

1.4.21 Global Configurations

<code>reset([item])</code>	Reset one or all (if None is received) configuration values.
<code>to_pandas()</code>	Load all configurations on a Pandas DataFrame.

awswrangler.config.reset

`config.reset(item: str | None = None) → None`

Reset one or all (if None is received) configuration values.

Parameters

item (*str, optional*) – Configuration item name.

Returns

None.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> wr.config.reset("database") # Reset one specific configuration
>>> wr.config.reset() # Reset all
```

awswrangler.config.to_pandas

`config.to_pandas() → DataFrame`

Load all configurations on a Pandas DataFrame.

Returns

Configuration DataFrame.

Return type

pd.DataFrame

Examples

```
>>> import awswrangler as wr
>>> wr.config.to_pandas()
```


Symbols

`__init__()` (*aws wrangler.data_api.rds.RdsDataApi* method), 287

`__init__()` (*aws wrangler.data_api.redshift.RedshiftDataApi* method), 286

A

`add_column()` (*in module aws wrangler.catalog*), 185

`add_csv_partitions()` (*in module aws wrangler.catalog*), 186

`add_parquet_partitions()` (*in module aws wrangler.catalog*), 187

B

`build_spark_step()` (*in module aws wrangler.emr*), 323

`build_step()` (*in module aws wrangler.emr*), 324

C

`cancel_ingestion()` (*in module aws wrangler.quicksight*), 340

`cancel_transaction()` (*in module aws wrangler.lakeformation*), 247

`commit_transaction()` (*in module aws wrangler.lakeformation*), 248

`connect()` (*in module aws wrangler.data_api.rds*), 288

`connect()` (*in module aws wrangler.data_api.redshift*), 286

`connect()` (*in module aws wrangler.mysql*), 271

`connect()` (*in module aws wrangler.neptune*), 304

`connect()` (*in module aws wrangler.opensearch*), 294

`connect()` (*in module aws wrangler.oracle*), 281

`connect()` (*in module aws wrangler.postgresql*), 266

`connect()` (*in module aws wrangler.redshift*), 251

`connect()` (*in module aws wrangler.sqlserver*), 276

`connect_temp()` (*in module aws wrangler.redshift*), 252

`copy()` (*in module aws wrangler.redshift*), 253

`copy_from_files()` (*in module aws wrangler.redshift*), 256

`copy_objects()` (*in module aws wrangler.s3*), 134

`create_athena_bucket()` (*in module aws wrangler.athena*), 222

`create_athena_data_source()` (*in module aws wrangler.quicksight*), 341

`create_athena_dataset()` (*in module aws wrangler.quicksight*), 342

`create_cluster()` (*in module aws wrangler.emr*), 325

`create_collection()` (*in module aws wrangler.opensearch*), 295

`create_csv_table()` (*in module aws wrangler.catalog*), 188

`create_ctas_table()` (*in module aws wrangler.athena*), 222

`create_database()` (*in module aws wrangler.catalog*), 191

`create_database()` (*in module aws wrangler.timestream*), 318

`create_index()` (*in module aws wrangler.opensearch*), 296

`create_ingestion()` (*in module aws wrangler.quicksight*), 344

`create_json_table()` (*in module aws wrangler.catalog*), 192

`create_parquet_table()` (*in module aws wrangler.catalog*), 195

`create_recommendation_ruleset()` (*in module aws wrangler.data_quality*), 289

`create_ruleset()` (*in module aws wrangler.data_quality*), 290

`create_table()` (*in module aws wrangler.timestream*), 318

D

`databases()` (*in module aws wrangler.catalog*), 197

`delete_all_dashboards()` (*in module aws wrangler.quicksight*), 344

`delete_all_data_sources()` (*in module aws wrangler.quicksight*), 345

`delete_all_datasets()` (*in module aws wrangler.quicksight*), 345

`delete_all_partitions()` (*in module aws wrangler.catalog*), 200

`delete_all_templates()` (in module `aws wrangler.quicksight`), 346
`delete_column()` (in module `aws wrangler.catalog`), 198
`delete_dashboard()` (in module `aws wrangler.quicksight`), 346
`delete_data_source()` (in module `aws wrangler.quicksight`), 347
`delete_database()` (in module `aws wrangler.catalog`), 199
`delete_database()` (in module `aws wrangler.timestream`), 319
`delete_dataset()` (in module `aws wrangler.quicksight`), 348
`delete_index()` (in module `aws wrangler.opensearch`), 297
`delete_items()` (in module `aws wrangler.dynamodb`), 308
`delete_objects()` (in module `aws wrangler.s3`), 135
`delete_partitions()` (in module `aws wrangler.catalog`), 199
`delete_table()` (in module `aws wrangler.timestream`), 320
`delete_table_if_exists()` (in module `aws wrangler.catalog`), 201
`delete_template()` (in module `aws wrangler.quicksight`), 348
`describe_dashboard()` (in module `aws wrangler.quicksight`), 349
`describe_data_source()` (in module `aws wrangler.quicksight`), 350
`describe_data_source_permissions()` (in module `aws wrangler.quicksight`), 350
`describe_dataset()` (in module `aws wrangler.quicksight`), 351
`describe_ingestion()` (in module `aws wrangler.quicksight`), 352
`describe_log_streams()` (in module `aws wrangler.cloudwatch`), 337
`describe_objects()` (in module `aws wrangler.s3`), 136
`describe_transaction()` (in module `aws wrangler.lakeformation`), 248
`does_object_exist()` (in module `aws wrangler.s3`), 137
`does_table_exist()` (in module `aws wrangler.catalog`), 202
`download()` (in module `aws wrangler.s3`), 138
`drop_duplicated_columns()` (in module `aws wrangler.catalog`), 203

E

`evaluate_ruleset()` (in module `aws wrangler.data_quality`), 291

`execute_gremlin()` (in module `aws wrangler.neptune`), 304
`execute_opencypher()` (in module `aws wrangler.neptune`), 305
`execute_sparql()` (in module `aws wrangler.neptune`), 305
`execute_statement()` (in module `aws wrangler.dynamodb`), 309
`extend_transaction()` (in module `aws wrangler.lakeformation`), 249
`extract_athena_types()` (in module `aws wrangler.catalog`), 203

F

`filter_log_events()` (in module `aws wrangler.cloudwatch`), 338
`flatten_nested_df()` (in module `aws wrangler.neptune`), 306

G

`generate_create_query()` (in module `aws wrangler.athena`), 224
`get_account_id()` (in module `aws wrangler.sts`), 364
`get_bucket_region()` (in module `aws wrangler.s3`), 139
`get_cluster_state()` (in module `aws wrangler.emr`), 330
`get_columns_comments()` (in module `aws wrangler.catalog`), 204
`get_csv_partitions()` (in module `aws wrangler.catalog`), 205
`get_current_identity_arn()` (in module `aws wrangler.sts`), 364
`get_current_identity_name()` (in module `aws wrangler.sts`), 365
`get_dashboard_id()` (in module `aws wrangler.quicksight`), 352
`get_dashboard_ids()` (in module `aws wrangler.quicksight`), 353
`get_data_source_arn()` (in module `aws wrangler.quicksight`), 353
`get_data_source_arns()` (in module `aws wrangler.quicksight`), 354
`get_data_source_id()` (in module `aws wrangler.quicksight`), 355
`get_data_source_ids()` (in module `aws wrangler.quicksight`), 355
`get_databases()` (in module `aws wrangler.catalog`), 206
`get_dataset_id()` (in module `aws wrangler.quicksight`), 356
`get_dataset_ids()` (in module `aws wrangler.quicksight`), 356

- `get_named_query_statement()` (in module `aws wrangler.athena`), 228
- `get_parquet_partitions()` (in module `aws wrangler.catalog`), 207
- `get_partitions()` (in module `aws wrangler.catalog`), 208
- `get_query_columns_types()` (in module `aws wrangler.athena`), 225
- `get_query_execution()` (in module `aws wrangler.athena`), 226
- `get_query_executions()` (in module `aws wrangler.athena`), 226
- `get_query_results()` (in module `aws wrangler.athena`), 227
- `get_ruleset()` (in module `aws wrangler.data_quality`), 293
- `get_secret()` (in module `aws wrangler.secretsmanager`), 365
- `get_secret_json()` (in module `aws wrangler.secretsmanager`), 366
- `get_step_state()` (in module `aws wrangler.emr`), 330
- `get_table()` (in module `aws wrangler.dynamodb`), 310
- `get_table_description()` (in module `aws wrangler.catalog`), 209
- `get_table_location()` (in module `aws wrangler.catalog`), 210
- `get_table_number_of_versions()` (in module `aws wrangler.catalog`), 211
- `get_table_parameters()` (in module `aws wrangler.catalog`), 212
- `get_table_types()` (in module `aws wrangler.catalog`), 212
- `get_table_versions()` (in module `aws wrangler.catalog`), 213
- `get_tables()` (in module `aws wrangler.catalog`), 214
- `get_template_id()` (in module `aws wrangler.quicksight`), 357
- `get_template_ids()` (in module `aws wrangler.quicksight`), 357
- `get_work_group()` (in module `aws wrangler.athena`), 228
- I**
- `index_csv()` (in module `aws wrangler.opensearch`), 298
- `index_df()` (in module `aws wrangler.opensearch`), 300
- `index_documents()` (in module `aws wrangler.opensearch`), 299
- `index_json()` (in module `aws wrangler.opensearch`), 301
- L**
- `list_buckets()` (in module `aws wrangler.s3`), 139
- `list_dashboards()` (in module `aws wrangler.quicksight`), 358
- `list_data_sources()` (in module `aws wrangler.quicksight`), 358
- `list_datasets()` (in module `aws wrangler.quicksight`), 359
- `list_directories()` (in module `aws wrangler.s3`), 140
- `list_group_memberships()` (in module `aws wrangler.quicksight`), 360
- `list_groups()` (in module `aws wrangler.quicksight`), 359
- `list_iam_policy_assignments()` (in module `aws wrangler.quicksight`), 360
- `list_iam_policy_assignments_for_user()` (in module `aws wrangler.quicksight`), 361
- `list_ingestions()` (in module `aws wrangler.quicksight`), 362
- `list_objects()` (in module `aws wrangler.s3`), 141
- `list_query_executions()` (in module `aws wrangler.athena`), 229
- `list_templates()` (in module `aws wrangler.quicksight`), 362
- `list_user_groups()` (in module `aws wrangler.quicksight`), 363
- `list_users()` (in module `aws wrangler.quicksight`), 363
- M**
- `merge_datasets()` (in module `aws wrangler.s3`), 142
- `merge_upsert_table()` (in module `aws wrangler.s3`), 143
- O**
- `overwrite_table_parameters()` (in module `aws wrangler.catalog`), 215
- P**
- `post_message()` (in module `aws wrangler.chime`), 366
- `put_csv()` (in module `aws wrangler.dynamodb`), 310
- `put_df()` (in module `aws wrangler.dynamodb`), 311
- `put_items()` (in module `aws wrangler.dynamodb`), 312
- `put_json()` (in module `aws wrangler.dynamodb`), 313
- Q**
- `query()` (in module `aws wrangler.timestream`), 320
- R**
- `RdsDataApi` (class in `aws wrangler.data_api.rds`), 287
- `read_csv()` (in module `aws wrangler.s3`), 144
- `read_deltalake()` (in module `aws wrangler.s3`), 146
- `read_excel()` (in module `aws wrangler.s3`), 147
- `read_fwf()` (in module `aws wrangler.s3`), 148
- `read_items()` (in module `aws wrangler.dynamodb`), 314
- `read_json()` (in module `aws wrangler.s3`), 150
- `read_logs()` (in module `aws wrangler.cloudwatch`), 334
- `read_parquet()` (in module `aws wrangler.s3`), 152

- [read_parquet_metadata\(\)](#) (in module `aws wrangler.s3`), 155
[read_parquet_table\(\)](#) (in module `aws wrangler.s3`), 157
[read_partiql_query\(\)](#) (in module `aws wrangler.dynamodb`), 317
[read_sql_query\(\)](#) (in module `aws wrangler.athena`), 229
[read_sql_query\(\)](#) (in module `aws wrangler.data_api.rds`), 288
[read_sql_query\(\)](#) (in module `aws wrangler.data_api.redshift`), 287
[read_sql_query\(\)](#) (in module `aws wrangler.lakeformation`), 244
[read_sql_query\(\)](#) (in module `aws wrangler.mysql`), 272
[read_sql_query\(\)](#) (in module `aws wrangler.oracle`), 282
[read_sql_query\(\)](#) (in module `aws wrangler.postgresql`), 267
[read_sql_query\(\)](#) (in module `aws wrangler.redshift`), 259
[read_sql_query\(\)](#) (in module `aws wrangler.sqlserver`), 277
[read_sql_table\(\)](#) (in module `aws wrangler.athena`), 234
[read_sql_table\(\)](#) (in module `aws wrangler.lakeformation`), 246
[read_sql_table\(\)](#) (in module `aws wrangler.mysql`), 273
[read_sql_table\(\)](#) (in module `aws wrangler.oracle`), 283
[read_sql_table\(\)](#) (in module `aws wrangler.postgresql`), 268
[read_sql_table\(\)](#) (in module `aws wrangler.redshift`), 260
[read_sql_table\(\)](#) (in module `aws wrangler.sqlserver`), 278
[RedshiftDataApi](#) (class in `aws wrangler.data_api.redshift`), 285
[repair_table\(\)](#) (in module `aws wrangler.athena`), 238
[reset\(\)](#) (`aws wrangler.config` method), 367
[run_query\(\)](#) (in module `aws wrangler.cloudwatch`), 335
- ## S
- [sanitize_column_name\(\)](#) (in module `aws wrangler.catalog`), 216
[sanitize_dataframe_columns_names\(\)](#) (in module `aws wrangler.catalog`), 216
[sanitize_table_name\(\)](#) (in module `aws wrangler.catalog`), 217
[search\(\)](#) (in module `aws wrangler.opensearch`), 302
[search_by_sql\(\)](#) (in module `aws wrangler.opensearch`), 303
[search_tables\(\)](#) (in module `aws wrangler.catalog`), 218
- [select_query\(\)](#) (in module `aws wrangler.s3`), 159
[size_objects\(\)](#) (in module `aws wrangler.s3`), 161
[start_query\(\)](#) (in module `aws wrangler.cloudwatch`), 336
[start_query_execution\(\)](#) (in module `aws wrangler.athena`), 239
[start_transaction\(\)](#) (in module `aws wrangler.lakeformation`), 249
[stop_query_execution\(\)](#) (in module `aws wrangler.athena`), 241
[store_parquet_metadata\(\)](#) (in module `aws wrangler.s3`), 162
[submit_ecr_credentials_refresh\(\)](#) (in module `aws wrangler.emr`), 331
[submit_spark_step\(\)](#) (in module `aws wrangler.emr`), 331
[submit_step\(\)](#) (in module `aws wrangler.emr`), 332
[submit_steps\(\)](#) (in module `aws wrangler.emr`), 333
- ## T
- [table\(\)](#) (in module `aws wrangler.catalog`), 218
[tables\(\)](#) (in module `aws wrangler.catalog`), 219
[terminate_cluster\(\)](#) (in module `aws wrangler.emr`), 333
[to_csv\(\)](#) (in module `aws wrangler.s3`), 165
[to_excel\(\)](#) (in module `aws wrangler.s3`), 171
[to_json\(\)](#) (in module `aws wrangler.s3`), 172
[to_pandas\(\)](#) (`aws wrangler.config` method), 367
[to_parquet\(\)](#) (in module `aws wrangler.s3`), 176
[to_property_graph\(\)](#) (in module `aws wrangler.neptune`), 306
[to_rdf_graph\(\)](#) (in module `aws wrangler.neptune`), 307
[to_sql\(\)](#) (in module `aws wrangler.mysql`), 274
[to_sql\(\)](#) (in module `aws wrangler.oracle`), 284
[to_sql\(\)](#) (in module `aws wrangler.postgresql`), 269
[to_sql\(\)](#) (in module `aws wrangler.redshift`), 261
[to_sql\(\)](#) (in module `aws wrangler.sqlserver`), 279
- ## U
- [unload\(\)](#) (in module `aws wrangler.athena`), 242
[unload\(\)](#) (in module `aws wrangler.redshift`), 263
[unload_to_files\(\)](#) (in module `aws wrangler.redshift`), 265
[update_ruleset\(\)](#) (in module `aws wrangler.data_quality`), 293
[upload\(\)](#) (in module `aws wrangler.s3`), 181
[upsert_table_parameters\(\)](#) (in module `aws wrangler.catalog`), 220
- ## W
- [wait_objects_exist\(\)](#) (in module `aws wrangler.s3`), 182
[wait_objects_not_exist\(\)](#) (in module `aws wrangler.s3`), 183

`wait_query()` (in module *awswrangler.athena*), 243
`wait_query()` (in module *awswrangler.cloudwatch*), 336
`wait_query()` (in module *awswrangler.lakeformation*),
250
`write()` (in module *awswrangler.timestream*), 321