AWS SDK for pandas

Release 3.16.1

Amazon Web Services

Jun 16, 2026

CONTENTS

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

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

```
>>> # Optional modules are installed with:
>>> pip install 'awswrangler[redshift]'
```

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

CHAPTER

ONE

READ THE DOCS

1.1 What is AWS SDK for pandas?

An AWS Professional Service open source python initiative that extends the power of the pandas library to AWS, connecting **DataFrames** and AWS data & analytics 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 your usual ETL tasks like load/unloading data from **Data Lakes**, **Data Warehouses** and **Databases**, even at scale.

Check our tutorials or the list of functionalities.

1.2 Install

AWS SDK for pandas runs on Python `3.8`, `3.9`, `3.10`, `3.11`, `3.12`, `3.13`, and `3.14` 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.

1.2.1 PyPI (pip)

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

```
>>> # Optional modules are installed with:
>>> pip install 'awswrangler[redshift]'
```

1.2.2 Conda

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

1.2.3 At scale

AWS SDK for pandas can also run your workflows at scale by leveraging modin and ray.

```
>>> pip install "awswrangler[modin,ray]"
```

As a result existing scripts can run on significantly larger datasets with no code rewrite.

1.2.4 Optional dependencies

Starting version 3.0, some `awswrangler` modules are optional and must be installed explicitly using:

```
>>> pip install 'awswrangler[optional-module1, optional-module2]'
```

The optional modules are:

- redshift
- mysql
- postgres
- sqlserver
- oracle
- gremlin
- sparql
- opencypher
- openpyxl
- opensearch
- deltalake
- pyiceberg

Calling these modules without the required dependencies raises an error prompting you to install the missing package.

1.2.5 AWS Lambda Layer

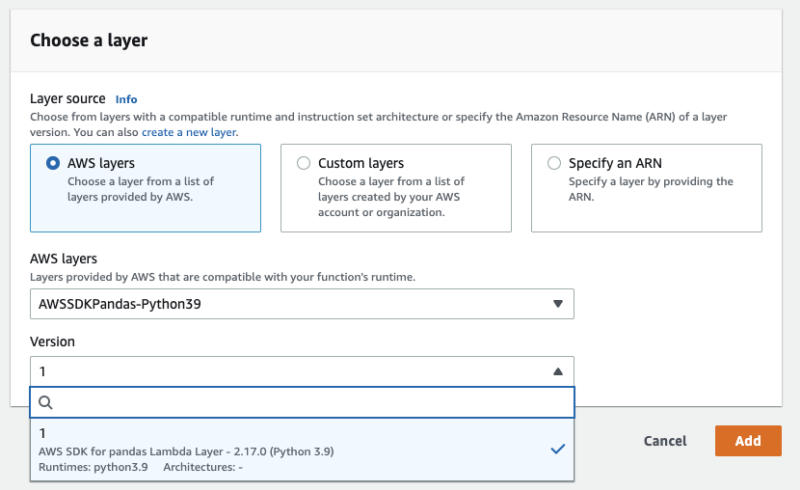
Managed Layer

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

Warning: Lambda Functions using the layer with a memory size of less than 512MB may be insufficient for some workloads.

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:



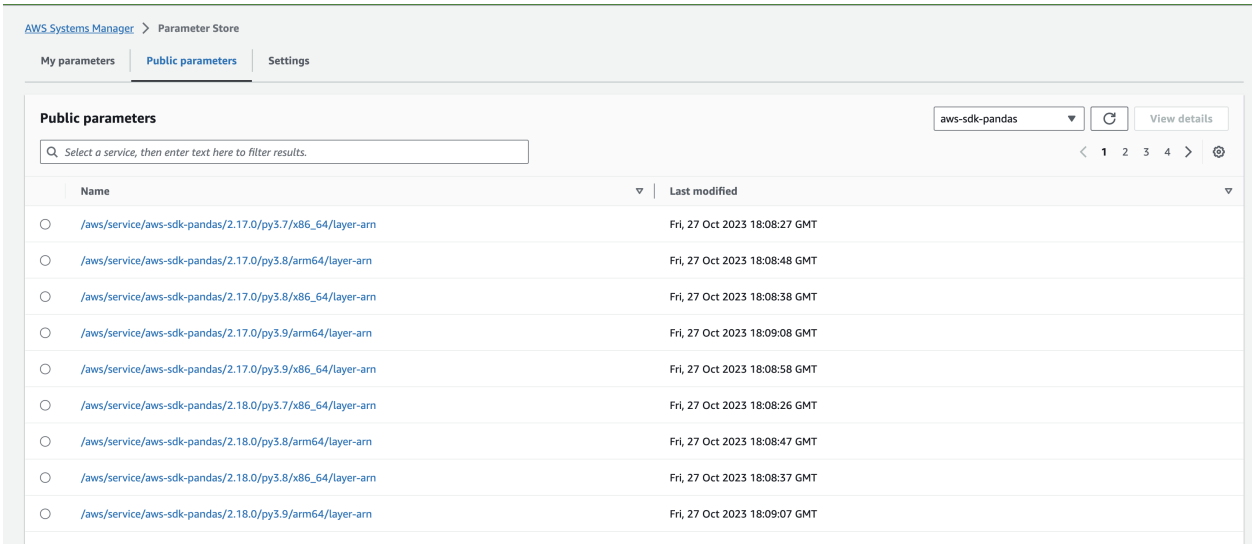
Or via its ARN: `arn:aws:lambda:<region>:336392948345:layer:AWSSDKPandas-Python<python-version>:<layer-version>`.

For example: `arn:aws:lambda:us-east-1:336392948345:layer:AWSSDKPandas-Python38:1`.

The full list of ARNs is available here.

Layer ARNs can also be obtained from SSM public parameters.

Console:



CLI: Find all layers for a version of the library.

```
aws ssm describe-parameters --parameter-filters "Key=Name, Option=BeginsWith, Values=/
↪aws/service/aws-sdk-pandas/3.4.0/"
```

CDK:

```
sdk_for_pandas_layer_arn = ssm.StringParameter.from_string_parameter_attributes(self,
↪"MyValue",
  parameter_name="/aws/service/aws-sdk-pandas/3.4.0/py3.10/x86_64/layer-arn"
).string_value
```

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-py3-8	arn:aws:serverlessrepo:us-east-1:336392948345:applicatio 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:applicatio sdk-pandas-layer-py3-9	Layer for `Python 3.9.x` runtimes
aws-sdk-pandas-layer-py3-10	arn:aws:serverlessrepo:us-east-1:336392948345:applicatio sdk-pandas-layer-py3-10	Layer for `Python 3.10.x` runtimes
aws-sdk-pandas-layer-py3-11	arn:aws:serverlessrepo:us-east-1:336392948345:applicatio sdk-pandas-layer-py3-11	Layer for `Python 3.11.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)
```

(continues on next page)

(continued from previous page)

```
    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="3.0.0",  # Get the latest version from https://serverlessrepo.
↪aws.amazon.com/applications
      ),
    )

    aws_sdk_pandas_layer_arn = aws_sdk_pandas_layer.get_att("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"),
      handler='lambda_function.lambda_handler',
      layers=[aws_sdk_pandas_layer_version]
    )
```

1.2.6 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:

Using pip

1 - In the AWS console, open your Glue Python Shell job's *Job details* tab.

2 - Scroll down and expand the *Advanced properties*.

3 - In the *Job parameters* section, add *–additional-python-modules* as *Key* and *awswrangler* as *Value*.

You can also specify optional dependencies or set a version in the *Value* field, e.g. *awswrangler[redshift]==3.9.0*. For details, see reference below.

Using a Whl file

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 of the *Job details* tab.

Official Glue Python Shell Reference

1.2.7 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: `awswrangler`

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

- Value: `pyarrow==14,pandas==1.5.3,awswrangler==3.16.1`

Official Glue PySpark Reference

1.2.8 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/3.0.0/awswrangler-layer-3.0.0-py3.8.zip`

You can check the bucket to find the latest version.

1.2.9 Amazon SageMaker Notebook

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

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

Platform compatibility notice:

`awswrangler>=3.14.0` defaults to PyArrow 21.0.0+, which requires CMake 3.25+ to build. On AL2-V3 notebook instances (CMake 2.8), you have the following options:

- Pin PyArrow to an older version:

```
>>> !pip install awswrangler "pyarrow<21"
```

- Upgrade CMake version to 3.25+
- Update your notebook instance platform type to AL2023-V1 or later using `UpdateNotebookInstance`

1.2.10 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=awswrangler

# 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
```

1.2.11 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 awswrangler
```

1.2.12 From Source

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

1.2.13 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 OBDC drivers and *pyobdc*.

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.

1.2.14 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.

1.3 At scale

AWS SDK for pandas supports Ray and Modin, enabling you to scale your pandas workflows from a single machine to a multi-node environment, with no code changes. Head to our tutorial to set up a self-managed Ray cluster on Amazon Elastic Compute Cloud (Amazon EC2).

1.3.1 Getting Started

Install the library with the these two optional dependencies to enable distributed mode:

```
>>> pip install "awswrangler[ray,modin]"
```

Once installed, you can use the library in your code as usual:

```
>>> import awswrangler as wr
```

At import, SDK for pandas checks if `ray` and `modin` are in the installation path and enables distributed mode. To confirm that you are in distributed mode, run:

```
>>> print(f"Execution Engine: {wr.engine.get()}")
>>> print(f"Memory Format: {wr.memory_format.get()}")
```

which show that both Ray and Modin are enabled as an execution engine and memory format, respectively. You can switch back to non-distributed mode at any point (See *Switching modes* below).

Initialization of the Ray cluster is lazy and only triggered when the first distributed API is executed. At that point, SDK for pandas looks for an environment variable called `WR_ADDRESS`. If found, it is used to send commands to a remote cluster. If not found, a local Ray runtime is initialized on your machine instead. Alternatively, you can trigger Ray initialization with:

```
>>> wr.engine.initialize()
```

In distributed mode, the same `awswrangler` APIs can now handle much larger datasets:

```
# Read 1.6 Gb Parquet data
df = wr.s3.read_parquet(path="s3://ursa-labs-taxi-data/2017/")

# Drop vendor_id column
df.drop("vendor_id", axis=1, inplace=True)

# Filter trips over 1 mile
df1 = df[df["trip_distance"] > 1]
```

In the example above, New York City Taxi data is read from Amazon S3 into a distributed Modin data frame. Modin is a drop-in replacement for Pandas. It exposes the same APIs but enables you to use all of the cores on your machine, or all of the workers in an entire cluster, leading to improved performance and scale. To use it, make sure to replace your pandas import statement with modin:

```
>>> import modin.pandas as pd  # instead of import pandas as pd
```

Failing to do so means that all operations run on a single thread instead of leveraging the entire cluster resources.

Note that in distributed mode, all `awswrangler` APIs return and accept Modin data frames, not pandas.

1.3.2 Supported APIs

This table lists the `awswrangler` APIs available in distributed mode (i.e. that can run at scale):

Service	API	Implementation
S3	read_parquet	
	read_parquet_metadata	
	read_parquet_table	
	read_csv	
	read_json	
	read_fwf	
	to_parquet	
	to_csv	
	to_json	
	select_query	
	store_parquet_metadata	
	delete_objects	
	describe_objects	
	size_objects	
	wait_objects_exist	
	wait_objects_not_exist	
	merge_datasets	
	copy_objects	
Redshift	copy	
	unload	
Athena	describe_table	
	get_query_results	
	read_sql_query	
	read_sql_table	
	show_create_table	
	to_iceberg	

continues on next page

Table 2 – continued from previous page

Service	API	Implementation
	delete_from_iceberg_table	
DynamoDB	read_items	
	put_df	
	put_csv	
	put_json	
	put_items	
Neptune	bulk_load	
Timestream	batch_load	
	write	
	unload	

1.3.3 Switching modes

The following commands showcase how to switch between distributed and non-distributed modes:

```
# Switch to non-distributed
wr.engine.set("python")
wr.memory_format.set("pandas")

# Switch to distributed
wr.engine.set("ray")
wr.memory_format.set("modin")
```

Similarly, you can set the `WR_ENGINE` and `WR_MEMORY_FORMAT` environment variables to the desired engine and memory format, respectively.

1.3.4 Caveats

S3FS Filesystem

When Ray is chosen as an engine, S3Fs is used instead of boto3 for certain API calls. These include listing a large number of S3 objects for example. This choice was made for performance reasons as a boto3 implementation can be much slower in some cases. As a side effect, users won't be able to use the `s3_additional_kwargs` input parameter as it's currently not supported by S3Fs.

Unsupported kwargs

Most AWS SDK for pandas calls support passing the `boto3_session` argument. While this is acceptable for an application running in a single process, distributed applications require the session to be serialized and passed to the worker nodes in the cluster. This constitutes a security risk. As a result, passing `boto3_session` when using the Ray runtime is not supported.

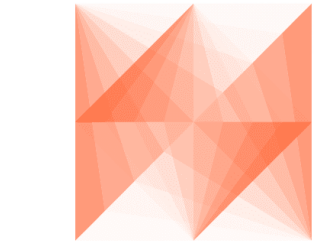
1.3.5 To learn more

Head to our latest tutorials to discover even more features.

A runbook with common errors when running the library with Ray is available here.

1.4 Tutorials

Note: You can also find all Tutorial Notebooks on GitHub.



AWS SDK for pandas

1.4.1 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 3.8, 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.

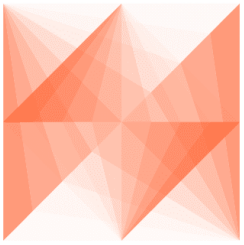
Let's Install it!

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

Restart your kernel after the installation!

```
[1]: import awswrangler as wr

     wr.__version__
```

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



AWS SDK for pandas

1.4.2 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 boto3

     import awswrangler as wr
```

Using the default Boto3 Session

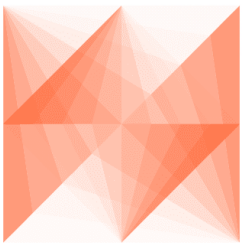
```
[2]: wr.s3.does_object_exist("s3://noaa-ghcn-pds/fake")
[2]: False
```

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

1.4.3 3 - Amazon S3

Table of Contents

[1]:
```
from datetime import datetime

import boto3
import pandas as pd
import pytz

import awswrangler as wr

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

(continues on next page)

(continued from previous page)

```
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 releated 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\n2    Lambusart 14-06-18\n3 Spormaggiore 15-04-18"
boto3.client("s3").put_object(Body=content, Bucket=bucket, Key="fwf/file1.txt")

content = "4    Buizingen 05-09-19\n5   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 datime 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
```

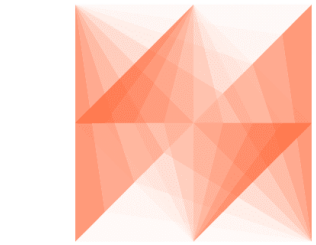
8.2 Upload object from a file-like object in binary mode

[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}/")
```



1.4.4 4 - Parquet Datasets

awswrangler 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 pandas as pd

import awswrangler as wr
```

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
↪"])

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)

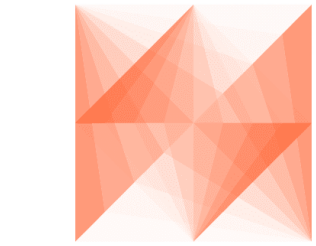
[7]:
```
df = pd.DataFrame({"id": [2, 3], "value": ["xoo", "bar"], "date": [date(2020, 1, 2),␣
↪date(2020, 1, 3)]})

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
```



1.4.5 5 - Glue Catalog

awswrangler makes heavy use of Glue Catalog to store metadata of tables and connections.

[1]:
```
import pandas as pd

import awswrangler as wr
```

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

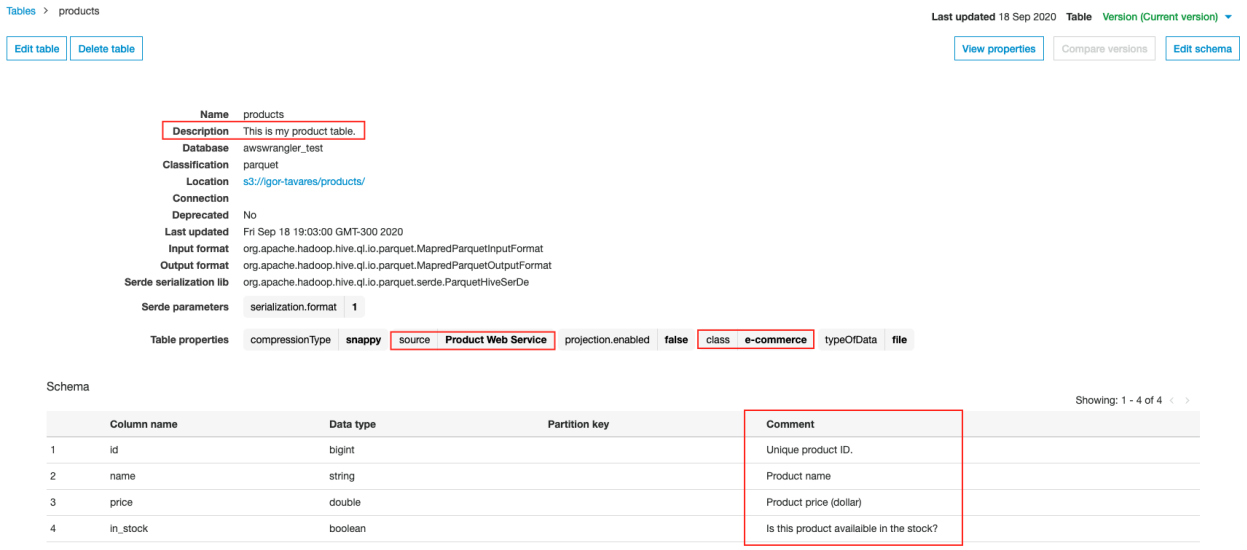
[6]:
```
wr.catalog.tables(database="awswrangler_test")
```

[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 availaible in the stock?",
}

res = wr.s3.to_parquet(
    df=df,
    path=f"s3://{bucket}/products/",
    dataset=True,
    database="awswrangler_test",
    table="products",
    mode="overwrite",
    glue_table_settings=wr.typing.GlueTableSettings(description=desc, parameters=param,␣
↪columns_comments=comments),
)
```

Checking Glue Catalog (AWS Console)



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.
```

(continues on next page)

(continued from previous page)

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

Getting tables details

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

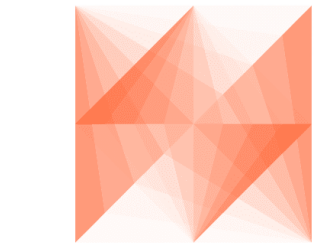
[12]:
```
  Column Name     Type  Partition                                  Comment
0          id   bigint      False                       Unique product ID.
1        name   string      False                             Product name
2       price   double      False                   Product price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



1.4.6 6 - Amazon Athena

awswrangler 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="awswrangler_test")
```

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

[ ]:
```
%%time

wr.athena.read_sql_query(
    "SELECT * FROM noaa",
    database="awswrangler_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="awswrangler_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="awswrangler_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="awswrangler_test",␣
↪chunksize=100_000_000)

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

Parameterized queries

Client-side parameter resolution

The `params` parameter allows client-side resolution of parameters, which are specified with `:col_name`, when `paramstyle` is set to `named`. Additionally, Python types will map to the appropriate Athena definitions. For example, the value `dt.date(2023, 1, 1)` will resolve to `DATE '2023-01-01`.

For the example below, the following query will be sent to Athena:

```
SELECT * FROM noaa WHERE S_FLAG = 'E'
```

[ ]:
```
%%time

wr.athena.read_sql_query(
    "SELECT * FROM noaa WHERE S_FLAG = :flag_value",
    database="awswrangler_test",
```

(continues on next page)

(continued from previous page)

```
    params={
        "flag_value": "E",
    },
)
```

Server-side parameter resolution

Alternatively, Athena supports server-side parameter resolution when `paramstyle` is defined as `qmark`. The SQL statement sent to Athena will not contain the values passed in `params`. Instead, they will be passed as part of a separate `params` parameter in `boto3`.

The downside of using this approach is that types aren't automatically resolved. The values sent to `params` must be strings. Therefore, if one of the values is a date, the value passed in `params` has to be `DATE 'XXXX-XX-XX'`.

The upside, however, is that these parameters can be used with prepared statements.

For more information, see "Using parameterized queries".

[ ]:
```
%%time

wr.athena.read_sql_query(
    "SELECT * FROM noaa WHERE S_FLAG = ?",
    database="awswrangler_test",
    params=["E"],
    paramstyle="qmark",
)
```

Prepared statements

[ ]:
```
wr.athena.create_prepared_statement(
    sql="SELECT * FROM noaa WHERE S_FLAG = ?",
    statement_name="statement",
)

# Resolve parameter using Athena execution parameters
wr.athena.read_sql_query(
    sql="EXECUTE statement",
    database="awswrangler_test",
    params=["E"],
    paramstyle="qmark",
)

# Resolve parameter using Athena execution parameters (same effect as above)
wr.athena.read_sql_query(
    sql="EXECUTE statement USING ?",
    database="awswrangler_test",
    params=["E"],
    paramstyle="qmark",
)

# Resolve parameter using client-side formatter
```

(continues on next page)

(continued from previous page)

```
wr.athena.read_sql_query(
    sql="EXECUTE statement USING :flag_value",
    database="awswrangler_test",
    params={
        "flag_value": "E",
    },
    paramstyle="named",
)
```

[ ]:
```
# Clean up prepared statement
wr.athena.delete_prepared_statement(statement_name="statement")
```

Cleaning Up S3

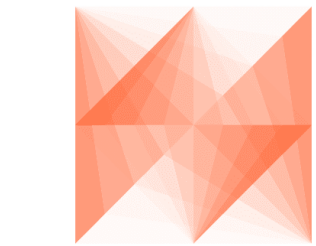
[ ]:
```
wr.s3.delete_objects(path)
```

Delete table

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

Delete Database

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



1.4.7 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()

[ ]:
```
# Install the optional modules first
!pip install 'awswrangler[redshift, postgres, mysql, sqlserver, oracle]'
```

[1]:
```
import pandas as pd

import awswrangler as wr

df = pd.DataFrame({"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

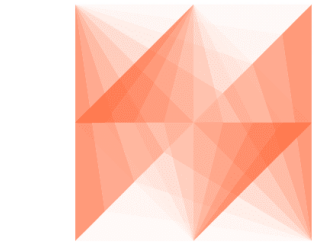
[4]:
```
wr.redshift.to_sql(df, con_redshift, schema="public", table="tutorial", mode="overwrite")
wr.mysql.to_sql(df, con_mysql, schema="test", table="tutorial", mode="overwrite")
wr.postgresql.to_sql(df, con_postgresql, schema="public", table="tutorial", mode=
↪"overwrite")
wr.sqlserver.to_sql(df,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

1.4.8 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.

[ ]:
```
# Install the optional modules first
!pip install 'awswrangler[redshift]'
```

[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,
    path=path,
    keep_files=True,
)
```

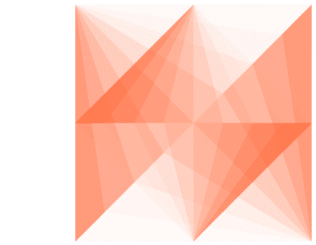
```
CPU times: user 10 s, sys: 1.14 s, total: 11.2 s
Wall time: 27.5 s
```

[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()
```



1.4.9 9 - Redshift - Append, Overwrite and Upsert

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

1 - `append`

2 - `overwrite`

3 - `upsert`

[ ]:
```
# Install the optional modules first
!pip install 'awswrangler[redshift]'
```

[2]:
```
from datetime import date

import pandas as pd

import awswrangler as wr

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, 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

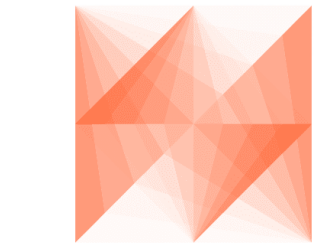
[12]:
```
df = pd.DataFrame({"id": [2, 3], "value": ["xoo", "bar"], "date": [date(2020, 1, 2),␣
↪date(2020, 1, 3)]})

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

1.4.10 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
```

[3]:

	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
...	...	...	...	...	...	...	...	...
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=1890/06a519afcf8e48c9b08c8908f30adcfe.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", 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
```

(continues on next page)

(continued from previous page)

```
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	year
0	USC00195145	1890-01-01	TMIN	-28	<NA>	<NA>	6	<NA>	1890
1	USC00196770	1890-01-01	PRCP	0	P	<NA>	6	<NA>	1890
2	USC00196770	1890-01-01	SNOW	0	<NA>	<NA>	6	<NA>	1890
3	USC00196915	1890-01-01	PRCP	0	P	<NA>	6	<NA>	1890
4	USC00196915	1890-01-01	SNOW	0	<NA>	<NA>	6	<NA>	1890
...	...	...	...	...	...	...	...	...	...
6139	ASN00022006	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	1890
6140	ASN00022007	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	1890
6141	ASN00022008	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	1890
6142	ASN00022009	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	1890
6143	ASN00022011	1890-12-03	PRCP	0	<NA>	<NA>	a	<NA>	1890

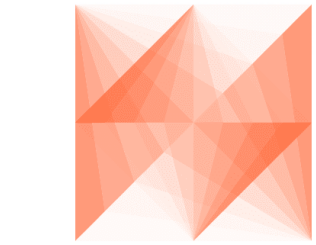
[1276246 rows x 9 columns]

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"])
```



1.4.11 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 pandas as pd

import awswrangler as wr
```

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, 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)

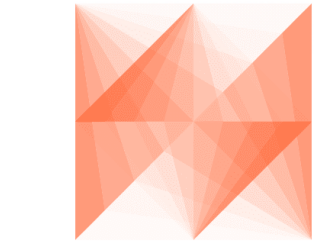
[8]:
```
df = pd.DataFrame({"id": [2, 3], "value": ["xoo", "bar"], "date": [date(2020, 1, 2),␣
↪date(2020, 1, 3)]})

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
```



1.4.12 12 - CSV Crawler

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

[1]:
```
from datetime import datetime

import pandas as pd

import awswrangler as wr
```

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"],
    }
)

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,
```

(continues on next page)

(continued from previous page)

```
    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
```

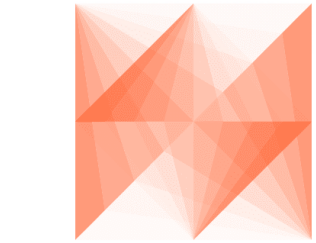
[14]:
```
id                     Int64
string                string
float                float64
date                   object
timestamp    datetime64[ns]
bool                 boolean
par0                   Int64
par1                  string
dtype: object
```

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
```



1.4.13 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 pandas as pd

import awswrangler as wr
```

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
```

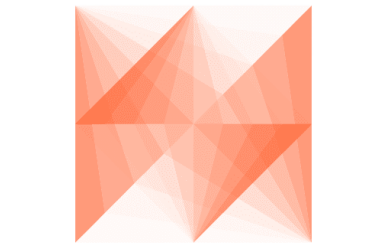
Merging (Dataset 2 -> Dataset 1) (OVERWRITE)

[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)
```



1.4.14 14 - Schema Evolution

awswrangler 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 pandas as pd
```

(continues on next page)

(continued from previous page)

```
import awswrangler as wr
```

Enter your bucket name:

[2]:
```
import getpass

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

```
 ........................................
```

Creating the Dataset

Parquet Create

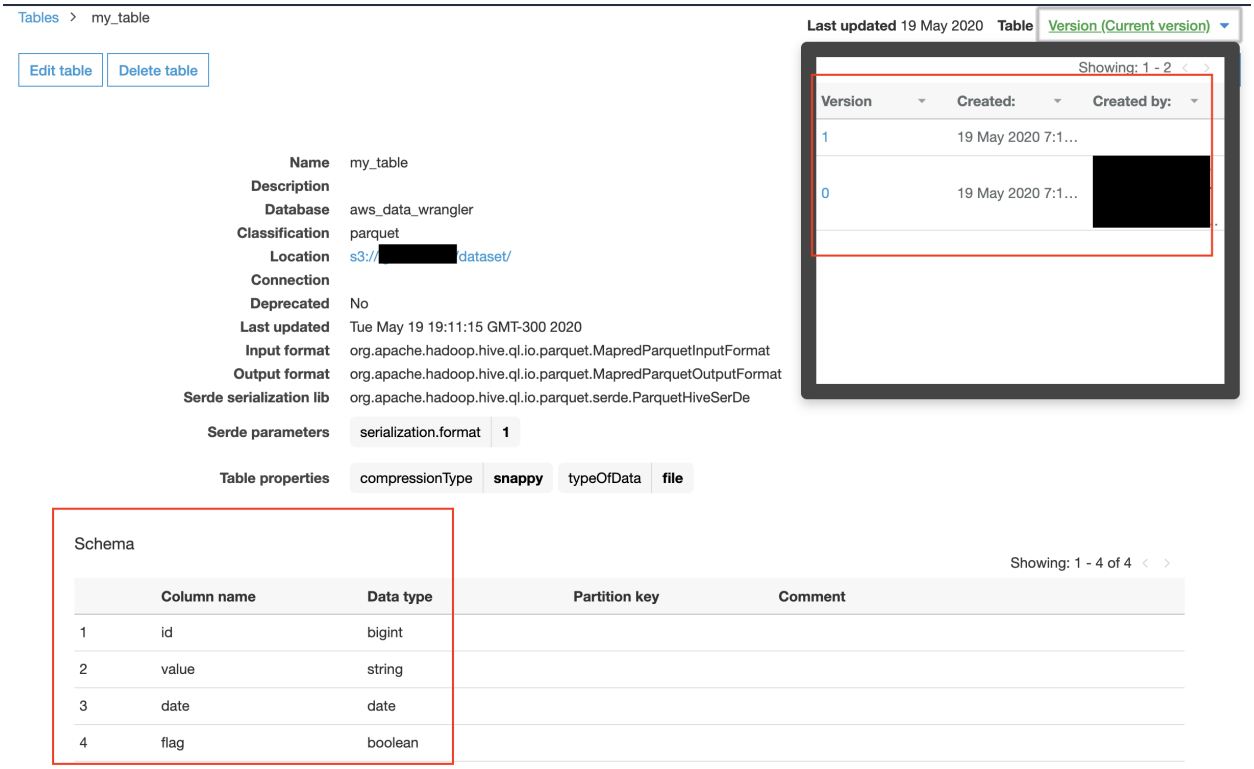
[3]:
```
df = pd.DataFrame(
    {
        "id": [1, 2],
        "value": ["foo", "boo"],
    }
)

wr.s3.to_parquet(df=df, 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 Create

[ ]:
```
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)



Appending with NEW COLUMNS

Parquet Append

[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)

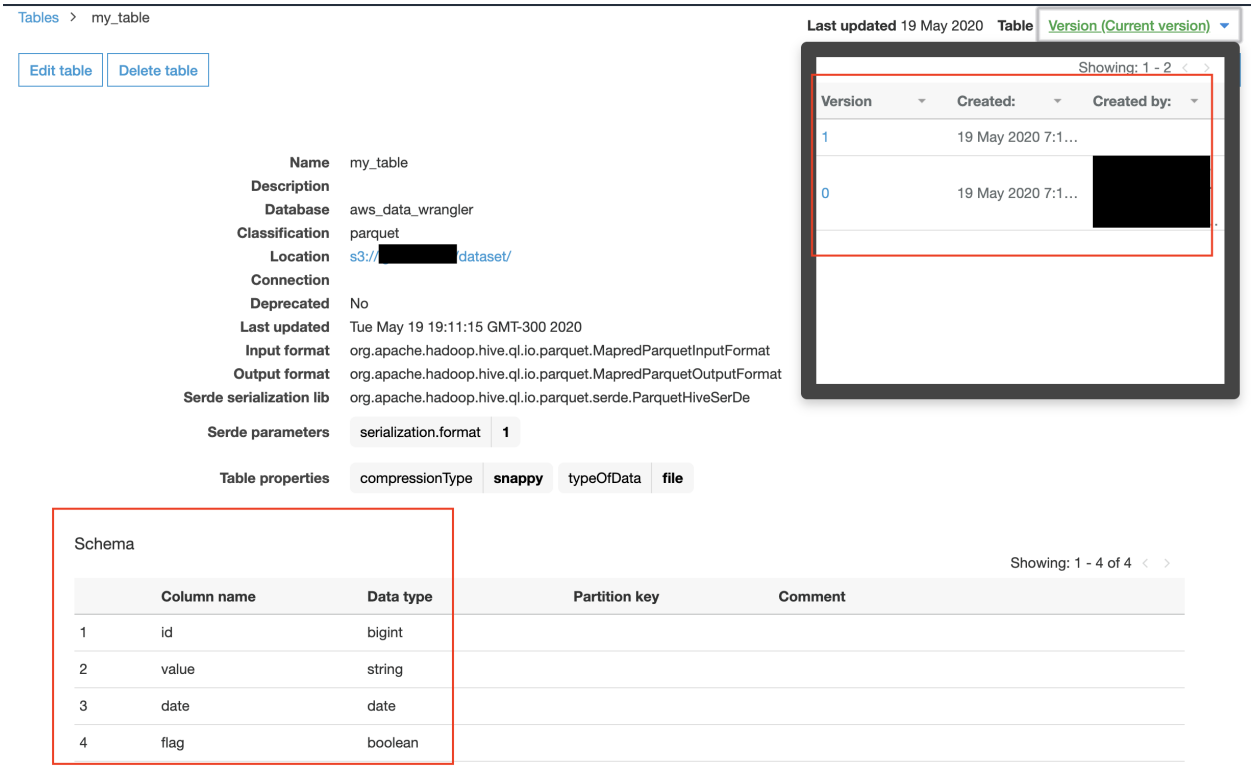
```
2   1   foo         NaN    NaN
3   2   boo         NaN    NaN
```

CSV Append

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)



Reading from Athena

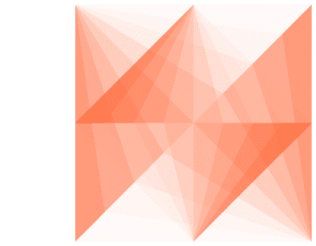
```
[5]: wr.athena.read_sql_table(table="my_table", database="aws_sdk_pandas")
```

```
[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

1.4.15 15 - EMR

[1]:
```
import boto3

import awswrangler as wr
```

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

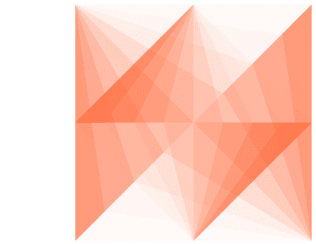
[11]:
```
step_id = wr.emr.submit_step(cluster_id, command=f"spark-submit s3://{bucket}/test.py")
```

Wait Step

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

Terminate Cluster

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



1.4.16 16 - EMR & Docker

[ ]:
```
import getpass

import boto3

import awswrangler as wr
```

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:
↪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-1B0O45RWJL8CL'
```

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

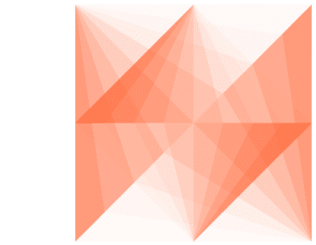
[9]:
```
cluster_id = wr.emr.create_cluster(
    cluster_name="my-demo-cluster-v2",
    logging_s3_path=f"s3://{bucket}/emr-logs/",
    emr_release="emr-6.7.0",
    subnet_id=subnet,
    emr_ec2_role="EMR_EC2_DefaultRole",
    emr_role="EMR_DefaultRole",
    instance_type_master="m5.2xlarge",
    instance_type_core="m5.2xlarge",
    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"

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
```

(continues on next page)

(continued from previous page)

```
wr.emr.terminate_cluster(cluster_id)
```



1.4.17 17 - Partition Projection

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

[1]:
```
import getpass
from datetime import datetime

import pandas as pd

import awswrangler as wr
```

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]})

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,
```

(continues on next page)

(continued from previous page)

```
    partition_cols=["year", "month", "day"],
    database="default",
    table="table_integer",
    athena_partition_projection_settings={
        "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("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,
    path=f"s3://{bucket}/table_enum/",
    dataset=True,
    partition_cols=["city"],
    database="default",
    table="table_enum",
    athena_partition_projection_settings={
        "projection_types": {
            "city": "enum",
        },
        "projection_values": {"city": "São Paulo,Tokio,Seattle"},
    },
)
```

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

[8]:
```
   value       city
0      1  São Paulo
```

(continues on next page)

(continued from previous page)

```
1      3    Seattle
2      2      Tokio
```

Date projection

[9]:
```
def ts(x):
    return datetime.strptime(x, "%Y-%m-%d %H:%M:%S")


def dt(x):
    return 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",
    athena_partition_projection_settings={
        "projection_types": {
            "dt": "date",
            "ts": "date",
        },
        "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("SELECT * FROM table_date", database="default")
```

[11]:
```
   value          dt                  ts
0      1  2020-01-01 2020-01-01 00:00:00
```

(continues on next page)

(continued from previous page)

```
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",
    athena_partition_projection_settings={
        "projection_types": {
            "uuid": "injected",
        }
    },
)
```

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

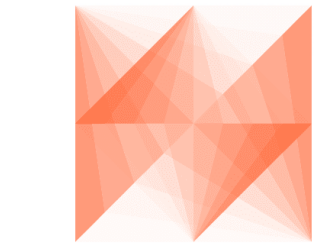
[14]:
```
   value                                  uuid
0      2  b89ed095-8179-4635-9537-88592c0f6bc3
```

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")
```

[ ]:
```
```



1.4.18 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]:
```
from time import sleep

import awswrangler as wr
```

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   awswrangler_test
2           covid-19
3            default                          Default Hive database
```

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

[4]:
```
    Database                                    Table  \
0   covid-19        alleninstitute_comprehend_medical
1   covid-19                  alleninstitute_metadata
2   covid-19                            country_codes
3   covid-19                       county_populations
4   covid-19              covid_knowledge_graph_edges
5   covid-19       covid_knowledge_graph_nodes_author
6   covid-19      covid_knowledge_graph_nodes_concept
7   covid-19  covid_knowledge_graph_nodes_institution
8   covid-19        covid_knowledge_graph_nodes_paper
9   covid-19        covid_knowledge_graph_nodes_topic
10  covid-19                covid_testing_states_daily
11  covid-19                    covid_testing_us_daily
12  covid-19                    covid_testing_us_total
13  covid-19                           covidcast_data
14  covid-19                       covidcast_metadata
15  covid-19                                enigma_jhu
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

                                         Description  \
0   Comprehend Medical results run against Allen I...
1   Metadata on papers pulled from the Allen Insti...
2                     Lookup table for country codes
3   Lookup table for population for each county ba...
4               AWS Knowledge Graph for COVID-19 data
5               AWS Knowledge Graph for COVID-19 data
6               AWS Knowledge Graph for COVID-19 data
7               AWS Knowledge Graph for COVID-19 data
8               AWS Knowledge Graph for COVID-19 data
9               AWS Knowledge Graph for COVID-19 data
10  USA total test daily trend by state.  Sourced ...
11  USA total test daily trend.  Sourced from covi...
12  USA total tests.  Sourced from covidtracking.c...
13            CMU Delphi's COVID-19 Surveillance Data
14        CMU Delphi's COVID-19 Surveillance Metadata
15  Johns Hopkins University Consolidated data on ...
```

(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={"users": ["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".nytimes_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={"users": ["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")
]
```

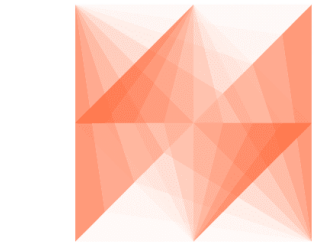
[13]:
```
[{'time': datetime.datetime(2020, 6, 12, 15, 13, 46, 996000, tzinfo=tzlocal()),
  'source': 'MANUAL'},
 {'time': datetime.datetime(2020, 6, 12, 15, 13, 42, 344000, tzinfo=tzlocal()),
  'source': 'MANUAL'}]
```

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={"users": ["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")
```



1.4.19 19 - Amazon Athena Cache

awswrangler has a cache strategy that is disabled by default and can be enabled by passing `max_cache_seconds` bigger than 0 as part of the `athena_cache_settings` parameter. 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.

[18]:
```
import awswrangler as wr
```

Enter your bucket name:

[19]:
```
import getpass

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

Checking/Creating Glue Catalog Databases

[20]:
```
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

[21]:
```
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/1865.csv", names=cols, parse_
↪dates=["dt", "obs_time"])

df
```

[21]:

	id	dt	element	value	m_flag	q_flag	s_flag	\
0	ID	DATE	ELEMENT	DATA_VALUE	M_FLAG	Q_FLAG	S_FLAG	
1	AGE00135039	18650101	PRCP	0	NaN	NaN	E	
2	ASN00019036	18650101	PRCP	0	NaN	NaN	a	
3	ASN00021001	18650101	PRCP	0	NaN	NaN	a	
4	ASN00021010	18650101	PRCP	0	NaN	NaN	a	
...	...	...	...	...	...	...	...	
37918	USC00288878	18651231	TMIN	-44	NaN	NaN	6	

(continues on next page)

(continued from previous page)

```
37919  USC00288878  18651231     PRCP           0       P     NaN       6
37920  USC00288878  18651231     SNOW           0       P     NaN       6
37921  USC00361920  18651231     PRCP           0     NaN     NaN       F
37922  USP00CA0001  18651231     PRCP           0     NaN     NaN       F

       obs_time
0      OBS_TIME
1           NaN
2           NaN
3           NaN
4           NaN
...         ...
37918       NaN
37919       NaN
37920       NaN
37921       NaN
37922       NaN

[37923 rows x 8 columns]
```

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

[23]:
```
wr.catalog.table(database="awswrangler_test", table="noaa")
```

[23]:
```
  Column Name    Type  Partition Comment
0          id  string      False
1          dt  string      False
2     element  string      False
3       value  string      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.

[24]:
```
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...

[25]:
```
%%time

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

```
CPU times: user 1.59 s, sys: 166 ms, total: 1.75 s
Wall time: 5.62 s
```

[25]:

	element	cnt
0	PRCP	12044499
1	MDTX	1460
2	DATX	1460
3	ELEMENT	1
4	WT01	22260
5	WT03	840
6	DATN	1460
7	DWPR	490
8	TMIN	7012479
9	MDTN	1460
10	MDPR	2683
11	SNOW	1086762
12	DAPR	1330
13	SNWD	783532
14	TMAX	6533103

Second execution with CACHE (400x faster)

[26]:
```
%%time

wr.athena.read_sql_query(query, database="awswrangler_test", athena_cache_settings={"max_
↪cache_seconds": 900})
```

```
CPU times: user 689 ms, sys: 68.1 ms, total: 757 ms
Wall time: 1.11 s
```

[26]:

	element	cnt
0	PRCP	12044499
1	MDTX	1460
2	DATX	1460
3	ELEMENT	1
4	WT01	22260
5	WT03	840
6	DATN	1460
7	DWPR	490
8	TMIN	7012479
9	MDTN	1460
10	MDPR	2683

(continues on next page)

(continued from previous page)

```
11      SNOW   1086762
12      DAPR      1330
13      SNWD    783532
14      TMAX   6533103
```

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

[27]:
```
%%time

wr.athena.read_sql_query(
    query,
    database="awswrangler_test",
    athena_cache_settings={"max_cache_seconds": 900, "max_cache_query_inspections": 500},
)
```

```
CPU times: user 715 ms, sys: 44.9 ms, total: 760 ms
Wall time: 1.03 s
```

[27]:
```
    element       cnt
0      PRCP  12044499
1      MDTX      1460
2      DATX      1460
3   ELEMENT         1
4      WT01     22260
5      WT03       840
6      DATN      1460
7      DWPR       490
8      TMIN   7012479
9      MDTN      1460
10     MDPR      2683
11     SNOW   1086762
12     DAPR      1330
13     SNWD    783532
14     TMAX   6533103
```

Cleaning Up S3

[28]:
```
wr.s3.delete_objects(path)
```

Delete table

[29]:
```
wr.catalog.delete_table_if_exists(database="awswrangler_test", table="noaa")
```

[29]:
```
True
```

Delete Database

[30]:
```
wr.catalog.delete_database("awswrangler_test")
```

1.4.20 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")
```

1.4.21 21 - Global Configurations

awswrangler 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 thefunction API docto 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
```

[2]:
```
import botocore

import awswrangler as wr
```

[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")
```

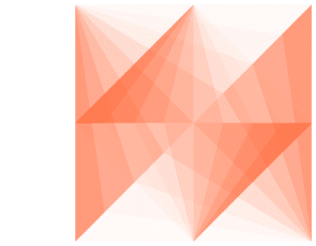
[6]:
```
   foo
0    1
```

Visualizing

[7]:
```
wr.config
```

[7]:
```
<awswrangler._config._Config at 0x1376ece80>
```

[ ]:
```

```



AWS SDK for pandas

1.4.22 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

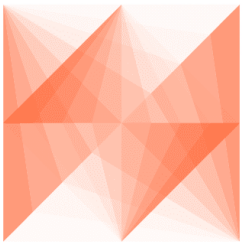
[4]:
```
%%time
%%memit

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

1.4.23 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 pandas as pd

import awswrangler as wr
```

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

Parquet Example 1

[4]:
```
def my_filter(x):
    return x["value"].endswith("oo")


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

[4]:

	id	value
0	2	boo
1	1	foo

Parquet 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
```

CSV Example 1

[7]:
```
def my_filter(x):
    return x["value"].endswith("oo")


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

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

CSV 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
```

1.4.24 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
```

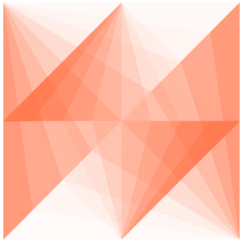
[5]:
```
df = wr.athena.read_sql_query("SELECT 1 AS foo")

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

1.4.25 25 - Redshift - Loading Parquet files with Spectrum

Enter your bucket name:

[ ]:
```
# Install the optional modules first
!pip install 'awswrangler[redshift]'
```

[1]:
```
import getpass

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

```
 ··········
```

Mocking some Parquet Files on S3

[2]:
```
import pandas as pd

import awswrangler as wr

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"],
    }
)

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

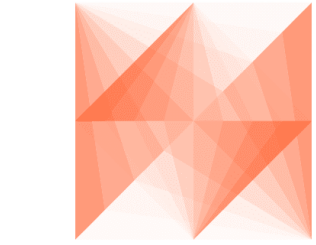
[11]:
```
query = "SELECT * FROM public.test"
```

[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

1.4.26 26 - Amazon Timestream

Creating resources

[10]:
```
from datetime import datetime

import pandas as pd

import awswrangler as wr

database = "sampleDB"
table_1 = "sampleTable1"
table_2 = "sampleTable2"
wr.timestream.create_database(database)
wr.timestream.create_table(database, table_1, memory_retention_hours=1, magnetic_
↪retention_days=1)
wr.timestream.create_table(database, table_2, memory_retention_hours=1, magnetic_
↪retention_days=1)
```

Write

Single measure WriteRecord

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

rejected_records = wr.timestream.write(
    df=df,
    database=database,
    table=table_1,
    time_col="time",
    measure_col="measure",
    dimensions_cols=["dim0", "dim1"],
)

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

```
Number of rejected records: 0
```

Multi measure WriteRecord

[ ]:
```
df = pd.DataFrame(
    {
        "time": [datetime.now()] * 3,
        "measure_1": ["10", "20", "30"],
        "measure_2": ["100", "200", "300"],
        "measure_3": ["1000", "2000", "3000"],
        "tag": ["tag123", "tag456", "tag789"],
    }
)
rejected_records = wr.timestream.write(
    df=df,
    database=database,
    table=table_2,
    time_col="time",
    measure_col=["measure_1", "measure_2", "measure_3"],
    dimensions_cols=["tag"],
)

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

Query

[12]:
```
wr.timestream.query(
    f'SELECT time, measure_value::double, dim0, dim1 FROM "{database}"."{table_1}" ORDER␣
↪BY time DESC LIMIT 3'
)
```

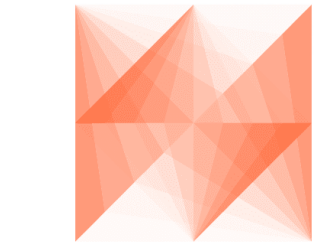
[12]:
```
                     time  measure_value::double dim0 dim1
0 2020-12-08 19:15:32.468                     1.0  foo    1
1 2020-12-08 19:15:32.468                     1.2  bar    3
2 2020-12-08 19:15:32.468                     1.1  boo    2
```

Unload

[ ]:
```
df = wr.timestream.unload(
    sql=f'SELECT time, measure_value, dim0, dim1 FROM "{database}"."{table_1}"',
    path="s3://bucket/extracted_parquet_files/",
    partition_cols=["dim1"],
)
```

Deleting resources

[13]:
```
wr.timestream.delete_table(database, table_1)
wr.timestream.delete_table(database, table_2)
wr.timestream.delete_database(database)
```



1.4.27 27 - Amazon Timestream - Example 2

Reading test data

[1]:
```
from datetime import datetime

import pandas as pd

import awswrangler as wr

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
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
```

(continues on next page)

(continued from previous page)

```
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
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
```

Batch Load 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
```

(continues on next page)

(continued from previous page)

```
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
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]:
```
response = wr.timestream.batch_load(
    df=df_memory,
    path="s3://bucket/prefix/",
    database="sampleDB",
    table="sampleTable",
    time_col="time",
    measure_cols=["memory_utilization"],
    dimensions_cols=["index", "region", "az", "hostname"],
    measure_name_col="measure_kind",
    report_s3_configuration={"BucketName": "error_bucket", "ObjectKeyPrefix": "error_
↪prefix"},
)
assert response["BatchLoadTaskDescription"]["ProgressReport"]["RecordIngestionFailures"]␣
↪== 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-OgvFx   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
```

(continues on next page)

(continued from previous page)

```
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
```

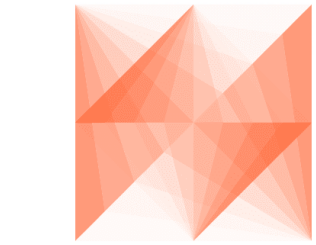
Querying MEMORY_UTILIZATION

[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")
```



1.4.28 28 - Amazon DynamoDB

Writing Data

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

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

import awswrangler as wr
```

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 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))
)
```

Read PartiQL

[29]:
```
wr.dynamodb.read_partiql_query(
    query=f"SELECT * FROM {table_name} WHERE title=? AND year=?",
    parameters=["Snatch", 2000],
)
```

[29]:
```
   year        genre   title
0  2000  caper story  Snatch
```

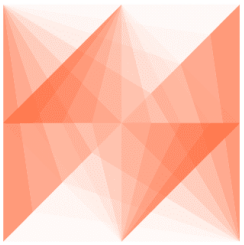
Executing statements

[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],
)

# 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

1.4.29 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:

- 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 multiple Parquet files from an S3 prefix

[1]:
```
import awswrangler as wr

df = wr.s3.select_query(
    sql='SELECT * FROM s3object s where s."trip_distance" > 30',
    path="s3://ursa-labs-taxi-data/2019/01/",
    input_serialization="Parquet",
    input_serialization_params={},
)

df.head()
```

[1]:
```
  vendor_id                 pickup_at                dropoff_at  \
0         2  2019-01-01T00:48:10.000Z  2019-01-01T01:36:58.000Z
1         2  2019-01-01T00:38:36.000Z  2019-01-01T01:21:33.000Z
2         2  2019-01-01T00:10:43.000Z  2019-01-01T01:23:59.000Z
3         1  2019-01-01T00:13:17.000Z  2019-01-01T01:06:13.000Z
4         2  2019-01-01T00:29:11.000Z  2019-01-01T01:29:05.000Z

   passenger_count  trip_distance rate_code_id store_and_fwd_flag  \
0                1      31.570000            1                  N
1                2      33.189999            5                  N
2                1      33.060001            1                  N
3                1      44.099998            5                  N
4                2      31.100000            1                  N

   pickup_location_id  dropoff_location_id payment_type  fare_amount  extra  \
0                 138                  138            2         82.5    0.5
1                 107                  265            1        121.0    0.0
2                 243                   42            2         92.0    0.5
3                 132                  265            2        150.0    0.0
4                 169                  201            1         85.5    0.5

   mta_tax  tip_amount  tolls_amount  improvement_surcharge  total_amount  \
0      0.5        0.00          0.00                    0.3     83.800003
```

(continues on next page)

(continued from previous page)

```
1      0.0        0.08         10.50                    0.3    131.880005
2      0.5        0.00          5.76                    0.3     99.059998
3      0.0        0.00          0.00                    0.3    150.300003
4      0.5        0.00          7.92                    0.3     94.720001

   congestion_surcharge
0                   NaN
1                   NaN
2                   NaN
3                   NaN
4                   NaN
```

Read full CSV file

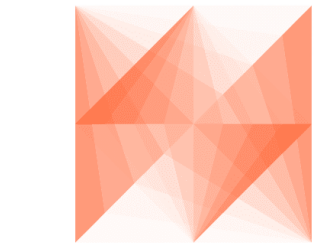
[5]:
```
df = wr.s3.select_query(
    sql="SELECT * FROM s3object",
    path="s3://humor-detection-pds/Humorous.csv",
    input_serialization="CSV",
    input_serialization_params={
        "FileHeaderInfo": "Use",
        "RecordDelimiter": "\r\n",
    },
    scan_range_chunk_size=1024 * 1024 * 32,  # override range of bytes to query, by default 1Mb
    use_threads=True,
)
df.head()
```

[5]:
```
                                          question  \
0         Will the volca sample get me a girlfriend?
1   Can u communicate with spirits even on Saturday?
2                          I won't get hunted right?
3  I have a few questions.. Can you get possessed...
4  Has anyone asked where the treasure is? What w...

                product_description  \
0    Korg Amplifier Part VOLCASAMPLE
1  Winning Moves Games Classic Ouija
2  Winning Moves Games Classic Ouija
3  Winning Moves Games Classic Ouija
4  Winning Moves Games Classic Ouija

                                        image_url label
0  http://ecx.images-amazon.com/images/I/81I1XZea...     1
1  http://ecx.images-amazon.com/images/I/81kcYEG5...     1
2  http://ecx.images-amazon.com/images/I/81kcYEG5...     1
3  http://ecx.images-amazon.com/images/I/81kcYEG5...     1
4  http://ecx.images-amazon.com/images/I/81kcYEG5...     1
```

Filter JSON file

[3]:
```
wr.s3.select_query(
    sql="SELECT * FROM s3object[*] s where s.\"family_name\" = 'Biden'",
    path="s3://awsglue-datasets/examples/us-legislators/all/persons.json",
    input_serialization="JSON",
    input_serialization_params={
        "Type": "Document",
    },
)
```

[3]:
```
  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  [{'identifier': 'B000444', 'scheme': 'bioguide...

                                         other_names      sort_name  \
0  [{'lang': None, 'name': 'Joe Biden', 'note': '...  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
```



AWS SDK for pandas

1.4.30 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

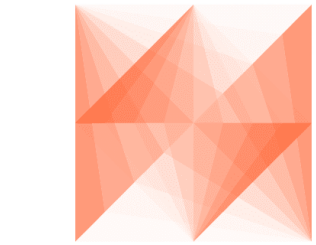
- wr.data_api.redshift.connect()
- wr.data_api.rds.connect()

[ ]:
```
con_redshift = wr.data_api.redshift.connect(
    cluster_id="aws-sdk-pandas-1xn5lqxrdxrv3",
    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,
)
```



1.4.31 31 - OpenSearch

Table of Contents

1. Initialize

[ ]:
```
# Install the optional modules first
!pip install 'awswrangler[opensearch]'
```

[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},
        "business_name": "Soup Unlimited",
        "business_postal_code": "94102",
```

(continues on next page)

(continued from previous page)

```
        "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",
    },
    # duplicate record
```

(continues on next page)

(continued from previous page)

```
    {
        "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

    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,
```

(continues on next page)

(continued from previous page)

```
    },  # 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
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

Search results are returned as Pandas DataFrame

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
```

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
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
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"],
    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
```

(continues on next page)

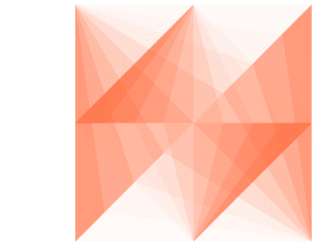
(continued from previous page)

```
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).  ...
..                                             ...  ...
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)
```

(continues on next page)

(continued from previous page)

```
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]
```



1.4.32 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[gremlin, opencypher, 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 pandas as pd

import awswrangler as wr

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 exists, 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 exists, 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 random
import string
import uuid


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 random
import string
import uuid


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"
```

(continues on next page)

(continued from previous page)

```
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)


# 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(s: str = "foo"):
         return {
             "s": s,
             "p": str(uuid.uuid4()),
             "o": random.randint(0, 1000),
         }


     label = f"foo_{uuid.uuid4()}"
     data = [_create_dummy_triple(label), _create_dummy_triple(label), _create_dummy_
     ↪triple(label)]
     df = pd.DataFrame(data)
     res = wr.neptune.to_rdf_graph(client, df)

     query = f"SELECT ?p ?o WHERE {{ <{label}> ?p ?o .}}"
     df = wr.neptune.execute_sparql(client, query)
     display(df)
```

Write Quads

```
[ ]: def _create_dummy_quad(s: str):
         data = _create_dummy_triple(s)
         data["g"] = "bar"
         return data


     label = f"foo_{uuid.uuid4()}"
     query = f"SELECT ?p ?o FROM <bar> WHERE {{ <{label}> ?p ?o .}}"

     data = [_create_dummy_quad(label), _create_dummy_quad(label), _create_dummy_quad(label)]
     df = pd.DataFrame(data)
     res = wr.neptune.to_rdf_graph(client, df)
     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 separator

```
[ ]: df_new = wr.neptune.flatten_nested_df(df, separator="|")
     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)
```

Bulk Load

Data can be written using the Neptune Bulk Loader by way of S3. The Bulk Loader is fast and optimized for large datasets.

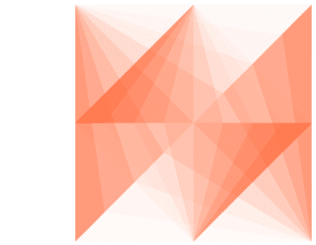
For details on the IAM permissions needed to set this up, see here.

[ ]:
```
df = pd.DataFrame([_create_dummy_edge() for _ in range(1000)])

wr.neptune.bulk_load(
    client=client,
    df=df,
    path="s3://my-bucket/stage-files/",
    iam_role="arn:aws:iam::XXX:role/XXX",
)
```

Alternatively, if the data is already on S3 in CSV format, you can use the `neptune.bulk_load_from_files` function. This is also useful if the data is written to S3 as a byproduct of an AWS Athena command, as the example below will show.

[ ]:
```
sql = """
SELECT
    <col_id> AS "~id"
  , <label_id> AS "~label"
  , *
FROM <database>.<table>
"""

wr.athena.start_query_execution(
    sql=sql,
    s3_output="s3://my-bucket/stage-files-athena/",
    wait=True,
)

wr.neptune.bulk_load_from_files(
    client=client,
    path="s3://my-bucket/stage-files-athena/",
    iam_role="arn:aws:iam::XXX:role/XXX",
)
```

Both the `bulk_load` and `bulk_load_from_files` functions are suitable at scale. The latter simply invokes the Neptune Bulk Loader on existing data in S3. The former, however, involves writing CSV data to S3. With `ray` and `modin` installed, this operation can also be distributed across multiple workers in a Ray cluster.



1.4.33 34 - Distributing Calls Using Ray

AWS SDK for pandas supports distribution of specific calls using ray and modin.

When enabled, data loading methods return modin dataframes instead of pandas dataframes. Modin provides seamless integration and compatibility with existing pandas code, with the benefit of distributing operations across your Ray instance and operating at a much larger scale.

[1]:
```
!pip install "awswrangler[modin,ray,redshift]"
```

Importing `awswrangler` when `ray` and `modin` are installed will automatically initialize a local Ray instance.

[3]:
```
import awswrangler as wr

print(f"Execution Engine: {wr.engine.get()}")
print(f"Memory Format: {wr.memory_format.get()}")
```

```
Execution Engine: EngineEnum.RAY
Memory Format: MemoryFormatEnum.MODIN
```

Read data at scale

Data is read using all cores on a single machine or multiple nodes on a cluster

[2]:
```
df = wr.s3.read_parquet(path="s3://ursa-labs-taxi-data/2019/")
df.head(5)
```

```
2023-09-15 12:24:44,457 INFO worker.py:1621 -- Started a local Ray instance.
2023-09-15 12:25:10,728 INFO read_api.py:374 -- To satisfy the requested parallelism of␣
↪200, each read task output will be split into 34 smaller blocks.
```

```
[dataset]: Run `pip install tqdm` to enable progress reporting.
```

```
UserWarning: When using a pre-initialized Ray cluster, please ensure that the runtime␣
↪env sets environment variable __MODIN_AUTOIMPORT_PANDAS__ to 1
```

[2]:
```
  vendor_id           pickup_at          dropoff_at  passenger_count  \
0         1 2019-01-01 00:46:40 2019-01-01 00:53:20                1
1         1 2019-01-01 00:59:47 2019-01-01 01:18:59                1
2         2 2018-12-21 13:48:30 2018-12-21 13:52:40                3
3         2 2018-11-28 15:52:25 2018-11-28 15:55:45                5
4         2 2018-11-28 15:56:57 2018-11-28 15:58:33                5
```

(continues on next page)

(continued from previous page)

```
   trip_distance rate_code_id store_and_fwd_flag  pickup_location_id  \
0            1.5            1                  N                 151
1            2.6            1                  N                 239
2            0.0            1                  N                 236
3            0.0            1                  N                 193
4            0.0            2                  N                 193

   dropoff_location_id payment_type  fare_amount  extra  mta_tax  tip_amount  \
0                  239            1          7.0    0.5      0.5        1.65
1                  246            1         14.0    0.5      0.5        1.00
2                  236            1          4.5    0.5      0.5        0.00
3                  193            2          3.5    0.5      0.5        0.00
4                  193            2         52.0    0.0      0.5        0.00

   tolls_amount  improvement_surcharge  total_amount  congestion_surcharge
0           0.0                    0.3      9.950000                   NaN
1           0.0                    0.3     16.299999                   NaN
2           0.0                    0.3      5.800000                   NaN
3           0.0                    0.3      7.550000                   NaN
4           0.0                    0.3     55.549999                   NaN
```

The data type is a modin DataFrame

[4]:
```
type(df)
```

[4]:
```
modin.pandas.dataframe.DataFrame
```

However, this type is interoperable with standard pandas calls:

[4]:
```
filtered_df = df[df.trip_distance > 30]
excluded_columns = ["vendor_id", "passenger_count", "store_and_fwd_flag"]
filtered_df = filtered_df.loc[:, ~filtered_df.columns.isin(excluded_columns)]
```

Enter your bucket name:

[7]:
```
bucket = "BUCKET"
```

Write data at scale

The write operation is parallelized, leading to significant speed-ups

[9]:
```
result = wr.s3.to_parquet(
    filtered_df,
    path=f"s3://{bucket}/taxi/",
    dataset=True,
)
print(f"Data has been written to {len(result['paths'])} files")
```

```
Data has been written to 408 files
```

```
2023-09-15 12:32:28,917 WARNING plan.py:567 -- Warning: The Ray cluster currently does␣
↪not have any available CPUs. The Dataset job will hang unless more CPUs are freed up.␣
```

(continues on next page)

(continued from previous page)

```
↪A common reason is that cluster resources are used by Actors or Tune trials; see the␣
↪following link for more details: https://docs.ray.io/en/master/data/dataset-internals.
↪html#datasets-and-tune
2023-09-15 12:32:31,094 INFO streaming_executor.py:92 -- Executing DAG␣
↪InputDataBuffer[Input] -> TaskPoolMapOperator[Write]
2023-09-15 12:32:31,095 INFO streaming_executor.py:93 -- Execution config:␣
↪ExecutionOptions(resource_limits=ExecutionResources(cpu=None, gpu=None, object_store_
↪memory=None), locality_with_output=False, preserve_order=False, actor_locality_
↪enabled=True, verbose_progress=False)
2023-09-15 12:32:31,096 INFO streaming_executor.py:95 -- Tip: For detailed progress␣
↪reporting, run `ray.data.DataContext.get_current().execution_options.verbose_progress␣
↪= True`
```

```
Data has been written to 408 files
```

Copy to Redshift at scale...

Data is first staged in S3 then a COPY command is executed against the Redshift cluster to load it. Both operations are distributed: S3 write with Ray and COPY in the Redshift cluster

[12]:
```
# Connect to the Redshift instance
con = wr.redshift.connect("aws-sdk-pandas-redshift")

path = f"s3://{bucket}/stage/"
iam_role = "ROLE"
schema = "public"
table = "taxi"

wr.redshift.copy(
    df=filtered_df,
    path=path,
    con=con,
    schema=schema,
    table=table,
    mode="overwrite",
    iam_role=iam_role,
    max_rows_by_file=None,
)
```

```
2023-09-15 12:52:24,155 INFO streaming_executor.py:92 -- Executing DAG␣
↪InputDataBuffer[Input] -> TaskPoolMapOperator[Write]
2023-09-15 12:52:24,157 INFO streaming_executor.py:93 -- Execution config:␣
↪ExecutionOptions(resource_limits=ExecutionResources(cpu=None, gpu=None, object_store_
↪memory=None), locality_with_output=False, preserve_order=False, actor_locality_
↪enabled=True, verbose_progress=False)
2023-09-15 12:52:24,157 INFO streaming_executor.py:95 -- Tip: For detailed progress␣
↪reporting, run `ray.data.DataContext.get_current().execution_options.verbose_progress␣
↪= True`
```

... and UNLOAD it back

Parallel calls can also be leveraged when reading from the cluster. The UNLOAD command distributes query processing in Redshift to dump files in S3 which are then read in parallel into a dataframe

[13]:
```
df = wr.redshift.unload(
    sql=f"SELECT * FROM {schema}.{table} where trip_distance > 30",
    con=con,
    iam_role=iam_role,
    path=path,
    keep_files=True,
)

df.head()
```

```
2023-09-15 12:56:53,838 INFO read_api.py:374 -- To satisfy the requested parallelism of␣
↪16, each read task output will be split into 8 smaller blocks.
```

[13]:

	pickup_at	dropoff_at	trip_distance	rate_code_id	\
0	2019-01-22 17:40:04	2019-01-22 18:33:48	30.469999	4	
1	2019-01-22 18:36:34	2019-01-22 19:52:50	33.330002	5	
2	2019-01-22 19:11:08	2019-01-22 20:16:10	32.599998	1	
3	2019-01-22 19:14:15	2019-01-22 20:09:57	36.220001	4	
4	2019-01-22 19:51:56	2019-01-22 20:48:39	33.040001	5	

	pickup_location_id	dropoff_location_id	payment_type	fare_amount	extra	\
0	132	265	1	142.000000	1.0	
1	51	221	1	96.019997	0.0	
2	231	205	1	88.000000	1.0	
3	132	265	1	130.500000	1.0	
4	132	265	1	130.000000	0.0	

	mta_tax	tip_amount	tolls_amount	improvement_surcharge	total_amount	\
0	0.5	28.760000	0.00	0.3	172.559998	
1	0.5	0.000000	11.52	0.3	108.339996	
2	0.5	0.000000	0.00	0.3	89.800003	
3	0.5	27.610001	5.76	0.3	165.669998	
4	0.5	29.410000	16.26	0.3	176.470001	

	congestion_surcharge
0	0.0
1	0.0
2	0.0
3	0.0
4	0.0

Find a needle in a hay stack with S3 Select

[3]:
```
import awswrangler as wr

# Run S3 Select query against all objects for 2019 year to find trips starting from a␣
↪particular location
wr.s3.select_query(
    sql='SELECT * FROM s3object s where s."pickup_location_id" = 138',
    path="s3://ursa-labs-taxi-data/2019/",
    input_serialization="Parquet",
    input_serialization_params={},
    scan_range_chunk_size=32 * 1024 * 1024,
)
```

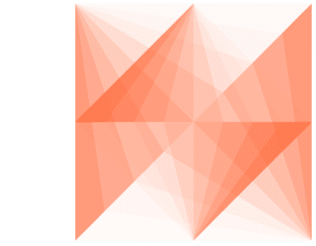
[3]:
```
        vendor_id                 pickup_at                dropoff_at  \
0               1  2019-01-01T00:19:55.000Z  2019-01-01T00:57:56.000Z
1               2  2019-01-01T00:48:10.000Z  2019-01-01T01:36:58.000Z
2               1  2019-01-01T00:39:58.000Z  2019-01-01T00:58:58.000Z
3               1  2019-01-01T00:07:45.000Z  2019-01-01T00:34:12.000Z
4               2  2019-01-01T00:27:40.000Z  2019-01-01T00:52:15.000Z
...           ...                       ...                       ...
1167508         2  2019-06-30T23:42:24.000Z  2019-07-01T00:10:28.000Z
1167509         2  2019-06-30T23:07:34.000Z  2019-06-30T23:25:09.000Z
1167510         2  2019-06-30T23:00:36.000Z  2019-06-30T23:20:18.000Z
1167511         1  2019-06-30T23:08:06.000Z  2019-06-30T23:30:20.000Z
1167512         2  2019-06-30T23:15:13.000Z  2019-06-30T23:35:18.000Z

         passenger_count  trip_distance rate_code_id store_and_fwd_flag  \
0                      1          12.30            1                  N
1                      1          31.57            1                  N
2                      2           8.90            1                  N
3                      4           9.60            1                  N
4                      1          12.89            1                  N
...                  ...            ...          ...                ...
1167508                1          15.66            1                  N
1167509                1           7.38            1                  N
1167510                1          11.24            1                  N
1167511                1           7.50            1                  N
1167512                2           8.73            1                  N

         pickup_location_id  dropoff_location_id payment_type  fare_amount  \
0                       138                   50            1         38.0
1                       138                  138            2         82.5
2                       138                  224            1         26.0
3                       138                  239            1         29.0
4                       138                   87            2         36.0
...                     ...                  ...          ...          ...
1167508                 138                  265            2         44.0
1167509                 138                  262            1         22.0
1167510                 138                  107            1         31.0
1167511                 138                  229            1         24.0
1167512                 138                  262            1         25.5
```

(continues on next page)

(continued from previous page)

```
         extra  mta_tax  tip_amount  tolls_amount  improvement_surcharge  \
0          0.5      0.5        4.00          5.76                    0.3
1          0.5      0.5        0.00          0.00                    0.3
2          0.5      0.5        8.25          5.76                    0.3
3          0.5      0.5        7.20          5.76                    0.3
4          0.5      0.5        0.00          0.00                    0.3
...        ...      ...         ...           ...                    ...
1167508    0.5      0.5        0.00          0.00                    0.3
1167509    0.5      0.5        7.98          6.12                    0.3
1167510    0.5      0.5        8.18          6.12                    0.3
1167511    3.0      0.5        4.00          0.00                    0.3
1167512    0.5      0.5        1.77          6.12                    0.3

         total_amount  congestion_surcharge
0           49.060001                   NaN
1           83.800003                   NaN
2           41.310001                   NaN
3           43.259998                   NaN
4           37.299999                   NaN
...               ...                   ...
1167508     45.299999                   0.0
1167509     39.900002                   2.5
1167510     49.099998                   2.5
1167511     31.799999                   2.5
1167512     37.189999                   2.5

[1167513 rows x 18 columns]
```

[ ]:



AWS SDK for pandas

1.4.34 35 - Distributing Calls on Ray Remote Cluster

AWS SDK for pandas supports distribution of specific calls on a cluster of EC2s using ray.

Note that this tutorial creates a cluster of EC2 nodes which will incur a charge in your account. Please make sure to delete the cluster at the end.

Install the library

```
[ ]: !pip install "awswrangler[modin,ray]"
```

Configure and Build Ray Cluster on AWS

Build Prerequisite Infrastructure

Click on the link below to provision an AWS CloudFormation stack. It builds a security group and IAM instance profile for the Ray Cluster to use. A valid CIDR range (encompassing your local machine IP) and a VPC ID are required.

|befad84501f44e238b8c5aa107f3e77d|

Configure Ray Cluster Configuration

Start with a cluster configuration file (YAML).

```
[ ]: !touch config.yml
```

Replace all values to match your desired region, account number and name of resources deployed by the above CloudFormation Stack.

A limited set of AWS regions is currently supported (Python 3.8 and above). The example configuration below uses the AMI for `us-east-1`.

Then edit `config.yml` file with your custom configuration.

```
cluster_name: pandas-sdk-cluster

min_workers: 2
max_workers: 2

provider:
    type: aws
    region: us-east-1 # Change AWS region as necessary
    availability_zone: us-east-1a,us-east-1b,us-east-1c # Change as necessary
    security_group:
        GroupName: ray-cluster
    cache_stopped_nodes: False

available_node_types:
  ray.head.default:
    node_config:
      InstanceType: m4.xlarge
      IamInstanceProfile:
```

(continues on next page)

(continued from previous page)

```
        # Replace with your account id and profile name if you did not use the default␣
↪value
        Arn: arn:aws:iam::{ACCOUNT ID}:instance-profile/ray-cluster
      # Replace ImageId if using a different region / python version
      ImageId: ami-02ea7c238b7ba36af
      TagSpecifications:  # Optional tags
        - ResourceType: "instance"
          Tags:
              - Key: Platform
                Value: "ray"

  ray.worker.default:
      min_workers: 2
      max_workers: 2
      node_config:
        InstanceType: m4.xlarge
        IamInstanceProfile:
          # Replace with your account id and profile name if you did not use the default␣
↪value
          Arn: arn:aws:iam::{ACCOUNT ID}:instance-profile/ray-cluster
        # Replace ImageId if using a different region / python version
        ImageId: ami-02ea7c238b7ba36af
        TagSpecifications:  # Optional tags
          - ResourceType: "instance"
            Tags:
                - Key: Platform
                  Value: "ray"


setup_commands:
- pip install "awswrangler[modin,ray]"
```

Provision Ray Cluster

The command below creates a Ray cluster in your account based on the aforementioned config file. It consists of one head node and 2 workers (m4xlarge EC2s). The command takes a few minutes to complete.

[ ]:
```
!ray up -y config.yml
```

Once the cluster is up and running, we set the `RAY_ADDRESS` environment variable to the head node Ray Cluster Address

[ ]:
```
import os
import subprocess

head_node_ip = subprocess.check_output(["ray", "get-head-ip", "config.yml"]).decode("utf-
↪8").split("\n")[-2]
os.environ["RAY_ADDRESS"] = f"ray://{head_node_ip}:10001"
```

As a result, `awswrangler` API calls now run on the cluster, not on your local machine. The SDK detects the required dependencies for its distributed mode and parallelizes supported methods on the cluster.

```
[ ]: import modin.pandas as pd

     import awswrangler as wr

     print(f"Execution engine: {wr.engine.get()}")
     print(f"Memory format: {wr.memory_format.get()}")
```

Enter bucket Name

```
[ ]: bucket = "BUCKET_NAME"
```

Read & write some data at scale on the cluster

```
[ ]: # Read last 3 months of Taxi parquet compressed data (400 Mb)
     df = wr.s3.read_parquet(path="s3://ursa-labs-taxi-data/2018/1*.parquet")
     df["month"] = df["pickup_at"].dt.month

     # Write it back to S3 partitioned by month
     path = f"s3://{bucket}/taxi-data/"
     database = "ray_test"
     wr.catalog.create_database(name=database, exist_ok=True)
     table = "nyc_taxi"

     wr.s3.to_parquet(
         df=df,
         path=path,
         dataset=True,
         database=database,
         table=table,
         partition_cols=["month"],
     )
```

Read it back via Athena UNLOAD

The UNLOAD command distributes query processing in Athena to dump results in S3 which are then read in parallel into a dataframe

```
[ ]: unload_path = f"s3://{bucket}/unload/nyc_taxi/"

     # Athena UNLOAD requires that the S3 path is empty
     # Note that s3.delete_objects is also a distributed call
     wr.s3.delete_objects(unload_path)

     wr.athena.read_sql_query(
         f"SELECT * FROM {table}",
         database=database,
         ctas_approach=False,
         unload_approach=True,
         s3_output=unload_path,
     )
```

The EC2 cluster must be terminated or it will incur a charge.

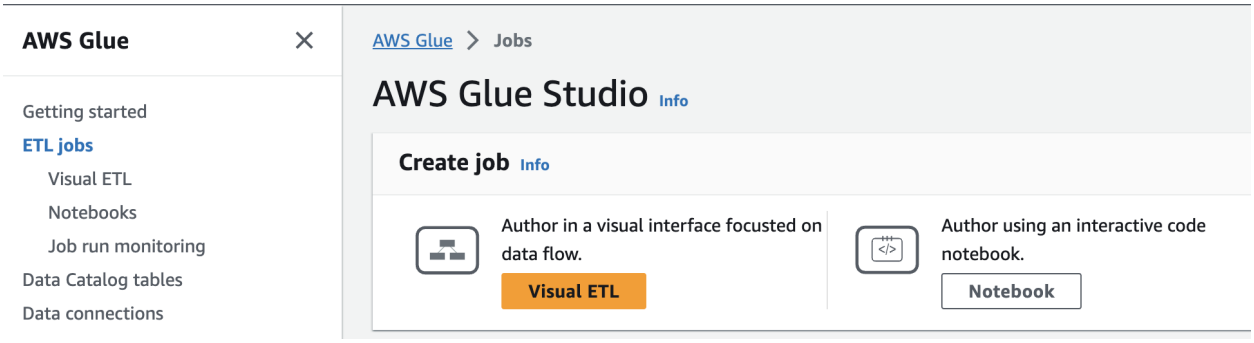
```
[ ]: !ray down -y ./config.yml
```

More Info on Ray Clusters on AWS

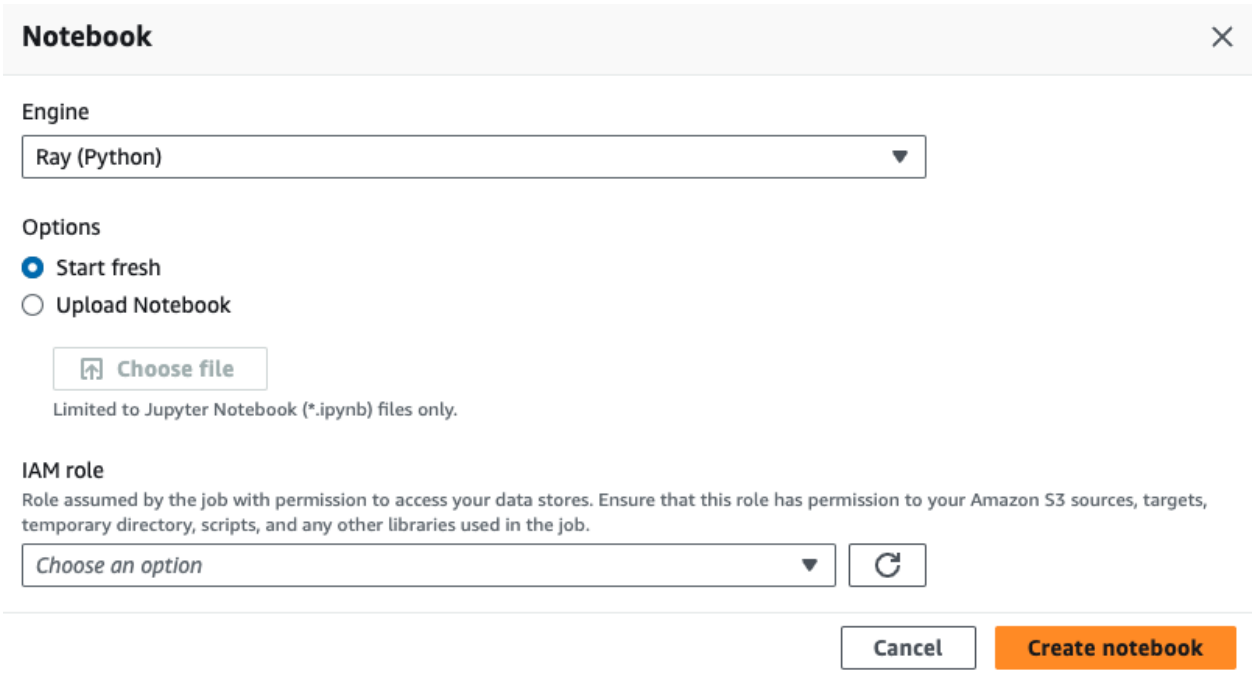
1.4.35 36 - Distributing Calls on Glue Interactive sessions

AWS SDK for pandas is pre-loaded into AWS Glue interactive sessions with Ray kernel, making it by far the easiest way to experiment with the library at scale.

In AWS Glue Studio, choose `Notebook` to create an AWS Glue interactive session:



Then select `Ray` as the kernel. The IAM role must trust the AWS Glue service principal.



Once the notebook is up and running you can import the library. You can install `awswrangler` and `modin` as additional dependencies.

[16]:
```
%additional_python_modules awswrangler,modin
```

```
Additional python modules to be included:
awswrangler
modin
```

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

```
Authenticating with environment variables and user-defined glue_role_arn: arn:aws:iam::
↪463623607974:role/service-role/AmazonSageMakerServiceCatalogProductsGlueRole
Trying to create a Glue session for the kernel.
Worker Type: Z.2X
Number of Workers: 5
Session ID: 32566e82-34d2-4db7-adac-cbee573e20bf
Job Type: glueray
Applying the following default arguments:
--glue_kernel_version 0.38.1
--enable-glue-datacatalog true
--auto-scaling-ray-min-workers 1
--additional-python-modules awswrangler,modin
Waiting for session 32566e82-34d2-4db7-adac-cbee573e20bf to get into ready status...
Session 32566e82-34d2-4db7-adac-cbee573e20bf has been created.
```

[8]:
```
df = wr.s3.read_parquet(path="s3://ursa-labs-taxi-data/2017/")
```

[9]:
```
df.head()
```

```
  vendor_id           pickup_at  ... improvement_surcharge  total_amount
0         1 2017-01-09 11:13:28  ...                    0.3     15.300000
```

(continues on next page)

(continued from previous page)

```
1          1 2017-01-09 11:32:27  ...                   0.3      7.250000
2          1 2017-01-09 11:38:20  ...                   0.3      7.300000
3          1 2017-01-09 11:52:13  ...                   0.3      8.500000
4          2 2017-01-01 00:00:00  ...                   0.3     52.799999

[5 rows x 17 columns]
```

To avoid incurring a charge, make sure to delete the Jupyter Notebook when you are done experimenting.

1.4.36 37 - 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 pandas as pd

import awswrangler as wr

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
8   Rule_9  ColumnValues "c2" in [0,1,2]   PASS
9  Rule_10        Uniqueness "c2" > 0.95   FAIL

                                          ResultId  \
0  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
1  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
2  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
3  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
4  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
5  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
6  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
7  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
8  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5
9  dqresult-be413b527c0e5520ad843323fecd9cf2e2edbdd5

                                EvaluationMessage
0                                             NaN
1                                             NaN
2                                             NaN
3                                             NaN
```

(continues on next page)

(continued from previous page)

```
4                                                   NaN
5                                                   NaN
6                                                   NaN
7                                                   NaN
8                                                   NaN
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"

dqdl_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,
    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(
```

(continues on next page)

(continued from previous page)

```
    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
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"])
```

(continues on next page)

(continued from previous page)

```
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
```

1.4.37 38 - 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.

[ ]:
```
# Install the optional modules first
!pip install 'awswrangler[opensearch]'
```

[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(),
        ],
    }
]
```

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",
```

(continues on next page)

(continued from previous page)

```
    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,
```

(continues on next page)

(continued from previous page)

```
    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:

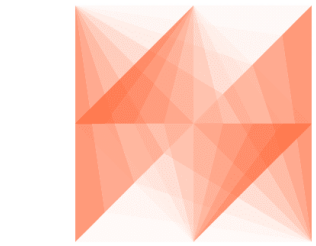
[27]:
```
wr.opensearch.search(client, index=index, search_body={"query": {"match": {"name": "Julia
↪"}}})
```

[27]:
```
  _id   name
0   3  Julia
```

Delete index

To delete an index, run:

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



AWS SDK for pandas

1.4.38 39 - Athena Iceberg

Athena supports read, time travel, write, and DDL queries for Apache Iceberg tables that use the Apache Parquet format for data and the AWS Glue catalog for their metastore. More in User Guide.

Create Iceberg table

[50]:
```
import getpass

bucket_name = getpass.getpass()
```

[2]:
```
import awswrangler as wr

glue_database = "aws_sdk_pandas"
glue_table = "iceberg_test"
path = f"s3://{bucket_name}/iceberg_test/"
temp_path = f"s3://{bucket_name}/iceberg_test_temp/"

# Cleanup table before create
wr.catalog.delete_table_if_exists(database=glue_database, table=glue_table)
```

[2]:
```
True
```

Create table & insert data

It is possible to insert Pandas data frame into Iceberg table using `wr.athena.to_iceberg`. If the table does not exist, it will be created:

[ ]:
```
import pandas as pd

df = pd.DataFrame({"id": [1, 2, 3], "name": ["John", "Lily", "Richard"]})

wr.athena.to_iceberg(
    df=df,
    database=glue_database,
    table=glue_table,
    table_location=path,
    temp_path=temp_path,
)
```

Alternatively, it is also possible to insert by directly running `INSERT INTO ... VALUES`:

[53]:
```
wr.athena.start_query_execution(
    sql=f"INSERT INTO {glue_table} VALUES (1,'John'), (2, 'Lily'), (3, 'Richard')",
    database=glue_database,
    wait=True,
)
```

[53]:
```
{'QueryExecutionId': 'e339fcd2-9db1-43ac-bb9e-9730e6395b51',
 'Query': "INSERT INTO iceberg_test VALUES (1,'John'), (2, 'Lily'), (3, 'Richard')",
 'StatementType': 'DML',
 'ResultConfiguration': {'OutputLocation': 's3://aws-athena-query-results-...-us-east-1/
↪e339fcd2-9db1-43ac-bb9e-9730e6395b51'},
 'ResultReuseConfiguration': {'ResultReuseByAgeConfiguration': {'Enabled': False}},
 'QueryExecutionContext': {'Database': 'aws_sdk_pandas'},
 'Status': {'State': 'SUCCEEDED',
  'SubmissionDateTime': datetime.datetime(2023, 3, 16, 10, 40, 8, 612000,␣
↪tzinfo=tzlocal()),
```

(continues on next page)

(continued from previous page)

```
 'CompletionDateTime': datetime.datetime(2023, 3, 16, 10, 40, 11, 143000,␣
↪tzinfo=tzlocal())},
 'Statistics': {'EngineExecutionTimeInMillis': 2242,
  'DataScannedInBytes': 0,
  'DataManifestLocation': 's3://aws-athena-query-results-...-us-east-1/e339fcd2-9db1-
↪43ac-bb9e-9730e6395b51-manifest.csv',
  'TotalExecutionTimeInMillis': 2531,
  'QueryQueueTimeInMillis': 241,
  'QueryPlanningTimeInMillis': 179,
  'ServiceProcessingTimeInMillis': 48,
  'ResultReuseInformation': {'ReusedPreviousResult': False}},
 'WorkGroup': 'primary',
 'EngineVersion': {'SelectedEngineVersion': 'Athena engine version 3',
  'EffectiveEngineVersion': 'Athena engine version 3'}}
```

[54]:
```
wr.athena.start_query_execution(
    sql=f"INSERT INTO {glue_table} VALUES (4,'Anne'), (5, 'Jacob'), (6, 'Leon')",
    database=glue_database,
    wait=True,
)
```

[54]:
```
{'QueryExecutionId': '922c8f02-4c00-4050-b4a7-7016809efa2b',
 'Query': "INSERT INTO iceberg_test VALUES (4,'Anne'), (5, 'Jacob'), (6, 'Leon')",
 'StatementType': 'DML',
 'ResultConfiguration': {'OutputLocation': 's3://aws-athena-query-results-...-us-east-1/
↪922c8f02-4c00-4050-b4a7-7016809efa2b'},
 'ResultReuseConfiguration': {'ResultReuseByAgeConfiguration': {'Enabled': False}},
 'QueryExecutionContext': {'Database': 'aws_sdk_pandas'},
 'Status': {'State': 'SUCCEEDED',
  'SubmissionDateTime': datetime.datetime(2023, 3, 16, 10, 40, 24, 582000,␣
↪tzinfo=tzlocal()),
  'CompletionDateTime': datetime.datetime(2023, 3, 16, 10, 40, 27, 352000,␣
↪tzinfo=tzlocal())},
 'Statistics': {'EngineExecutionTimeInMillis': 2414,
  'DataScannedInBytes': 0,
  'DataManifestLocation': 's3://aws-athena-query-results-...-us-east-1/922c8f02-4c00-
↪4050-b4a7-7016809efa2b-manifest.csv',
  'TotalExecutionTimeInMillis': 2770,
  'QueryQueueTimeInMillis': 329,
  'QueryPlanningTimeInMillis': 189,
  'ServiceProcessingTimeInMillis': 27,
  'ResultReuseInformation': {'ReusedPreviousResult': False}},
 'WorkGroup': 'primary',
 'EngineVersion': {'SelectedEngineVersion': 'Athena engine version 3',
  'EffectiveEngineVersion': 'Athena engine version 3'}}
```

Query

[65]:
```
wr.athena.read_sql_query(
    sql=f'SELECT * FROM "{glue_table}"',
    database=glue_database,
    ctas_approach=False,
    unload_approach=False,
)
```

[65]:
```
   id     name
0   1     John
1   4     Anne
2   2     Lily
3   3  Richard
4   5    Jacob
5   6     Leon
```

Read query metadata

In a SELECT query, you can use the following properties after `table_name` to query Iceberg table metadata:

- `$files` Shows a table's current data files
- `$manifests` Shows a table's current file manifests
- `$history` Shows a table's history
- `$partitions` Shows a table's current partitions

[55]:
```
wr.athena.read_sql_query(
    sql=f'SELECT * FROM "{glue_table}$files"',
    database=glue_database,
    ctas_approach=False,
    unload_approach=False,
)
```

[55]:
```
   content                                          file_path file_format  \
0        0  s3://.../iceberg_test/data/089a...     PARQUET
1        0  s3://.../iceberg_test/data/5736...     PARQUET

   record_count  file_size_in_bytes  column_sizes value_counts  \
0             3                 360  {1=48, 2=63}   {1=3, 2=3}
1             3                 355  {1=48, 2=61}   {1=3, 2=3}

  null_value_counts nan_value_counts   lower_bounds      upper_bounds  \
0        {1=0, 2=0}               {}  {1=1, 2=John}  {1=3, 2=Richard}
1        {1=0, 2=0}               {}  {1=4, 2=Anne}     {1=6, 2=Leon}

  key_metadata split_offsets equality_ids
0         <NA>           NaN          NaN
1         <NA>           NaN          NaN
```

[56]:
```
wr.athena.read_sql_query(
    sql=f'SELECT * FROM "{glue_table}$manifests"',
```

(continues on next page)

(continued from previous page)

```
    database=glue_database,
    ctas_approach=False,
    unload_approach=False,
)
```

[56]:
```
                                                path  length  \
0  s3://.../iceberg_test/metadata/...    6538
1  s3://.../iceberg_test/metadata/...    6548

   partition_spec_id    added_snapshot_id  added_data_files_count  \
0                  0  4379263637983206651                       1
1                  0  2934717851675145063                       1

   added_rows_count  existing_data_files_count  existing_rows_count  \
0                 3                          0                    0
1                 3                          0                    0

   deleted_data_files_count  deleted_rows_count partitions
0                         0                  0         []
1                         0                  0         []
```

[58]:
```
df = wr.athena.read_sql_query(
    sql=f'SELECT * FROM "{glue_table}$history"',
    database=glue_database,
    ctas_approach=False,
    unload_approach=False,
)

# Save snapshot id
snapshot_id = df.snapshot_id[0]

df
```

[58]:
```
                  made_current_at          snapshot_id            parent_id  \
0 2023-03-16 09:40:10.438000+00:00  2934717851675145063                 <NA>
1 2023-03-16 09:40:26.754000+00:00  4379263637983206651  2934717851675144704

   is_current_ancestor
0                 True
1                 True
```

[59]:
```
wr.athena.read_sql_query(
    sql=f'SELECT * FROM "{glue_table}$partitions"',
    database=glue_database,
    ctas_approach=False,
    unload_approach=False,
)
```

[59]:
```
   record_count  file_count  total_size  \
0             6           2         715

                                                data
0  {id={min=1, max=6, null_count=0, nan_count=nul...
```

Time travel

[60]:
```
wr.athena.read_sql_query(
    sql=f"SELECT * FROM {glue_table} FOR TIMESTAMP AS OF (current_timestamp - interval '5
↪' second)",
    database=glue_database,
)
```

[60]:
```
   id     name
0   1     John
1   4     Anne
2   2     Lily
3   3  Richard
4   5    Jacob
5   6     Leon
```

Version travel

[61]:
```
wr.athena.read_sql_query(
    sql=f"SELECT * FROM {glue_table} FOR VERSION AS OF {snapshot_id}",
    database=glue_database,
)
```

[61]:
```
   id     name
0   1     John
1   2     Lily
2   3  Richard
```

Optimize

The `OPTIMIZE table REWRITE DATA` compaction action rewrites data files into a more optimized layout based on their size and number of associated delete files. For syntax and table property details, see OPTIMIZE.

[62]:
```
wr.athena.start_query_execution(
    sql=f"OPTIMIZE {glue_table} REWRITE DATA USING BIN_PACK",
    database=glue_database,
    wait=True,
)
```

[62]:
```
{'QueryExecutionId': '94666790-03ae-42d7-850a-fae99fa79a68',
 'Query': 'OPTIMIZE iceberg_test REWRITE DATA USING BIN_PACK',
 'StatementType': 'DDL',
 'ResultConfiguration': {'OutputLocation': 's3://aws-athena-query-results-...-us-east-1/
↪tables/94666790-03ae-42d7-850a-fae99fa79a68'},
 'ResultReuseConfiguration': {'ResultReuseByAgeConfiguration': {'Enabled': False}},
 'QueryExecutionContext': {'Database': 'aws_sdk_pandas'},
 'Status': {'State': 'SUCCEEDED',
  'SubmissionDateTime': datetime.datetime(2023, 3, 16, 10, 49, 42, 857000,␣
↪tzinfo=tzlocal()),
  'CompletionDateTime': datetime.datetime(2023, 3, 16, 10, 49, 45, 655000,␣
↪tzinfo=tzlocal())},
 'Statistics': {'EngineExecutionTimeInMillis': 2622,
```

(continues on next page)

(continued from previous page)

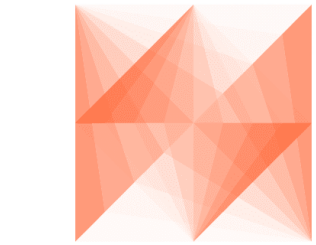
```
  'DataScannedInBytes': 220,
  'DataManifestLocation': 's3://aws-athena-query-results-...-us-east-1/tables/94666790-
↪03ae-42d7-850a-fae99fa79a68-manifest.csv',
  'TotalExecutionTimeInMillis': 2798,
  'QueryQueueTimeInMillis': 124,
  'QueryPlanningTimeInMillis': 252,
  'ServiceProcessingTimeInMillis': 52,
  'ResultReuseInformation': {'ReusedPreviousResult': False}},
 'WorkGroup': 'primary',
 'EngineVersion': {'SelectedEngineVersion': 'Athena engine version 3',
  'EffectiveEngineVersion': 'Athena engine version 3'}}
```

Vacuum

`VACUUM` performs snapshot expiration and orphan file removal. These actions reduce metadata size and remove files not in the current table state that are also older than the retention period specified for the table. For syntax details, see VACUUM.

[64]:
```
wr.athena.start_query_execution(
    sql=f"VACUUM {glue_table}",
    database=glue_database,
    wait=True,
)
```

[64]:
```
{'QueryExecutionId': '717a7de6-b873-49c7-b744-1b0b402f24c9',
 'Query': 'VACUUM iceberg_test',
 'StatementType': 'DML',
 'ResultConfiguration': {'OutputLocation': 's3://aws-athena-query-results-...-us-east-1/
↪717a7de6-b873-49c7-b744-1b0b402f24c9.csv'},
 'ResultReuseConfiguration': {'ResultReuseByAgeConfiguration': {'Enabled': False}},
 'QueryExecutionContext': {'Database': 'aws_sdk_pandas'},
 'Status': {'State': 'SUCCEEDED',
  'SubmissionDateTime': datetime.datetime(2023, 3, 16, 10, 50, 41, 14000,␣
↪tzinfo=tzlocal()),
  'CompletionDateTime': datetime.datetime(2023, 3, 16, 10, 50, 43, 441000,␣
↪tzinfo=tzlocal())},
 'Statistics': {'EngineExecutionTimeInMillis': 2229,
  'DataScannedInBytes': 0,
  'TotalExecutionTimeInMillis': 2427,
  'QueryQueueTimeInMillis': 153,
  'QueryPlanningTimeInMillis': 30,
  'ServiceProcessingTimeInMillis': 45,
  'ResultReuseInformation': {'ReusedPreviousResult': False}},
 'WorkGroup': 'primary',
 'EngineVersion': {'SelectedEngineVersion': 'Athena engine version 3',
  'EffectiveEngineVersion': 'Athena engine version 3'}}
```



1.4.39 40 - EMR Serverless

Amazon EMR Serverless is a new deployment option for Amazon EMR. EMR Serverless provides a serverless runtime environment that simplifies the operation of analytics applications that use the latest open source frameworks, such as Apache Spark and Apache Hive. With EMR Serverless, you don't have to configure, optimize, secure, or operate clusters to run applications with these frameworks. More in User Guide.

Spark

Create a Spark application

[1]:
```
import awswrangler as wr

spark_application_id: str = wr.emr_serverless.create_application(
    name="my-spark-application",
    application_type="Spark",
    release_label="emr-6.10.0",
)
```

```
/var/folders/_n/7dm3ff5d5fb01gjt6ms150km0000gs/T/ipykernel_11468/3968622978.py:3:␣
↪SDKPandasExperimentalWarning: `create_application`: This API is experimental and may␣
↪change in future AWS SDK for Pandas releases.
  spark_application_id: str = wr.emr_serverless.create_application(
```

Run a Spark job

[ ]:
```
iam_role_arn = "arn:aws:iam::...:role/..."

wr.emr_serverless.run_job(
    application_id=spark_application_id,
    execution_role_arn=iam_role_arn,
    job_driver_args={
        "entryPoint": "/usr/lib/spark/examples/jars/spark-examples.jar",
        "entryPointArguments": ["1"],
        "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi --conf spark.
↪executor.cores=4 --conf spark.executor.memory=20g --conf spark.driver.cores=4 --conf␣
↪spark.driver.memory=8g --conf spark.executor.instances=1",
```

(continues on next page)

(continued from previous page)

```
    },
    job_type="Spark",
)
```

Hive

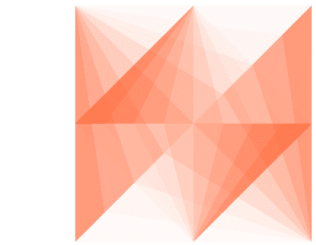
Create a Hive application

[2]:
```
hive_application_id: str = wr.emr_serverless.create_application(
    name="my-hive-application",
    application_type="Hive",
    release_label="emr-6.10.0",
)
```

```
/var/folders/_n/7dm3ff5d5fb01gjt6ms150km0000gs/T/ipykernel_11468/3826130602.py:1:␣
↪SDKPandasExperimentalWarning: `create_application`: This API is experimental and may␣
↪change in future AWS SDK for Pandas releases.
  hive_application_id: str = wr.emr_serverless.create_application(
```

Run a Hive job

[ ]:
```
path = "s3://my-bucket/path"

wr.emr_serverless.run_job(
    application_id=hive_application_id,
    execution_role_arn="arn:aws:iam::...:role/...",
    job_driver_args={
        "query": f"{path}/hive-query.ql",
        "parameters": f"--hiveconf hive.exec.scratchdir={path}/scratch --hiveconf hive.
↪metastore.warehouse.dir={path}/warehouse",
    },
    job_type="Hive",
)
```



AWS SDK for pandas

1.4.40 41 - Apache Spark on Amazon Athena

Amazon Athena makes it easy to interactively run data analytics and exploration using Apache Spark without the need to plan for, configure, or manage resources. Running Apache Spark applications on Athena means submitting Spark code for processing and receiving the results directly without the need for additional configuration.

More in User Guide.

Run a Spark calculation

For this tutorial, you will need Spark-enabled Athena Workgroup. For the steps to create one, visit Getting started with Apache Spark on Amazon Athena.

[ ]:
```
import awswrangler as wr

workgroup: str = "my-spark-workgroup"

result = wr.athena.run_spark_calculation(
    code="print(spark)",
    workgroup=workgroup,
)
```

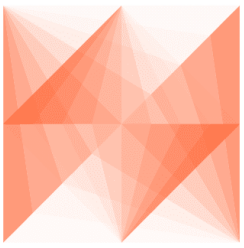
Create and re-use a session

It is possible to create a session and re-use it launching multiple calculations with the same resources. To create a session, use:

[ ]:
```
session_id: str = wr.athena.create_spark_session(
    workgroup=workgroup,
)
```

Now, to use the session, pass `session_id`:

[ ]:
```
result = wr.athena.run_spark_calculation(
    code="print(spark)",
    workgroup=workgroup,
    session_id=session_id,
)
```



AWS SDK for pandas

1.4.41 42 - Amazon S3 Tables

Amazon S3 Tables provide analytics-optimized tabular storage using Apache Iceberg format. S3 Tables introduce **table buckets**, **namespaces**, and **tables**.

AWS SDK for pandas supports S3 Tables through the `wr.s3` module. Read and write operations require the `pyiceberg` optional dependency:

```
pip install awswrangler[pyiceberg]
```

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

[18]:
```
import getpass

import pandas as pd

import awswrangler as wr
```

[19]:
```
bucket_name = getpass.getpass("Enter a table bucket name:")
```

Creating resources

Create a Table Bucket

[ ]:
```
bucket_arn = wr.s3.create_table_bucket(name=bucket_name)
print(f"Table bucket ARN: {bucket_arn}")
```

Create a Namespace

[ ]:
```
namespace = "tutorial"

wr.s3.create_namespace(
    table_bucket_arn=bucket_arn,
    namespace=namespace,
)
```

Write

Writing a DataFrame

`to_iceberg` automatically creates the table if it does not exist.

[ ]:
```
df = pd.DataFrame(
    {
        "order_id": [1, 2, 3],
        "amount": [10.50, 20.00, 15.75],
        "region": ["us", "eu", "us"],
    }
```

(continues on next page)

(continued from previous page)

```
)

wr.s3.to_iceberg(
    df=df,
    table_bucket_arn=bucket_arn,
    namespace=namespace,
    table_name="orders",
)
```

Appending data

[ ]:
```
df_new = pd.DataFrame(
    {
        "order_id": [4, 5],
        "amount": [30.00, 12.25],
        "region": ["eu", "us"],
    }
)

wr.s3.to_iceberg(
    df=df_new,
    table_bucket_arn=bucket_arn,
    namespace=namespace,
    table_name="orders",
    mode="append",
)
```

Overwriting data

[ ]:
```
df_replace = pd.DataFrame(
    {
        "order_id": [100, 200],
        "amount": [99.99, 49.99],
        "region": ["ap", "ap"],
    }
)

wr.s3.to_iceberg(
    df=df_replace,
    table_bucket_arn=bucket_arn,
    namespace=namespace,
    table_name="orders",
    mode="overwrite",
)
```

Read

Read entire table

```
[ ]: df = wr.s3.from_iceberg(
         table_bucket_arn=bucket_arn,
         namespace=namespace,
         table_name="orders",
     )
     df
```

Column selection and row filtering

```
[ ]: df = wr.s3.from_iceberg(
         table_bucket_arn=bucket_arn,
         namespace=namespace,
         table_name="orders",
         columns=["order_id", "amount"],
         row_filter="amount > 50.0",
     )
     df
```

Limiting rows

```
[ ]: df = wr.s3.from_iceberg(
         table_bucket_arn=bucket_arn,
         namespace=namespace,
         table_name="orders",
         limit=1,
     )
     df
```

Using the AWS Glue Iceberg REST endpoint

By default, read and write operations use the S3 Tables REST endpoint. To use the AWS Glue Iceberg REST endpoint instead, set `wr.config.s3tables_catalog_endpoint_url`. This enables integration with services that work through the Glue Data Catalog (e.g. Amazon Athena, Amazon Redshift).

Prerequisites

Before using the Glue endpoint, your table bucket must be integrated with the AWS Glue Data Catalog. This requires:

1. **An IAM role for Lake Formation** with `s3tables:*` permissions and a trust policy allowing `lakeformation.amazonaws.com` to assume it.
2. **A Lake Formation resource registration** for `arn:aws:s3tables:<region>:<account>:bucket/*` with `WithFederation=True` and `HybridAccessEnabled=True`.
3. **A Glue federated catalog** named `s3tablescatalog` linked to S3 Tables via the `aws:s3tables` connection.

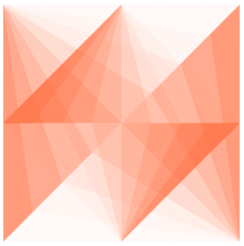
4. **Lake Formation permissions** granting the caller access to the catalog, databases, and tables.

For step-by-step instructions, see Integrating S3 Tables with AWS analytics services.

[ ]:
```
# Point read/write at the Glue Iceberg REST endpoint
wr.config.s3tables_catalog_endpoint_url = "https://glue.<region>.amazonaws.com/iceberg"

df = wr.s3.from_iceberg(
    table_bucket_arn=bucket_arn,
    namespace=namespace,
    table_name="orders",
)

# Reset to default (S3 Tables endpoint)
wr.config.s3tables_catalog_endpoint_url = None
```

Deleting resources

[ ]:
```
wr.s3.delete_table(
    table_bucket_arn=bucket_arn,
    namespace=namespace,
    table_name="orders",
)
wr.s3.delete_namespace(
    table_bucket_arn=bucket_arn,
    namespace=namespace,
)
wr.s3.delete_table_bucket(
    table_bucket_arn=bucket_arn,
)
```



AWS SDK for pandas

1.4.42 43 - Amazon S3 Vectors

Amazon S3 Vectors provides cost-optimized native vector storage on S3 for similarity search and RAG. The hierarchy is **vector bucket** → **vector index** → **vectors**, where each vector is a tuple of `(key, float32[], metadata)`.

AWS SDK for pandas wraps the `s3vectors` boto3 service and exposes 14 functions on `wr.s3` covering the full bucket / index / data lifecycle, with DataFrame-friendly I/O and optional on-the-fly embedding via Amazon Bedrock.

[1]:
```
import getpass

import pandas as pd

import awswrangler as wr
```

[4]:
```
bucket_name = getpass.getpass("Enter a vector bucket name:")
index_name = "tutorial"
# Match the embedding model's output dimension. Titan v2 supports 256, 512, or 1024.
dimension = 256
```

Creating resources

A vector bucket holds many indexes; an index has a fixed dimension and distance metric and holds the actual vectors. All vectors written to an index must match its dimension.

[5]:
```
bucket_arn = wr.s3.create_vector_bucket(name=bucket_name)
print(f"Vector bucket ARN: {bucket_arn}")
```

```
Vector bucket ARN: arn:aws:s3vectors:eu-west-1:123456789012:bucket/test-aws-sdk-pandas-
↪s3-vectors-1
```

[6]:
```
index_arn = wr.s3.create_vector_index(
    vector_bucket=bucket_name,
    name=index_name,
    dimension=dimension,
    distance_metric="cosine",
)
print(f"Vector index ARN: {index_arn}")
```

```
Vector index ARN: arn:aws:s3vectors:eu-west-1:123456789012:bucket/test-aws-sdk-pandas-s3-
↪vectors-1/index/tutorial
```

Discovery

`list_vector_buckets`, `get_vector_bucket`, `list_vector_indexes`, and `get_vector_index` describe what's in an account / bucket. List operations paginate internally.

[7]:
```
print("Buckets:", [b["vectorBucketName"] for b in wr.s3.list_vector_buckets()])
print("This bucket:", wr.s3.get_vector_bucket(name=bucket_name))
print("Indexes:", [i["indexName"] for i in wr.s3.list_vector_indexes(vector_
↪bucket=bucket_name)])
wr.s3.get_vector_index(name=index_name, vector_bucket=bucket_name)
```

```
Buckets: ['test-aws-sdk-pandas-s3-vectors-1']
This bucket: {'vectorBucketName': 'test-aws-sdk-pandas-s3-vectors-1', 'vectorBucketArn':
↪'arn:aws:s3vectors:eu-west-1:123456789012:bucket/test-aws-sdk-pandas-s3-vectors-1',
↪'creationTime': datetime.datetime(2026, 5, 10, 0, 47, 57, tzinfo=tzlocal()),
↪'encryptionConfiguration': {'sseType': 'AES256'}}
Indexes: ['tutorial']
```

[7]:
```
{'vectorBucketName': 'test-aws-sdk-pandas-s3-vectors-1',
 'indexName': 'tutorial',
 'indexArn': 'arn:aws:s3vectors:eu-west-1:123456789012:bucket/test-aws-sdk-pandas-s3-
↪vectors-1/index/tutorial',
 'creationTime': datetime.datetime(2026, 5, 10, 0, 48, tzinfo=tzlocal()),
 'dataType': 'float32',
 'dimension': 256,
 'distanceMetric': 'cosine',
 'encryptionConfiguration': {'sseType': 'AES256'}}
```

Writing vectors

`put_vectors_from_df` is the DataFrame entry point. The most natural workflow is to pass `text_column` together with a Bedrock embedding model — awswrangler then calls Bedrock for each row and writes the resulting vectors plus any other columns as metadata.

> **Prerequisite:** the calling identity needs `bedrock:InvokeModel` permission and **Bedrock model access** for the chosen embedding model in the current region (request access in the Bedrock console). Supported model id prefixes: `amazon.titan-embed-text-*`, `cohere.embed-*`.

If you already have embeddings (computed by your own model or a different SDK), pass them as a column instead via `vector_column`:

```
wr.s3.put_vectors_from_df(
    df=my_df,
    key_column="id",
    vector_column="embedding",  # precomputed list[float] / np.ndarray per row
    vector_bucket=bucket_name,
    index=index_name,
)
```

[9]:
```
df = pd.DataFrame(
    {
        "id": ["m-1", "m-2", "m-3", "m-4", "m-5"],
        "title": [
            "A wildlife photographer documents the rebirth of a forest after a wildfire.
↪",
            "An investigative reporter uncovers fraud at a multinational bank.",
            "A coming-of-age comedy about siblings inheriting a struggling family bakery.
↪",
            "A heart-warming tale of an elderly widower and a stray dog.",
            "A documentary tracing the history of jazz from New Orleans to Tokyo.",
        ],
        "genre": ["documentary", "drama", "comedy", "drama", "documentary"],
        "year": [2022, 2019, 2024, 2023, 2018],
    }
```

(continues on next page)

(continued from previous page)

```
)
df
```

[9]:

	id	title	genre	year
0	m-1	A wildlife photographer documents the rebirth ...	documentary	2022
1	m-2	An investigative reporter uncovers fraud at a ...	drama	2019
2	m-3	A coming-of-age comedy about siblings inheriti...	comedy	2024
3	m-4	A heart-warming tale of an elderly widower and...	drama	2023
4	m-5	A documentary tracing the history of jazz from...	documentary	2018

[10]:
```
wr.s3.put_vectors_from_df(
    df=df,
    key_column="id",
    text_column="title",
    bedrock_model_id="amazon.titan-embed-text-v2:0",
    bedrock_model_kwargs={"dimensions": dimension},
    vector_bucket=bucket_name,
    index=index_name,
)
```

Querying by text

`query_vectors` runs an approximate-nearest-neighbour search. Pass `query_text` + `bedrock_model_id` to embed the query on the fly, or `query_vector` to query with a precomputed embedding (shown later).

[11]:
```
wr.s3.query_vectors(
    query_text="a touching story about loneliness and companionship",
    bedrock_model_id="amazon.titan-embed-text-v2:0",
    bedrock_model_kwargs={"dimensions": dimension},
    top_k=3,
    return_distance=True,
    return_metadata=True,
    vector_bucket=bucket_name,
    index=index_name,
)
```

[11]:

	key	distance	metadata
0	m-4	0.560630	{'year': 2023, 'genre': 'drama'}
1	m-2	0.701137	{'genre': 'drama', 'year': 2019}
2	m-5	0.751206	{'year': 2018, 'genre': 'documentary'}

Filtering on metadata

Filters use MongoDB-style operators (`$eq`, `$ne`, `$gt`, `$gte`, `$lt`, `$lte`, `$in`, `$nin`, `$exists`, `$and`, `$or`). They are evaluated *during* the search, not as a post-filter.

[12]:
```
wr.s3.query_vectors(
    query_text="music history",
    bedrock_model_id="amazon.titan-embed-text-v2:0",
    bedrock_model_kwargs={"dimensions": dimension},
```

(continues on next page)

(continued from previous page)

```
    top_k=5,
    filter={"$and": [{"genre": {"$eq": "documentary"}}, {"year": {"$gte": 2020}}]},
    return_metadata=True,
    vector_bucket=bucket_name,
    index=index_name,
)
```

[12]:
```
   key  distance                                  metadata
0  m-1  0.776892  {'year': 2022, 'genre': 'documentary'}
```

Working with vectors directly

`get_vectors` retrieves vectors by key, optionally returning the embedding data and/or metadata. Useful for inspection, re-querying, or exporting subsets.

[13]:
```
fetched = wr.s3.get_vectors(
    keys=["m-1", "m-3"],
    return_data=True,
    return_metadata=True,
    vector_bucket=bucket_name,
    index=index_name,
)
fetched
```

[13]:
```
   key                                              vector  \
0  m-3  [-0.03277716785669327, 0.10634636878967285, 0...
1  m-1  [-0.12095249444246292, 0.10887987911701202, 0...

                              metadata
0       {'year': 2024, 'genre': 'comedy'}
1  {'year': 2022, 'genre': 'documentary'}
```

Querying with a precomputed vector

Any `list[float]` / `np.ndarray` of the right dimension works. Here we re-use a vector we just fetched — in practice, this is where you'd plug in embeddings from your own model.

[14]:
```
wr.s3.query_vectors(
    query_vector=fetched.iloc[0]["vector"],
    top_k=3,
    return_distance=True,
    return_metadata=True,
    vector_bucket=bucket_name,
    index=index_name,
)
```

[14]:
```
   key  distance                            metadata
0  m-3  0.000384  {'genre': 'comedy', 'year': 2024}
1  m-2  0.733959   {'genre': 'drama', 'year': 2019}
2  m-4  0.755624   {'year': 2023, 'genre': 'drama'}
```

Deleting vectors by key

[15]:
```
wr.s3.delete_vectors(
    keys=["m-3"],
    vector_bucket=bucket_name,
    index=index_name,
)
```

Bulk export

`list_vectors` walks the entire index. With `use_threads=True` it parallelises across up to 16 segments under the hood; pass `return_data=True` / `return_metadata=True` to include the full payload.

[16]:
```
wr.s3.list_vectors(
    return_metadata=True,
    vector_bucket=bucket_name,
    index=index_name,
    use_threads=4,
)
```

[16]:

	key	metadata
0	m-1	{'year': 2022, 'genre': 'documentary'}
1	m-5	{'genre': 'documentary', 'year': 2018}
2	m-4	{'year': 2023, 'genre': 'drama'}
3	m-2	{'genre': 'drama', 'year': 2019}

Cleanup

[17]:
```
wr.s3.delete_vector_index(name=index_name, vector_bucket=bucket_name)
wr.s3.delete_vector_bucket(name=bucket_name)
```

1.5 Architectural Decision Records

A collection of records for "architecturally significant" decisions: those that affect the structure, non-functional characteristics, dependencies, interfaces, or construction techniques.

These decisions are made by the team which maintains *AWS SDK for pandas*. However, suggestions can be submitted by any contributor via issues or pull requests.

Note: You can also find all ADRs on GitHub.

1.5.1 1. Record architecture decisions

Date: 2023-03-08

Status

Accepted

Context

We need to record the architectural decisions made on this project.

Decision

We will use Architecture Decision Records, as described by Michael Nygard.

Consequences

See Michael Nygard's article, linked above. For a lightweight ADR toolset, see Nat Pryce's adr-tools.

1.5.2 2. Handling unsupported arguments in distributed mode

Date: 2023-03-09

Status

Accepted

Context

Many of the API functions allow the user to pass their own `boto3` session, which will then be used by all the underlying `boto3` calls. With distributed computing, one of the limitations we have is that we cannot pass the `boto3` session to the worker nodes.

Boto3 session are not thread-safe, and therefore cannot be passed to Ray workers. The credentials behind a `boto3` session cannot be sent to Ray workers either, since sending credentials over the network is considered a security risk.

This raises the question of what to do when, in distributed mode, the customer passes arguments that are normally supported, but aren't supported in distributed mode.

Decision

When a user passes arguments that are unsupported by distributed mode, the function should fail immediately.

The main alternative to this approach would be if a parameter such as a `boto3` session is passed, we should use it where possible. This could result in a situation where, when reading Parquet files from S3, the process of listing the files uses the `boto3` session whereas the reading of the Parquet files doesn't. This could result in inconsistent behavior, as part of the function uses the extra parameters while the other part of it doesn't.

Another alternative would simply be to ignore the unsupported parameters, while potentially outputting a warning. The main issue with this approach is that if a customer tells our API functions to use certain parameters, they expect those parameters to be used. By ignoring them, the the AWS SDK for pandas API would be doing something different from what the customer asked, without properly notifying them, and would thus lose the customer's trust.

Consequences

In PR#2501, the `validate_distributed_kwargs` annotation was introduced which can check for the presence of arguments that are unsupported in the distributed mode.

The annotation has also been applied for arguments such as `s3_additional_kwargs` and `version_id` when reading/writing data on S3.

1.5.3 3. Use TypedDict to group similar parameters

Date: 2023-03-10

Status

Accepted

Context

AWS SDK for pandas API methods contain many parameters which are related to a specific behaviour or setting. For example, methods which have an option to update the Glue AWScatalog, such as `to_csv` and `to_parquet`, contain a list of parameters that define the settings for the table in AWS Glue. These settings include the table description, column comments, the table type, etc.

As a consequence, some of our functions have grown to include dozens of parameters. When reading the function signatures, it can be unclear which parameters are related to which functionality. For example, it's not immediately obvious that the parameter `column_comments` in `s3.to_parquet` only writes the column comments into the AWS Glue catalog, and not to S3.

Decision

Parameters that are related to similar functionality will be replaced by a single parameter of type TypedDict. This will allow us to reduce the amount of parameters for our API functions, and also make it clearer that certain parameters are only related to specific functionalities.

For example, parameters related to Athena cache settings will be extracted into a parameter of type `AthenaCacheSettings`, parameters related to Ray settings will be extracted into `RayReadParquetSettings`, etc.

The usage of `TypedDict` allows the user to define the parameters as regular dictionaries with string keys, while empowering type checkers such as `mypy`. Alternately, implementations such as `AthenaCacheSettings` can be instantiated as classes.

Alternatives

The main alternative that was considered was the idea of using `dataclass` instead of `TypedDict`. The advantage of this alternative would be that default values for parameters could be defined directly in the class signature, rather than needing to be defined in the function which uses the parameter.

On the other hand, the main issue with using `dataclass` is that it would require the customer figure out which class needs to be imported. With `TypedDict`, this is just one of the options; the parameters can simply be passed as a typical Python dictionary.

This alternative was discussed in more detail as part of PR#1855.

Consequences

Subclasses of `TypedDict` such as `GlueCatalogParameters`, `AthenaCacheSettings`, `AthenaUNLOADSettings`, `AthenaCTASSettings` and `RaySettings` have been created. They are defined in the `wrangler.typing` module.

These parameters grouping can used in either of the following two ways:

```
wr.athena.read_sql_query(
    "SELECT * FROM ...",,
    ctas_approach=True,
    athena_cache_settings={"max_cache_seconds": 900},
)

wr.athena.read_sql_query(
    "SELECT * FROM ...",,
    ctas_approach=True,
    athena_cache_settings=wr.typing.AthenaCacheSettings(
        max_cache_seconds=900,
    ),
)
```

Many of our functions signatures have been changes to take advantage of this refactor. Many of these are breaking changes which will be released as part of the next major version: `3.0.0`.

1.5.4 4. AWS SDK for pandas does not alter IAM permissions

Date: 2023-03-15

Status

Accepted

Context

AWS SDK for pandas requires permissions to execute AWS API calls. Permissions are granted using AWS Identity and Access Management Policies that are attached to IAM entities - users or roles.

Decision

AWS SDK for pandas does not alter (create, update, delete) IAM permissions policies attached to the IAM entities.

Consequences

It is users responsibility to ensure IAM entities they are using to execute the calls have the required permissions.

1.5.5 5. Move dependencies to optional

Date: 2023-03-15

Status

Accepted

Context

AWS SDK for pandas relies on external dependencies in some of its modules. These include `redshift-connector`, `gremlinpython` and `pymysql` to cite a few.

In versions 2.x and below, most of these packages were set as required, meaning they were installed regardless of whether the user actually needed them. This has introduced two major risks and issues as the number of dependencies increased:

1. **Security risk**: Unused dependencies increase the attack surface to manage. Users must scan them and ensure that they are kept up to date even though they don't need them
2. **Dependency hell**: Users must resolve dependencies for packages that they are not using. It can lead to dependency hell and prevent critical updates related to security patches and major bugs

Decision

A breaking change is introduced in version 3.x where the number of required dependencies is reduced to the most important ones, namely:

- boto3
- pandas
- numpy
- pyarrow
- typing-extensions

Consequences

All other dependencies are moved to optional and must be installed by the user separately using pip install `awswrangler[dependency]`. For instance, the command to use the redshift APIs is `pip install awswrangler[redshift]`. Failing to do so raises an exception informing the user that the package is missing and how to install it

1.5.6 6. Deprecate wr.s3.merge_upsert_table

Date: 2023-03-15

Status

Accepted

Context

AWS SDK for pandas `wr.s3.merge_upsert_table` is used to perform upsert (update else insert) onto an existing AWS Glue Data Catalog table. It is a much simplified version of upsert functionality that is supported natively by Apache Hudi and Athena Iceberg tables, and does not, for example, handle partitioned datasets.

Decision

To avoid poor user experience `wr.s3.merge_upsert_table` is deprecated and will be removed in 3.0 release.

Consequences

In PR#2076, `wr.s3.merge_upsert_table` function was removed.

1.5.7 7. Design of engine and memory format

Date: 2023-03-16

Status

Accepted

Context

Ray and Modin are the two frameworks used to support running `awswrangler` APIs at scale. Adding them to the codebase requires significant refactoring work. The original approach considered was to handle both distributed and non-distributed code within the same modules. This quickly turned out to be undesirable as it affected the readability, maintainability and scalability of the codebase.

Decision

Version 3.x of the library introduces two new constructs, `engine` and `memory_format`, which are designed to address the aforementioned shortcomings of the original approach, but also provide additional functionality.

Currently `engine` takes one of two values: `python` (default) or `ray`, but additional engines could be onboarded in the future. The value is determined at import based on installed dependencies. The user can override this value with `wr.engine.set("engine_name")`. Likewise, `memory_format` can be set to `pandas` (default) or `modin` and overridden with `wr.memory_format.set("memory_format_name")`.

A custom dispatcher is used to register functions based on the execution and memory format values. For instance, if the `ray` engine is detected at import, then methods distributed with Ray are used instead of the default AWS SDK for pandas code.

Consequences

The good:

Clear separation of concerns: Distributed methods live outside non-distributed code, eliminating ugly if conditionals, allowing both to scale independently and making them easier to maintain in the future

Better dispatching: Adding a new engine/memory format is as simple as creating a new directory with its methods and registering them with the custom dispatcher based on the value of the engine or memory format

Custom engine/memory format classes: Give more flexibility than config when it comes to interacting with the engine and managing its state (initialising, registering, get/setting…)

The bad:

Managing state: Adding a custom dispatcher means that we must maintain its state. For instance, unregistering methods when a user sets a different engine (e.g. moving from ray to dask at execution time) is currently unsupported

Detecting the engine: Conditionals are simpler/easier when it comes to detecting an engine. With a custom dispatcher, the registration and dispatching process is more opaque/convoluted. For example, there is a higher risk of not realising that we are using a given engine vs another

The ugly:

Unused arguments: Each method registered with the dispatcher must accept the union of both non-distributed and distributed arguments, even though some would be unused. As the list of supported engines grows, so does the number of unused arguments. It also means that we must maintain the same list of arguments across the different versions of the method

1.5.8 8. Switching between PyArrow and Pandas based datasources for CSV/JSON I/O

Date: 2023-03-16

Status

Accepted

Context

The reading and writing operations for CSV/JSON data in *AWS SDK for pandas* make use of the underlying functions in Pandas. For example, `wr.s3.read_csv` will open a stream of data from S3 and then invoke `pandas.read_csv`. This allows the library to fully support all the arguments which are supported by the underlying Pandas functions. Functions such as `wr.s3.read_csv` or `wr.s3.to_json` accept a `**kwargs` parameter which forwards all parameters to `pandas.read_csv` and `pandas.to_json` automatically.

From version 3.0.0 onward, *AWS SDK for pandas* supports Ray and Modin. When those two libraries are installed, all aforementioned I/O functions will be distributed on a Ray cluster. In the background, this means that all the I/O functions for S3 are running as part of a custom Ray data source. Data is then returned in blocks, which form the Modin DataFrame.

The issue is that the Pandas I/O functions work very slowly in the Ray datasource compared with the equivalent I/O functions in PyArrow. Therefore, calling `pyarrow.csv.read_csv` is significantly faster than calling `pandas.read_csv` in the background.

However, the PyArrow I/O functions do not support the same set of parameters as the ones in Pandas. As a consequence, whereas the PyArrow functions offer greater performance, they come at the cost of feature parity between the non-distributed mode and the distributed mode.

For reference, loading 5 GiB of CSV data with the PyArrow functions took around 30 seconds, compared to 120 seconds with the Pandas functions in the same scenario. For writing back to S3, the speed-up is around 2x.

Decision

In order to maximize both performance without losing feature parity, we implemented logic whereby if the user passes a set of parameters which are supported by PyArrow, the library uses PyArrow for reading/writing. If not, the library defaults to the slower Pandas functions, which will support the set of parameter.

The following example will illustrate the difference:

```
# This will be loaded by PyArrow, as `doublequote` is supported
wr.s3.read_csv(
    path="s3://my-bucket/my-path/",
    dataset=True,
    doublequote=False,
)

# This will be loaded using the Pandas I/O functions, as `comment` is not supported by␣
↪PyArrow
wr.s3.read_csv(
    path="s3://my-bucket/my-path/",
    dataset=True,
    comment="#",
)
```

This logic is applied to the following functions:

1. `wr.s3.read_csv`
2. `wr.s3.read_json`

3. `wr.s3.to_json`
4. `wr.s3.to_csv`

Consequences

The logic of switching between using PyArrow or Pandas functions in background was implemented as part of #1699. It was later expanded to support more parameters in #2008 and #2019.

1.5.9 9. Engine selection and lazy initialization

Date: 2023-05-17

Status

Accepted

Context

In distributed mode, three approaches are possible when it comes to selecting and initializing a Ray engine:

1. Initialize the Ray runtime at import (current default). This option causes the least friction to the user but assumes that installing Ray as an optional dependency is enough to enable distributed mode. Moreover, the user cannot prevent/delay Ray initialization (as it's done at import)
2. Initialize the Ray runtime on the first distributed API call. The user can prevent Ray initialization by switching the engine/memory format with environment variables or between import and the first awswrangler distributed API call. However, by default this approach still assumes that installing Ray is equivalent to enabling distributed mode
3. Wait for the user to enable distributed mode, via environment variables and/or via `wr.engine.set`. This option makes no assumption on which mode to use (distributed vs non-distributed). Non-distributed would be the default and it's up to the user to switch the engine/memory format

Decision

Option #1 is inflexible and gives little control to the user, while option #3 introduces too much friction and puts the burden on the user. Option #2 on the other hand gives full flexibility to the user while providing a sane default.

Consequences

The only difference between the current default and the suggested approach is to delay engine initialization, which is not a breaking change. However, it means that in certain situations more than one Ray instance is initialized. For instance, when running tests across multiple threads, each thread runs its own Ray runtime.

1.6 API Reference

1.6.1 Amazon S3

copy_objects(paths, source_path, target_path)	Copy a list of S3 objects to another S3 directory.
delete_objects(path[, use_threads, ...])	Delete Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
describe_objects(path[, version_id, ...])	Describe Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
does_object_exist(path[, ...])	Check if object exists on S3.
download(path, local_file[, version_id, ...])	Download file from a received S3 path to local file.
get_bucket_region(bucket[, boto3_session])	Get bucket region name.
list_buckets([boto3_session])	List Amazon S3 buckets.
list_directories(path[, chunked, ...])	List Amazon S3 objects from a prefix.
list_objects(path[, suffix, ignore_suffix, ...])	List Amazon S3 objects from a prefix.
merge_datasets(source_path, target_path[, ...])	Merge a source dataset into a target dataset.
read_csv(path[, path_suffix, ...])	Read CSV file(s) from a received S3 prefix or list of S3 objects paths.
read_excel(path[, version_id, use_threads, ...])	Read EXCEL file(s) from a received S3 path.
read_fwf(path[, path_suffix, ...])	Read fixed-width formatted file(s) from a received S3 prefix or list of S3 objects paths.
read_json(path[, path_suffix, ...])	Read JSON file(s) from a received S3 prefix or list of S3 objects paths.
read_parquet(path[, path_root, dataset, ...])	Read Parquet file(s) from an S3 prefix or list of S3 objects paths.
read_parquet_metadata(path[, dataset, ...])	Read Apache Parquet file(s) metadata from an S3 prefix or list of S3 objects paths.
read_parquet_table(table, database[, ...])	Read Apache Parquet table registered in the AWS Glue Catalog.
read_orc(path[, path_root, dataset, ...])	Read ORC file(s) from an S3 prefix or list of S3 objects paths.
read_orc_metadata(path[, dataset, ...])	Read Apache ORC file(s) metadata from an S3 prefix or list of S3 objects paths.
read_orc_table(table, database[, ...])	Read Apache ORC table registered in the AWS Glue Catalog.
read_deltalake(path[, version, partitions, ...])	Load a Deltalake table data from an S3 path.
select_query(sql, path, input_serialization, ...)	Filter contents of Amazon S3 objects based on SQL statement.
size_objects(path[, version_id, ...])	Get the size (ContentLength) in bytes of Amazon S3 objects from a received S3 prefix or list of S3 objects paths.
store_parquet_metadata(path, database, table)	Infer and store parquet metadata on AWS Glue Catalog.
to_csv(df[, path, sep, index, columns, ...])	Write CSV file or dataset on Amazon S3.
to_excel(df, path[, boto3_session, ...])	Write EXCEL file on Amazon S3.
to_json(df[, path, index, columns, ...])	Write JSON file on Amazon S3.
to_parquet(df[, path, index, compression, ...])	Write Parquet file or dataset on Amazon S3.
to_orc(df[, path, index, compression, ...])	Write ORC file or dataset on Amazon S3.
to_deltalake(df, path[, index, mode, dtype, ...])	Write a DataFrame to S3 as a DeltaLake table.
upload(local_file, path[, use_threads, ...])	Upload file from a local file to received S3 path.
wait_objects_exist(paths[, delay, ...])	Wait Amazon S3 objects exist.
wait_objects_not_exist(paths[, delay, ...])	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] | None) – 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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}

Return type
list[str]

Returns
List of new objects paths.

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"]
```

awswrangler.s3.delete_objects

awswrangler.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** (`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** (`datetime | None`) – 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 | None`) – 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** (`dict[str, Any] | None`) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (`Session | None`) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type

None

Returns
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** (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** (str | dict[str, str] | None) – 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** (datetime | None) – 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 | None) – 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** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type
dict[str, dict[str, Any]]

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

Examples

```
>>> import awswrangler as wr
>>> descs0 = wr.s3.describe_objects(['s3://bucket/key0', 's3://bucket/key1'])  #␣
↪Describe both objects
>>> descs1 = 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** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **version_id** (str | None) – Specific version of the object that should exist.

Return type
bool

Returns
True if exists, False otherwise.

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** (str | Any) – A file-like object in binary mode or a path to local file (e.g. ./local/path/to/key0).
- **version_id** (str | None) – 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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (dict[str, Any] | None) – Forward to botocore requests, only "SSECustomerAlgorithm", "SSECustomerKey" and "RequestPayer" arguments will be considered.

Return type
None

Returns
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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type
str

Returns
Region code (e.g. 'us-east-1').

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 (Session | None) – Boto3 Session. The default boto3 session to use, default to None.

Return type
list[str]

Returns
List of bucket names.

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** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type
list[str] | Iterator[list[str]]

Returns
List of objects paths.

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** (`str | list[str] | None`) – Suffix or List of suffixes for filtering S3 keys.
- **ignore_suffix** (`str | list[str] | None`) – Suffix or List of suffixes for S3 keys to be ignored.
- **last_modified_begin** (`datetime | None`) – 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 | None`) – 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** (`dict[str, Any] | None`) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (`Session | None`) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type
`list[str] | Iterator[list[str]]`

Returns
List of objects paths.

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: Literal['append', 'overwrite', 'overwrite_partitions'] = '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** (Literal['append', 'overwrite', 'overwrite_partitions']) – 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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}

Return type
list[str]

Returns
List of new objects paths.

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.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*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *chunksize: int | None = None*, *dataset: bool = False*, *partition_filter: Callable[[dict[str, str]], bool] | None = None*, *ray_args: RaySettings | 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** (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** (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** (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** (str | dict[str, str] | None) – 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** (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** (datetime | None) – 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 | None) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **pyarrow_additional_kwargs** – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **chunksize** (int | None) – 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** (Callable[[dict[str, str]], bool] | None) – 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/3.16.1/tutorials/023%20-%20Flexible%20Partitions%20Filter.html
- **s3_additional_kwargs** (dict[str, Any] | None) – Forwarded to botocore requests.

- **ray_args** (*RaySettings* | None) –

  Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

  – boto3_session

  Parameters of the Ray Modin settings. Only used when distributed computing is used with Ray and Modin installed.

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 awswrangler 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

Return type
DataFrame | Iterator[DataFrame]

Returns
Pandas DataFrame or a Generator in case of *chunksize != 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_excel

awswrangler.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.

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** (`str | None`) – 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 given an int will use the given amount of threads. If integer is provided, specified number is used.
- **boto3_session** (`Session | None`) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (`dict[str, Any] | None`) – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.
- **pandas_kwargs** (`Any`) – KEYWORD arguments forwarded to pandas.read_excel(). 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_excel("s3://bucket/key.xlsx", na_rep="", verbose=True) https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

Return type
`DataFrame`

Returns
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*, *ray_args: RaySettings | 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** (`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** (`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** (`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** (`str | dict[str, str] | None`) – 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** (`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** (datetime | None) – 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 | None) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **pyarrow_additional_kwargs** – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.
- **chunksize** (int | None) – 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** (Callable[[dict[str, str]], bool] | None) – 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/3.16.1/tutorials/023%20-%20Flexible%20Partitions%20Filter.html
- **s3_additional_kwargs** (dict[str, Any] | None) – Forwarded to botocore requests.
- **ray_args** (*RaySettings* | None) –

  Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

  – boto3_session

  Parameters of the Ray Modin settings. Only used when distributed computing is used with Ray and Modin installed.

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

Return type
DataFrame | Iterator[DataFrame]

Returns
Pandas DataFrame or a Generator in case of *chunksize != 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
↪'])
```

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*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *chunksize: int | None = None*, *dataset: bool = False*, *partition_filter: Callable[[dict[str, str]], bool] | None = None*, *ray_args: RaySettings | 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** (`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** (`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** (`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** (`str | dict[str, str] | None`) – 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** (`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** (`datetime | None`) – 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 | None`) – Filter the s3 files by the Last modified date of the object. The filter is applied only after list all s3 files.
- **boto3_session** (`Session | None`) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **pyarrow_additional_kwargs** – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.
- **dtype_backend** (`Literal['numpy_nullable', 'pyarrow']`) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **chunksize** (`int | None`) – 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** (`Callable[[dict[str, str]], bool] | None`) – 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/3.16.1/tutorials/023%20-%20Flexible%20Partitions%20Filter.html

- **s3_additional_kwargs** (`dict[str, Any] | None`) – Forwarded to botocore requests.
- **ray_args** (*RaySettings* | None) –

Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

– boto3_session

Parameters of the Ray Modin settings. Only used when distributed computing is used with Ray and Modin installed.

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

Return type
`DataFrame | Iterator[DataFrame]`

Returns
Pandas DataFrame or a Generator in case of *chunksize != 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*, *dataset: bool = False*, *path_suffix: str | list[str] | None = None*, *path_ignore_suffix: str | list[str] | None = None*, *ignore_empty: bool = True*, *partition_filter: Callable[[dict[str, str]], bool] | None = None*, *columns: list[str] | None = None*, *validate_schema: bool = False*, *coerce_int96_timestamp_unit: str | None = None*, *schema: Schema | None = None*, *last_modified_begin: datetime | None = None*, *last_modified_end: datetime | None = None*, *version_id: str | dict[str, str] | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *chunked: bool | int = False*, *use_threads: bool | int = True*, *ray_args: RayReadParquetSettings | None = None*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*, *decryption_configuration: ArrowDecryptionConfiguration | None = None*) → DataFrame | Iterator[DataFrame]

Read Parquet file(s) from an S3 prefix or list of S3 objects paths.

The concept of *dataset* 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 argument to this function.

Note: `Batching` (*chunked* argument) (Memory Friendly):

Used to return an Iterable of DataFrames instead of a regular DataFrame.

Two batching strategies are available:

- If **chunked=True**, depending on the size of the data, one or more data frames are returned per file in the path/dataset. Unlike **chunked=INTEGER**, rows from different files are not mixed in the resulting data frames.
- If **chunked=INTEGER**, awswrangler iterates 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.

Note: If *use_threads=True*, the number of threads is obtained from os.cpu_count().

Note: Filtering by *last_modified begin* and *last_modified end* is applied after listing all S3 files

---

Parameters

- **path** (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** (str | None) – Root path of the dataset. If dataset=`True`, it is used as a starting point to load partition columns.
- **dataset** (bool) – If *True*, read a parquet dataset instead of individual file(s), loading all related partitions as columns.
- **path_suffix** (str | list[str] | None) – Suffix or List of suffixes to be read (e.g. [".gz.parquet", ".snappy.parquet"]). If None, reads all files. (default)
- **path_ignore_suffix** (str | list[str] | None) – Suffix or List of suffixes to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, reads all files. (default)
- **ignore_empty** (bool) – Ignore files with 0 bytes.
- **partition_filter** (Callable[[dict[str, str]], bool] | None) – 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 must be strings and the 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/3.16.1/tutorials/023%20-%20Flexible%20Partitions%20Filter.html
- **columns** (list[str] | None) – List of columns to read from the file(s).
- **validate_schema** (bool) – Check that the schema is consistent across individual files.
- **coerce_int96_timestamp_unit** (str | None) – Cast timestamps that are stored in INT96 format to a particular resolution (e.g. "ms"). Setting to None is equivalent to "ns" and therefore INT96 timestamps are inferred as in nanoseconds.
- **schema** (Schema | None) – Schema to use whem reading the file.
- **last_modified_begin** (datetime | None) – Filter S3 objects by Last modified date. Filter is only applied after listing all objects.
- **last_modified_end** (datetime | None) – Filter S3 objects by Last modified date. Filter is only applied after listing all objects.
- **version_id** (str | dict[str, str] | None) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimential. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **chunked** (bool | int) – If passed, the data is split into an iterable of DataFrames (Memory friendly). If *True* an iterable of DataFrames is returned without guarantee of chunksize. If an *INTEGER* is passed, an iterable of DataFrames is returned with maximum rows equal to the received INTEGER.
- **use_threads** (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.

- **ray_args** (*RayReadParquetSettings* | None) –

  Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

  – dtype_backend

  Check out the Global Configurations Tutorial for details.

  Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

  – boto3_session

  – version_id

  – s3_additional_kwargs

  – dtype_backend

  Parameters of the Ray Modin settings. Only used when distributed computing is used with Ray and Modin installed.

boto3_session
Boto3 Session. The default boto3 session is used if None is received.

s3_additional_kwargs
Forward to S3 botocore requests.

pyarrow_additional_kwargs
Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

decryption_configuration
`pyarrow.parquet.encryption.CryptoFactory` and `pyarrow.parquet.encryption.KmsConnectionConfig` objects dict used to create a PyArrow `CryptoFactory.file_decryption_properties` object to forward to PyArrow reader. see: https://arrow.apache.org/docs/python/parquet.html#decryption-configuration Client Decryption is not supported in distributed mode.

Return type
`DataFrame` | `Iterator[DataFrame]`

Returns
Pandas DataFrame or a Generator in case of *chunked=True*.

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]*, *dataset: bool = False*, *version_id: str | dict[str, str] | None = None*, *path_suffix: str | None = None*, *path_ignore_suffix: str | list[str] | None = None*, *ignore_empty: bool = True*, *ignore_null: bool = False*, *dtype: dict[str, str] | None = None*, *sampling: float = 1.0*, *coerce_int96_timestamp_unit: str | None = None*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*) → *_ReadTableMetadataReturnValue*

Read Apache Parquet file(s) metadata from an S3 prefix or list of S3 objects paths.

The concept of *dataset* 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 argument to this function.

Note: If *use_threads=True*, the number of threads is obtained from os.cpu_count().

Parameters

- **path** (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]).
- **dataset** (bool) – If *True*, read a parquet dataset instead of individual file(s), loading all related partitions as columns.

- **version_id** (str | dict[str, str] | None) – 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** (str | None) – Suffix or List of suffixes to be read (e.g. [".gz.parquet", ".snappy.parquet"]). If None, reads all files. (default)
- **path_ignore_suffix** (str | list[str] | None) – Suffix or List of suffixes to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, reads all files. (default)
- **ignore_empty** (bool) – Ignore files with 0 bytes.
- **ignore_null** (bool) – Ignore columns with null type.
- **dtype** (dict[str, str] | None) – Dictionary of columns names and Athena/Glue types to cast. Use when you have columns with undetermined data types as partitions columns. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **sampling** (float) – Ratio of files metadata to inspect. Must be *0.0 < sampling <= 1.0*. The higher, the more accurate. The lower, the faster.
- **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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (dict[str, Any] | None) – Forward to S3 botocore requests.

Return type

_ReadTableMetadataReturnValue

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'}).

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*, *coerce_int96_timestamp_unit: str | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *chunked: bool | int = False*, *use_threads: bool | int = True*, *ray_args: RayReadParquetSettings | None = None*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*, *decryption_configuration: ArrowDecryptionConfiguration | None = None*) → DataFrame | Iterator[DataFrame]

Read Apache Parquet table registered in the AWS Glue Catalog.

Note: `Batching` (*chunked* argument) (Memory Friendly):

Used to return an Iterable of DataFrames instead of a regular DataFrame.

Two batching strategies are available:

- If **chunked=True**, depending on the size of the data, one or more data frames are returned per file in the path/dataset. Unlike **chunked=INTEGER**, rows from different files will not be mixed in the resulting data frames.
- 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 the number of rows.

Note: If *use_threads=True*, the number of threads is obtained from os.cpu_count().

Parameters

- **table** (`str`) – AWS Glue Catalog table name.
- **database** (`str`) – AWS Glue Catalog database name.
- **filename_suffix** (`str | list[str] | None`) – Suffix or List of suffixes to be read (e.g. [".gz.parquet", ".snappy.parquet"]). If None, read all files. (default)
- **filename_ignore_suffix** (`str | list[str] | None`) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, read all files. (default)
- **catalog_id** (`str | None`) – 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** (`Callable[[dict[str, str]], bool] | None`) – 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 must be strings and the 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/3.16.1/tutorials/023%20-%20Flexible%20Partitions%20Filter.html

- **columns** (`list[str] | None`) – List of columns to read from the file(s).
- **validate_schema** (`bool`) – Check that the schema is consistent across individual files.
- **coerce_int96_timestamp_unit** (`str | None`) – Cast timestamps that are stored in INT96 format to a particular resolution (e.g. "ms"). Setting to None is equivalent to "ns" and therefore INT96 timestamps are inferred as in nanoseconds.
- **dtype_backend** (`Literal['numpy_nullable', 'pyarrow']`) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **chunked** (`bool | int`) – If passed, the data is split into an iterable of DataFrames (Memory friendly). If *True* an iterable of DataFrames is returned without guarantee of chunksize. If an *INTEGER* is passed, an iterable of DataFrames is returned with maximum rows equal to the received INTEGER.
- **use_threads** (`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.
- **ray_args** (*RayReadParquetSettings* | `None`) –

  Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

  – catalog_id

  – database

  – dtype_backend

  Check out the Global Configurations Tutorial for details.

  Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

  – boto3_session

  – s3_additional_kwargs

  – dtype_backend

  Parameters of the Ray Modin settings. Only used when distributed computing is used with Ray and Modin installed.

boto3_session
Boto3 Session. The default boto3 session is used if None is received.

s3_additional_kwargs
Forward to S3 botocore requests.

pyarrow_additional_kwargs
Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame.

Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

decryption_configuration
`pyarrow.parquet.encryption.CryptoFactory` and `pyarrow.parquet.encryption.KmsConnectionConfig` objects dict used to create a PyArrow `CryptoFactory.file_decryption_properties` object to forward to PyArrow reader. Client Decryption is not supported in distributed mode.

Return type
`DataFrame | Iterator[DataFrame]`

Returns
Pandas DataFrame or a Generator in case of *chunked=True*.

Examples

Reading Parquet Table

```
>>> import awswrangler as wr
>>> df = wr.s3.read_parquet_table(database='...', table='...')
```

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.read_orc

awswrangler.s3.**read_orc**(*path: str | list[str]*, *path_root: str | None = None*, *dataset: bool = False*, *path_suffix: str | list[str] | None = None*, *path_ignore_suffix: str | list[str] | None = None*, *ignore_empty: bool = True*, *partition_filter: Callable[[dict[str, str]], bool] | None = None*, *columns: list[str] | None = None*, *validate_schema: bool = False*, *last_modified_begin: datetime | None = None*, *last_modified_end: datetime | None = None*, *version_id: str | dict[str, str] | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *use_threads: bool | int = True*, *ray_args: RaySettings | None = None*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*) → DataFrame

Read ORC file(s) from an S3 prefix or list of S3 objects paths.

The concept of *dataset* 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 argument to this function.

Note: If *use_threads=True*, the number of threads is obtained from os.cpu_count().

Note: Filtering by *last_modified begin* and *last_modified end* is applied after listing all S3 files

Parameters

- **path** (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** (str | None) – Root path of the dataset. If dataset=`True`, it is used as a starting point to load partition columns.
- **dataset** (bool) – If *True*, read an ORC dataset instead of individual file(s), loading all related partitions as columns.
- **path_suffix** (str | list[str] | None) – Suffix or List of suffixes to be read (e.g. [".gz.orc", ".snappy.orc"]). If None, reads all files. (default)
- **path_ignore_suffix** (str | list[str] | None) – Suffix or List of suffixes to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, reads all files. (default)
- **ignore_empty** (bool) – Ignore files with 0 bytes.
- **partition_filter** (Callable[[dict[str, str]], bool] | None) – 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 must be strings and the 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/3.16.1/tutorials/023%20-%20Flexible%20Partitions%20Filter.html
- **columns** (list[str] | None) – List of columns to read from the file(s).
- **validate_schema** (bool) – Check that the schema is consistent across individual files.
- **last_modified_begin** (datetime | None) – Filter S3 objects by Last modified date. Filter is only applied after listing all objects.
- **last_modified_end** (datetime | None) – Filter S3 objects by Last modified date. Filter is only applied after listing all objects.
- **version_id** (str | dict[str, str] | None) – Version id of the object or mapping of object path to version id. (e.g. {'s3://bucket/key0': '121212', 's3://bucket/key1': '343434'})
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

- **use_threads** (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.
- **ray_args** (*RaySettings* | None) –

  Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

  – dtype_backend

  Check out the Global Configurations Tutorial for details.

  Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

  – boto3_session

  – version_id

  – s3_additional_kwargs

  – dtype_backend

  Parameters of the Ray Modin settings. Only used when distributed computing is used with Ray and Modin installed.

boto3_session
Boto3 Session. The default boto3 session is used if None is received.

s3_additional_kwargs
Forward to S3 botocore requests.

pyarrow_additional_kwargs
Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

Return type
DataFrame

Returns
Pandas DataFrame.

Examples

Reading all ORC files under a prefix

```
>>> import awswrangler as wr
>>> df = wr.s3.read_orc(path='s3://bucket/prefix/')
```

Reading all ORC files from a list

```
>>> import awswrangler as wr
>>> df = wr.s3.read_orc(path=['s3://bucket/filename0.orc', 's3://bucket/filename1.
↪orc'])
```

Reading ORC 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_orc(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.read_orc_metadata

awswrangler.s3.**read_orc_metadata**(*path: str | list[str]*, *dataset: bool = False*, *version_id: str | dict[str, str] | None = None*, *path_suffix: str | None = None*, *path_ignore_suffix: str | list[str] | None = None*, *ignore_empty: bool = True*, *ignore_null: bool = False*, *dtype: dict[str, str] | None = None*, *sampling: float = 1.0*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*) → *_ReadTableMetadataReturnValue*

Read Apache ORC file(s) metadata from an S3 prefix or list of S3 objects paths.

The concept of *dataset* 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 argument to this function.

Note: If *use_threads=True*, the number of threads is obtained from os.cpu_count().

Parameters

- **path** (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]).
- **dataset** (bool) – If *True*, read an ORC dataset instead of individual file(s), loading all related partitions as columns.
- **version_id** (str | dict[str, str] | None) – 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** (str | None) – Suffix or List of suffixes to be read (e.g. [".gz.orc", ".snappy.orc"]). If None, reads all files. (default)
- **path_ignore_suffix** (str | list[str] | None) – Suffix or List of suffixes to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, reads all files. (default)
- **ignore_empty** (bool) – Ignore files with 0 bytes.
- **ignore_null** (bool) – Ignore columns with null type.
- **dtype** (dict[str, str] | None) – Dictionary of columns names and Athena/Glue types to cast. Use when you have columns with undetermined data types as partitions columns. (e.g. {'col name': 'bigint', 'col2 name': 'int'})
- **sampling** (float) – Ratio of files metadata to inspect. Must be *0.0 < sampling <= 1.0*. The higher, the more accurate. The lower, the faster.
- **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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (dict[str, Any] | None) – Forward to S3 botocore requests.

Return type
_ReadTableMetadataReturnValue

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'}).

Examples

Reading all ORC files (with partitions) metadata under a prefix

```
>>> import awswrangler as wr
>>> columns_types, partitions_types = wr.s3.read_orc_metadata(path='s3://bucket/
↪prefix/', dataset=True)
```

Reading all ORC files metadata from a list

```
>>> import awswrangler as wr
>>> columns_types, partitions_types = wr.s3.read_orc_metadata(path=[
...     's3://bucket/filename0.orc',
...     's3://bucket/filename1.orc',
... ])
```

awswrangler.s3.read_orc_table

awswrangler.s3.**read_orc_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*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *use_threads: bool | int = True*, *ray_args: RaySettings | None = None*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*) → DataFrame

Read Apache ORC table registered in the AWS Glue Catalog.

Note: If *use_threads=True*, the number of threads is obtained from os.cpu_count().

Parameters

- **table** (str) – AWS Glue Catalog table name.
- **database** (str) – AWS Glue Catalog database name.
- **filename_suffix** (str | list[str] | None) – Suffix or List of suffixes to be read (e.g. [".gz.orc", ".snappy.orc"]). If None, read all files. (default)

- **filename_ignore_suffix** (str | list[str] | None) – Suffix or List of suffixes for S3 keys to be ignored.(e.g. [".csv", "_SUCCESS"]). If None, read all files. (default)
- **catalog_id** (str | None) – 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** (Callable[[dict[str, str]], bool] | None) – 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 must be strings and the 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/3.16.1/tutorials/023%20-%20Flexible%20Partitions%20Filter.html
- **columns** (list[str] | None) – List of columns to read from the file(s).
- **validate_schema** (bool) – Check that the schema is consistent across individual files.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **use_threads** (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.
- **ray_args** (*RaySettings* | None) –

  Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

  – catalog_id

  – database

  – dtype_backend

  Check out the Global Configurations Tutorial for details.

  Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

  – boto3_session

  – s3_additional_kwargs

  – dtype_backend

  Parameters of the Ray Modin settings. Only used when distributed computing is used with Ray and Modin installed.

boto3_session
Boto3 Session. The default boto3 session is used if None is received.

s3_additional_kwargs
Forward to S3 botocore requests.

pyarrow_additional_kwargs
Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

Return type
DataFrame

Returns
Pandas DataFrame.

Examples

Reading ORC Table

```
>>> import awswrangler as wr
>>> df = wr.s3.read_orc_table(database='...', table='...')
```

Reading ORC 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_orc_table(path, dataset=True, partition_filter=my_filter)
```

awswrangler.s3.read_deltalake

awswrangler.s3.**read_deltalake**(*path: str*, *version: int | None = None*, *partitions: list[tuple[str, str, Any]] | None = None*, *columns: list[str] | None = None*, *without_files: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *use_threads: bool = True*, *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** (str) – The path of the DeltaTable.
- **version** (int | None) – The version of the DeltaTable.
- **partitions** (list[tuple[str, str, Any]] | None) – A list of partition filters, see help(DeltaTable.files_by_partitions) for filter syntax.
- **columns** (list[str] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

- **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** (Session | None) – Boto3 Session. If None, the default boto3 session is used.
- **s3_additional_kwargs** (dict[str, str] | None) – Forwarded to the Delta Table class for the storage options of the S3 backend.
- **pyarrow_additional_kwargs** (dict[str, Any] | None) – Forwarded to the PyArrow to_pandas method.

Return type
DataFrame

Returns
DataFrame with the results.

See also:

deltalake.DeltaTable
Create a DeltaTable instance with the deltalake library.

awswrangler.s3.select_query

awswrangler.s3.**select_query**(*sql: str*, *path: str | list[str]*, *input_serialization: str*, *input_serialization_params: dict[str, bool | str]*, *compression: str | None = None*, *scan_range_chunk_size: int | None = None*, *path_suffix: str | list[str] | None = None*, *path_ignore_suffix: str | list[str] | None = None*, *ignore_empty: bool = True*, *use_threads: bool | int = True*, *last_modified_begin: datetime | None = None*, *last_modified_end: datetime | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*) → DataFrame

Filter contents of Amazon S3 objects 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 aforementioned conditions are not met, leading to lower performance.

Parameters

- **sql** (str) – SQL statement used to query the object.
- **path** (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]).
- **input_serialization** (str) – Format of the S3 object queried. Valid values: "CSV", "JSON", or "Parquet". Case sensitive.
- **input_serialization_params** (dict[str, bool | str]) – Dictionary describing the serialization of the S3 object.
- **compression** (str | None) – Compression type of the S3 object. Valid values: None, "gzip", or "bzip2". gzip and bzip2 are only valid for CSV and JSON objects.

- **scan_range_chunk_size** (int | None) – Chunk size used to split the S3 object into scan ranges. 1,048,576 by default.
- **path_suffix** (str | list[str] | None) – Suffix or List of suffixes to be read (e.g. [".csv"]). If None, read all files. (default)
- **path_ignore_suffix** (str | list[str] | None) – Suffix or List of suffixes for S3 keys to be ignored. (e.g. ["_SUCCESS"]). If None, read all files. (default)
- **ignore_empty** (bool) – Ignore files with 0 bytes.
- **use_threads** (bool | int) – True (default) 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.
- **last_modified_begin** (datetime | None) – Filter S3 objects by Last modified date. Filter is only applied after listing all objects.
- **last_modified_end** (datetime | None) – Filter S3 objects by Last modified date. Filter is only applied after listing all objects.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **boto3_session** (Session | None) – The default boto3 session is used if none is provided.
- **s3_additional_kwargs** (dict[str, Any] | None) – Forwarded to botocore requests. Valid values: "SSECustomerAlgorithm", "SSECustomerKey", "ExpectedBucketOwner". e.g. s3_additional_kwargs={'SSECustomerAlgorithm': 'md5'}.
- **pyarrow_additional_kwargs** (dict[str, Any] | None) – Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

Return type
DataFrame

Returns
Pandas DataFrame with results from query.

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',
...     },
```

(continues on next page)

(continued from previous page)

```
...     compression="gzip",
... )
```

Reading multiple CSV objects from a prefix

```
>>> import awswrangler as wr
>>> df = wr.s3.select_query(
...     sql='SELECT * FROM s3object',
...     path='s3://bucket/prefix/',
...     input_serialization='CSV',
...     input_serialization_params={
...         'FileHeaderInfo': 'Use',
...         'RecordDelimiter': '\r\n'
...     },
... )
```

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** (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** (str | dict[str, str] | None) – 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** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type
dict[str, int | None]

Returns
Dictionary where the key is the object path and the value is the object size.

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 | list[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*, *description: str | None = None*, *parameters: dict[str, str] | None = None*, *columns_comments: dict[str, str] | None = None*, *compression: str | None = None*, *mode: Literal['append', 'overwrite'] = 'overwrite'*, *catalog_versioning: bool = False*, *regular_partitions: bool = True*, *athena_partition_projection_settings: AthenaPartitionProjectionSettings | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*, *boto3_session: Session | None = None*) → tuple[dict[str, str], dict[str, str] | None, dict[str, list[str]] | None]

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().

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 | None) – 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** (str | None) – Suffix or List of suffixes for filtering S3 keys.
- **path_ignore_suffix** (str | list[str] | None) – Suffix or List of suffixes for S3 keys to be ignored.
- **ignore_empty** (bool) – Ignore files with 0 bytes.
- **ignore_null** (bool) – Ignore columns with null type.
- **dtype** (dict[str, str] | None) – 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 <= 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 | None) – Glue/Athena catalog: Table description
- **parameters** (dict[str, str] | None) – Glue/Athena catalog: Key/value pairs to tag the table.
- **columns_comments** (dict[str, str] | None) – Glue/Athena catalog: Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **compression** (str | None) – Compression style (None, snappy, gzip, etc).
- **mode** (Literal['append', 'overwrite']) – '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.
- **athena_partition_projection_settings** (*AthenaPartitionProjectionSettings* | None) –

Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

– boto3_session

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 of the Athena Partition Projection (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html). AthenaPartitionProjectionSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaPartitionProjectionSettings or as a regular Python dict.

Following projection parameters are supported:

Table 4: Projection Parameters

Name	Type	Description
projection_types	Optional[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_range	Optional[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_value	Optional[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_interv	Optional[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[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_forma	Optional[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_stora	Optional[	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}/)

s3_additional_kwargs
Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}

boto3_session
The default boto3 session will be used if boto3_session receive None.

Return type
`tuple[dict[str, str], dict[str, str] | None, dict[str, list[str]] | 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']}).

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
... )
```

awswrangler.s3.to_csv

awswrangler.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: Literal['append', 'overwrite', 'overwrite_partitions'] | None = None*, *catalog_versioning: bool = False*, *schema_evolution: bool = False*, *dtype: dict[str, str] | None = None*, *database: str | None = None*, *table: str | None = None*, *glue_table_settings:* GlueTableSettings *| None = None*, *athena_partition_projection_settings:* AthenaPartitionProjectionSettings *| None = None*, *catalog_id: str | None = None*, ***pandas_kwargs: Any*) → *_S3WriteDataReturnValue*

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: In case of *use_threads=True* the number of threads that will be spawned will be gotten from os.cpu_count().

Parameters

- **df** (DataFrame) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (str | None) – 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** (list[str] | None) – 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** (Session | None) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **s3_additional_kwargs** (dict[str, Any] | None) – 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_params, catalog_id, schema_evolution.
- **filename_prefix** (str | None) – If dataset=True, add a filename prefix to the output files.
- **partition_cols** (list[str] | None) – List of column names that will be used to create partitions. Only takes effect if dataset=True.
- **bucketing_info** (Tuple[List[str], int] | None) – 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/3.16.1/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html
- **mode** (Literal['append', 'overwrite', 'overwrite_partitions'] | None) – append (Default), overwrite, overwrite_partitions. Only takes effect if dataset=True. For details check the related tutorial: https://aws-sdk-pandas.readthedocs.io/en/3.16.1/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/3.16.1/tutorials/014%20-%20Schema%20Evolution.html
- **database** (str | None) – Glue/Athena catalog: Database name.
- **table** (str | None) – Glue/Athena catalog: Table name.

- **glue_table_settings** (*GlueTableSettings* | None) – Settings for writing to the Glue table.
- **dtype** (dict[str, str] | None) – 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'})
- **athena_partition_projection_settings** (*AthenaPartitionProjectionSettings* | None) –

Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

– boto3_session

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 of the Athena Partition Projection (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html). AthenaPartitionProjectionSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaPartitionProjectionSettings or as a regular Python dict.

Following projection parameters are supported:

Table 5: Projection Parameters

Name	Type	Description
projection_types	Optional[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_range	Optional[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_value	Optional[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_interv	Optional[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[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_forma	Optional[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_stora	Optional[	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}/)

catalog_id
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 awswrangler 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

Return type
_S3WriteDataReturnValue

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.

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(
...     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({
```

(continues on next page)

(continued from previous page)

```
...         '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 partitioned dataset with partition projection

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> from datetime import datetime
>>> dt = lambda x: datetime.strptime(x, "%Y-%m-%d").date()
>>> wr.s3.to_csv(
...     df=pd.DataFrame({
...         "id": [1, 2, 3],
...         "value": [1000, 1001, 1002],
...         "category": ['A', 'B', 'C'],
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['value', 'category'],
...     athena_partition_projection_settings={
...        "projection_types": {
...             "value": "integer",
...             "category": "enum",
...         },
...         "projection_ranges": {
...             "value": "1000,2000",
...             "category": "A,B,C",
...         },
...     },
... )
{
    'paths': [
        's3://.../value=1000/category=A/x.json', ...
    ],
    'partitions_values': {
        's3://.../value=1000/category=A/': [
            '1000',
            'A',
        ], ...
    }
}
```

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 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]
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     database='default',  # Athena/Glue database
```

(continues on next page)

(continued from previous page)

```
...     table='my_table'  # Athena/Glue table
...     dtype={'col3': 'date'}
... )
{
    'paths': ['s3://.../x.csv'],
    'partitions_values: {}
}
```

awswrangler.s3.to_excel

awswrangler.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.

Note: In case of *use_threads=True* the number of threads that will be spawned will be gotten from os.cpu_count().

Parameters

- **df** (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** (Session | None) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **pyarrow_additional_kwargs** – 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** (Any) – 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 awswrangler 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

Return type

str

Returns
Written S3 path.

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: Literal['append', 'overwrite', 'overwrite_partitions'] | None = None*, *catalog_versioning: bool = False*, *schema_evolution: bool = True*, *dtype: dict[str, str] | None = None*, *database: str | None = None*, *table: str | None = None*, *glue_table_settings: GlueTableSettings | None = None*, *athena_partition_projection_settings: AthenaPartitionProjectionSettings | None = None*, *catalog_id: str | None = None*, ***pandas_kwargs: Any*) → *_S3WriteDataReturnValue*

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().

Parameters

- **df** (DataFrame) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (str | None) – Amazon S3 path (e.g. s3://bucket/filename.json).
- **index** (bool) – Write row names (index).
- **columns** (list[str] | None) – 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** (Session | None) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **s3_additional_kwarg** – 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_params, catalog_id, schema_evolution.
- **filename_prefix** (str | None) – If dataset=True, add a filename prefix to the output files.
- **partition_cols** (list[str] | None) – List of column names that will be used to create partitions. Only takes effect if dataset=True.
- **bucketing_info** (Tuple[List[str], int] | None) – 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/3.16.1/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html
- **mode** (Literal['append', 'overwrite', 'overwrite_partitions'] | None) – append (Default), overwrite, overwrite_partitions. Only takes effect if dataset=True. For details check the related tutorial: https://aws-sdk-pandas.readthedocs.io/en/3.16.1/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/3.16.1/tutorials/014%20-%20Schema%20Evolution.html
- **database** (str | None) – Glue/Athena catalog: Database name.
- **table** (str | None) – Glue/Athena catalog: Table name.
- **glue_table_settings** (*GlueTableSettings* | None) – Settings for writing to the Glue table.
- **dtype** (dict[str, str] | None) – 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'})
- **athena_partition_projection_settings** (*AthenaPartitionProjectionSettings* | None) –

Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

– boto3_session

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 of the Athena Partition Projection (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html). AthenaPartitionProjectionSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaPartitionProjectionSettings or as a regular Python dict.

Following projection parameters are supported:

Table 6: Projection Parameters

Name	Type	Description
projection_types	Optional[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_range	Optional[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_value	Optional[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_inter	Optional[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[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_forma	Optional[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_stora	Optional[	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}/)

catalog_id
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 awswrangler 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

Return type
_S3WriteDataReturnValue

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.

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'
...     }
... )
```

Writing partitioned dataset with partition projection

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> from datetime import datetime
>>> dt = lambda x: datetime.strptime(x, "%Y-%m-%d").date()
>>> wr.s3.to_json(
...     df=pd.DataFrame({
...         "id": [1, 2, 3],
...         "value": [1000, 1001, 1002],
...         "category": ['A', 'B', 'C'],
```

(continues on next page)

(continued from previous page)

```
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['value', 'category'],
...     athena_partition_projection_settings={
...        "projection_types": {
...             "value": "integer",
...             "category": "enum",
...         },
...         "projection_ranges": {
...             "value": "1000,2000",
...             "category": "A,B,C",
...         },
...     },
... )
{
    'paths': [
        's3://.../value=1000/category=A/x.json', ...
    ],
    'partitions_values': {
        's3://.../value=1000/category=A/': [
            '1000',
            'A',
        ], ...
    }
}
```

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: Literal['append', 'overwrite', 'overwrite_partitions'] | None = None*, *catalog_versioning: bool = False*, *schema_evolution: bool = True*, *database: str | None = None*, *table: str | None = None*, *glue_table_settings: GlueTableSettings | None = None*, *dtype: dict[str, str] | None = None*, *athena_partition_projection_settings: AthenaPartitionProjectionSettings | None = None*, *catalog_id: str | None = None*, *encryption_configuration: ArrowEncryptionConfiguration | None = None*) → *_S3WriteDataReturnValue*

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().

Parameters

- **df** (DataFrame) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (str | None) – 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. Is not supported in conjunction with *max_rows_by_file* when running the library with Ray/Modin.
- **compression** (str | None) – Compression style (None, snappy, gzip, zstd).
- **pyarrow_additional_kwargs** (dict[str, Any] | None) – 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 | None) – Max number of rows in each file. Default is None i.e. don't split the files. (e.g. 33554432, 268435456) Is not supported in conjunction with *index=True* when running the library with Ray/Modin.
- **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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (dict[str, Any] | None) – 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_params, catalog_id, schema_evolution.
- **filename_prefix** (str | None) – If dataset=True, add a filename prefix to the output files.
- **partition_cols** (list[str] | None) – List of column names that will be used to create partitions. Only takes effect if dataset=True.
- **bucketing_info** (Tuple[List[str], int] | None) – 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/3.16.1/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html
- **mode** (Literal['append', 'overwrite', 'overwrite_partitions'] | None) – append (Default), overwrite, overwrite_partitions. Only takes effect if dataset=True. For details check the related tutorial: https://aws-sdk-pandas.readthedocs.io/en/3.16.1/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/3.16.1/tutorials/014%20-%20Schema%20Evolution.html
- **database** (str | None) – Glue/Athena catalog: Database name.
- **table** (str | None) – Glue/Athena catalog: Table name.
- **glue_table_settings** (*GlueTableSettings* | None) – Settings for writing to the Glue table.
- **dtype** (dict[str, str] | None) – 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'})
- **athena_partition_projection_settings** (*AthenaPartitionProjectionSettings* | None) –

Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

– boto3_session

– s3_additional_kwargs

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 of the Athena Partition Projection (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html). AthenaPartitionProjectionSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaPartitionProjectionSettings or as a regular Python dict.

Following projection parameters are supported:

Table 7: Projection Parameters

Name	Type	Description
projection_types	Optional[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_range	Optional[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_value	Optional[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_inter	Optional[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[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_form	Optional[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_stora	Optional[	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}/)

catalog_id
The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

encryption_configuration
For Arrow client-side encryption provide materials as follows {'crypto_factory': pyarrow.parquet.encryption.CryptoFactory, 'kms_connection_config': pyarrow.parquet.encryption.KmsConnectionConfig, 'encryption_config': pyarrow.parquet.encryption.EncryptionConfiguration} see: https://arrow.apache.org/docs/python/parquet.html#parquet-modular-encryption-columnar-encryption Client Encryption is not supported in distributed mode.

Return type
_S3WriteDataReturnValue

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.

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',
...     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 partitioned dataset with partition projection

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> from datetime import datetime
>>> dt = lambda x: datetime.strptime(x, "%Y-%m-%d").date()
>>> wr.s3.to_parquet(
...     df=pd.DataFrame({
...         "id": [1, 2, 3],
...         "value": [1000, 1001, 1002],
...         "category": ['A', 'B', 'C'],
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['value', 'category'],
...     athena_partition_projection_settings={
...        "projection_types": {
...             "value": "integer",
...             "category": "enum",
...         },
...         "projection_ranges": {
...             "value": "1000,2000",
...             "category": "A,B,C",
...         },
...     },
... )
{
    'paths': [
        's3://.../value=1000/category=A/x.snappy.parquet', ...
    ],
    'partitions_values': {
        's3://.../value=1000/category=A/': [
            '1000',
            'A',
        ], ...
    }
}
```

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 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.to_orc

awswrangler.s3.**to_orc**(*df: DataFrame*, *path: str | None = None*, *index: bool = False*, *compression: str | None = None*, *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: Literal['append', 'overwrite', 'overwrite_partitions'] | None = None*, *catalog_versioning: bool = False*, *schema_evolution: bool = True*, *database: str | None = None*, *table: str | None = None*, *glue_table_settings: GlueTableSettings | None = None*, *dtype: dict[str, str] | None = None*, *athena_partition_projection_settings: AthenaPartitionProjectionSettings | None = None*, *catalog_id: str | None = None*) → *_S3WriteDataReturnValue*

Write ORC 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().

Parameters

- **df** (DataFrame) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **path** (str | None) – S3 path (for file e.g. s3://bucket/prefix/filename.orc) (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. Is not supported in conjunction with *max_rows_by_file* when running the library with Ray/Modin.
- **compression** (str | None) – Compression style (None, snappy, gzip, zstd).
- **pyarrow_additional_kwargs** (dict[str, Any] | None) – 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 | None) – Max number of rows in each file. Default is None i.e. don't split the files. (e.g. 33554432, 268435456) Is not supported in conjunction with *index=True* when running the library with Ray/Modin.
- **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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **s3_additional_kwargs** (*dict[str, Any], optional*) – 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 orc 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_params, catalog_id, schema_evolution.
- **filename_prefix** (str | None) – If dataset=True, add a filename prefix to the output files.
- **partition_cols** (list[str] | None) – List of column names that will be used to create partitions. Only takes effect if dataset=True.
- **bucketing_info** (Tuple[List[str], int] | None) – 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/3.16.1/tutorials/022%20-%20Writing%20Partitions%20Concurrently.html
- **mode** (Literal['append', 'overwrite', 'overwrite_partitions'] | None) – append (Default), overwrite, overwrite_partitions. Only takes effect if dataset=True.
- **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/3.16.1/tutorials/014%20-%20Schema%20Evolution.html
- **database** (str | None) – Glue/Athena catalog: Database name.
- **table** (str | None) – Glue/Athena catalog: Table name.
- **glue_table_settings** (*GlueTableSettings* | None) – Settings for writing to the Glue table.
- **dtype** (dict[str, str] | None) – 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'})
- **athena_partition_projection_settings** (*AthenaPartitionProjectionSettings* | None) –

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.

Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

– boto3_session

– s3_additional_kwargs

Parameters of the Athena Partition Projection (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html). AthenaPartitionProjectionSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaPartitionProjectionSettings or as a regular Python dict.

Following projection parameters are supported:

Table 8: Projection Parameters

Name	Type	Description
projection_types	Optional[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_range	Optional[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_value	Optional[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_interv	Optional[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[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_forma	Optional[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_stora	Optional[	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}/)

catalog_id
The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

Return type
_S3WriteDataReturnValue

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.

Examples

Writing single file

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_orc(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/prefix/my_file.orc',
... )
{
    'paths': ['s3://bucket/prefix/my_file.orc'],
    'partitions_values': {}
}
```

Writing single file encrypted with a KMS key

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_orc(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     path='s3://bucket/prefix/my_file.orc',
...     s3_additional_kwargs={
...         'ServerSideEncryption': 'aws:kms',
...         'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'
...     }
... )
{
    'paths': ['s3://bucket/prefix/my_file.orc'],
    'partitions_values': {}
}
```

Writing partitioned dataset

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_orc(
...     df=pd.DataFrame({
...         'col': [1, 2, 3],
...         'col2': ['A', 'A', 'B']
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
```

(continues on next page)

(continued from previous page)

```
...     partition_cols=['col2']
... )
{
    'paths': ['s3://.../col2=A/x.orc', 's3://.../col2=B/y.orc'],
    'partitions_values: {
        's3://.../col2=A/': ['A'],
        's3://.../col2=B/': ['B']
    }
}
```

Writing partitioned dataset with partition projection

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> from datetime import datetime
>>> dt = lambda x: datetime.strptime(x, "%Y-%m-%d").date()
>>> wr.s3.to_orc(
...     df=pd.DataFrame({
...         "id": [1, 2, 3],
...         "value": [1000, 1001, 1002],
...         "category": ['A', 'B', 'C'],
...     }),
...     path='s3://bucket/prefix',
...     dataset=True,
...     partition_cols=['value', 'category'],
...     athena_partition_projection_settings={
...        "projection_types": {
...             "value": "integer",
...             "category": "enum",
...         },
...         "projection_ranges": {
...             "value": "1000,2000",
...             "category": "A,B,C",
...         },
...     },
... )
{
    'paths': [
        's3://.../value=1000/category=A/x.snappy.orc', ...
    ],
    'partitions_values': {
        's3://.../value=1000/category=A/': [
            '1000',
            'A',
        ], ...
    }
}
```

Writing bucketed dataset

```
>>> import awswrangler as wr
>>> import pandas as pd
```

(continues on next page)

(continued from previous page)

```
>>> wr.s3.to_orc(
...     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_orc(
...     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.orc', 's3://.../col2=B/y.orc'],
    'partitions_values: {
        's3://.../col2=A/': ['A'],
        's3://.../col2=B/': ['B']
    }
}
```

awswrangler.s3.to_deltalake

awswrangler.s3.**to_deltalake**(*df: DataFrame*, *path: str*, *index: bool = False*, *mode: Literal['error', 'append', 'overwrite', 'ignore'] = 'append'*, *dtype: dict[str, str] | None = None*, *partition_cols: list[str] | None = None*, *schema_mode: Literal['overwrite'] | None = None*, *lock_dynamodb_table: str | None = None*, *s3_allow_unsafe_rename: bool = False*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, str] | None = None*) → None

Write a DataFrame to S3 as a DeltaLake table.

This function requires the deltalake package.

Parameters

- **df** (DataFrame) – Pandas DataFrame

- **path** (str) – S3 path for a directory where the DeltaLake table will be stored.
- **index** (bool) – True to store the DataFrame index in file, otherwise False to ignore it.
- **mode** (Literal['error', 'append', 'overwrite', 'ignore']) – append (Default), overwrite, ignore, error
- **dtype** (dict[str, str] | None) – 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'})
- **partition_cols** (list[str] | None) – List of columns to partition the table by. Only required when creating a new table.
- **schema_mode** (Literal['overwrite'] | None) – If set to "overwrite", allows replacing the schema of the table. Set to "merge" to merge with existing schema.
- **lock_dynamodb_table** (str | None) – DynamoDB table to use as a locking provider. A locking mechanism is needed to prevent unsafe concurrent writes to a delta lake directory when writing to S3. If you don't want to use a locking mechanism, you can choose to set s3_allow_unsafe_rename to True.

  For information on how to set up the lock table, please check this page.
- **s3_allow_unsafe_rename** (bool) – Allows using the default S3 backend without support for concurrent writers.
- **boto3_session** (Session | None) – If None, the default boto3 session is used.
- **pyarrow_additional_kwargs** – Forwarded to the Delta Table class for the storage options of the S3 backend.

Return type
None

Examples

Writing a Pandas DataFrame into a DeltaLake table in S3.

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_deltalake(
...     df=pd.DataFrame({"col": [1, 2, 3]}),
...     path="s3://bucket/prefix/",
...     lock_dynamodb_table="my-lock-table",
... )
```

See also:

deltalake.DeltaTable
Create a DeltaTable instance with the deltalake library.

deltalake.write_deltalake
Write to a DeltaLake table.

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** (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** (Session | None) – The default boto3 session will be used if boto3_session receive None.
- **pyarrow_additional_kwargs** – Forward to botocore requests, only "SSECustomerAlgorithm" and "SSECustomerKey" arguments will be considered.

Return type
None

Returns
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** (float | None) – The amount of time in seconds to wait between attempts. Default: 5
- **max_attempts** (int | None) – 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** (Session | None) – The default boto3 session will be used if boto3_session receive None.

Return type
None

Returns
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** (float | None) – The amount of time in seconds to wait between attempts. Default: 5
- **max_attempts** (int | None) – 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** (Session | None) – The default boto3 session will be used if boto3_session receive None.

Return type
None

Returns
None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.wait_objects_not_exist(['s3://bucket/key0', 's3://bucket/key1'])  # wait␣
↪both objects not exist
```

1.6.2 AWS Glue Catalog

add_column(database, table, column_name[, ...])	Add a column in a AWS Glue Catalog table.
add_csv_partitions(database, table, ...[, ...])	Add partitions (metadata) to a CSV Table in the AWS Glue Catalog.
add_parquet_partitions(database, table, ...)	Add partitions (metadata) to a Parquet Table in the AWS Glue Catalog.
create_csv_table(database, table, path, ...)	Create a CSV Table (Metadata Only) in the AWS Glue Catalog.
create_database(name[, description, ...])	Create a database in AWS Glue Catalog.
create_json_table(database, table, path, ...)	Create a JSON Table (Metadata Only) in the AWS Glue Catalog.
create_parquet_table(database, table, path, ...)	Create a Parquet Table (Metadata Only) in the AWS Glue Catalog.
databases([limit, catalog_id, boto3_session])	Get a Pandas DataFrame with all listed databases.
delete_column(database, table, column_name)	Delete a column in a AWS Glue Catalog table.
delete_database(name[, catalog_id, ...])	Delete a database in AWS Glue Catalog.
delete_partitions(table, database, ...[, ...])	Delete specified partitions in a AWS Glue Catalog table.
delete_all_partitions(table, database[, ...])	Delete all partitions in a AWS Glue Catalog table.
delete_table_if_exists(database, table[, ...])	Delete Glue table if exists.

continues on next page

Table 9 – continued from previous page

does_table_exist(database, table[, ...])	Check if the table exists.
drop_duplicated_columns(df)	Drop all repeated columns (duplicated names).
extract_athena_types(df[, index, ...])	Extract columns and partitions types (Amazon Athena) from Pandas DataFrame.
get_columns_comments(database, table[, ...])	Get all columns comments.
get_columns_parameters(database, table[, ...])	Get all columns parameters.
get_csv_partitions(database, table[, ...])	Get all partitions from a Table in the AWS Glue Catalog.
get_databases([catalog_id, boto3_session])	Get an iterator of databases.
get_parquet_partitions(database, table[, ...])	Get all partitions from a Table in the AWS Glue Catalog.
get_partitions(database, table[, ...])	Get all partitions from a Table in the AWS Glue Catalog.
get_table_description(database, table[, ...])	Get table description.
get_table_location(database, table[, ...])	Get table's location on Glue catalog.
get_table_number_of_versions(database, table)	Get total number of versions.
get_table_parameters(database, table[, ...])	Get all parameters.
get_table_types(database, table[, ...])	Get all columns and types from a table.
get_table_versions(database, table[, ...])	Get all versions.
get_tables([catalog_id, database, ...])	Get an iterator of tables.
overwrite_table_parameters(parameters, ...)	Overwrite all existing parameters.
sanitize_column_name(column)	Convert the column name to be compatible with Amazon Athena and the AWS Glue Catalog.
sanitize_dataframe_columns_names(df[, ...])	Normalize all columns names to be compatible with Amazon Athena.
sanitize_table_name(table)	Convert the table name to be compatible with Amazon Athena and the AWS Glue Catalog.
search_tables(text[, catalog_id, boto3_session])	Get Pandas DataFrame of tables filtered by a search string.
table(database, table[, catalog_id, ...])	Get table details as Pandas DataFrame.
tables([limit, catalog_id, database, ...])	Get a DataFrame with tables filtered by a search term, prefix, suffix.
upsert_table_parameters(parameters, ...[, ...])	Insert or Update the received parameters.

awswrangler.catalog.add_column

awswrangler.catalog.**add_column**(*database: str*, *table: str*, *column_name: str*, *column_type: str = 'string'*, *column_comment: str | None = None*, *boto3_session: Session | None = None*, *catalog_id: str | None = None*) → None

Add a column in a AWS Glue Catalog table.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **column_name** (str) – Column name
- **column_type** (str) – Column type.
- **column_comment** (str | None) – Column Comment
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **catalog_id** (str | None) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.

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*) → None

Add partitions (metadata) to a CSV Table in the AWS Glue Catalog.

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] | None) – 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 | None) – 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 | None) – Compression style (None, gzip, etc).
- **sep** (str) – String of length 1. Field delimiter for the output file.
- **serde_library** (str | None) – 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** (dict[str, str] | None) – Dictionary of initialization parameters for the SerDe. The default is *{"field.delim": sep, "escape.delim": "\"}*.
- **boto3_session** (Session | None) – The default boto3 session will be used if boto3_session receive None.
- **columns_types** (dict[str, str] | None) – 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** (dict[str, str] | None) – Dictionary with key-value pairs defining partition parameters.

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*) → None

Add partitions (metadata) to a Parquet Table in the AWS Glue Catalog.

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] | None) – 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 | None) – 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 | None) – Compression style (None, snappy, gzip, etc).
- **boto3_session** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.
- **columns_types** (dict[str, str] | None) – 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** (dict[str, str] | None) – Dictionary with key-value pairs defining partition parameters.

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

awswrangler.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*, *columns_parameters: dict[str, dict[str, str]] | None = None*, *mode: Literal['overwrite', 'append'] = '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*, *boto3_session: Session | None = None*, *athena_partition_projection_settings: AthenaPartitionProjectionSettings | None = None*, *catalog_id: str | None = None*) → None

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.

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 | None) – The type of the Glue Table. Set to EXTERNAL_TABLE if None.
- **partitions_types** (dict[str, str] | None) – Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).
- **bucketing_info** (Tuple[List[str], int] | None) – 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 | None) – Compression style (None, gzip, etc).
- **description** (str | None) – Table description
- **parameters** (dict[str, str] | None) – Key/value pairs to tag the table.
- **columns_comments** (dict[str, str] | None) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **columns_parameters** (dict[str, dict[str, str]] | None) – Columns names and the related parameters (e.g. {'col0': {'par0': 'Param 0', 'par1': 'Param 1'}}).
- **mode** (Literal['overwrite', 'append']) – '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.
- **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/3.16.1/tutorials/014%20-%20Schema%20Evolution.html
- **sep** (str) – String of length 1. Field delimiter for the output file.
- **skip_header_line_count** (int | None) – Number of Lines to skip regarding to the header.
- **serde_library** (str | None) – 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** (dict[str, str] | None) – Dictionary of initialization parameters for the SerDe. The default is *{"field.delim": sep, "escape.delim": "\"}*.
- **athena_partition_projection_settings** (*AthenaPartitionProjectionSettings* | None) –

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 of the Athena Partition Projection (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html). AthenaPartitionProjectionSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaPartitionProjectionSettings or as a regular Python dict.

Following projection parameters are supported:

Table 10: Projection Parameters

Name	Type	Description
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_interva	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_formal	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_storag	Optional[s	Value which is allows Athena to properly map partition values if the S3 file locations do not follow

Return type

None 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

The default boto3 session will be used if **boto3_session** receive None.

catalog_id

The ID of the Data Catalog from which to retrieve Databases. If None is provided, the AWS account ID is used by default.

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*, *database_input_args: dict[str, Any] | None = None*, *boto3_session: Session | None = None*) → None

Create a database in AWS Glue Catalog.

Parameters

- **name** (str) – Database name.
- **description** (str | None) – A description for the Database.
- **catalog_id** (str | None) – 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.
- **database_input_args** (dict[str, Any] | None) – Additional metadata to pass to database creation. Supported arguments listed here: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.create_database
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive 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*, *columns_parameters: dict[str, dict[str, str]] | None = None*, *mode: Literal['overwrite', 'append'] = 'overwrite'*, *catalog_versioning: bool = False*, *schema_evolution: bool = False*, *serde_library: str | None = None*, *serde_parameters: dict[str, str] | None = None*, *boto3_session: Session | None = None*, *athena_partition_projection_settings: AthenaPartitionProjectionSettings | None = None*, *catalog_id: str | None = None*) → None

Create a JSON Table (Metadata Only) in the AWS Glue Catalog.

'https://docs.aws.amazon.com/athena/latest/ug/data-types.html'

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 | None) – The type of the Glue Table. Set to EXTERNAL_TABLE if None.
- **partitions_types** (dict[str, str] | None) – Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).
- **bucketing_info** (Tuple[List[str], int] | None) – 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 | None) – Compression style (None, gzip, etc).
- **description** (str | None) – Table description
- **parameters** (dict[str, str] | None) – Key/value pairs to tag the table.
- **columns_comments** (dict[str, str] | None) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **columns_parameters** (dict[str, dict[str, str]] | None) – Columns names and the related parameters (e.g. {'col0': {'par0': 'Param 0', 'par1': 'Param 1'}}).

- **mode** (Literal['overwrite', 'append']) – '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.
- **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/3.16.1/tutorials/014%20-%20Schema%20Evolution.html
- **serde_library** (str | None) – 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** (dict[str, str] | None) – Dictionary of initialization parameters for the SerDe. The default is *{"field.delim": sep, "escape.delim": "\"}*.
- **athena_partition_projection_settings** (*AthenaPartitionProjectionSettings* | None) –

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 of the Athena Partition Projection (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html). AthenaPartitionProjectionSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaPartitionProjectionSettings or as a regular Python dict.

Following projection parameters are supported:

Table 11: Projection Parameters

Name	Type	Description
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_interva	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_format	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_storag	Optional[s	Value which is allows Athena to properly map partition values if the S3 file locations do not follow

Return type

None 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

The default boto3 session will be used if **boto3_session** receive `None`.

catalog_id

The ID of the Data Catalog from which to retrieve Databases. If `None` is provided, the AWS account ID is used by default.

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*, *columns_parameters: dict[str, dict[str, str]] | None = None*, *mode: Literal['overwrite', 'append'] = 'overwrite'*, *catalog_versioning: bool = False*, *athena_partition_projection_settings: AthenaPartitionProjectionSettings | None = None*, *boto3_session: Session | None = None*) → None

Create a Parquet Table (Metadata Only) in the AWS Glue Catalog.

'https://docs.aws.amazon.com/athena/latest/ug/data-types.html'

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 | None) – The type of the Glue Table. Set to EXTERNAL_TABLE if None.
- **partitions_types** (dict[str, str] | None) – Dictionary with keys as partition names and values as data types (e.g. {'col2': 'date'}).
- **bucketing_info** (Tuple[List[str], int] | None) – 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 | None) – 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 | None) – Compression style (None, snappy, gzip, etc).

- **description** (str | None) – Table description
- **parameters** (dict[str, str] | None) – Key/value pairs to tag the table.
- **columns_comments** (dict[str, str] | None) – Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).
- **columns_parameters** (dict[str, dict[str, str]] | None) – Columns names and the related parameters (e.g. {'col0': {'par0': 'Param 0', 'par1': 'Param 1'}}).
- **mode** (Literal['overwrite', 'append']) – '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.
- **athena_partition_projection_settings** (*AthenaPartitionProjectionSettings* | None) –

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 of the Athena Partition Projection (https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html). AthenaPartitionProjectionSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaPartitionProjectionSettings or as a regular Python dict.

Following projection parameters are supported:

Table 12: Projection Parameters

Name	Type	Description
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_interva	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_format	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_storag	Optional[s	Value which is allows Athena to properly map partition values if the S3 file locations do not follow

Return type

None 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

The default boto3 session will be used if **boto3_session** receive 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'},
```

(continues on next page)

(continued from previous page)

```
...     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*) → DataFrame

Get a Pandas DataFrame with all listed databases.

Parameters

- **limit** (int) – Max number of tables to be returned.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
DataFrame

Returns
Pandas DataFrame filled by formatted table information.

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*, *boto3_session: Session | None = None*, *catalog_id: str | None = None*) → None

Delete a column in a AWS Glue Catalog table.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **column_name** (str) – Column name
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.
- **catalog_id** (str | None) – The ID of the Data Catalog from which to retrieve Databases. If None is provided, the AWS account ID is used by default.

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*) → None

Delete a database in AWS Glue Catalog.

Parameters

- **name** (str) – Database name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive 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*) → None

Delete specified partitions in a AWS Glue Catalog table.

Parameters

- **table** (str) – Table name.
- **database** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive 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*) → list[list[str]]

Delete all partitions in a AWS Glue Catalog table.

Parameters

- **table** (str) – Table name.
- **database** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
list[list[str]]

Returns
Partitions values.

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*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → bool

Delete Glue table if exists.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
bool

Returns
True if deleted, otherwise False.

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*) → bool

Check if the table exists.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.
- **catalog_id** (str | None) – The ID of the Data Catalog from which to retrieve Databases. If None is provided, the AWS account ID is used by default.

Return type
bool

Returns
True if exists, otherwise False.

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 (DataFrame) – Original Pandas DataFrame.

Return type

DataFrame

Returns

Pandas DataFrame without duplicated columns.

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** (DataFrame) – Pandas DataFrame.
- **index** (bool) – Should consider the DataFrame index as a column?.
- **partition_cols** (list[str] | None) – List of partitions names.

- **dtype** (dict[str, str] | None) – 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) – File format to be considered to place the index column: "parquet" | "csv".

Return type
tuple[dict[str, str], dict[str, str]]

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'}).

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*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → dict[str, str | None]

Get all columns comments.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, str | None]

Returns
Columns comments. e.g. {"col1": "foo boo bar", "col2": None}.

Examples

```
>>> import awswrangler as wr
>>> pars = wr.catalog.get_columns_comments(database="...", table="...")
```

awswrangler.catalog.get_columns_parameters

awswrangler.catalog.**get_columns_parameters**(*database: str*, *table: str*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → dict[str, dict[str, str] | None]

Get all columns parameters.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, dict[str, str] | None]

Returns
Columns parameters.

Examples

```
>>> import awswrangler as wr
>>> pars = wr.catalog.get_columns_parameters(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*) → dict[str, list[str]]

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

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **expression** (str | None) – An expression that filters the partitions to be returned.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, list[str]]

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']}).

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 | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type

Iterator[dict[str, Any]]

Returns

Iterator of Databases.

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*) → dict[str, list[str]]

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

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **expression** (str | None) – An expression that filters the partitions to be returned.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type

dict[str, list[str]]

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']}).

Examples

Fetch all partitions

```
>>> import awswrangler as wr
>>> wr.catalog.get_parquet_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_parquet_partitions(
```

(continues on next page)

(continued from previous page)

```
...     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*) → dict[str, list[str]]

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

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **expression** (str | None) – An expression that filters the partitions to be returned.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type

dict[str, list[str]]

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']}).

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*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → str | None

Get table description.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
str | None

Returns
Description if exists.

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*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → str

Get table's location on Glue catalog.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
str

Returns
Table's location.

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*) → int

Get total number of versions.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
int

Returns
Total number of versions.

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*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → dict[str, str]

Get all parameters.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.

- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, str]

Returns
Dictionary of parameters.

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*, *catalog_id: str | None = None*, *filter_iceberg_current: bool = False*, *boto3_session: Session | None = None*) → dict[str, str] | None

Get all columns and types from a table.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – The ID of the Data Catalog from which to retrieve Databases. If None is provided, the AWS account ID is used by default.
- **filter_iceberg_current** (bool) – If True, returns only current iceberg fields (fields marked with iceberg.field.current: true). Otherwise, returns the all fields. False by default (return all fields).
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, str] | None

Returns
If table exists, a dictionary like {'col name': 'col data type'}. Otherwise None.

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*) → list[dict[str, Any]]

Get all versions.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
list[dict[str, Any]]

Returns
List of table inputs: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_table_versions

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*, *name_contains: str | None = None*, *name_prefix: str | None = None*, *name_suffix: str | None = None*, *boto3_session: Session | None = None*) → Iterator[dict[str, 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.

Parameters

- **catalog_id** (str | None) – 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 | None) – Database name.
- **name_contains** (str | None) – Select by a specific string on table name
- **name_prefix** (str | None) – Select by a specific prefix on table name
- **name_suffix** (str | None) – Select by a specific suffix on table name
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
Iterator[dict[str, Any]]

Returns
Iterator of tables.

Examples

```
>>> import awswrangler as wr
>>> tables = wr.catalog.get_tables()
```

awswrangler.catalog.overwrite_table_parameters

awswrangler.catalog.**overwrite_table_parameters**(*parameters: dict[str, str]*, *database: str*, *table: str*, *catalog_versioning: bool = False*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → dict[str, str]

Overwrite all existing parameters.

Parameters

- **parameters** (dict[str, str]) – e.g. {"source": "mysql", "destination": "datalake"}
- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_versioning** (bool) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, str]

Returns
All parameters after the overwrite (The same received).

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.

Return type
str

Returns
Normalized column name.

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** (DataFrame) – Original Pandas DataFrame.
- **handle_duplicate_columns** (str | None) – 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”.

Return type
DataFrame

Returns
Original Pandas DataFrame with columns names normalized.

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.

Return type
str

Returns
Normalized table name.

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.

Parameters

- **text** (str) – Select only tables with the given string in table's properties.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
Iterator[dict[str, Any]]

Returns
Iterator of tables.

Examples

```
>>> import awswrangler as wr
>>> df_tables = wr.catalog.search_tables(text='my_property')
```

awswrangler.catalog.table

awswrangler.catalog.**table**(*database: str*, *table: str*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → DataFrame

Get table details as Pandas DataFrame.

Parameters

- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
DataFrame

Returns
Pandas DataFrame filled by formatted table information.

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*, *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*) → DataFrame

Get a DataFrame with tables filtered by a search term, prefix, suffix.

Parameters

- **limit** (int) – Max number of tables to be returned.
- **catalog_id** (str | None) – 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 | None) – Database name.
- **search_text** (str | None) – Select only tables with the given string in table's properties.

- **name_contains** (str | None) – Select by a specific string on table name
- **name_prefix** (str | None) – Select by a specific prefix on table name
- **name_suffix** (str | None) – Select by a specific suffix on table name
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
DataFrame

Returns
Pandas DataFrame filled by formatted table information.

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*, *catalog_versioning: bool = False*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → dict[str, str]

Insert or Update the received parameters.

Parameters

- **parameters** (dict[str, str]) – e.g. {"source": "mysql", "destination": "datalake"}
- **database** (str) – Database name.
- **table** (str) – Table name.
- **catalog_versioning** (bool) – If True and *mode="overwrite"*, creates an archived version of the table catalog before updating it.
- **catalog_id** (str | None) – 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** (Session | None) – Boto3 Session. The default boto3 session will be used if boto3_session receive None.

Return type
dict[str, str]

Returns
All parameters after the upsert.

Examples

```
>>> import awswrangler as wr
>>> pars = wr.catalog.upsert_table_parameters(
...     parameters={"source": "mysql", "destination":  "datalake"},
...     database="...",
...     table="...",
... )
```

1.6.3 Amazon Athena

create_athena_bucket([boto3_session])	Create the default Athena bucket if it doesn't exist.
create_spark_session(workgroup[, ...])	Create session and wait until ready to accept calculations.
create_ctas_table(sql[, database, ...])	Create a new table populated with the results of a SELECT query.
generate_create_query(table[, database, ...])	Generate the query that created a table(EXTERNAL_TABLE) or a view(VIRTUAL_TABLE).
get_query_columns_types(query_execution_id)	Get the data type of all columns queried.
get_query_execution(query_execution_id[, ...])	Fetch query execution details.
get_query_executions(query_execution_ids[, ...])	From specified query execution IDs, return a DataFrame of query execution details.
get_query_results(query_execution_id[, ...])	Get AWS Athena SQL query results as a Pandas DataFrame.
get_named_query_statement(named_query_id[, ...])	Get the named query statement string from a query ID.
get_work_group(workgroup[, boto3_session])	Return information about the workgroup with the specified name.
list_query_executions([workgroup, ...])	Fetch list query execution IDs ran in specified workgroup or primary work group if not specified.
read_sql_query(sql, database[, ...])	Execute any SQL query on AWS Athena and return the results as a Pandas DataFrame.
read_sql_table(table, database[, ...])	Extract the full table AWS Athena and return the results as a Pandas DataFrame.
repair_table(table[, database, data_source, ...])	Run the Hive's metastore consistency check: 'MSCK REPAIR TABLE table;'.
run_spark_calculation(code, workgroup[, ...])	Execute Spark Calculation and wait for completion.
show_create_table(table[, database, ...])	Generate the query that created it: 'SHOW CREATE TABLE table;'.
start_query_execution(sql[, database, ...])	Start a SQL Query against AWS Athena.
stop_query_execution(query_execution_id[, ...])	Stop a query execution.
to_iceberg(df, database, table[, temp_path, ...])	Insert into Athena Iceberg table using INSERT INTO .
delete_from_iceberg_table(df, database, ...)	Delete rows from an Iceberg table.
unload(sql, path, database[, file_format, ...])	Write query results from a SELECT statement to the specified data format using UNLOAD.
wait_query(query_execution_id[, ...])	Wait for the query end.
create_prepared_statement(sql, statement_name)	Create a SQL statement with the name statement_name to be run at a later time.
list_prepared_statements([workgroup, ...])	List the prepared statements in the specified workgroup.
delete_prepared_statement(statement_name[, ...])	Delete the prepared statement with the specified name from the specified workgroup.

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.

The bucket name is derived from the caller's account ID and region (`aws-athena-query-results-{account_id}-{region}`). Because S3 bucket names are global, this function verifies that the bucket is owned by the caller's account before returning; if another account owns it, a `InvalidArgumentValue` is raised to prevent query results from being written to a bucket controlled by a third party.

Parameters

boto3_session (`Session | None`) – The default boto3 session will be used if **boto3_session** receive `None`.

Return type

`str`

Returns

Bucket s3 path (E.g. s3://aws-athena-query-results-ACCOUNT-REGION/)

Examples

```
>>> import awswrangler as wr
>>> wr.athena.create_athena_bucket()
's3://aws-athena-query-results-ACCOUNT-REGION/'
```

awswrangler.athena.create_spark_session

awswrangler.athena.**create_spark_session**(*workgroup: str*, *coordinator_dpu_size: int = 1*, *max_concurrent_dpus: int = 5*, *default_executor_dpu_size: int = 1*, *additional_configs: dict[str, Any] | None = None*, *spark_properties: dict[str, Any] | None = None*, *notebook_version: str | None = None*, *idle_timeout: int = 15*, *boto3_session: Session | None = None*) → str

Create session and wait until ready to accept calculations.

Parameters

- **workgroup** (`str`) – Athena workgroup name. Must be Spark-enabled.
- **coordinator_dpu_size** (`int`) – The number of DPUs to use for the coordinator. A coordinator is a special executor that orchestrates processing work and manages other executors in a notebook session. The default is 1.
- **max_concurrent_dpus** (`int`) – The maximum number of DPUs that can run concurrently. The default is 5.
- **default_executor_dpu_size** (`int`) – The default number of DPUs to use for executors. The default is 1.
- **additional_configs** (`dict[str, Any] | None`) – Contains additional engine parameter mappings in the form of key-value pairs.
- **spark_properties** (`dict[str, Any] | None`) – Contains SparkProperties in the form of key-value pairs.Specifies custom jar files and Spark properties for use cases like cluster encryption, table formats, and general Spark tuning.

- **notebook_version** (str | None) – The notebook version. This value is supplied automatically for notebook sessions in the Athena console and is not required for programmatic session access. The only valid notebook version is Athena notebook version 1. If you specify a value for NotebookVersion, you must also specify a value for NotebookId
- **idle_timeout** (int) – The idle timeout in minutes for the session. The default is 15.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
str

Returns
Session ID

Examples

```
>>> import awswrangler as wr
>>> df = wr.athena.create_spark_session(workgroup="...", max_concurrent_dpus=10)
```

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 = 'primary'*, *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 = 1.0*, *execution_params: list[str] | None = None*, *params: dict[str, Any] | list[str] | None = None*, *paramstyle: Literal['qmark', 'named'] = 'named'*, *boto3_session: Session | None = None*) → dict[str, str | awswrangler.athena._utils._QueryMetadata]

Create a new table populated with the results of a SELECT query.

https://docs.aws.amazon.com/athena/latest/ug/create-table-as.html

Parameters

- **sql** (str) – SELECT SQL query.
- **database** (str | None) – The name of the database where the original table is stored.
- **ctas_table** (str | None) – The name of the CTAS table. If None, a name with a random string is used.
- **ctas_database** (str | None) – 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** (str | None) – 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** (str | None) – The storage format for the CTAS query results, such as ORC, PARQUET, AVRO, JSON, or TEXTFILE. PARQUET by default.
- **write_compression** (str | None) – The compression type to use for any storage format that allows compression to be specified.
- **partitioning_info** (list[str] | None) – A list of columns by which the CTAS table will be partitioned.
- **bucketing_info** (Tuple[List[str], int] | None) – 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** (str | None) – The single-character field delimiter for files in CSV, TSV, and text files.
- **schema_only** (bool) – _description_, by default False
- **workgroup** (str) – Athena workgroup. Primary by default.
- **data_source** (str | None) – Data Source / Catalog name. If None, 'AwsDataCatalog' is used.
- **encryption** (str | None) – Valid values: [None, 'SSE_S3', 'SSE_KMS']. Note: 'CSE_KMS' is not supported.
- **kms_key** (str | None) – For SSE-KMS, this is the KMS key ARN or ID.
- **categories** (list[str] | None) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **wait** (bool) – Whether to wait for the query to finish and return a dictionary with the Query metadata.
- **athena_query_wait_polling_delay** (float) – Interval in seconds for how often the function will check if the Athena query has completed.
- **execution_params** (list[str] | None) – [**DEPRECATED**] A list of values for the parameters that are used in the SQL query. This parameter is on a deprecation path. Use params and *paramstyle* ` instead.
- **params** (dict[str, Any] | list[str] | None) – Dictionary or list of parameters to pass to execute method. The syntax used to pass parameters depends on the configuration of paramstyle.
- **paramstyle** (Literal['qmark', 'named']) – The syntax style to use for the parameters. Supported values are named and qmark. The default is named.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, str | _QueryMetadata]

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.

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*, *catalog_id: str | None = None*, *boto3_session: Session | None = None*) → str

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.

Parameters

- **table** (str) – Table name.
- **database** (str | None) – Database name.
- **catalog_id** (str | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
str

Returns
The query that created the table or view.

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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, str]

Returns
Dictionary with all data types.

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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, Any]

Returns
Dictionary with the get_query_execution response.

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[pandas.DataFrame, pandas.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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
tuple[DataFrame, DataFrame] | DataFrame

Returns
DataFrame containing either information about query execution details. Optionally, another DataFrame containing 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*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *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 = 1.0*) → DataFrame | Iterator[DataFrame]

Get AWS Athena SQL query results as a Pandas DataFrame.

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** (`Session | None`) – The default boto3 session will be used if **boto3_session** receive `None`.
- **categories** (`list[str] | None`) – List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.
- **dtype_backend** (`Literal['numpy_nullable', 'pyarrow']`) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimential. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **chunksize** (`int | bool | None`) – 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** (`dict[str, Any] | None`) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **pyarrow_additional_kwargs** (`dict[str, Any] | None`) – Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.
- **athena_query_wait_polling_delay** (`float`) – Interval in seconds for how often the function will check if the Athena query has completed.

Return type

`DataFrame | Iterator[DataFrame]`

Returns

Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
str

Returns
The named query statement string

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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

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

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*, *max_results: int | 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 | None) – 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 workgroup is returned.
- **max_results** (int | None) – The maximum number of query execution IDs to return in this request. If not present, all execution IDs will be returned.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
List of query execution IDs.

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.list_query_executions(workgroup='workgroup-name')
```

awswrangler.athena.read_sql_query

awswrangler.athena.**read_sql_query**(*sql: str*, *database: str*, *ctas_approach: bool = True*, *unload_approach: bool = False*, *ctas_parameters:* AthenaCTASSettings | *None = None*, *unload_parameters:* AthenaUNLOADSettings | *None = None*, *categories: list[str] | None = None*, *chunksize: int | bool | None = None*, *s3_output: str | None = None*, *workgroup: str = 'primary'*, *encryption: str | None = None*, *kms_key: str | None = None*, *keep_files: bool = True*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*, *client_request_token: str | None = None*, *athena_cache_settings:* AthenaCacheSettings | *None = None*, *data_source: str | None = None*, *athena_query_wait_polling_delay: float = 1.0*, *params: dict[str, Any] | list[str] | None = None*, *paramstyle: Literal['qmark', 'named'] = 'named'*, *result_reuse_configuration: dict[str, Any] | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *s3_additional_kwargs: dict[str, Any] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*) → DataFrame | Iterator[DataFrame]

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**, depending on the size of the data, one or more data frames are returned per file in the query result. Unlike **chunksize=INTEGER**, rows from different files are not mixed in the resulting data frames.
- If **chunksize=INTEGER**, awswrangler iterates on the data by number of rows equal to 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 data frame.

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().

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.
- **ctas_parameters** (*AthenaCTASSettings* | None) –

  Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

  – boto3_session

  – s3_additional_kwargs

  This function has arguments which can be configured globally through *wr.config* or environment variables:

  – ctas_approach

  – database

  – athena_cache_settings

  – athena_query_wait_polling_delay

  – workgroup

  – chunksize

  – dtype_backend

  Check out the Global Configurations Tutorial for details.

  Parameters of the CTAS such as database, temp_table_name, bucketing_info, and compression.

unload_parameters
Parameters of the UNLOAD such as format, compression, field_delimiter, and partitioned_by.

categories
List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.

chunksize
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
Amazon S3 path. Not required for the regular query path (*ctas_approach=False*, *unload_approach=False*) when the workgroup uses managed query results. Still used for CTAS/UNLOAD paths.

workgroup
Athena workgroup. Primary by default.

encryption
Valid values: [None, 'SSE_S3', 'SSE_KMS']. Notice: 'CSE_KMS' is not supported.

kms_key
For SSE-KMS, this is the KMS key ARN or ID.

keep_files
Whether staging files produced by Athena are retained. 'True' by default.

use_threads
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
The default boto3 session will be used if **boto3_session** receive `None`.

client_request_token
A unique case-sensitive string used to ensure the request to create the query is idempotent (executes only once). If another StartQueryExecution request is received, the same response is returned and another query is not created. If a parameter has changed, for example, the QueryString , an error is returned. If you pass the same client_request_token value with different parameters the query fails with error message "Idempotent parameters do not match". Use this only with ctas_approach=False and unload_approach=False and disabled cache.

athena_cache_settings
Parameters of the Athena cache settings such as max_cache_seconds, max_cache_query_inspections, max_remote_cache_entries, and max_local_cache_entries. AthenaCacheSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaCacheSettings or as a regular Python dict. 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.

data_source
Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.

athena_query_wait_polling_delay
Interval in seconds for how often the function will check if the Athena query has completed.

params
Parameters that will be used for constructing the SQL query. Only named or question mark parameters are supported. The parameter style needs to be specified in the `paramstyle` parameter.

For `paramstyle="named"`, this value needs to be a dictionary. The dict needs to contain the information in the form `{'name': 'value'}` and the SQL query needs to contain `:name`. The formatter will be applied client-side in this scenario.

For `paramstyle="qmark"`, this value needs to be a list of strings. The formatter will be applied server-side. The values are applied sequentially to the parameters in the query in the order in which the parameters occur.

paramstyle
Determines the style of `params`. Possible values are:

- `named`

- qmark

result_reuse_configuration
A structure that contains the configuration settings for reusing query results. This parameter is only valid when both *ctas_approach* and *unload_approach* are set to *False*. See also: https://docs.aws.amazon.com/athena/latest/ug/reusing-query-results.html

dtype_backend
Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

s3_additional_kwargs
Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}

pyarrow_additional_kwargs
Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

Return type
DataFrame | Iterator[DataFrame]

Returns
Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

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"}
... )
```

```
>>> import awswrangler as wr
>>> df = wr.athena.read_sql_query(
...     sql="...",
...     database="...",
...     athena_cache_settings={
...         "max_cache_seconds": 90,
...     },
... )
```

awswrangler.athena.read_sql_table

awswrangler.athena.**read_sql_table**(*table: str*, *database: str*, *unload_approach: bool = False*, *unload_parameters:* AthenaUNLOADSettings | *None = None*, *ctas_approach: bool = True*, *ctas_parameters:* AthenaCTASSettings | *None = None*, *categories: list[str] | None = None*, *chunksize: int | bool | None = None*, *s3_output: str | None = None*, *workgroup: str = 'primary'*, *encryption: str | None = None*, *kms_key: str | None = None*, *keep_files: bool = True*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*, *client_request_token: str | None = None*, *athena_cache_settings:* AthenaCacheSettings | *None = None*, *data_source: str | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *s3_additional_kwargs: dict[str, Any] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*) → DataFrame | Iterator[DataFrame]

Extract the full table 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**, depending on the size of the data, one or more data frames are returned per file in the query result. Unlike **chunksize=INTEGER**, rows from different files are not mixed in the resulting data frames.
- If **chunksize=INTEGER**, awswrangler iterates on the data by number of rows equal to 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 data frame.

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().

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.
- **ctas_parameters** (*AthenaCTASSettings* | None) –

  Note: Following arguments are not supported in distributed mode with engine *EngineEnum.RAY*:

  – boto3_session

  – s3_additional_kwargs

  This function has arguments which can be configured globally through *wr.config* or environment variables:

  – ctas_approach

  – database

  – athena_cache_settings

  – workgroup

  – chunksize

  – dtype_backend

  Check out the Global Configurations Tutorial for details.

  Parameters of the CTAS such as database, temp_table_name, bucketing_info, and compression.

unload_parameters
Parameters of the UNLOAD such as format, compression, field_delimiter, and partitioned_by.

categories
List of columns names that should be returned as pandas.Categorical. Recommended for memory restricted environments.

chunksize
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
AWS S3 path.

workgroup
Athena workgroup. Primary by default.

encryption
Valid values: [None, 'SSE_S3', 'SSE_KMS']. Notice: 'CSE_KMS' is not supported.

kms_key
For SSE-KMS, this is the KMS key ARN or ID.

keep_files
Should awswrangler delete or keep the staging files produced by Athena?

use_threads
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
The default boto3 session will be used if **boto3_session** receive `None`.

client_request_token
A unique case-sensitive string used to ensure the request to create the query is idempotent (executes only once). If another StartQueryExecution request is received, the same response is returned and another query is not created. If a parameter has changed, for example, the QueryString , an error is returned. If you pass the same client_request_token value with different parameters the query fails with error message "Idempotent parameters do not match". Use this only with ctas_approach=False and unload_approach=False and disabled cache.

athena_cache_settings
Parameters of the Athena cache settings such as max_cache_seconds, max_cache_query_inspections, max_remote_cache_entries, and max_local_cache_entries. AthenaCacheSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaCacheSettings or as a regular Python dict. 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.

data_source
Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.

dtype_backend
Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

s3_additional_kwargs
Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}

pyarrow_additional_kwargs
Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

Return type
`DataFrame | Iterator[DataFrame]`

Returns
Pandas DataFrame or Generator of Pandas DataFrames if chunksize is passed.

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 = 'primary'*, *encryption: str | None = None*, *kms_key: str | None = None*, *athena_query_wait_polling_delay: float = 1.0*, *boto3_session: Session | None = None*) → str

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/)

Parameters

- **table** (str) – Table name.
- **database** (str | None) – AWS Glue/Athena database name.
- **data_source** (str | None) – Data Source / Catalog name. If None, 'AwsDataCatalog' is used.
- **s3_output** (str | None) – AWS S3 path.
- **workgroup** (str) – Athena workgroup. Primary by default.
- **encryption** (str | None) – None, 'SSE_S3', 'SSE_KMS', 'CSE_KMS'.
- **kms_key** (str | None) – For SSE-KMS and CSE-KMS , this is the KMS key ARN or ID.
- **athena_query_wait_polling_delay** (float) – Interval in seconds for how often the function will check if the Athena query has completed.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type

str

Returns

Query final state ('SUCCEEDED', 'FAILED', 'CANCELLED').

Examples

```
>>> import awswrangler as wr
>>> query_final_state = wr.athena.repair_table(table='...', database='...')
```

awswrangler.athena.run_spark_calculation

awswrangler.athena.**run_spark_calculation**(*code: str*, *workgroup: str*, *session_id: str | None = None*, *coordinator_dpu_size: int = 1*, *max_concurrent_dpus: int = 5*, *default_executor_dpu_size: int = 1*, *additional_configs: dict[str, Any] | None = None*, *spark_properties: dict[str, Any] | None = None*, *notebook_version: str | None = None*, *idle_timeout: int = 15*, *boto3_session: Session | None = None*) → dict[str, Any]

Execute Spark Calculation and wait for completion.

Parameters

- **code** (str) – A string that contains the code for the calculation.
- **workgroup** (str) – Athena workgroup name. Must be Spark-enabled.
- **session_id** (str | None) – The session id. If not passed, a session will be started.
- **coordinator_dpu_size** (int) – The number of DPUs to use for the coordinator. A coordinator is a special executor that orchestrates processing work and manages other executors in a notebook session. The default is 1.
- **max_concurrent_dpus** (int) – The maximum number of DPUs that can run concurrently. The default is 5.
- **default_executor_dpu_size** (int) – The default number of DPUs to use for executors. The default is 1.
- **additional_configs** (dict[str, Any] | None) – Contains additional engine parameter mappings in the form of key-value pairs.
- **spark_properties** (dict[str, Any] | None) – Contains SparkProperties in the form of key-value pairs.Specifies custom jar files and Spark properties for use cases like cluster encryption, table formats, and general Spark tuning.
- **notebook_version** (str | None) – The notebook version. This value is supplied automatically for notebook sessions in the Athena console and is not required for programmatic session access. The only valid notebook version is Athena notebook version 1. If you specify a value for NotebookVersion, you must also specify a value for NotebookId
- **idle_timeout** (int) – The idle timeout in minutes for the session. The default is 15.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
dict[str, Any]

Returns
Calculation response

Examples

```
>>> import awswrangler as wr
>>> df = wr.athena.run_spark_calculation(
...     code="print(spark)",
...     workgroup="...",
... )
```

awswrangler.athena.show_create_table

awswrangler.athena.**show_create_table**(*table: str*, *database: str | None = None*, *s3_output: str | None = None*, *workgroup: str = 'primary'*, *encryption: str | None = None*, *kms_key: str | None = None*, *athena_query_wait_polling_delay: float = 1.0*, *s3_additional_kwargs: dict[str, Any] | None = None*, *boto3_session: Session | None = None*) → str

Generate the query that created it: 'SHOW CREATE TABLE table;'.

Analyzes an existing table named table_name to generate the query that created it.

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/)

Parameters

- **table** (str) – Table name.
- **database** (str | None) – AWS Glue/Athena database name.
- **s3_output** (str | None) – AWS S3 path.
- **workgroup** (str) – Athena workgroup. Primary by default.
- **encryption** (str | None) – None, 'SSE_S3', 'SSE_KMS', 'CSE_KMS'.
- **kms_key** (str | None) – For SSE-KMS and CSE-KMS , this is the KMS key ARN or ID.
- **athena_query_wait_polling_delay** (float) – Interval in seconds for how often the function will check if the Athena query has completed.
- **s3_additional_kwargs** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type

str

Returns

The query that created the table.

Examples

```
>>> import awswrangler as wr
>>> df_table = wr.athena.show_create_table(table='my_table', database='default')
```

awswrangler.athena.start_query_execution

awswrangler.athena.**start_query_execution**(*sql: str*, *database: str | None = None*, *s3_output: str | None = None*, *workgroup: str = 'primary'*, *encryption: str | None = None*, *kms_key: str | None = None*, *params: dict[str, Any] | list[str] | None = None*, *paramstyle: Literal['qmark', 'named'] = 'named'*, *result_reuse_configuration: dict[str, Any] | None = None*, *boto3_session: Session | None = None*, *client_request_token: str | None = None*, *athena_cache_settings: AthenaCacheSettings | None = None*, *athena_query_wait_polling_delay: float = 1.0*, *data_source: str | None = None*, *wait: bool = False*) → str | dict[str, Any]

Start a SQL Query against AWS Athena.

Note: Create the default Athena bucket if it doesn't exist and s3_output is None. Not required when the workgroup uses managed query results. (E.g. s3://aws-athena-query-results-ACCOUNT-REGION/)

Parameters

- **sql** (str) – SQL query.
- **database** (str | None) – AWS Glue/Athena database name.
- **s3_output** (str | None) – AWS S3 path. Not required when the workgroup uses managed query results.
- **workgroup** (str) – Athena workgroup. Primary by default.
- **encryption** (str | None) – None, 'SSE_S3', 'SSE_KMS', 'CSE_KMS'.
- **kms_key** (str | None) – For SSE-KMS and CSE-KMS , this is the KMS key ARN or ID.
- **params** (dict[str, Any] | list[str] | None) –

  Note: This function has arguments which can be configured globally through *wr.config* or environment variables:

  – database

  – athena_cache_settings

  – athena_query_wait_polling_delay

  – workgroup

  Check out the Global Configurations Tutorial for details.

  Parameters that will be used for constructing the SQL query. Only named or question mark parameters are supported. The parameter style needs to be specified in the paramstyle parameter.

For `paramstyle="named"`, this value needs to be a dictionary. The dict needs to contain the information in the form `{'name': 'value'}` and the SQL query needs to contain `:name`. The formatter will be applied client-side in this scenario.

For `paramstyle="qmark"`, this value needs to be a list of strings. The formatter will be applied server-side. The values are applied sequentially to the parameters in the query in the order in which the parameters occur.

paramstyle
Determines the style of `params`. Possible values are:

- `named`
- `qmark`

result_reuse_configuration
A structure that contains the configuration settings for reusing query results. See also: https://docs.aws.amazon.com/athena/latest/ug/reusing-query-results.html

boto3_session
The default boto3 session will be used if **boto3_session** receive `None`.

client_request_token
A unique case-sensitive string used to ensure the request to create the query is idempotent (executes only once). If another StartQueryExecution request is received, the same response is returned and another query is not created. If a parameter has changed, for example, the QueryString , an error is returned. If you pass the same client_request_token value with different parameters the query fails with error message "Idempotent parameters do not match". Use this only with ctas_approach=False and unload_approach=False and disabled cache.

athena_cache_settings
Parameters of the Athena cache settings such as max_cache_seconds, max_cache_query_inspections, max_remote_cache_entries, and max_local_cache_entries. AthenaCacheSettings is a *TypedDict*, meaning the passed parameter can be instantiated either as an instance of AthenaCacheSettings or as a regular Python dict. 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.

athena_query_wait_polling_delay
Interval in seconds for how often the function will check if the Athena query has completed.

data_source
Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.

wait
Indicates whether to wait for the query to finish and return a dictionary with the query execution response.

Return type
`str | dict[str, Any]`

Returns
Query execution ID if *wait* is set to *False*, dictionary with the get_query_execution response otherwise.

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** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> wr.athena.stop_query_execution(query_execution_id='query-execution-id')
```

awswrangler.athena.to_iceberg

awswrangler.athena.**to_iceberg**(*df: DataFrame*, *database: str*, *table: str*, *temp_path: str | None = None*, *index: bool = False*, *table_location: str | None = None*, *partition_cols: list[str] | None = None*, *merge_cols: list[str] | None = None*, *merge_condition: Literal['update', 'ignore'] = 'update'*, *merge_match_nulls: bool = False*, *keep_files: bool = True*, *data_source: str | None = None*, *s3_output: str | None = None*, *workgroup: str = 'primary'*, *mode: Literal['append', 'overwrite', 'overwrite_partitions'] = 'append'*, *encryption: str | None = None*, *kms_key: str | None = None*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*, *additional_table_properties: dict[str, Any] | None = None*, *dtype: dict[str, str] | None = None*, *catalog_id: str | None = None*, *schema_evolution: bool = False*, *fill_missing_columns_in_df: bool = True*, *glue_table_settings: GlueTableSettings | None = None*) → None

Insert into Athena Iceberg table using INSERT INTO … SELECT. Will create Iceberg table if it does not exist.

Creates temporary external table, writes staged files and inserts via INSERT INTO … SELECT.

Parameters

- **df** (DataFrame) – Pandas DataFrame.
- **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*).
- **table** (str) – AWS Glue/Athena table name.
- **temp_path** (str | None) – Amazon S3 location to store temporary results. Workgroup config will be used if not provided.
- **index** (bool) – Should consider the DataFrame index as a column?.
- **table_location** (str | None) – Amazon S3 location for the table. Will only be used to create a new table if it does not exist.
- **partition_cols** (list[str] | None) – List of column names that will be used to create partitions, including support for transform functions (e.g. "day(ts)").

  https://docs.aws.amazon.com/athena/latest/ug/querying-iceberg-creating-tables.html#querying-iceberg-partitioning
- **merge_cols** (list[str] | None) – List of column names that will be used for conditional inserts and updates.

  https://docs.aws.amazon.com/athena/latest/ug/merge-into-statement.html
- **merge_condition** (Literal['update', 'ignore']) – The condition to be used in the MERGE INTO statement. Valid values: ['update', 'ignore']. Default is update.
- **merge_match_nulls** (bool) – Instruct whether to have nulls in the merge condition match other nulls.
- **keep_files** (bool) – Whether staging files produced by Athena are retained. Default is True.
- **data_source** (str | None) – Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.
- **s3_output** (str | None) – Amazon S3 path used for query execution.
- **workgroup** (str) – Athena workgroup. Primary by default.
- **mode** (Literal['append', 'overwrite', 'overwrite_partitions']) – append (default), overwrite, overwrite_partitions.
- **encryption** (str | None) – Valid values: [None, 'SSE_S3', 'SSE_KMS']. Notice: 'CSE_KMS' is not supported.
- **kms_key** (str | None) – For SSE-KMS, this is the KMS key ARN or ID.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.
- **pyarrow_additional_kwargs** (dict[str, Any] | None) – Additional parameters forwarded to pyarrow. e.g. pyarrow_additional_kwargs={'coerce_timestamps': 'ns', 'use_deprecated_int96_timestamps': False, 'allow_truncated_timestamps'=False}

- **s3_additional_kwargs** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'RequestPayer': 'requester'}
- **additional_table_properties** (dict[str, Any] | None) – Additional table properties. e.g. additional_table_properties={'write_target_data_file_size_bytes': '536870912'}

  https://docs.aws.amazon.com/athena/latest/ug/querying-iceberg-creating-tables.html#querying-iceberg-table-properties
- **dtype** (dict[str, str] | None) – 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'}
- **catalog_id** (str | None) – The ID of the Data Catalog from which to retrieve Databases. If none is provided, the AWS account ID is used by default.
- **schema_evolution** (bool) – If True allows schema evolution for new columns or changes in column types. Columns missing from the DataFrame that are present in the Iceberg schema will throw an error unless fill_missing_columns_in_df is set to True. Default is False.
- **fill_missing_columns_in_df** (bool) – If True, fill columns that was missing in the DataFrame with NULL values. Default is True.
- **columns_comments** – Glue/Athena catalog: Settings for writing to the Glue table. Currently only the 'columns_comments' attribute is supported for this function. Columns comments can only be added with this function when creating a new table.

Return type
None

Examples

Insert into an existing Iceberg table

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.athena.to_iceberg(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     database='my_database',
...     table='my_table',
...     temp_path='s3://bucket/temp/',
... )
```

Create Iceberg table and insert data (table doesn't exist, requires table_location)

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.athena.to_iceberg(
...     df=pd.DataFrame({'col': [1, 2, 3]}),
...     database='my_database',
...     table='my_table2',
...     table_location='s3://bucket/my_table2/',
...     temp_path='s3://bucket/temp/',
... )
```

awswrangler.athena.delete_from_iceberg_table

awswrangler.athena.**delete_from_iceberg_table**(*df: DataFrame*, *database: str*, *table: str*, *merge_cols: list[str]*, *temp_path: str | None = None*, *keep_files: bool = True*, *data_source: str | None = None*, *s3_output: str | None = None*, *workgroup: str = 'primary'*, *encryption: str | None = None*, *kms_key: str | None = None*, *dtype: dict[str, str] | None = None*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, Any] | None = None*, *catalog_id: str | None = None*) → None

Delete rows from an Iceberg table.

Creates temporary external table, writes staged files and then deletes any rows which match the contents of the temporary table.

Parameters

- **df** (DataFrame) – Pandas DataFrame containing the IDs of rows that are to be deleted from the Iceberg table.
- **database** (str) – Database name.
- **table** (str) – Table name.
- **merge_cols** (list[str]) – List of columns to be used to determine which rows of the Iceberg table should be deleted.

  MERGE INTO
- **temp_path** (str | None) – S3 path to temporarily store the DataFrame.
- **keep_files** (bool) – Whether staging files produced by Athena are retained. True by default.
- **data_source** (str | None) – The AWS KMS key ID or alias used to encrypt the data.
- **s3_output** (str | None) – Amazon S3 path used for query execution.
- **workgroup** (str) – Athena workgroup name.
- **encryption** (str | None) – Valid values: [None, "SSE_S3", "SSE_KMS"]. Notice: "CSE_KMS" is not supported.
- **kms_key** (str | None) – For SSE-KMS, this is the KMS key ARN or ID.
- **dtype** (dict[str, str] | None) – 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'})
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.
- **s3_additional_kwargs** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. `s3_additional_kwargs={"RequestPayer": "requester"}`
- **catalog_id** (str | None) – The ID of the Data Catalog which contains the database and table. If none is provided, the AWS account ID is used by default.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> df = pd.DataFrame({"id": [1, 2, 3], "col": ["foo", "bar", "baz"]})
>>> wr.athena.to_iceberg(
...     df=df,
...     database="my_database",
...     table="my_table",
...     temp_path="s3://bucket/temp/",
... )
>>> df_delete = pd.DataFrame({"id": [1, 3]})
>>> wr.athena.delete_from_iceberg_table(
...     df=df_delete,
...     database="my_database",
...     table="my_table",
...     merge_cols=["id"],
... )
>>> wr.athena.read_sql_table(table="my_table", database="my_database")
   id  col
0   2   bar
```

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 = 'primary'*, *encryption: str | None = None*, *kms_key: str | None = None*, *boto3_session: Session | None = None*, *data_source: str | None = None*, *params: dict[str, Any] | list[str] | None = None*, *paramstyle: Literal['qmark', 'named'] = 'named'*, *athena_query_wait_polling_delay: float = 1.0*) → _QueryMetadata

Write query results from a SELECT statement to the specified data format using UNLOAD.

https://docs.aws.amazon.com/athena/latest/ug/unload.html

Parameters

- **sql** (str) – SQL query.
- **path** (str) – 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** (str | None) – 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 | None) – A single-character field delimiter for files in CSV, TSV, and other text formats.
- **partitioned_by** (list[str] | None) – An array list of columns by which the output is partitioned.

- **workgroup** (str) – Athena workgroup. Primary by default.
- **encryption** (str | None) – Valid values: [None, 'SSE_S3', 'SSE_KMS']. Notice: 'CSE_KMS' is not supported.
- **kms_key** (str | None) – For SSE-KMS, this is the KMS key ARN or ID.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.
- **data_source** (str | None) – Data Source / Catalog name. If None, 'AwsDataCatalog' will be used by default.
- **params** (dict[str, Any] | list[str] | None) –

  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 that will be used for constructing the SQL query. Only named or question mark parameters are supported. The parameter style needs to be specified in the paramstyle parameter.

  For paramstyle="named", this value needs to be a dictionary. The dict needs to contain the information in the form {'name': 'value'} and the SQL query needs to contain :name. The formatter will be applied client-side in this scenario.

  For paramstyle="qmark", this value needs to be a list of strings. The formatter will be applied server-side. The values are applied sequentially to the parameters in the query in the order in which the parameters occur.

paramstyle
Determines the style of params. Possible values are:

- named
- qmark

athena_query_wait_polling_delay
Interval in seconds for how often the function will check if the Athena query has completed.

Return type
_QueryMetadata

Returns
Query metadata including query execution id, dtypes, manifest & output location.

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 = 1.0*) → dict[str, Any]

Wait for the query end.

Parameters

- **query_execution_id** (str) – Athena query execution ID.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** receive None.
- **athena_query_wait_polling_delay** (float) – Interval in seconds for how often the function will check if the Athena query has completed.

Return type
dict[str, Any]

Returns
Dictionary with the get_query_execution response.

Examples

```
>>> import awswrangler as wr
>>> res = wr.athena.wait_query(query_execution_id='query-execution-id')
```

awswrangler.athena.create_prepared_statement

awswrangler.athena.**create_prepared_statement**(*sql: str*, *statement_name: str*, *workgroup: str = 'primary'*, *mode: Literal['update', 'error'] = 'update'*, *boto3_session: Session | None = None*) → None

Create a SQL statement with the name statement_name to be run at a later time. The statement can include parameters represented by question marks.

https://docs.aws.amazon.com/athena/latest/ug/sql-prepare.html

Parameters

- **sql** (str) – The query string for the prepared statement.
- **statement_name** (str) – The name of the prepared statement.
- **workgroup** (str) – The name of the workgroup to which the prepared statement belongs. Primary by default.
- **mode** (Literal['update', 'error']) – Determines the behaviour if the prepared statement already exists:

- `update` - updates statement if already exists
- `error` - throws an error if table exists

- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** receive `None`.

Return type
`None`

Examples

```
>>> import awswrangler as wr
>>> wr.athena.create_prepared_statement(
...     sql="SELECT * FROM my_table WHERE name = ?",
...     statement_name="statement",
... )
```

awswrangler.athena.list_prepared_statements

awswrangler.athena.**list_prepared_statements**(*workgroup: str = 'primary'*, *boto3_session: Session | None = None*) → list[dict[str, Any]]

List the prepared statements in the specified workgroup.

Parameters

- **workgroup** (`str`) – The name of the workgroup to which the prepared statement belongs. Primary by default.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** receive `None`.

Return type
`list[dict[str, Any]]`

Returns
List of prepared statements in the workgroup. Each item is a dictionary with the keys `StatementName` and `LastModifiedTime`.

awswrangler.athena.delete_prepared_statement

awswrangler.athena.**delete_prepared_statement**(*statement_name: str*, *workgroup: str = 'primary'*, *boto3_session: Session | None = None*) → None

Delete the prepared statement with the specified name from the specified workgroup.

https://docs.aws.amazon.com/athena/latest/ug/sql-deallocate-prepare.html

Parameters

- **statement_name** (`str`) – The name of the prepared statement.
- **workgroup** (`str`) – The name of the workgroup to which the prepared statement belongs. Primary by default.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** receive `None`.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> wr.athena.delete_prepared_statement(
...     statement_name="statement",
... )
```

1.6.4 Amazon Redshift

connect([connection, secret_id, catalog_id, ...])	Return a redshift_connector connection from a Glue Catalog or Secret Manager.
connect_temp(cluster_identifier, user[, ...])	Return a redshift_connector temporary connection (No password required).
copy(df, path, con, table, schema[, ...])	Load Pandas DataFrame as a Table on Amazon Redshift using parquet files on S3 as stage.
copy_from_files(path, con, table, schema[, ...])	Load files from S3 to a Table on Amazon Redshift (Through COPY command).
read_sql_query(sql, con[, index_col, ...])	Return a DataFrame corresponding to the result set of the query string.
read_sql_table(table, con[, schema, ...])	Return a DataFrame corresponding the table.
to_sql(df, con, table, schema[, mode, ...])	Write records stored in a DataFrame into Redshift.
unload(sql, path, con[, iam_role, ...])	Load Pandas DataFrame from a Amazon Redshift query result using Parquet files on s3 as stage.
unload_to_files(sql, path, con[, iam_role, ...])	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*) → redshift_connector.core.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 | None) – Glue Catalog Connection name.

- **secret_id** (str | None) – 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 | None) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (str | None) – Optional database name to overwrite the stored one.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is 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 | None) – 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='...'`

Return type
Connection

Returns
redshift_connector connection.

Examples

Fetching Redshift connection from Glue Catalog

```
>>> import awswrangler as wr
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     with con.cursor() as cursor:
...         cursor.execute("SELECT 1")
...         print(cursor.fetchall())
```

Fetching Redshift connection from Secrets Manager

```
>>> import awswrangler as wr
>>> with wr.redshift.connect(secret_id="MY_SECRET") as con:
...     with con.cursor() as cursor:
...         cursor.execute("SELECT 1")
...         print(cursor.fetchall())
```

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*) → redshift_connector.core.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) – The name of a database user.
- **database** (str | None) – Database name. If None, the default Database is used.
- **duration** (int) – 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] | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** is 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 | None) – 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='...'

Return type
Connection

Returns
redshift_connector connection.

Examples

```
>>> import awswrangler as wr
>>> with wr.redshift.connect_temp(cluster_identifier="my-cluster", user="test") as␣
↪con:
...     with con.cursor() as cursor:
...         cursor.execute("SELECT 1")
...         print(cursor.fetchall())
```

awswrangler.redshift.copy

awswrangler.redshift.**copy**(*df: DataFrame*, *path: str*, *con: redshift_connector.core.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: Literal['append', 'overwrite', 'upsert'] = 'append'*, *overwrite_method: Literal['drop', 'cascade', 'truncate', 'delete'] = 'drop'*, *diststyle: Literal['AUTO', 'EVEN', 'ALL', 'KEY'] = 'AUTO'*, *distkey: str | None = None*, *sortstyle: Literal['COMPOUND', 'INTERLEAVED'] = '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*, *add_new_columns: bool = False*, *pyarrow_additional_kwargs: dict[str, str] | None = None*) → 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** (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** (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 | None) – AWS IAM role with the related permissions.
- **aws_access_key_id** (str | None) – The access key for your AWS account.
- **aws_secret_access_key** (str | None) – The secret key for your AWS account.
- **aws_session_token** (str | None) – 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] | None) – 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** (Literal['append', 'overwrite', 'upsert']) – Append, overwrite or upsert.
- **overwrite_method** (Literal['drop', 'cascade', 'truncate', 'delete']) – 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** (Literal['AUTO', 'EVEN', 'ALL', 'KEY']) – Redshift distribution styles. Must be in ["AUTO", "EVEN", "ALL", "KEY"]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html
- **distkey** (str | None) – Specifies a column name or positional number for the distribution key.
- **sortstyle** (Literal['COMPOUND', 'INTERLEAVED']) – Sorting can be "COMPOUND" or "INTERLEAVED". https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html
- **sortkey** (list[str] | None) – List of columns to be sorted.
- **primary_keys** (list[str] | None) – 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] | None) – 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
- **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** (list[str] | None) – Additional copy parameters to pass to the command. For example: ["STATUPDATE ON"]
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **s3_additional_kwargs** (dict[str, str] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}
- **max_rows_by_file** (int | None) – Max number of rows in each file. (e.g. 33554432, 268435456)
- **precombine_key** (str | None) – 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*.
- **add_new_columns** (bool) – If True, it automatically adds the new DataFrame columns into the target table.
- **pyarrow_additional_kwargs** (dict[str, str] | None) – Forwarded to pyarrow. e.g. pyarrow_additional_kwargs={'coerce_timestamps': 'us', 'allow_truncated_timestamps': False}

Return type
None

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     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",
...     )
```

awswrangler.redshift.copy_from_files

awswrangler.redshift.**copy_from_files**(*path: str*, *con: redshift_connector.core.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*, *data_format: Literal['parquet', 'orc', 'csv'] = 'parquet'*, *redshift_column_types: dict[str, str] | None = None*, *parquet_infer_sampling: float = 1.0*, *mode: Literal['append', 'overwrite', 'upsert'] = 'append'*, *overwrite_method: Literal['drop', 'cascade', 'truncate', 'delete'] = 'drop'*, *diststyle: Literal['AUTO', 'EVEN', 'ALL', 'KEY'] = 'AUTO'*, *distkey: str | None = None*, *sortstyle: Literal['COMPOUND', 'INTERLEAVED'] = '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 | list[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*, *add_new_columns: bool = False*) → None

Load 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/ORC/CSV metadata to infer the columns data types. If the data is in the CSV format, the Redshift column types need to be specified manually using `redshift_column_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** (`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 | None`) – AWS IAM role with the related permissions.
- **aws_access_key_id** (`str | None`) – The access key for your AWS account.
- **aws_secret_access_key** (`str | None`) – The secret key for your AWS account.
- **aws_session_token** (`str | None`) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **data_format** (`Literal['parquet', 'orc', 'csv']`) – Data format to be loaded. Supported values are Parquet, ORC, and CSV. Default is Parquet.

- **redshift_column_types** (dict[str, str] | None) – Dictionary with keys as column names and values as Redshift column types. Only used when data_format is CSV.

  e.g. `{'col1': 'BIGINT', 'col2': 'VARCHAR(256)'}`

- **parquet_infer_sampling** (float) – Random sample ratio of files that will have the metadata inspected. Must be *0.0 < sampling <= 1.0*. The higher, the more accurate. The lower, the faster.

- **mode** (Literal['append', 'overwrite', 'upsert']) – Append, overwrite or upsert.

- **overwrite_method** (Literal['drop', 'cascade', 'truncate', 'delete']) – 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** (Literal['AUTO', 'EVEN', 'ALL', 'KEY']) – Redshift distribution styles. Must be in [“AUTO”, “EVEN”, “ALL”, “KEY”]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html

- **distkey** (str | None) – Specifies a column name or positional number for the distribution key.

- **sortstyle** (Literal['COMPOUND', 'INTERLEAVED']) – Sorting can be “COMPOUND” or “INTERLEAVED”. https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html

- **sortkey** (list[str] | None) – List of columns to be sorted.

- **primary_keys** (list[str] | None) – 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] | None) – 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** (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** (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 | None) – If set to true path argument accepts a S3 uri to a manifest file.

- **sql_copy_extra_params** (list[str] | None) – Additional copy parameters to pass to the command. For example: ["STATUPDATE ON"]
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **s3_additional_kwargs** (dict[str, str] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}
- **precombine_key** (str | None) – 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] | None) – List of column names to map source data fields to the target columns.
- **add_new_columns** (bool) – If True, it automatically adds the new DataFrame columns into the target table.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     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"
...     )
```

awswrangler.redshift.read_sql_query

awswrangler.redshift.**read_sql_query**(*sql: str*, *con: redshift_connector.core.Connection*, *index_col: str | list[str] | None = None*, *params: list[Any] | tuple[Any, ...] | dict[Any, Any] | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *chunksize: int | None = None*, *dtype: dict[str, pyarrow.lib.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** (Connection) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.

- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **chunksize** (int | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     df = wr.redshift.read_sql_query(
...         sql="SELECT * FROM public.my_table",
...         con=con
...     )
```

awswrangler.redshift.read_sql_table

awswrangler.redshift.**read_sql_table**(*table: str*, *con: redshift_connector.core.Connection*, *schema: str | None = None*, *index_col: str | list[str] | None = None*, *params: list[Any] | tuple[Any, ...] | dict[Any, Any] | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *chunksize: int | None = None*, *dtype: dict[str, pyarrow.lib.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** (Connection) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **schema** (str | None) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **chunksize** (int | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     df = wr.redshift.read_sql_table(
...         table="my_table",
...         schema="public",
...         con=con
...     )
```

awswrangler.redshift.to_sql

awswrangler.redshift.**to_sql**(*df: DataFrame*, *con: redshift_connector.core.Connection*, *table: str*, *schema: str*, *mode: Literal['append', 'overwrite', 'upsert'] = 'append'*, *overwrite_method: Literal['drop', 'cascade', 'truncate', 'delete'] = 'drop'*, *index: bool = False*, *dtype: dict[str, str] | None = None*, *diststyle: Literal['AUTO', 'EVEN', 'ALL', 'KEY'] = 'AUTO'*, *distkey: str | None = None*, *sortstyle: Literal['COMPOUND', 'INTERLEAVED'] = '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*, *add_new_columns: bool = False*) → None

Write records stored in a DataFrame into Redshift.

Note: For large DataFrames (1K+ rows) consider the function **wr.redshift.copy()**.

Parameters

- **df** (DataFrame) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **con** (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** (Literal['append', 'overwrite', 'upsert']) – Append, overwrite or upsert.
- **overwrite_method** (Literal['drop', 'cascade', 'truncate', 'delete']) – 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] | None) – 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** (Literal['AUTO', 'EVEN', 'ALL', 'KEY']) – Redshift distribution styles. Must be in ["AUTO", "EVEN", "ALL", "KEY"]. https://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html
- **distkey** (str | None) – Specifies a column name or positional number for the distribution key.

- **sortstyle** (Literal['COMPOUND', 'INTERLEAVED']) – Sorting can be "COMPOUND" or "INTERLEAVED". https://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html
- **sortkey** (list[str] | None) – List of columns to be sorted.
- **primary_keys** (list[str] | None) – 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] | None) – 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 | None) – 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.
- **add_new_columns** (bool) – If True, it automatically adds the new DataFrame columns into the target table.

Return type
None

Examples

Writing to Redshift using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     wr.redshift.to_sql(
...         df=df,
...         table="my_table",
...         schema="public",
...         con=con,
...     )
```

awswrangler.redshift.unload

awswrangler.redshift.**unload**(*sql: str*, *path: str*, *con: redshift_connector.core.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*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *chunked: bool | int = False*, *keep_files: bool = False*, *parallel: bool = True*, *cleanpath: bool = False*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, str] | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | 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**, depending on the size of the data, one or more data frames are returned per file. Unlike **chunked=INTEGER**, rows from different files are not be mixed in the resulting data frames.
- If **chunked=INTEGER**, awswrangler iterates 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** (`str`) – S3 path to write stage files (e.g. s3://bucket_name/any_name/)
- **con** (`Connection`) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **iam_role** (`str | None`) – AWS IAM role with the related permissions.
- **aws_access_key_id** (`str | None`) – The access key for your AWS account.
- **aws_secret_access_key** (`str | None`) – The secret key for your AWS account.
- **aws_session_token** (`str | None`) – The session key for your AWS account. This is only needed when you are using temporary credentials.

- **region** (str | None) – 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 | None) – 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 | None) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3.
- **keep_files** (bool) – Should keep stage files?
- **parallel** (bool) – Whether to unload to multiple files in parallel. Defaults to True. By default, UNLOAD writes data in parallel to multiple files, according to the number of slices in the cluster. If parallel is False, UNLOAD writes to one or more data files serially, sorted absolutely according to the ORDER BY clause, if one is used.
- **cleanpath** (bool) – Use CLEANPATH instead of ALLOWOVERWRITE. When True, uses CLEANPATH to remove existing files located in the Amazon S3 path before unloading files. When False (default), uses ALLOWOVERWRITE to overwrite existing files, including the manifest file. These options are mutually exclusive.

  ALLOWOVERWRITE: By default, UNLOAD fails if it finds files that it would possibly overwrite. If ALLOWOVERWRITE is specified, UNLOAD overwrites existing files, including the manifest file.

  CLEANPATH: Removes existing files located in the Amazon S3 path specified in the TO clause before unloading files to the specified location. If you include the PARTITION BY clause, existing files are removed only from the partition folders to receive new files generated by the UNLOAD operation. You must have the s3:DeleteObject permission on the Amazon S3 bucket. Files removed using CLEANPATH are permanently deleted and can't be recovered.

  For more information, see: https://docs.aws.amazon.com/redshift/latest/dg/r_UNLOAD.html
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimential. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **chunked** (bool | int) – 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.
- **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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **s3_additional_kwargs** (dict[str, str] | None) – Forward to botocore requests.

- **pyarrow_additional_kwargs** (dict[str, Any] | None) – Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

```
>>> import awswrangler as wr
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     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"
...     )
>>> # Using CLEANPATH instead of ALLOWOVERWRITE
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     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",
...         cleanpath=True
...     )
```

awswrangler.redshift.unload_to_files

awswrangler.redshift.**unload_to_files**(*sql: str*, *path: str*, *con: redshift_connector.core.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: Literal['CSV', 'PARQUET'] | None = None*, *parallel: bool = True*, *max_file_size: float | None = None*, *kms_key_id: str | None = None*, *manifest: bool = False*, *partition_cols: list[str] | None = None*, *cleanpath: bool = False*, *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** (str) – S3 path to write stage files (e.g. s3://bucket_name/any_name/)
- **con** (Connection) – Use redshift_connector.connect() to use " "credentials directly or wr.redshift.connect() to fetch it from the Glue Catalog.
- **iam_role** (str | None) – AWS IAM role with the related permissions.
- **aws_access_key_id** (str | None) – The access key for your AWS account.
- **aws_secret_access_key** (str | None) – The secret key for your AWS account.
- **aws_session_token** (str | None) – The session key for your AWS account. This is only needed when you are using temporary credentials.
- **region** (str | None) – 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** (Literal['CSV', 'PARQUET'] | None) – Format of the unloaded S3 objects from the query. Valid values: "CSV", "PARQUET". Case sensitive. Defaults to PARQUET.
- **parallel** (bool) – Whether to unload to multiple files in parallel. Defaults to True. By default, UNLOAD writes data in parallel to multiple files, according to the number of slices in the cluster. If parallel is False, UNLOAD writes to one or more data files serially, sorted absolutely according to the ORDER BY clause, if one is used.
- **max_file_size** (float | None) – 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 | None) – 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] | None) – Specifies the partition keys for the unload operation.
- **cleanpath** (bool) – Use CLEANPATH instead of ALLOWOVERWRITE. When True, uses CLEANPATH to remove existing files located in the Amazon S3 path before unloading files. When False (default), uses ALLOWOVERWRITE to overwrite existing files, including the manifest file. These options are mutually exclusive.

  ALLOWOVERWRITE: By default, UNLOAD fails if it finds files that it would possibly overwrite. If ALLOWOVERWRITE is specified, UNLOAD overwrites existing files, including the manifest file.

  CLEANPATH: Removes existing files located in the Amazon S3 path specified in the TO clause before unloading files to the specified location. If you include the PARTITION BY clause, existing files are removed only from the partition folders to receive new files generated by the UNLOAD operation. You must have the s3:DeleteObject permission on the Amazon S3 bucket. Files removed using CLEANPATH are permanently deleted and can't be recovered.

  For more information, see: https://docs.aws.amazon.com/redshift/latest/dg/r_UNLOAD.html
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     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"
...     )
>>> # Using CLEANPATH instead of ALLOWOVERWRITE
>>> with wr.redshift.connect("MY_GLUE_CONNECTION") as con:
...     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",
...         cleanpath=True
...     )
```

1.6.5 PostgreSQL

connect([connection, secret_id, catalog_id, ...])	Return a pg8000 connection from a Glue Catalog Connection.
read_sql_query(sql, con[, index_col, ...])	Return a DataFrame corresponding to the result set of the query string.
read_sql_table(table, con[, schema, ...])	Return a DataFrame corresponding the table.
to_sql(df, con, table, schema[, mode, ...])	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*) → pg8000.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.dr8vkeyrb9m1.us-east-1.rds.amazonaws.com", "username":"test", "password":"test", "engine":"postgresql", "port":"3306", "dbname": "mydb" # Optional }

Parameters

- **connection** (str | None) – Glue Catalog Connection name.
- **secret_id** (str | None) – 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 | None) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (str | None) – Optional database name to overwrite the stored one.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **ssl_context** (bool | SSLContext | None) – This governs SSL encryption for TCP/IP sockets. This parameter is forward to pg8000. https://github.com/tlocke/pg8000#functions
- **timeout** (int | None) – 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 forwarded to pg8000. https://github.com/tlocke/pg8000#functions

Return type
Connection

Returns
pg8000 connection.

Examples

```
>>> import awswrangler as wr
>>> with wr.postgresql.connect("MY_GLUE_CONNECTION") as con:
...     with con.cursor() as cursor:
...         cursor.execute("SELECT 1")
...         print(cursor.fetchall())
```

awswrangler.postgresql.read_sql_query

awswrangler.postgresql.**read_sql_query**(*sql: str*, *con: pg8000.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, pyarrow.lib.DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (str) – SQL query.
- **con** (Connection) – Use pg8000.connect() to use credentials directly or wr.postgresql.connect() to fetch it from the Glue Catalog.
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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 | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.postgresql.connect("MY_GLUE_CONNECTION") as con:
...     df = wr.postgresql.read_sql_query(
...         sql="SELECT * FROM public.my_table",
...         con=con,
...     )
```

awswrangler.postgresql.read_sql_table

awswrangler.postgresql.**read_sql_table**(*table: str*, *con: pg8000.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, pyarrow.lib.DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding the table.

Parameters

- **table** (str) – Table name.
- **con** (Connection) – Use pg8000.connect() to use credentials directly or wr.postgresql.connect() to fetch it from the Glue Catalog.
- **schema** (str | None) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).

- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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 | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.postgresql.connect("MY_GLUE_CONNECTION") as con:
>>>     df = wr.postgresql.read_sql_table(
...         table="my_table",
...         schema="public",
...         con=con,
...     )
```

awswrangler.postgresql.to_sql

awswrangler.postgresql.**to_sql**(*df: DataFrame*, *con: pg8000.Connection*, *table: str*, *schema: str*, *mode: Literal['append', 'overwrite', 'upsert'] = 'append'*, *overwrite_method: Literal['drop', 'cascade', 'truncate', 'truncate cascade'] = 'drop'*, *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*, *commit_transaction: bool = True*) → None

Write records stored in a DataFrame into PostgreSQL.

Parameters

- **df** (DataFrame) – Pandas DataFrame

- **con** (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** (Literal['append', 'overwrite', 'upsert']) – 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.
- **overwrite_method** (Literal['drop', 'cascade', 'truncate', 'truncate cascade']) – Drop, cascade, truncate, or truncate cascade. 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 ... RESTRICT - truncates the table. Fails if any of the tables have foreign-key references from tables that are not listed in the command.
  - "truncate cascade" - TRUNCATE ... CASCADE - truncates the table, and all tables that have foreign-key references to any of the named tables.
- **index** (bool) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (dict[str, str] | None) – 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] | None) – 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] | None) – 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] | None) – 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'.
- **commit_transaction** (bool) – Whether to commit the transaction. True by default.

Return type
None

Examples

Writing to PostgreSQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.postgresql.connect("MY_GLUE_CONNECTION") as con:
...     wr.postgresql.to_sql(
...         df=df,
...         table="my_table",
...         schema="public",
...         con=con
...     )
```

1.6.6 MySQL

connect([connection, secret_id, catalog_id, ...])	Return a pymysql connection from a Glue Catalog Connection or Secrets Manager.
read_sql_query(sql, con[, index_col, ...])	Return a DataFrame corresponding to the result set of the query string.
read_sql_table(table, con[, schema, ...])	Return a DataFrame corresponding the table.
to_sql(df, con, table, schema[, mode, ...])	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: Session | None = None*, *read_timeout: int | None = None*, *write_timeout: int | None = None*, *connect_timeout: int = 10*, *cursorclass: type[pymysql.cursors.Cursor] | None = None*) → pymysql.connections.Connection

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 | None) – Glue Catalog Connection name.

- **secret_id** (str | None) – 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 | None) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (str | None) – Optional database name to overwrite the stored one.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **read_timeout** (int | None) – 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** (int | None) – 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** (type[Cursor] | None) – Cursor class to use, e.g. SSCursor; defaults to pymysql.cursors.Cursor https://pymysql.readthedocs.io/en/latest/modules/cursors.html

Return type
Connection

Returns
pymysql connection.

Examples

```
>>> import awswrangler as wr
>>> with wr.mysql.connect("MY_GLUE_CONNECTION") as con:
...     with con.cursor() as cursor:
...         cursor.execute("SELECT 1")
...         print(cursor.fetchall())
```

awswrangler.mysql.read_sql_query

awswrangler.mysql.**read_sql_query**(*sql: str*, *con: pymysql.connections.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, pyarrow.lib.DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (str) – SQL query.

- **con** (Connection) – Use pymysql.connect() to use credentials directly or wr.mysql.connect() to fetch it from the Glue Catalog.
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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 | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.mysql.connect("MY_GLUE_CONNECTION") as con:
...     df = wr.mysql.read_sql_query(
...         sql="SELECT * FROM test.my_table",
...         con=con,
...     )
```

awswrangler.mysql.read_sql_table

awswrangler.mysql.**read_sql_table**(*table: str*, *con: pymysql.connections.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, pyarrow.lib.DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding the table.

Parameters

- **table** (str) – Table name.
- **con** (Connection) – Use pymysql.connect() to use credentials directly or wr.mysql.connect() to fetch it from the Glue Catalog.
- **schema** (str | None) – Name of SQL schema in database to query. Uses default schema if None.
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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 | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.mysql.connect("MY_GLUE_CONNECTION") as con:
...     df = wr.mysql.read_sql_table(
...         table="my_table",
...         schema="test",
...         con=con
...     )
```

awswrangler.mysql.to_sql

awswrangler.mysql.**to_sql**(*df: DataFrame*, *con: pymysql.connections.Connection*, *table: str*, *schema: str*, *mode: Literal['append', 'overwrite', 'upsert_replace_into', 'upsert_duplicate_key', 'upsert_distinct', 'ignore'] = '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*, *cursorclass: type[pymysql.cursors.Cursor] | None = None*) → None

Write records stored in a DataFrame into MySQL.

Parameters

- **df** (DataFrame) – Pandas DataFrame
- **con** (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** (Literal['append', 'overwrite', 'upsert_replace_into', 'upsert_duplicate_key', 'upsert_distinct', 'ignore']) – Supports the following modes:
  - 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] | None) – 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] | None) – 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** (type[Cursor] | None) – Cursor class to use, e.g. SSCrusor; defaults to pymysql.cursors.Cursor https://pymysql.readthedocs.io/en/latest/modules/cursors.html

Return type
None

Examples

Writing to MySQL using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.mysql.connect("MY_GLUE_CONNECTION") as con:
...     wr.mysql.to_sql(
...         df=df,
...         table="my_table",
...         schema="test",
...         con=con
...     )
```

Microsoft SQL Server

connect([connection, secret_id, catalog_id, ...])	Return a pyodbc connection from a Glue Catalog Connection.
read_sql_query(sql, con[, index_col, ...])	Return a DataFrame corresponding to the result set of the query string.
read_sql_table(table, con[, schema, ...])	Return a DataFrame corresponding the table.
to_sql(df, con, table, schema[, mode, ...])	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*) → pyodbc.Connection

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** (str | None) – Glue Catalog Connection name.
- **secret_id** (str | None) – 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 | None) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.

- **dbname** (str | None) – Optional database name to overwrite the stored one.
- **odbc_driver_version** (int) – Major version of the OBDC Driver version that is installed and should be used.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **timeout** (int | None) – 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

Return type
Connection

Returns
pyodbc connection.

Examples

```
>>> import awswrangler as wr
>>> with wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↪version=17) as con:
...     with con.cursor() as cursor:
...         cursor.execute("SELECT 1")
...         print(cursor.fetchall())
```

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, pyarrow.lib.DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (str) – SQL query.
- **con** (Connection) – Use pyodbc.connect() to use credentials directly or wr.sqlserver.connect() to fetch it from the Glue Catalog.
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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 | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from Microsoft SQL Server using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↪version=17) as con:
...     df = wr.sqlserver.read_sql_query(
...         sql="SELECT * FROM dbo.my_table",
...         con=con,
...     )
```

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, pyarrow.lib.DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding the table.

Parameters

- **table** (str) – Table name.
- **con** (Connection) – Use pyodbc.connect() to use credentials directly or wr.sqlserver.connect() to fetch it from the Glue Catalog.
- **schema** (str | None) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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 | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimential. The "pyarrow" backend is only supported with Pandas 2.0 or above.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from Microsoft SQL Server using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↪version=17) as con:
...     df = wr.sqlserver.read_sql_table(
...         table="my_table",
...         schema="dbo",
...         con=con,
...     )
```

awswrangler.sqlserver.to_sql

awswrangler.sqlserver.**to_sql**(*df: DataFrame*, *con: pyodbc.Connection*, *table: str*, *schema: str*, *mode: Literal['append', 'overwrite', 'upsert'] = 'append'*, *index: bool = False*, *dtype: dict[str, str] | None = None*, *varchar_lengths: dict[str, int] | None = None*, *use_column_names: bool = False*, *upsert_conflict_columns: list[str] | None = None*, *chunksize: int = 200*, *fast_executemany: bool = False*) → None

Write records stored in a DataFrame into Microsoft SQL Server.

Parameters

- **df** (DataFrame) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **con** (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** (Literal['append', 'overwrite', 'upsert']) – 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 column names of the Dataframe will be used for this operation, as if use_column_names was set to True.
- **index** (bool) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (dict[str, str] | None) – 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] | None) – 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*.
- **uspert_conflict_columns** – List of columns to be used as conflict columns in the upsert operation.
- **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#executemanysql-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.

Return type
None

Examples

Writing to Microsoft SQL Server using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.sqlserver.connect(connection="MY_GLUE_CONNECTION", odbc_driver_
↪version=17) as con:
...     wr.sqlserver.to_sql(
...         df=df,
...         table="table",
```

(continues on next page)

(continued from previous page)

```
...         schema="dbo",
...         con=con
...     )
```

Oracle

connect([connection, secret_id, catalog_id, ...])	Return a oracledb connection from a Glue Catalog Connection.
read_sql_query(sql, con[, index_col, ...])	Return a DataFrame corresponding to the result set of the query string.
read_sql_table(table, con[, schema, ...])	Return a DataFrame corresponding the table.
to_sql(df, con, table, schema[, mode, ...])	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*) → oracledb.Connection

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** (str | None) – Glue Catalog Connection name.
- **secret_id** (str | None) – 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 | None) – The ID of the Data Catalog. If none is provided, the AWS account ID is used by default.
- **dbname** (str | None) – Optional database name to overwrite the stored one.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **call_timeout** (int | None) – 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

Return type

Connection

Returns
oracledb connection.

Examples

```
>>> import awswrangler as wr
>>> with wr.oracle.connect(connection="MY_GLUE_CONNECTION") as con:
...     with con.cursor() as cursor:
...         cursor.execute("SELECT 1 FROM DUAL")
...         print(cursor.fetchall())
```

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, pyarrow.lib.DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding to the result set of the query string.

Parameters

- **sql** (str) – SQL query.
- **con** (Connection) – Use oracledb.connect() to use credentials directly or wr.oracle.connect() to fetch it from the Glue Catalog.
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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 | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Reading from Oracle Database using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.oracle.connect(connection="MY_GLUE_CONNECTION") as con:
...     df = wr.oracle.read_sql_query(
...         sql="SELECT * FROM test.my_table",
...         con=con,
...     )
```

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, pyarrow.lib.DataType] | None = None*, *safe: bool = True*, *timestamp_as_object: bool = False*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*) → DataFrame | Iterator[DataFrame]

Return a DataFrame corresponding the table.

Parameters

- **table** (str) – Table name.
- **con** (Connection) – Use oracledb.connect() to use credentials directly or wr.oracle.connect() to fetch it from the Glue Catalog.
- **schema** (str | None) – Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).
- **index_col** (str | list[str] | None) – Column(s) to set as index(MultiIndex).
- **params** (list[Any] | tuple[Any, ...] | dict[Any, Any] | None) – 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 | None) – If specified, return an iterator where chunksize is the number of rows to include in each chunk.
- **dtype** (dict[str, DataType] | None) – 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.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

The dtype_backends are still experimental. The "pyarrow" backend is only supported with Pandas 2.0 or above.

Return type
`DataFrame | Iterator[DataFrame]`

Returns
Result as Pandas DataFrame(s).

Examples

Reading from Oracle Database using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.oracle.connect(connection="MY_GLUE_CONNECTION") as con:
...     df = wr.oracle.read_sql_table(
...         table="my_table",
...         schema="test",
...         con=con,
...     )
```

awswrangler.oracle.to_sql

awswrangler.oracle.**to_sql**(*df: DataFrame*, *con: oracledb.Connection*, *table: str*, *schema: str*, *mode: Literal['append', 'overwrite', 'upsert'] = 'append'*, *index: bool = False*, *dtype: dict[str, str] | None = None*, *varchar_lengths: dict[str, int] | None = None*, *use_column_names: bool = False*, *primary_keys: list[str] | None = None*, *chunksize: int = 200*) → None

Write records stored in a DataFrame into Oracle Database.

Parameters

- **df** (`DataFrame`) – Pandas DataFrame https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
- **con** (`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** (`Literal['append', 'overwrite', 'upsert']`) – Append, overwrite or upsert.
- **index** (`bool`) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (`dict[str, str] | None`) – 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] | None`) – 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*.

- **primary_keys** (list[str] | None) – Primary keys.
- **chunksize** (int) – Number of rows which are inserted with each SQL query. Defaults to inserting 200 rows per query.

Return type
None

Examples

Writing to Oracle Database using a Glue Catalog Connections

```
>>> import awswrangler as wr
>>> with wr.oracle.connect(connection="MY_GLUE_CONNECTION") as con:
...     wr.oracle.to_sql(
...         df=df,
...         table="table",
...         schema="ORCL",
...         con=con,
...     )
```

1.6.7 Data API Redshift

RedshiftDataApi([cluster_id, database, ...])	Provides access to a Redshift cluster via the Data API.
connect([cluster_id, database, ...])	Create a Redshift Data API connection.
read_sql_query(sql, con[, database, parameters])	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*) → 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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

__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, ...])	
batch_execute(sql[, database, ...])	Batch execute SQL statements against a Data API Service.
begin_transaction([database, schema])	Start an SQL transaction.
close()	Close underlying endpoint connections.
commit_transaction(transaction_id)	Commit an SQL transaction.
execute(sql[, database, transaction_id, ...])	Execute SQL statement against a Data API Service.
rollback_transaction(transaction_id)	Roll back an SQL transaction.

awswrangler.data_api.redshift.connect

awswrangler.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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- ****kwargs** (Any) – Any additional kwargs are passed to the underlying RedshiftDataApi class.

Return type
RedshiftDataApi

Returns
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*, *parameters: list[dict[str, Any]] | 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 | None) – Database to run query on - defaults to the database specified by *con*.
- **parameters** (list[dict[str, Any]] | None) – A list of named parameters e.g. [{"name": "id", "value": "42"}].

Return type
DataFrame

Returns
A Pandas DataFrame containing the query results.

Examples

```
>>> import awswrangler as wr
>>> df = wr.data_api.redshift.read_sql_query(
>>>     sql="SELECT * FROM public.my_table",
>>>     con=con,
>>> )
```

```
>>> import awswrangler as wr
>>> df = wr.data_api.redshift.read_sql_query(
>>>     sql="SELECT * FROM public.my_table WHERE id >= :id",
>>>     con=con,
>>>     parameters=[
```

(continues on next page)

(continued from previous page)

```
>>>         {"name": "id", "value": "42"},
>>>     ],
>>> )
```

1.6.8 Data API RDS

RdsDataApi(resource_arn, database[, ...])	Provides access to the RDS Data API.
connect(resource_arn, database[, ...])	Create a RDS Data API connection.
read_sql_query(sql, con[, database, parameters])	Run an SQL query on an RdsDataApi connection and return the result as a DataFrame.
to_sql(df, con, table, database[, mode, ...])	Insert data using an SQL query on a Data API connection.

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*) → 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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

__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[, ...])	
`batch_execute`(sql[, database, ...])	Batch execute SQL statements against a Data API Service.
`begin_transaction`([database, schema])	Start an SQL transaction.
`close`()	Close underlying endpoint connections.
`commit_transaction`(transaction_id)	Commit an SQL transaction.
`execute`(sql[, database, transaction_id, ...])	Execute SQL statement against a Data API Service.
`rollback_transaction`(transaction_id)	Roll back an SQL transaction.

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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- ****kwargs** (Any) – Any additional kwargs are passed to the underlying RdsDataApi class.

Return type
RdsDataApi

Returns
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*, *parameters: list[dict[str, Any]] | 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 | None) – Database to run query on - defaults to the database specified by *con*.
- **parameters** (list[dict[str, Any]] | None) – A list of named parameters e.g. [{"name": "col", "value": {"stringValue": "val1"}}].

Return type
DataFrame

Returns
A Pandas DataFrame containing the query results.

Examples

```
>>> import awswrangler as wr
>>> df = wr.data_api.rds.read_sql_query(
>>>     sql="SELECT * FROM public.my_table",
>>>     con=con,
>>> )
```

```
>>> import awswrangler as wr
>>> df = wr.data_api.rds.read_sql_query(
>>>     sql="SELECT * FROM public.my_table WHERE col = :name",
>>>     con=con,
>>>     parameters=[
>>>        {"name": "col1", "value": {"stringValue": "val1"}}
>>>     ],
>>> )
```

awswrangler.data_api.rds.to_sql

awswrangler.data_api.rds.**to_sql**(*df: DataFrame*, *con:* RdsDataApi, *table: str*, *database: str*, *mode: Literal['append', 'overwrite'] = '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*, *sql_mode: str = 'mysql'*) → None

Insert data using an SQL query on a Data API connection.

Parameters

- **df** (DataFrame) – Pandas DataFrame
- **con** (*RdsDataApi*) – A RdsDataApi connection instance
- **database** (str) – Database to run query on - defaults to the database specified by *con*.
- **table** (str) – Table name
- **mode** (Literal['append', 'overwrite']) – *append* (inserts new records into table), *overwrite* (drops table and recreates)
- **index** (bool) – True to store the DataFrame index as a column in the table, otherwise False to ignore it.
- **dtype** (dict[str, str] | None) – 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] | None) – 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.
- **sql_mode** (str) – "mysql" for default MySQL identifiers (backticks) or "ansi" for ANSI-compatible identifiers (double quotes).

Return type
None

1.6.9 AWS Glue Data Quality

create_recommendation_ruleset(database, ...)	Create recommendation Data Quality ruleset.
create_ruleset(name, database, table[, ...])	Create Data Quality ruleset.
evaluate_ruleset(name, iam_role_arn[, ...])	Evaluate Data Quality ruleset.
get_ruleset(name[, boto3_session])	Get a Data Quality ruleset.
update_ruleset(name[, mode, df_rules, ...])	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*) → DataFrame

Create recommendation Data Quality ruleset.

Parameters

- **database** (str) – Glue database name.
- **table** (str) – Glue table name.
- **iam_role_arn** (str) – IAM Role ARN.
- **name** (str | None) – Ruleset name.
- **catalog_id** (str | None) – Glue Catalog id.
- **connection_name** (str | None) – Glue connection name.
- **additional_options** (dict[str, Any] | None) – 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) – The number of G.1X workers to be used in the run. The default is 5.
- **timeout** (int) – The timeout for a run in minutes. The default is 2880 (48 hours).

- **client_token** (str | None) – Random id used for idempotency. Is automatically generated if not provided.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
DataFrame

Returns
Data frame with recommended ruleset details.

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*) → None

Create Data Quality ruleset.

Parameters

- **name** (str) – Ruleset name.
- **database** (str) – Glue database name.
- **table** (str) – Glue table name.
- **df_rules** (DataFrame | None) – Data frame with *rule_type*, *parameter*, and *expression* columns.
- **dqdl_rules** (str | None) – Data Quality Definition Language definition.
- **description** (str) – Ruleset description.
- **client_token** (str | None) – Random id used for idempotency. Is automatically generated if not provided.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
None

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, str | bool] | None = None*, *client_token: str | None = None*, *boto3_session: Session | None = None*) → DataFrame

Evaluate Data Quality ruleset.

Parameters

- **name** (str | list[str]) – Ruleset name or list of names.
- **iam_role_arn** (str) – IAM Role ARN.
- **number_of_workers** (int) – The number of G.1X workers to be used in the run. The default is 5.
- **timeout** (int) – The timeout for a run in minutes. The default is 2880 (48 hours).
- **database** (str | None) – Glue database name. Database associated with the ruleset will be used if not provided.

- **table** (str | None) – Glue table name. Table associated with the ruleset will be used if not provided.
- **catalog_id** (str | None) – Glue Catalog id.
- **connection_name** (str | None) – Glue connection name.
- **additional_options** (dict[str, str] | None) – 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, str | bool] | None) – Additional run options. Supported keys:
  - *CloudWatchMetricsEnabled*: whether to enable CloudWatch metrics.
  - *ResultsS3Prefix*: prefix for Amazon S3 to store results.
- **client_token** (str | None) – Random id used for idempotency. Will be automatically generated if not provided.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
DataFrame

Returns
Data frame with ruleset evaluation results.

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,
... )
```

awswrangler.data_quality.get_ruleset

awswrangler.data_quality.**get_ruleset**(*name: str | list[str]*, *boto3_session: Session | None = None*) → DataFrame

Get a Data Quality ruleset.

Parameters

- **name** (str | list[str]) – Ruleset name or list of names.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
DataFrame

Returns
Data frame with ruleset(s) details.

Examples

Get single ruleset >>> import awswrangler 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 >>> import awswrangler as wr >>> df_rulesets = wr.data_quality.get_ruleset(name=["ruleset_1", "ruleset_2"])

awswrangler.data_quality.update_ruleset

awswrangler.data_quality.**update_ruleset**(*name: str*, *mode: Literal['overwrite', 'upsert'] = 'overwrite'*, *df_rules: DataFrame | None = None*, *dqdl_rules: str | None = None*, *description: str = ''*, *boto3_session: Session | None = None*) → None

Update Data Quality ruleset.

Parameters

- **name** (str) – Ruleset name.
- **mode** (Literal['overwrite', 'upsert']) – overwrite (default) or upsert.
- **df_rules** (DataFrame | None) – Data frame with *rule_type*, *parameter*, and *expression* columns.
- **dqdl_rules** (str | None) – Data Quality Definition Language definition.
- **description** (str) – Ruleset description.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
None

Examples

Overwrite rules in the existing ruleset. >>> wr.data_quality.update_ruleset(… name="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]", …)

1.6.10 OpenSearch

connect(host[, port, boto3_session, region, ...])	Create a secure connection to the specified Amazon OpenSearch domain.
create_collection(name[, collection_type, ...])	Create Amazon OpenSearch Serverless collection.
create_index(client, index[, doc_type, ...])	Create an index.
delete_index(client, index)	Delete an index.
index_csv(client, path, index[, doc_type, ...])	Index all documents from a CSV file to OpenSearch index.
index_documents(client, documents, index[, ...])	Index all documents to OpenSearch index.
index_df(client, df, index[, doc_type, ...])	Index all documents from a DataFrame to OpenSearch index.
index_json(client, path, index[, doc_type, ...])	Index all documents from JSON file to OpenSearch index.
search(client[, index, search_body, ...])	Return results matching query DSL as pandas DataFrame.
search_by_sql(client, sql_query, **kwargs)	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*) → opensearchpy.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 | None) – OpenSearch Service only accepts connections over port 80 (HTTP) or 443 (HTTPS)
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

- **region** (str | None) – AWS region of the Amazon OS domain. If not provided will be extracted from boto3_session.
- **username** (str | None) – Fine-grained access control username. Mandatory if OS Cluster uses Fine Grained Access Control.
- **password** (str | None) – Fine-grained access control password. Mandatory if OS Cluster uses Fine Grained Access Control.
- **service** (str | None) – 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** (Sequence[int] | None) – Set of HTTP status codes on which we should retry on a different node. Defaults to [500, 502, 503, 504].

Return type
OpenSearch

Returns
OpenSearch low-level client.

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)

Parameters

- **name** (str) – Collection name.
- **collection_type** (str) – Collection type. Allowed values are *SEARCH*, and *TIMESERIES*.
- **description** (str) – Collection description.
- **encryption_policy** (dict[str, Any] | list[dict[str, Any]] | None) – 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 | None) – Encryption key.

- **network_policy** (dict[str, Any] | list[dict[str, Any]] | None) – 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] | None) – List of VPC endpoints for access to non-public collection.
- **data_policy** (list[dict[str, Any]] | None) – Data policy of a form: [{ "Rules": [...] }]
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

Returns
Collection details

awswrangler.opensearch.create_index

awswrangler.opensearch.**create_index**(*client: opensearchpy.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 | None) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **settings** (dict[str, Any] | None) – Index settings https://opensearch.org/docs/opensearch/rest-api/create-index/#index-settings
- **mappings** (dict[str, Any] | None) – Index mappings https://opensearch.org/docs/opensearch/rest-api/create-index/#mappings

Return type
dict[str, Any]

Returns
OpenSearch rest api response https://opensearch.org/docs/opensearch/rest-api/create-index/#response.

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={
```

(continues on next page)

(continued from previous page)

```
...         "properties": {
...           "age":  { "type" : "integer" }
...         }
...     },
...     settings={
...         "index": {
...             "number_of_shards": 2,
...             "number_of_replicas": 1
...         }
...     }
... )
```

awswrangler.opensearch.delete_index

awswrangler.opensearch.**delete_index**(*client: opensearchpy.OpenSearch*, *index: str*) → dict[str, Any]

Delete an index.

Parameters

- **client** (OpenSearch) – instance of opensearchpy.OpenSearch to use.
- **index** (str) – Name of the index.

Return type
dict[str, Any]

Returns
OpenSearch rest api response

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: opensearchpy.OpenSearch*, *path: str*, *index: str*, *doc_type: str | None = None*, *pandas_kwargs: dict[str, Any] | None = None*, *use_threads: bool | int = False*, ***kwargs: Any*) → 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 | None) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **pandas_kwargs** (dict[str, Any] | None) – 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*
- **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.
- ****kwargs** (Any) – KEYWORD arguments forwarded to *index_documents()* which is used to execute the operation

Return type
Any

Returns
Response payload https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#response.

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']}
... )
```

awswrangler.opensearch.index_documents

awswrangler.opensearch.**index_documents**(*client: opensearchpy.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 = None*, *initial_backoff: int | None = None*, *max_backoff: int | None = None*, *use_threads: bool | int = False*, *enable_refresh_interval: bool = True*, ***kwargs: Any*) → dict[str, Any]

Index all documents to OpenSearch index.

Note: *max_retries*, *initial_backoff*, and *max_backoff* are not supported with parallel bulk (when *use_threads* is set to True).

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 | None`) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **keys_to_write** (`list[str] | None`) – list of keys to index. If not provided all keys will be indexed
- **id_keys** (`list[str] | None`) – 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** (`list[Any] | tuple[Any] | None`) – 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 | None`) – number of docs in one chunk sent to es (default: 500)
- **max_chunk_bytes** (`int | None`) – the maximum size of the request in bytes (default: 100MB)
- **max_retries** (`int | None`) – 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 | None`) – 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 | None) – maximum number of seconds a retry will wait (default: 600)
- **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.
- **enable_refresh_interval** (bool) – True (default) to enable refresh_interval modification to -1 (disabled) while indexing documents
- ****kwargs** (Any) – 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

Return type
dict[str, Any]

Returns
Response payload https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#response.

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: opensearchpy.OpenSearch*, *df: DataFrame*, *index: str*, *doc_type: str | None = None*, *use_threads: bool | int = False*, ***kwargs: Any*) → Any

Index all documents from a DataFrame to OpenSearch index.

Parameters

- **client** (OpenSearch) – instance of opensearchpy.OpenSearch to use.
- **df** (DataFrame) – Pandas DataFrame
- **index** (str) – Name of the index.
- **doc_type** (str | None) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **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.
- ****kwargs** (Any) – KEYWORD arguments forwarded to *index_documents()* which is used to execute the operation

Return type
Any

Returns

Response payload https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#response.

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: opensearchpy.OpenSearch*, *path: str*, *index: str*, *doc_type: str | None = None*, *boto3_session: Session | None = None*, *json_path: str | None = None*, *use_threads: bool | int = False*, ***kwargs: Any*) → 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 | None) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **json_path** (str | None) – JsonPath expression to specify explicit path to a single name element in a JSON hierarchical data structure. Read more about JsonPath
- **boto3_session** (Session | None) – Boto3 Session to be used to access S3 if **path** is provided. The default boto3 session will be used if **boto3_session** is None.
- **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.
- ****kwargs** (Any) – KEYWORD arguments forwarded to *index_documents()* which is used to execute the operation

Return type

Any

Returns

Response payload https://opensearch.org/docs/opensearch/rest-api/document-apis/bulk/#response.

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: opensearchpy.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 | None) – 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] | None) – The search definition using the Query DSL.
- **doc_type** (str | None) – Name of the document type (for Elasticsearch versions 5.x and earlier).
- **is_scroll** (bool | None) – 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** (str | Collection[str] | None) – Use the filter_path parameter to reduce the size of the OpenSearch Service response (default: ['hits.hits._id','hits.hits._source'])
- ****kwargs** (Any) – KEYWORD arguments forwarded to opensearchpy.OpenSearch.search and also to opensearchpy.helpers.scan if *is_scroll=True*

Return type
DataFrame

Returns
Results as 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: opensearchpy.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** (Any) – KEYWORD arguments forwarded to request url (e.g.: filter_path, etc.)

Return type
DataFrame

Returns
Results as 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",
... )
```

1.6.11 Amazon Neptune

connect(host, port[, iam_enabled])	Create a connection to a Neptune cluster.
execute_gremlin(client, query)	Return results of a Gremlin traversal as pandas DataFrame.
execute_opencypher(client, query)	Return results of a openCypher traversal as pandas DataFrame.
execute_sparql(client, query)	Return results of a SPARQL query as pandas DataFrame.
flatten_nested_df(df[, include_prefix, ...])	Flatten the lists and dictionaries of the input data frame.
to_property_graph(client, df[, batch_size, ...])	Write records stored in a DataFrame into Amazon Neptune.
to_rdf_graph(client, df[, batch_size, ...])	Write records stored in a DataFrame into Amazon Neptune.
bulk_load(client, df, path, iam_role[, ...])	Write records into Amazon Neptune using the Neptune Bulk Loader.
bulk_load_from_files(client, path, iam_role)	Load files from S3 into Amazon Neptune using the Neptune Bulk Loader.

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) – True if IAM is enabled on the cluster. Defaults to False.

Return type
NeptuneClient

Returns
[description]

awswrangler.neptune.execute_gremlin

awswrangler.neptune.**execute_gremlin**(*client: NeptuneClient*, *query: str*) → DataFrame

Return results of a Gremlin traversal as pandas DataFrame.

Parameters

- **client** (NeptuneClient) – instance of the neptune client to use
- **query** (str) – The gremlin traversal to execute

Return type
DataFrame

Returns
Results as 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

Return type
DataFrame

Returns
Results as 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*) → DataFrame

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

Return type
DataFrame

Returns
Results as 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** (DataFrame) – The input data frame
- **include_prefix** (bool) – If True, then it will prefix the new column name with the original column name. Defaults to True.
- **separator** (str) – The separator to use between field names when a dictionary is exploded. Defaults to "_".
- **recursive** (bool) – If True, then this will recurse the fields in the data frame. Defaults to True.

Return type
DataFrame

Returns
The flattened DataFrame

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.

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.

Existing ~id values will be overwritten. If no ~id, ~label, ~to, or ~from column exists, an InvalidArgumentValue 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 `use_header_cardinality=False`.

Parameters

- **client** (NeptuneClient) – instance of the neptune client to use
- **df** (DataFrame) – Pandas DataFrame
- **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

Return type
bool

Returns
True if records were written

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'*) → bool

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

- **client** (NeptuneClient) – Instance of the neptune client to use.
- **df** (DataFrame) – Pandas DataFrame.
- **batch_size** (int) – The number of rows in the DataFrame (i.e. triples) to write into Amazon Neptune in one query. Defaults to 50.
- **subject_column** (str) – The column name in the DataFrame for the subject. Defaults to 's'.
- **predicate_column** (str) – The column name in the DataFrame for the predicate. Defaults to 'p'.
- **object_column** (str) – The column name in the DataFrame for the object. Defaults to 'o'.
- **graph_column** (str) – The column name in the DataFrame for the graph if sending across quads. Defaults to 'g'.

Return type
bool

Returns
True if records were written

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
... )
```

awswrangler.neptune.bulk_load

awswrangler.neptune.**bulk_load**(*client: NeptuneClient*, *df: DataFrame*, *path: str*, *iam_role: str*, *neptune_load_wait_polling_delay: float = 0.25*, *load_parallelism: Literal['LOW', 'MEDIUM', 'HIGH', 'OVERSUBSCRIBE'] = 'HIGH'*, *parser_configuration: BulkLoadParserConfiguration | None = None*, *update_single_cardinality_properties: Literal['TRUE', 'FALSE'] = 'FALSE'*, *queue_request: Literal['TRUE', 'FALSE'] = 'FALSE'*, *dependencies: list[str] | None = None*, *keep_files: bool = False*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, str] | None = None*) → None

Write records into Amazon Neptune using the Neptune Bulk Loader.

The DataFrame will be written to S3 and then loaded to Neptune using the Bulk Loader.

Parameters

- **client** (NeptuneClient) – Instance of the neptune client to use
- **df** (DataFrame) – Pandas DataFrame to write to Neptune.
- **path** (str) – S3 Path that the Neptune Bulk Loader will load data from.
- **iam_role** (str) – The Amazon Resource Name (ARN) for an IAM role to be assumed by the Neptune DB instance for access to the S3 bucket. For information about creating a role that has access to Amazon S3 and then associating it with a Neptune cluster, see Prerequisites: IAM Role and Amazon S3 Access.
- **neptune_load_wait_polling_delay** (float) – Interval in seconds for how often the function will check if the Neptune bulk load has completed.
- **load_parallelism** (Literal['LOW', 'MEDIUM', 'HIGH', 'OVERSUBSCRIBE']) – Specifies the number of threads used by Neptune's bulk load process.
- **parser_configuration** (BulkLoadParserConfiguration | None) – An optional object with additional parser configuration values. Each of the child parameters is also optional: namedGraphUri, baseUri and allowEmptyStrings.
- **update_single_cardinality_properties** (Literal['TRUE', 'FALSE']) – An optional parameter that controls how the bulk loader treats a new value for single-cardinality vertex or edge properties.
- **queue_request** (Literal['TRUE', 'FALSE']) – An optional flag parameter that indicates whether the load request can be queued up or not.

If omitted or set to "FALSE", the load request will fail if another load job is already running.

- **dependencies** (list[str] | None) – An optional parameter that can make a queued load request contingent on the successful completion of one or more previous jobs in the queue.
- **keep_files** (bool) – Whether to keep stage files or delete them. False by 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.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **s3_additional_kwargs** (dict[str, str] | None) – Forwarded to botocore requests. e.g. s3_additional_kwargs={'ServerSideEncryption': 'aws:kms', 'SSEKMSKeyId': 'YOUR_KMS_KEY_ARN'}

Return type
None

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> client = wr.neptune.connect("MY_NEPTUNE_ENDPOINT", 8182)
>>> frame = pd.DataFrame([{"~id": "0", "~labels": ["version"], "~properties": {"type
↪": "version"}}])
>>> wr.neptune.bulk_load(
...     client=client,
...     df=frame,
...     path="s3://my-bucket/stage-files/",
...     iam_role="arn:aws:iam::XXX:role/XXX"
... )
```

awswrangler.neptune.bulk_load_from_files

awswrangler.neptune.**bulk_load_from_files**(*client: NeptuneClient*, *path: str*, *iam_role: str*, *format: Literal['csv', 'opencypher', 'ntriples', 'nquads', 'rdfxml', 'turtle'] = 'csv'*, *neptune_load_wait_polling_delay: float = 0.25*, *load_parallelism: Literal['LOW', 'MEDIUM', 'HIGH', 'OVERSUBSCRIBE'] = 'HIGH'*, *parser_configuration: BulkLoadParserConfiguration | None = None*, *update_single_cardinality_properties: Literal['TRUE', 'FALSE'] = 'FALSE'*, *queue_request: Literal['TRUE', 'FALSE'] = 'FALSE'*, *dependencies: list[str] | None = None*) → None

Load files from S3 into Amazon Neptune using the Neptune Bulk Loader.

For more information about the Bulk Loader see here.

Parameters

- **client** (NeptuneClient) – Instance of the neptune client to use
- **path** (str) – S3 Path that the Neptune Bulk Loader will load data from.

- **iam_role** (str) – The Amazon Resource Name (ARN) for an IAM role to be assumed by the Neptune DB instance for access to the S3 bucket. For information about creating a role that has access to Amazon S3 and then associating it with a Neptune cluster, see Prerequisites: IAM Role and Amazon S3 Access.
- **format** (Literal['csv', 'opencypher', 'ntriples', 'nquads', 'rdfxml', 'turtle']) – The format of the data.
- **neptune_load_wait_polling_delay** (float) – Interval in seconds for how often the function will check if the Neptune bulk load has completed.
- **load_parallelism** (Literal['LOW', 'MEDIUM', 'HIGH', 'OVERSUBSCRIBE']) – Specifies the number of threads used by Neptune's bulk load process.
- **parser_configuration** (BulkLoadParserConfiguration | None) – An optional object with additional parser configuration values. Each of the child parameters is also optional: namedGraphUri, baseUri and allowEmptyStrings.
- **update_single_cardinality_properties** (Literal['TRUE', 'FALSE']) – An optional parameter that controls how the bulk loader treats a new value for single-cardinality vertex or edge properties.
- **queue_request** (Literal['TRUE', 'FALSE']) – An optional flag parameter that indicates whether the load request can be queued up or not.

  If omitted or set to "FALSE", the load request will fail if another load job is already running.
- **dependencies** (list[str] | None) – An optional parameter that can make a queued load request contingent on the successful completion of one or more previous jobs in the queue.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> client = wr.neptune.connect("MY_NEPTUNE_ENDPOINT", 8182)
>>> wr.neptune.bulk_load_from_files(
...     client=client,
...     path="s3://my-bucket/stage-files/",
...     iam_role="arn:aws:iam::XXX:role/XXX",
...     format="csv",
... )
```

1.6.12 DynamoDB

delete_items(items, table_name[, boto3_session])	Delete all items in the specified DynamoDB table.
execute_statement(statement[, parameters, ...])	Run a PartiQL statement against a DynamoDB table.
get_table(table_name[, boto3_session])	Get DynamoDB table object for specified table name.
put_csv(path, table_name[, boto3_session, ...])	Write all items from a CSV file to a DynamoDB.
put_df(df, table_name[, boto3_session, ...])	Write all items from a DataFrame to a DynamoDB.
put_items(items, table_name[, ...])	Insert all items to the specified DynamoDB table.
put_json(path, table_name[, boto3_session, ...])	Write all items from JSON file to a DynamoDB.
read_items(table_name[, index_name, ...])	Read items from given DynamoDB table.
read_partiql_query(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*) → None

Delete all items in the specified DynamoDB table.

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** (Session | None) – The default boto3 session will be used if **boto3_session** is 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[list[dict[str, Any]]] | None

Run a PartiQL statement against a DynamoDB table.

Parameters

- **statement** (str) – The PartiQL statement.
- **parameters** (list[Any] | None) – 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** (Session | None) – Boto3 Session. If None, the default boto3 Session is used.

Return type
Iterator[list[dict[str, Any]]] | None

Returns
An iterator of the items from the statement response, if 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],
... )
```

awswrangler.dynamodb.get_table

awswrangler.dynamodb.**get_table**(*table_name: str*, *boto3_session: boto3.Session | None = None*) → Table

Get DynamoDB table object for specified table name.

Parameters

- **table_name** (str) – Name of the Amazon DynamoDB table.
- **boto3_session** (boto3.Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type

Table

Returns

Boto3 DynamoDB.Table object. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/dynamodb.html#DynamoDB.Table

awswrangler.dynamodb.put_csv

awswrangler.dynamodb.**put_csv**(*path: str | Path*, *table_name: str*, *boto3_session: Session | None = None*, *use_threads: bool | int = True*, ***pandas_kwargs: Any*) → None

Write all items from a CSV file to a DynamoDB.

Parameters

- **path** (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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **use_threads** (bool | int) – Used for Parallel Write requests. True (default) to enable concurrency, 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.
- **pandas_kwargs** (Any) – 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

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*, *use_threads: bool | int = True*) → None

Write all items from a DataFrame to a DynamoDB.

Parameters

- **df** (DataFrame) – Pandas DataFrame
- **table_name** (str) – Name of the Amazon DynamoDB table.
- **use_threads** (bool | int) – Used for Parallel Write requests. True (default) to enable concurrency, 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** (Session | None) – The default boto3 session will be used if **boto3_session** is 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*, *use_threads: bool | int = True*) → None

Insert all items to the specified DynamoDB table.

Parameters

- **items** (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** (Session | None) – Boto3 Session. The default boto3 Session will be used if boto3_session receive None.
- **use_threads** (bool | int) – Used for Parallel Write requests. True (default) to enable concurrency, 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.

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*, *use_threads: bool | int = True*) → None

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.

Parameters

- **path** (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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **use_threads** (bool | int) – Used for Parallel Write requests. True (default) to enable concurrency, 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.

Return type
None

Examples

Writing contents of JSON file

```
>>> import awswrangler as wr
>>> wr.dynamodb.put_json(
...     path='items.json',
...     table_name='table'
... )
```

awswrangler.dynamodb.read_items

awswrangler.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*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *as_dataframe: bool = True*, *chunked: bool = False*, *use_threads: bool | int = True*, *boto3_session: boto3.Session | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*, *key_schema: list[dict[str, str]] | None = None*) → pd.DataFrame | Iterator[pd.DataFrame] | _ItemsListType | Iterator[_ItemsListType]

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 entry point.

Under the hood, it wraps all the four available read actions: *get_item*, *batch_get_item*, *query* and *scan*.

Warning: To avoid a potentially costly Scan operation, please make sure to pass arguments such as *partition_values* or *max_items_evaluated*. Note that *filter_expression* is applied AFTER a Scan

Note: Number of Parallel Scan segments is based on the *use_threads* argument. A parallel scan with a large number of workers could consume all the provisioned throughput of the table or index. See: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Scan.html#Scan.ParallelScan

Note: If *max_items_evaluated* is specified, then *use_threads=False* is enforced. This is because it's not possible to limit the number of items in a Query/Scan operation across threads.

Parameters

- **table_name** (str) – DynamoDB table name.
- **index_name** (str | None) – Name of the secondary global or local index on the table. Defaults to None.
- **partition_values** (Sequence[Any] | None) – Partition key values to retrieve. Defaults to None.
- **sort_values** (Sequence[Any] | None) – Sort key values to retrieve. Defaults to None.
- **filter_expression** (ConditionBase | str | None) – Filter expression as string or combinations of boto3.dynamodb.conditions.Attr conditions. Defaults to None.
- **key_condition_expression** (ConditionBase | str | None) – Key condition expression as string or combinations of boto3.dynamodb.conditions.Key conditions. Defaults to None.
- **expression_attribute_names** (dict[str, str] | None) – Mapping of placeholder and target attributes. Defaults to None.

- **expression_attribute_values** (dict[str, Any] | None) – 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] | None) – 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 | None) – Limit the number of items evaluated in case of query or scan operations. Defaults to None (all matching items). When set, *use_threads* is enforced to False.
- **dtype_backend** (Literal['numpy_nullable', 'pyarrow']) – Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

  The dtype_backends are still experimential. The "pyarrow" backend is only supported with Pandas 2.0 or above.
- **as_dataframe** (bool) – If True, return items as pd.DataFrame, otherwise as list/dict. Defaults to True.
- **chunked** (bool) – If *True* an iterable of DataFrames/lists is returned. False by default.
- **use_threads** (bool | int) – Used for Parallel Scan requests. True (default) to enable concurrency, 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 | None) – The default boto3 session will be used if **boto3_session** is `None`.
- **pyarrow_additional_kwargs** (dict[str, Any] | None) – Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.
- **key_schema** (list[dict[str, str]] | None) – Key schema of the table (e.g. *[{"AttributeName": "key", "KeyType": "HASH"}]*). If provided, the library will bypass the *DescribeTable* API call, which can reduce network latency and prevent API throttling. Defaults to None.

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. Alternatively, the return type can be an iterable of either type when *chunked=True*.

Return type

```
pd.DataFrame | list[dict[str, Any]] | Iterable[pd.DataFrame] |
Iterable[list[dict[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',
```

(continues on next page)

(continued from previous page)

```
...     expression_attribute_values={':v1': 'val_1', ':v2': 'val_2'},
... )
```

Reading items matching a FilterExpression expressed with boto3.dynamodb.conditions.Attr Note that FilterExpression is applied AFTER a Scan operation

```
>>> 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** (list[Any] | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type

DataFrame | Iterator[DataFrame]

Returns

Result as Pandas 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"
... )
```

1.6.13 Amazon S3 Tables

create_table_bucket(name[, boto3_session])	Create an S3 Table Bucket.
create_namespace(table_bucket_arn, namespace)	Create a namespace in an S3 Table Bucket.
create_table(table_bucket_arn, namespace, ...)	Create a table in an S3 Table Bucket namespace.
delete_table_bucket(table_bucket_arn[, ...])	Delete an S3 Table Bucket.
delete_namespace(table_bucket_arn, namespace)	Delete a namespace from an S3 Table Bucket.
delete_table(table_bucket_arn, namespace, ...)	Delete a table from an S3 Table Bucket namespace.
from_iceberg(table_bucket_arn, namespace, ...)	Read an S3 Table into a Pandas DataFrame via PyIceberg.
to_iceberg(df, table_bucket_arn, namespace, ...)	Write a Pandas DataFrame to an S3 Table via PyIceberg.

awswrangler.s3.create_table_bucket

awswrangler.s3.**create_table_bucket**(*name: str*, *boto3_session: Session | None = None*) → str

Create an S3 Table Bucket.

Parameters

- **name** (*str*) – The name of the table bucket to create.
- **boto3_session** (*boto3.Session, optional*) – Boto3 Session. If None, the default boto3 session is used.

Returns

The ARN of the created table bucket.

Return type

str

Examples

```
>>> import awswrangler as wr
>>> arn = wr.s3.create_table_bucket(name="my-table-bucket")
```

awswrangler.s3.create_namespace

awswrangler.s3.**create_namespace**(*table_bucket_arn: str*, *namespace: str*, *boto3_session: Session | None = None*) → str

Create a namespace in an S3 Table Bucket.

Parameters

- **table_bucket_arn** (*str*) – The ARN of the table bucket.
- **namespace** (*str*) – The name of the namespace to create.
- **boto3_session** (*boto3.Session, optional*) – Boto3 Session. If None, the default boto3 session is used.

Returns
The namespace name.

Return type
str

Examples

```
>>> import awswrangler as wr
>>> ns = wr.s3.create_namespace(
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
...     namespace="my_namespace",
... )
```

awswrangler.s3.create_table

awswrangler.s3.**create_table**(*table_bucket_arn: str*, *namespace: str*, *table_name: str*, *format: str = 'ICEBERG'*, *boto3_session: Session | None = None*) → str

Create a table in an S3 Table Bucket namespace.

Parameters

- **table_bucket_arn** (*str*) – The ARN of the table bucket.
- **namespace** (*str*) – The namespace in which to create the table.
- **table_name** (*str*) – The name of the table to create.
- **format** (*str, optional*) – The table format. Default is "ICEBERG".
- **boto3_session** (*boto3.Session, optional*) – Boto3 Session. If None, the default boto3 session is used.

Returns
The ARN of the created table.

Return type
str

Examples

```
>>> import awswrangler as wr
>>> table_arn = wr.s3.create_table(
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
...     namespace="my_namespace",
...     table_name="my_table",
... )
```

awswrangler.s3.delete_table_bucket

awswrangler.s3.**delete_table_bucket**(*table_bucket_arn: str*, *boto3_session: Session | None = None*) → None

Delete an S3 Table Bucket.

Parameters

- **table_bucket_arn** (*str*) – The ARN of the table bucket to delete.
- **boto3_session** (*boto3.Session, optional*) – Boto3 Session. If None, the default boto3 session is used.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.delete_table_bucket(
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
... )
```

awswrangler.s3.delete_namespace

awswrangler.s3.**delete_namespace**(*table_bucket_arn: str*, *namespace: str*, *boto3_session: Session | None = None*) → None

Delete a namespace from an S3 Table Bucket.

Parameters

- **table_bucket_arn** (*str*) – The ARN of the table bucket.
- **namespace** (*str*) – The name of the namespace to delete.
- **boto3_session** (*boto3.Session, optional*) – Boto3 Session. If None, the default boto3 session is used.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.delete_namespace(
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
...     namespace="my_namespace",
... )
```

awswrangler.s3.delete_table

awswrangler.s3.**delete_table**(*table_bucket_arn: str*, *namespace: str*, *table_name: str*, *version_token: str | None = None*, *boto3_session: Session | None = None*) → None

Delete a table from an S3 Table Bucket namespace.

Parameters

- **table_bucket_arn** (*str*) – The ARN of the table bucket.
- **namespace** (*str*) – The namespace of the table.
- **table_name** (*str*) – The name of the table to delete.
- **version_token** (*str, optional*) – The version token of the table. If not provided, the current version is deleted.
- **boto3_session** (*boto3.Session, optional*) – Boto3 Session. If None, the default boto3 session is used.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> wr.s3.delete_table(
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
...     namespace="my_namespace",
...     table_name="my_table",
... )
```

awswrangler.s3.from_iceberg

awswrangler.s3.**from_iceberg**(*table_bucket_arn: str*, *namespace: str*, *table_name: str*, *columns: list[str] | None = None*, *row_filter: str | None = None*, *snapshot_id: int | None = None*, *limit: int | None = None*, *dtype_backend: Literal['numpy_nullable', 'pyarrow'] = 'numpy_nullable'*, *boto3_session: Session | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*) → DataFrame

Read an S3 Table into a Pandas DataFrame via PyIceberg.

This function requires the `pyiceberg` package. Install it with `pip install awswrangler[pyiceberg]`.

By default, the S3 Tables REST endpoint is used. To use the AWS Glue Iceberg REST endpoint instead, set `wr.config.s3tables_catalog_endpoint_url` (e.g. `"https://glue.<region>.amazonaws.com/iceberg`

"). See Integrating S3 Tables with AWS analytics services for the required Glue Data Catalog and Lake Formation setup.

Parameters

- **table_bucket_arn** (*str*) – The ARN of the S3 table bucket.
- **namespace** (*str*) – The namespace of the table.
- **table_name** (*str*) – The name of the table to read.
- **columns** (*list[str], optional*) – List of column names to read. If None, all columns are read.
- **row_filter** (*str, optional*) – A row filter expression (e.g. `"col > 5"`). If None, all rows are read.
- **snapshot_id** (*int, optional*) – A specific snapshot ID to read. If None, the latest snapshot is read.
- **limit** (*int, optional*) – Maximum number of rows to return. If None, all rows are returned.
- **dtype_backend** (*str, optional*) – Which dtype_backend to use. `"numpy_nullable"` or `"pyarrow"`.
- **boto3_session** (*boto3.Session, optional*) – Boto3 Session. If None, the default boto3 session is used.
- **pyarrow_additional_kwargs** (*dict[str, Any], optional*) – Additional keyword arguments forwarded to PyArrow's `to_pandas()` method.

Returns

DataFrame with the table data.

Return type

pd.DataFrame

Examples

Reading an entire table:

```
>>> import awswrangler as wr
>>> df = wr.s3.from_iceberg(
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
...     namespace="my_namespace",
...     table_name="my_table",
... )
```

Reading with row filtering and limit:

```
>>> df = wr.s3.from_iceberg(
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
...     namespace="my_namespace",
...     table_name="my_table",
...     row_filter="amount > 50.0",
...     limit=100,
... )
```

Reading via the Glue Iceberg REST endpoint:

```
>>> wr.config.s3tables_catalog_endpoint_url = "https://glue.us-east-1.amazonaws.com/
↪iceberg"
>>> df = wr.s3.from_iceberg(
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
...     namespace="my_namespace",
...     table_name="my_table",
... )
```

awswrangler.s3.to_iceberg

awswrangler.s3.**to_iceberg**(*df: DataFrame*, *table_bucket_arn: str*, *namespace: str*, *table_name: str*, *mode: Literal['append', 'overwrite'] = 'append'*, *index: bool = False*, *dtype: dict[str, str] | None = None*, *boto3_session: Session | None = None*) → None

Write a Pandas DataFrame to an S3 Table via PyIceberg.

If the table does not exist, it is automatically created with a schema inferred from the DataFrame.

This function requires the `pyiceberg` package. Install it with `pip install awswrangler[pyiceberg]`.

By default, the S3 Tables REST endpoint is used. To use the AWS Glue Iceberg REST endpoint instead, set `wr.config.s3tables_catalog_endpoint_url` (e.g. `"https://glue.<region>.amazonaws.com/iceberg"`). See Integrating S3 Tables with AWS analytics services for the required Glue Data Catalog and Lake Formation setup.

Parameters

- **df** (*pd.DataFrame*) – Pandas DataFrame to write.
- **table_bucket_arn** (*str*) – The ARN of the S3 table bucket.
- **namespace** (*str*) – The namespace of the table.
- **table_name** (*str*) – The name of the table to write to.
- **mode** (*str, optional*) – Write mode. `"append"` (default) adds rows to the table. `"overwrite"` replaces all existing data.
- **index** (*bool, optional*) – If True, include the DataFrame index as a column. Default is False.
- **dtype** (*dict[str, str], optional*) – Dictionary of column names and Athena/Glue types to cast. (e.g. `{"col_name": "bigint", "col2_name": "int"}`).
- **boto3_session** (*boto3.Session, optional*) – Boto3 Session. If None, the default boto3 session is used.

Return type

None

Examples

```
>>> import awswrangler as wr
>>> import pandas as pd
>>> wr.s3.to_iceberg(
...     df=pd.DataFrame({"col": [1, 2, 3]}),
...     table_bucket_arn="arn:aws:s3tables:us-east-1:123456789012:bucket/my-bucket",
...     namespace="my_namespace",
...     table_name="my_table",
... )
```

1.6.14 Amazon S3 Vectors

create_vector_bucket(name, *[, ...])	Create an Amazon S3 Vectors bucket.
delete_vector_bucket([name, arn, boto3_session])	Delete an Amazon S3 Vectors bucket.
list_vector_buckets([prefix, boto3_session])	List all Amazon S3 Vectors buckets in the account/region (paginates internally).
get_vector_bucket([name, arn, boto3_session])	Get attributes of a vector bucket.
create_vector_index(*, name, dimension[, ...])	Create a vector index inside an Amazon S3 Vectors bucket.
delete_vector_index(*[, name, arn, ...])	Delete a vector index.
list_vector_indexes(*[, vector_bucket, ...])	List all vector indexes in a bucket (paginates internally).
get_vector_index(*[, name, arn, ...])	Get attributes of a vector index.
put_vectors(*, vectors[, vector_bucket, ...])	Insert one or more vectors into an Amazon S3 Vectors index.
put_vectors_from_df(df, *, key_column[, ...])	Insert all rows of a DataFrame into an Amazon S3 Vectors index.
get_vectors(*, keys[, return_data, ...])	Retrieve vectors by key.
delete_vectors(*, keys[, vector_bucket, ...])	Delete vectors by key (chunks at 500 per underlying call).
list_vectors(*[, return_data, ...])	List all vectors in an index.
query_vectors(*[, query_vector, query_text, ...])	Approximate-nearest-neighbour query against an Amazon S3 Vectors index.

awswrangler.s3.create_vector_bucket

awswrangler.s3.**create_vector_bucket**(*name: str*, *, *encryption_kms_key_arn: str | None = None*, *sse_type: str | None = None*, *tags: dict[str, str] | None = None*, *boto3_session: Session | None = None*) → str

Create an Amazon S3 Vectors bucket.

Parameters

- **name** (str) – Name of the vector bucket to create. 3-63 chars.
- **encryption_kms_key_arn** (str | None) – Optional KMS key ARN for SSE-KMS encryption. Implies sse_type='aws:kms' if not specified.
- **sse_type** (str | None) – Server-side encryption type. 'AES256' (default if encryption block omitted) or 'aws:kms'.
- **tags** (dict[str, str] | None) – Resource tags as a dict.

- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
ARN of the created vector bucket.

Examples

```
>>> import awswrangler as wr
>>> arn = wr.s3.create_vector_bucket("my-vector-bucket")
```

awswrangler.s3.delete_vector_bucket

awswrangler.s3.**delete_vector_bucket**(*name: str | None = None*, ***, *arn: str | None = None*, *boto3_session: Session | None = None*) → None

Delete an Amazon S3 Vectors bucket. Specify either name or arn.

Return type
None

awswrangler.s3.list_vector_buckets

awswrangler.s3.**list_vector_buckets**(*prefix: str | None = None*, ***, *boto3_session: Session | None = None*) → list[dict[str, Any]]

List all Amazon S3 Vectors buckets in the account/region (paginates internally).

Parameters

- **prefix** (str | None) – Optional name prefix filter.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
List of vector bucket summaries (each a dict with vectorBucketName, vectorBucketArn, creationTime).

awswrangler.s3.get_vector_bucket

awswrangler.s3.**get_vector_bucket**(*name: str | None = None*, ***, *arn: str | None = None*, *boto3_session: Session | None = None*) → dict[str, Any]

Get attributes of a vector bucket. Specify either name or arn.

Return type
dict[str, Any]

awswrangler.s3.create_vector_index

awswrangler.s3.**create_vector_index**(*, *name: str*, *dimension: int*, *distance_metric: str = 'cosine'*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *data_type: str = 'float32'*, *non_filterable_metadata_keys: list[str] | None = None*, *encryption_kms_key_arn: str | None = None*, *sse_type: str | None = None*, *tags: dict[str, str] | None = None*, *boto3_session: Session | None = None*) → str

Create a vector index inside an Amazon S3 Vectors bucket.

Parameters

- **name** (str) – Index name (3-63 chars).
- **dimension** (int) – Vector dimension (1-4096). All vectors written to the index must match.
- **distance_metric** (str) – `'cosine'` (default) or `'euclidean'`.
- **vector_bucket_arn** (*vector_bucket /*) – Target vector bucket. Specify exactly one.
- **data_type** (str) – Vector element type. Currently only `'float32'` is supported.
- **non_filterable_metadata_keys** (list[str] | None) – Metadata keys excluded from filtering (up to 10). Cannot be changed after index creation.
- **encryption_kms_key_arn** (str | None) – Encryption overrides; default is to inherit from the bucket.
- **sse_type** (str | None) – Encryption overrides; default is to inherit from the bucket.
- **tags** (dict[str, str] | None) – Resource tags.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
ARN of the created vector index.

Examples

```
>>> import awswrangler as wr
>>> arn = wr.s3.create_vector_index(
...     vector_bucket="my-bucket", name="my-index", dimension=384
... )
```

awswrangler.s3.delete_vector_index

awswrangler.s3.**delete_vector_index**(*, *name: str | None = None*, *arn: str | None = None*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *boto3_session: Session | None = None*) → None

Delete a vector index. Specify either `arn`, or `name` together with `vector_bucket`/`vector_bucket_arn`.

Return type
None

awswrangler.s3.list_vector_indexes

awswrangler.s3.**list_vector_indexes**(*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *prefix: str | None = None*, *boto3_session: Session | None = None*) → list[dict[str, Any]]

List all vector indexes in a bucket (paginates internally).

Return type
list[dict[str, Any]]

awswrangler.s3.get_vector_index

awswrangler.s3.**get_vector_index**(*, *name: str | None = None*, *arn: str | None = None*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *boto3_session: Session | None = None*) → dict[str, Any]

Get attributes of a vector index.

Return type
dict[str, Any]

awswrangler.s3.put_vectors

awswrangler.s3.**put_vectors**(*, *vectors: list[dict[str, Any]]*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *index: str | None = None*, *index_arn: str | None = None*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*) → None

Insert one or more vectors into an Amazon S3 Vectors index.

Parameters

- **vectors** (list[dict[str, Any]]) – List of dicts, each shaped `{"key": str, "data": list[float] | dict[str, list[float]] | np.ndarray, "metadata": dict | None}`. `data` is automatically cast to float32. Up to 500 vectors per underlying API call.
- **index_arn** (*vector_bucket / vector_bucket_arn / index /*) – Target index. See module docstring for resolution rules.
- **use_threads** (bool | int) – Concurrency for batched calls.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
None

awswrangler.s3.put_vectors_from_df

awswrangler.s3.**put_vectors_from_df**(*df: DataFrame*, *, *key_column: str*, *vector_column: str | None = None*, *metadata_columns: list[str] | None = None*, *text_column: str | None = None*, *bedrock_model_id: str | None = None*, *bedrock_model_kwargs: dict[str, Any] | None = None*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *index: str | None = None*, *index_arn: str | None = None*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*) → None

Insert all rows of a DataFrame into an Amazon S3 Vectors index.

Either `vector_column` (precomputed embeddings) or `text_column` + `bedrock_model_id` (embed via Amazon Bedrock on the fly) must be provided.

Parameters

- **df** (`DataFrame`) – Input DataFrame.
- **key_column** (`str`) – Column containing the per-row vector key (string).
- **vector_column** (`str | None`) – Column containing the precomputed embedding (list[float] / np.ndarray per row).
- **metadata_columns** (`list[str] | None`) – Columns to attach as filterable/non-filterable metadata. `None` means "all columns except `key_column`, `vector_column` and `text_column`" — note `text_column` is excluded by default; pass it explicitly here (e.g. for RAG citations) to keep it. NaN / `pd.NA` / `None` cells are dropped per row.
- **text_column** (`str | None`) – Column containing input text to embed via Bedrock. Mutually exclusive with `vector_column`.
- **bedrock_model_id** (`str | None`) – Bedrock embedding model and optional model-specific kwargs (e.g. `{"dimensions": 256}`).
- **bedrock_model_kwargs** (`dict[str, Any] | None`) – Bedrock embedding model and optional model-specific kwargs (e.g. `{"dimensions": 256}`).
- **index_arn** (*vector_bucket / vector_bucket_arn / index /*) – Target index.
- **use_threads** (`bool | int`) – Concurrency for batched put calls and for parallel Bedrock embedding.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `None`.

Return type

`None`

Examples

Pre-computed vectors:

```
>>> import awswrangler as wr
>>> wr.s3.put_vectors_from_df(
...     df=my_df,
...     key_column="id",
...     vector_column="embedding",
...     vector_bucket="my-bucket",
...     index="my-index",
... )
```

Embed-on-write via Bedrock Titan:

```
>>> wr.s3.put_vectors_from_df(
...     df=my_df,
...     key_column="id",
...     text_column="content",
```

(continues on next page)

(continued from previous page)

```
...     bedrock_model_id="amazon.titan-embed-text-v2:0",
...     vector_bucket="my-bucket",
...     index="my-index",
... )
```

awswrangler.s3.get_vectors

awswrangler.s3.**get_vectors**(*, *keys: list[str]*, *return_data: bool = False*, *return_metadata: bool = False*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *index: str | None = None*, *index_arn: str | None = None*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*) → DataFrame

Retrieve vectors by key. Returns a DataFrame with columns `key` and (optionally) `vector`, `metadata`.

Up to 100 keys per underlying API call (chunked automatically).

Return type
DataFrame

awswrangler.s3.delete_vectors

awswrangler.s3.**delete_vectors**(*, *keys: list[str]*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *index: str | None = None*, *index_arn: str | None = None*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*) → None

Delete vectors by key (chunks at 500 per underlying call).

Return type
None

awswrangler.s3.list_vectors

awswrangler.s3.**list_vectors**(*, *return_data: bool = False*, *return_metadata: bool = False*, *max_items: int | None = None*, *chunked: bool | int = False*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *index: str | None = None*, *index_arn: str | None = None*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*) → DataFrame | Iterator[DataFrame]

List all vectors in an index. Uses parallel segments (up to 16) when `use_threads` enables it.

Parameters

- **return_data** (bool) – Whether to include each vector's data and metadata.
- **return_metadata** (bool) – Whether to include each vector's data and metadata.
- **max_items** (int | None) – Optional cap on total vectors returned across all pages/segments.
- **chunked** (bool | int) – Batching (memory-friendly). Returns an iterator of DataFrames instead of one frame:
  - True — yield one DataFrame per underlying API page.
  - INTEGER — yield DataFrames of exactly this many rows (final frame may be shorter).

  Chunked streaming is single-segment (sequential) regardless of `use_threads`.

- **index_arn** (*vector_bucket / vector_bucket_arn / index /*) – Target index.
- **use_threads** (bool | int) – Concurrency for parallel-segment listing. Ignored when chunked is truthy.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
DataFrame | Iterator[DataFrame]

Returns
DataFrame with columns key and (optionally) vector, metadata — or an iterator of such DataFrames when chunked is truthy.

awswrangler.s3.query_vectors

awswrangler.s3.**query_vectors**(*, *query_vector: list[float] | ndarray[Any, Any] | None = None*, *query_text: str | None = None*, *top_k: int = 10*, *filter: dict[str, Any] | None = None*, *return_distance: bool = True*, *return_metadata: bool = True*, *bedrock_model_id: str | None = None*, *bedrock_model_kwargs: dict[str, Any] | None = None*, *vector_bucket: str | None = None*, *vector_bucket_arn: str | None = None*, *index: str | None = None*, *index_arn: str | None = None*, *boto3_session: Session | None = None*) → DataFrame

Approximate-nearest-neighbour query against an Amazon S3 Vectors index.

Parameters

- **query_vector** (list[float] | ndarray[Any, Any] | None) – Pre-computed query embedding.
- **query_text** (str | None) – Text to embed via Bedrock (requires bedrock_model_id).
- **top_k** (int) – Number of nearest neighbours to return (1-100).
- **filter** (dict[str, Any] | None) – Metadata filter (MongoDB-style operators: $eq, $ne, $gt, $gte, $lt, $lte, $in, $nin, $exists, $and, $or).
- **return_distance** (bool) – Whether to include each result's distance and metadata.
- **return_metadata** (bool) – Whether to include each result's distance and metadata.
- **bedrock_model_id** (str | None) – Bedrock embedding configuration when using query_text.
- **bedrock_model_kwargs** (dict[str, Any] | None) – Bedrock embedding configuration when using query_text.
- **index_arn** (*vector_bucket / vector_bucket_arn / index /*) – Target index.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
DataFrame

Returns
DataFrame with columns key and (optionally) distance, metadata. The configured distance metric is exposed via df.attrs['distance_metric'].

Examples

```
>>> import awswrangler as wr
>>> df = wr.s3.query_vectors(
...     query_vector=[0.1, 0.2, 0.3],
...     top_k=5,
...     filter={"genre": {"$eq": "documentary"}},
...     vector_bucket="my-bucket",
...     index="my-index",
... )
```

1.6.15 Amazon Timestream

batch_load(df, path, database, table, ...[, ...])	Batch load a Pandas DataFrame into a Amazon Timestream table.
batch_load_from_files(path, database, table, ...)	Batch load files from S3 into a Amazon Timestream table.
create_database(database[, kms_key_id, ...])	Create a new Timestream database.
create_table(database, table, ...[, tags, ...])	Create a new Timestream database.
delete_database(database[, boto3_session])	Delete a given Timestream database.
delete_table(database, table[, boto3_session])	Delete a given Timestream table.
list_databases([boto3_session])	List all databases in timestream.
list_tables([database, boto3_session])	List tables in timestream.
query(sql[, chunked, pagination_config, ...])	Run a query and retrieve the result as a Pandas DataFrame.
wait_batch_load_task(task_id[, ...])	Wait for the Timestream batch load task to complete.
write(df, database, table[, time_col, ...])	Store a Pandas DataFrame into an Amazon Timestream table.
unload_to_files(sql, path[, unload_format, ...])	Unload query results to Amazon S3.
unload(sql, path[, unload_format, ...])	Unload query results to Amazon S3 and read the results as Pandas Data Frame.

awswrangler.timestream.batch_load

awswrangler.timestream.**batch_load**(*df: DataFrame*, *path: str*, *database: str*, *table: str*, *time_col: str*, *dimensions_cols: list[str]*, *measure_cols: list[str]*, *measure_name_col: str*, *report_s3_configuration:* TimestreamBatchLoadReportS3Configuration, *time_unit: Literal['MILLISECONDS', 'SECONDS', 'MICROSECONDS', 'NANOSECONDS'] = 'MILLISECONDS'*, *record_version: int = 1*, *timestream_batch_load_wait_polling_delay: float = 2*, *keep_files: bool = False*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*, *s3_additional_kwargs: dict[str, str] | None = None*) → dict[str, Any]

Batch load a Pandas DataFrame into a Amazon Timestream table.

Note: The supplied column names (time, dimension, measure) MUST match those in the Timestream table.

Note: Only `MultiMeasureMappings` is supported. See https://docs.aws.amazon.com/timestream/latest/developerguide/batch-load-data-model-mappings.html

Parameters

- **df** (`DataFrame`) – Pandas DataFrame.
- **path** (`str`) – S3 prefix to write the data.
- **database** (`str`) – Amazon Timestream database name.
- **table** (`str`) – Amazon Timestream table name.
- **time_col** (`str`) – Column name with the time data. It must be a long data type that represents the time since the Unix epoch.
- **dimensions_cols** (`list[str]`) – List of column names with the dimensions data.
- **measure_cols** (`list[str]`) – List of column names with the measure data.
- **measure_name_col** (`str`) – Column name with the measure name.
- **report_s3_configuration** (*TimestreamBatchLoadReportS3Configuration*) – Dictionary of the configuration for the S3 bucket where the error report is stored. https://docs.aws.amazon.com/timestream/latest/developerguide/API_ReportS3Configuration.html Example: {"BucketName": 'error-report-bucket-name'}
- **time_unit** (`Literal['MILLISECONDS', 'SECONDS', 'MICROSECONDS', 'NANOSECONDS']`) – Time unit for the time column. MILLISECONDS by default.
- **record_version** (`int`) – Record version.
- **timestream_batch_load_wait_polling_delay** (`float`) – Time to wait between two polling attempts.
- **keep_files** (`bool`) – Whether to keep the files after the operation.
- **use_threads** (`bool | int`) – True to enable concurrent requests, False to disable multiple threads.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `None`.
- **s3_additional_kwargs** (`dict[str, str] | None`) – Forwarded to S3 botocore requests.

Return type

`dict[str, Any]`

Returns

A dictionary of the batch load task response.

Examples

```
>>> import awswrangler as wr
```

```
>>> response = wr.timestream.batch_load(
>>>     df=df,
>>>     path='s3://bucket/path/',
>>>     database='sample_db',
>>>     table='sample_table',
>>>     time_col='time',
>>>     dimensions_cols=['region', 'location'],
>>>     measure_cols=['memory_utilization', 'cpu_utilization'],
>>>     report_s3_configuration={'BucketName': 'error-report-bucket-name'},
>>> )
```

awswrangler.timestream.batch_load_from_files

awswrangler.timestream.**batch_load_from_files**(*path: str*, *database: str*, *table: str*, *time_col: str*, *dimensions_cols: list[str]*, *measure_cols: list[str]*, *measure_types: list[str]*, *measure_name_col: str*, *report_s3_configuration:* TimestreamBatchLoadReportS3Configuration, *time_unit: Literal['MILLISECONDS', 'SECONDS', 'MICROSECONDS', 'NANOSECONDS'] = 'MILLISECONDS'*, *record_version: int = 1*, *data_source_csv_configuration: dict[str, str | bool] | None = None*, *timestream_batch_load_wait_polling_delay: float = 2*, *boto3_session: Session | None = None*) → dict[str, Any]

Batch load files from S3 into a Amazon Timestream table.

Note: The supplied column names (time, dimension, measure) MUST match those in the Timestream table.

Note: Only `MultiMeasureMappings` is supported. See https://docs.aws.amazon.com/timestream/latest/developerguide/batch-load-data-model-mappings.html

Parameters

- **path** (str) – S3 prefix to write the data.
- **database** (str) – Amazon Timestream database name.
- **table** (str) – Amazon Timestream table name.
- **time_col** (str) – Column name with the time data. It must be a long data type that represents the time since the Unix epoch.
- **dimensions_cols** (list[str]) – List of column names with the dimensions data.
- **measure_cols** (list[str]) – List of column names with the measure data.
- **measure_name_col** (str) – Column name with the measure name.

- **report_s3_configuration** (*TimestreamBatchLoadReportS3Configuration*) – Dictionary of the configuration for the S3 bucket where the error report is stored. https://docs.aws.amazon.com/timestream/latest/developerguide/API_ReportS3Configuration.html Example: {"BucketName": 'error-report-bucket-name'}
- **time_unit** (Literal['MILLISECONDS', 'SECONDS', 'MICROSECONDS', 'NANOSECONDS']) – Time unit for the time column. MILLISECONDS by default.
- **record_version** (int) – Record version.
- **data_source_csv_configuration** (dict[str, str | bool] | None) – Dictionary of the data source CSV configuration. https://docs.aws.amazon.com/timestream/latest/developerguide/API_CsvConfiguration.html
- **timestream_batch_load_wait_polling_delay** (float) – Time to wait between two polling attempts.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

Returns
A dictionary of the batch load task response.

Examples

```
>>> import awswrangler as wr
```

```
>>> response = wr.timestream.batch_load_from_files(
>>>     path='s3://bucket/path/',
>>>     database='sample_db',
>>>     table='sample_table',
>>>     time_col='time',
>>>     dimensions_cols=['region', 'location'],
>>>     measure_cols=['memory_utilization', 'cpu_utilization'],
>>>     report_s3_configuration={'BucketName': 'error-report-bucket-name'},
>>> )
```

awswrangler.timestream.create_database

awswrangler.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** (str | None) – 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** (dict[str, str] | None) – 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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
The Amazon Resource Name that uniquely identifies this database. (ARN)

Examples

Creating a database.

```
>>> import awswrangler as wr
>>> arn = wr.timestream.create_database("MyDatabase")
```

awswrangler.timestream.create_table

awswrangler.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** (dict[str, str] | None) – 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** (dict[str, Any] | None) – Forwarded to botocore requests. e.g. timestream_additional_kwargs={'MagneticStoreWriteProperties': {'EnableMagneticStoreWrites': True}}
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
The Amazon Resource Name that uniquely identifies this database. (ARN)

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** (Session | None) – The default boto3 session will be used if **boto3_session** is 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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
None

Examples

Deleting a table

```
>>> import awswrangler as wr
>>> arn = wr.timestream.delete_table("MyDatabase", "MyTable")
```

awswrangler.timestream.list_databases

awswrangler.timestream.**list_databases**(*boto3_session: Session | None = None*) → list[str]

List all databases in timestream.

Parameters
boto3_session (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
a list of available timestream databases.

Examples

Querying the list of all available databases

```
>>> import awswrangler as wr
>>> wr.timestream.list_databases()
["database1", "database2"]
```

awswrangler.timestream.list_tables

awswrangler.timestream.**list_tables**(*database: str | None = None*, *boto3_session: Session | None = None*) → list[str]

List tables in timestream.

Parameters

- **database** (str | None) – Database name. If None, all tables in Timestream will be returned. Otherwise, only the tables inside the given database are returned.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
A list of table names.

Examples

Listing all tables in timestream across databases

```
>>> import awswrangler as wr
>>> wr.timestream.list_tables()
["table1", "table2"]
```

Listing all tables in timestream in a specific database

```
>>> import awswrangler as wr
>>> wr.timestream.list_tables(DatabaseName="database1")
["table1"]
```

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] | None) – Pagination configuration dictionary of a form {'MaxItems': 10, 'PageSize': 10, 'StartingToken': '...'}
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
DataFrame | Iterator[DataFrame]

Returns
Pandas 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.wait_batch_load_task

awswrangler.timestream.**wait_batch_load_task**(*task_id: str*, *timestream_batch_load_wait_polling_delay: float = 2*, *boto3_session: Session | None = None*) → dict[str, Any]

Wait for the Timestream batch load task to complete.

Parameters

- **task_id** (str) – The ID of the batch load task.
- **timestream_batch_load_wait_polling_delay** (float) – Time to wait between two polling attempts.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

Returns
Dictionary with the describe_batch_load_task response.

Examples

```
>>> import awswrangler as wr
>>> res = wr.timestream.wait_batch_load_task(task_id='task-id')
```

Raises
exceptions.TimestreamLoadError – Error message raised by failed task.

awswrangler.timestream.write

awswrangler.timestream.**write**(*df: DataFrame*, *database: str*, *table: str*, *time_col: str | None = None*, *measure_col: str | list[str | None] | None = None*, *dimensions_cols: list[str | None] | None = None*, *version: int = 1*, *time_unit: Literal['MILLISECONDS', 'SECONDS', 'MICROSECONDS', 'NANOSECONDS'] = 'MILLISECONDS'*, *use_threads: bool | int = True*, *measure_name: str | None = None*, *common_attributes: dict[str, Any] | None = None*, *boto3_session: Session | None = None*) → list[dict[str, str]]

Store a Pandas DataFrame into an Amazon Timestream table.

Note: In case *use_threads=True*, the number of threads from os.cpu_count() is used.

If the Timestream service rejects a record(s), this function will not throw a Python exception. Instead it will return the rejection information.

Note: If `time_col` column is supplied, it must be of type timestamp. `time_unit` is set to MILLISECONDS by default. NANOSECONDS is not supported as python datetime objects are limited to microseconds precision.

Parameters

- **df** (`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 | None`) – DataFrame column name to be used as time. MUST be a timestamp column.
- **measure_col** (`str | list[str | None] | None`) – DataFrame column name(s) to be used as measure.
- **dimensions_cols** (`list[str | None] | None`) – 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.
- **time_unit** (`Literal['MILLISECONDS', 'SECONDS', 'MICROSECONDS', 'NANOSECONDS']`) – Time unit for the time column. MILLISECONDS by default.
- **use_threads** (`bool | int`) – True to enable concurrent writing, False to disable multiple threads. If enabled, os.cpu_count() is used as the number of threads. If integer is provided, specified number is used.
- **measure_name** (`str | None`) – Name that represents the data attribute of the time series. Overrides `measure_col` if specified.
- **common_attributes** (`dict[str, Any] | None`) – Dictionary of attributes shared across all records in the request. Using common attributes can optimize the cost of writes by reducing the size of request payloads. Values in `common_attributes` take precedence over all other arguments and data frame values. Dimension attributes are merged with attributes in record objects. Example: `{"Dimensions": [{"Name": "device_id", "Value": "12345"}], "MeasureValueType": "DOUBLE"}`.

- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, str]]

Returns
Rejected records. Possible reasons for rejection are described here: https://docs.aws.amazon.com/timestream/latest/developerguide/API_RejectedRecord.html

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
>>>     }
>>> ]
```

awswrangler.timestream.unload_to_files

awswrangler.timestream.**unload_to_files**(*sql: str*, *path: str*, *unload_format: Literal['CSV', 'PARQUET'] | None = None*, *compression: Literal['GZIP', 'NONE'] | None = None*, *partition_cols: list[str] | None = None*, *encryption: Literal['SSE_KMS', 'SSE_S3'] | None = None*, *kms_key_id: str | None = None*, *field_delimiter: str | None = ','*, *escaped_by: str | None = '\\'*, *boto3_session: Session | None = None*) → None

Unload query results to Amazon S3.

https://docs.aws.amazon.com/timestream/latest/developerguide/export-unload.html

Parameters

- **sql** (str) – SQL query
- **path** (str) – S3 path to write stage files (e.g. s3://bucket_name/any_name/)
- **unload_format** (Literal['CSV', 'PARQUET'] | None) – Format of the unloaded S3 objects from the query. Valid values: "CSV", "PARQUET". Case sensitive. Defaults to "PARQUET"
- **compression** (Literal['GZIP', 'NONE'] | None) – Compression of the unloaded S3 objects from the query. Valid values: "GZIP", "NONE". Defaults to "GZIP"
- **partition_cols** (list[str] | None) – Specifies the partition keys for the unload operation
- **encryption** (Literal['SSE_KMS', 'SSE_S3'] | None) – Encryption of the unloaded S3 objects from the query. Valid values: "SSE_KMS", "SSE_S3". Defaults to "SSE_S3"
- **kms_key_id** (str | None) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3
- **field_delimiter** (str | None) – A single ASCII character that is used to separate fields in the output file, such as pipe character (|), a comma (,), or tab (/t). Only used with CSV format
- **escaped_by** (str | None) – The character that should be treated as an escape character in the data file written to S3 bucket. Only used with CSV format
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type

None

Examples

Unload and read as Parquet (default).

```
>>> import awswrangler as wr
>>> wr.timestream.unload_to_files(
...     sql="SELECT time, measure, dimension FROM database.mytable",
...     path="s3://bucket/extracted_parquet_files/",
... )
```

Unload and read partitioned Parquet. Note: partition columns must be at the end of the table.

```
>>> import awswrangler as wr
>>> wr.timestream.unload_to_files(
...     sql="SELECT time, measure, dim1, dim2 FROM database.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     partition_cols=["dim2"],
... )
```

Unload and read as CSV.

```
>>> import awswrangler as wr
>>> wr.timestream.unload_to_files(
...     sql="SELECT time, measure, dimension FROM database.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     unload_format="CSV",
... )
```

awswrangler.timestream.unload

awswrangler.timestream.**unload**(*sql: str*, *path: str*, *unload_format: Literal['CSV', 'PARQUET'] | None = None*, *compression: Literal['GZIP', 'NONE'] | None = None*, *partition_cols: list[str] | None = None*, *encryption: Literal['SSE_KMS', 'SSE_S3'] | None = None*, *kms_key_id: str | None = None*, *field_delimiter: str | None = ','*, *escaped_by: str | 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*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*) → DataFrame | Iterator[DataFrame]

Unload query results to Amazon S3 and read the results as Pandas Data Frame.

https://docs.aws.amazon.com/timestream/latest/developerguide/export-unload.html

Parameters

- **sql** (str) – SQL query
- **path** (str) – S3 path to write stage files (e.g. s3://bucket_name/any_name/)
- **unload_format** (Literal['CSV', 'PARQUET'] | None) – Format of the unloaded S3 objects from the query. Valid values: “CSV”, “PARQUET”. Case sensitive. Defaults to “PARQUET”
- **compression** (Literal['GZIP', 'NONE'] | None) – Compression of the unloaded S3 objects from the query. Valid values: “GZIP”, “NONE”. Defaults to “GZIP”
- **partition_cols** (list[str] | None) – Specifies the partition keys for the unload operation
- **encryption** (Literal['SSE_KMS', 'SSE_S3'] | None) – Encryption of the unloaded S3 objects from the query. Valid values: “SSE_KMS”, “SSE_S3”. Defaults to “SSE_S3”
- **kms_key_id** (str | None) – Specifies the key ID for an AWS Key Management Service (AWS KMS) key to be used to encrypt data files on Amazon S3
- **field_delimiter** (str | None) – A single ASCII character that is used to separate fields in the output file, such as pipe character (|), a comma (,), or tab (/t). Only used with CSV format
- **escaped_by** (str | None) – The character that should be treated as an escape character in the data file written to S3 bucket. Only used with CSV format

- **chunked** (bool | int) – 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.
- **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.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **s3_additional_kwargs** (dict[str, str] | None) – Forward to botocore requests.
- **pyarrow_additional_kwargs** (dict[str, Any] | None) – Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}.

Return type
DataFrame | Iterator[DataFrame]

Returns
Result as Pandas DataFrame(s).

Examples

Unload and read as Parquet (default).

```
>>> import awswrangler as wr
>>> df = wr.timestream.unload(
...     sql="SELECT time, measure, dimension FROM database.mytable",
...     path="s3://bucket/extracted_parquet_files/",
... )
```

Unload and read partitioned Parquet. Note: partition columns must be at the end of the table.

```
>>> import awswrangler as wr
>>> df = wr.timestream.unload(
...     sql="SELECT time, measure, dim1, dim2 FROM database.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     partition_cols=["dim2"],
... )
```

Unload and read as CSV.

```
>>> import awswrangler as wr
>>> df = wr.timestream.unload(
...     sql="SELECT time, measure, dimension FROM database.mytable",
...     path="s3://bucket/extracted_parquet_files/",
...     unload_format="CSV",
... )
```

1.6.16 AWS Clean Rooms

read_sql_query([sql, analysis_template_arn, ...])	Execute Clean Rooms Protected SQL query and return the results as a Pandas DataFrame.
wait_query(membership_id, query_id[, ...])	Wait for the Clean Rooms protected query to end.

awswrangler.cleanrooms.read_sql_query

awswrangler.cleanrooms.**read_sql_query**(*sql: str | None = None*, *analysis_template_arn: str | None = None*, *membership_id: str = ''*, *output_bucket: str = ''*, *output_prefix: str = ''*, *keep_files: bool = True*, *params: dict[str, Any] | None = None*, *chunksize: int | bool | None = None*, *use_threads: bool | int = True*, *boto3_session: Session | None = None*, *pyarrow_additional_kwargs: dict[str, Any] | None = None*) → Iterator[DataFrame] | DataFrame

Execute Clean Rooms Protected SQL query and return the results as a Pandas DataFrame.

Note: One of *sql* or *analysis_template_arn* must be supplied, not both.

Parameters

- **sql** (str | None) – SQL query
- **analysis_template_arn** (str | None) – ARN of the analysis template
- **membership_id** (str) – Membership ID
- **output_bucket** (str) – S3 output bucket name
- **output_prefix** (str) – S3 output prefix
- **keep_files** (bool) – Whether files in S3 output bucket/prefix are retained. 'True' by default
- **params** (dict[str, Any] | None) – (Client-side) If used in combination with the *sql* parameter, it's the Dict of parameters used for constructing the SQL query. Only named parameters are supported. The dict must be in the form {'name': 'value'} and the SQL query must contain *:name*. Note that for varchar columns and similar, you must surround the value in single quotes.

  (Server-side) If used in combination with the *analysis_template_arn* parameter, it's the Dict of parameters supplied with the analysis template. It must be a string to string dict in the form {'name': 'value'}.
- **chunksize** (int | bool | None) – If passed, the data is split into an iterable of DataFrames (Memory friendly). If *True* an iterable of DataFrames is returned without guarantee of chunksize. If an *INTEGER* is passed, an iterable of DataFrames is returned with maximum rows equal to the received INTEGER
- **use_threads** (bool | int) – True to enable concurrent requests, False to disable multiple threads. If enabled os.cpu_count() is used as the maximum number of threads. If integer is provided, specified number is used
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

- **pyarrow_additional_kwargs** (dict[str, Any] | None) – Forwarded to *to_pandas* method converting from PyArrow tables to Pandas DataFrame. Valid values include "split_blocks", "self_destruct", "ignore_metadata". e.g. pyarrow_additional_kwargs={'split_blocks': True}

Return type
Iterator[DataFrame] | DataFrame

Returns
Pandas DataFrame or Generator of Pandas DataFrames if chunksize is provided.

Examples

```
>>> import awswrangler as wr
>>> df = wr.cleanrooms.read_sql_query(
...     sql='SELECT DISTINCT...',
...     membership_id='membership-id',
...     output_bucket='output-bucket',
...     output_prefix='output-prefix',
... )
```

```
>>> import awswrangler as wr
>>> df = wr.cleanrooms.read_sql_query(
...     analysis_template_arn='arn:aws:cleanrooms:...',
...     params={'param1': 'value1'},
...     membership_id='membership-id',
...     output_bucket='output-bucket',
...     output_prefix='output-prefix',
... )
```

awswrangler.cleanrooms.wait_query

awswrangler.cleanrooms.**wait_query**(*membership_id: str*, *query_id: str*, *boto3_session: boto3.Session | None = None*) → GetProtectedQueryOutputTypeDef

Wait for the Clean Rooms protected query to end.

Parameters

- **membership_id** (str) – Membership ID
- **query_id** (str) – Protected query execution ID
- **boto3_session** (boto3.Session | None) – The default boto3 session will be used if **boto3_session** is None.

Returns
Dictionary with the get_protected_query response.

Return type
Dict[str, Any]

Raises
exceptions.QueryFailed – Raises exception with error message if protected query is cancelled, times out or fails.

Examples

```
>>> import awswrangler as wr
>>> res = wr.cleanrooms.wait_query(membership_id='membership-id', query_id='query-id
↪')
```

1.6.17 Amazon EMR

build_spark_step(path[, args, deploy_mode, ...])	Build the Step structure (dictionary).
build_step(command[, name, ...])	Build the Step structure (dictionary).
create_cluster(subnet_id[, cluster_name, ...])	Create a EMR cluster with instance fleets configuration.
get_cluster_state(cluster_id[, boto3_session])	Get the EMR cluster state.
get_step_state(cluster_id, step_id[, ...])	Get EMR step state.
submit_ecr_credentials_refresh(cluster_id, path)	Update internal ECR credentials.
submit_spark_step(cluster_id, path[, args, ...])	Submit Spark Step.
submit_step(cluster_id, command[, name, ...])	Submit new job in the EMR Cluster.
submit_steps(cluster_id, steps[, boto3_session])	Submit a list of steps.
terminate_cluster(cluster_id[, boto3_session])	Terminate EMR cluster.

awswrangler.emr.build_spark_step

awswrangler.emr.**build_spark_step**(*path: str*, *args: list[str] | None = None*, *deploy_mode: Literal['cluster', 'client'] = 'cluster'*, *docker_image: str | None = None*, *name: str = 'my-step'*, *action_on_failure: Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'] = '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] | None) – CLI args to use with script
- **deploy_mode** (Literal['cluster', 'client']) – "cluster" | "client"
- **docker_image** (str | None) – e.g. "{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{IMAGE_NAME}:{TA
- **name** (str) – Step name.
- **action_on_failure** (Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE']) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **region** (str | None) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type

dict[str, Any]

Returns
Step structure.

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: Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'] = '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) – Step name.
- **action_on_failure** (Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE']) – '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 | None) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

Returns
Step structure.

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)
```

awswrangler.emr.create_cluster

awswrangler.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 | None) – 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] | None) – Bootstraps paths (e.g ["s3://BUCKET_NAME/script.sh"]).
- **debugging** (bool) – Debugging enabled?
- **applications** (list[str] | None) – 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 | None) – Key pair name.
- **security_group_master** (str | None) – The identifier of the Amazon EC2 security group for the master node.
- **security_groups_master_additional** (list[str] | None) – A list of additional Amazon EC2 security group IDs for the master node.
- **security_group_slave** (str | None) – The identifier of the Amazon EC2 security group for the core and task nodes.
- **security_groups_slave_additional** (list[str] | None) – A list of additional Amazon EC2 security group IDs for the core and task nodes.
- **security_group_service_access** (str | None) – 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] | None) – Additional docker registries.
- **spark_log_level** (str) – log4j.rootCategory log level (ALL, DEBUG, INFO, WARN, ERROR, FATAL, OFF, TRACE).
- **spark_jars_path** (list[str] | None) – spark.jars e.g. [s3://.../foo.jar, s3://.../boo.jar] https://spark.apache.org/docs/latest/configuration.html

- **spark_defaults** (dict[str, str] | None) – 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]] | None) – 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** (str | None) – The custom AMI ID to use for the provisioned instance group
- **steps** (list[dict[str, Any]] | None) – 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** (dict[str, int] | None) – 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] | None) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"})
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **configurations** (list[dict[str, Any]] | None) – 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"}}*

Return type
str

Returns
Cluster ID.

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",
>>>             },
>>>         }
>>>     ],
>>> )
```

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"],
```

(continues on next page)

(continued from previous page)

```
...     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"
...     })
```

awswrangler.emr.get_cluster_state

awswrangler.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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
State.

Examples

```
>>> import awswrangler as wr
>>> state = wr.emr.get_cluster_state("cluster-id")
```

awswrangler.emr.get_step_state

awswrangler.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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
State.

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: Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'] = '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** (Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE']) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Step ID.

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: Literal['cluster', 'client'] = 'cluster'*, *docker_image: str | None = None*, *name: str = 'my-step'*, *action_on_failure: Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'] = '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] | None) – CLI args to use with script eg. args = ["–name", "hello-world"]
- **deploy_mode** (Literal['cluster', 'client']) – "cluster" | "client"
- **docker_image** (str | None) – e.g. "{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{IMAGE_NAME}:{TA
- **name** (str) – Step name.
- **action_on_failure** (Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE']) – 'TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'
- **region** (str | None) – Region name to not get it from boto3.Session. (e.g. *us-east-1*)
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Step ID.

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: Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE'] = '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) – Step name.
- **action_on_failure** (Literal['TERMINATE_JOB_FLOW', 'TERMINATE_CLUSTER', 'CANCEL_AND_WAIT', 'CONTINUE']) – '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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Step ID.

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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
List of step IDs.

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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
None

Examples

```
>>> import awswrangler as wr
>>> wr.emr.terminate_cluster("cluster-id")
```

1.6.18 Amazon EMR Serverless

create_application(name, release_label[, ...])	Create an EMR Serverless application.
run_job(application_id, execution_role_arn, ...)	Run an EMR serverless job.
wait_job(application_id, job_run_id[, ...])	Wait for the EMR Serverless job to finish.

awswrangler.emr_serverless.create_application

awswrangler.emr_serverless.**create_application**(*name: str*, *release_label: str*, *application_type: Literal['Spark', 'Hive'] = 'Spark'*, *initial_capacity: dict[str, str] | None = None*, *maximum_capacity: dict[str, str] | None = None*, *tags: dict[str, str] | None = None*, *autostart: bool = True*, *autostop: bool = True*, *idle_timeout: int = 15*, *network_configuration: dict[str, str] | None = None*, *architecture: Literal['ARM64', 'X86_64'] = 'X86_64'*, *image_uri: str | None = None*, *worker_type_specifications: dict[str, str] | None = None*, *boto3_session: Session | None = None*) → str

Create an EMR Serverless application.

https://docs.aws.amazon.com/emr/latest/EMR-Serverless-UserGuide/emr-serverless.html

Parameters

- **name** (str) – Name of EMR Serverless appliation
- **release_label** (str) – Release label e.g. *emr-6.10.0*
- **application_type** (Literal['Spark', 'Hive']) – Application type: "Spark" or "Hive". Defaults to "Spark".
- **initial_capacity** (dict[str, str] | None) – The capacity to initialize when the application is created.
- **maximum_capacity** (dict[str, str] | None) – The maximum capacity to allocate when the application is created. This is cumulative across all workers at any given point in time, not just when an application is created. No new resources will be created once any one of the defined limits is hit.
- **tags** (dict[str, str] | None) – Key/Value collection to put tags on the application. e.g. {"foo": "boo", "bar": "xoo"})
- **autostart** (bool) – Enables the application to automatically start on job submission. Defaults to true.
- **autostop** (bool) – Enables the application to automatically stop after a certain amount of time being idle. Defaults to true.
- **idle_timeout** (int) – The amount of idle time in minutes after which your application will automatically stop. Defaults to 15 minutes.
- **network_configuration** (dict[str, str] | None) – The network configuration for customer VPC connectivity.
- **architecture** (Literal['ARM64', 'X86_64']) – The CPU architecture of an application: "ARM64" or "X86_64". Defaults to "X86_64".
- **image_uri** (str | None) – The URI of an image in the Amazon ECR registry.
- **worker_type_specifications** (dict[str, str] | None) – The key-value pairs that specify worker type.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Application Id.

awswrangler.emr_serverless.run_job

awswrangler.emr_serverless.**run_job**(*application_id: str*, *execution_role_arn: str*, *job_driver_args: dict[str, Any] | SparkSubmitJobArgs | HiveRunJobArgs*, *job_type: Literal['Spark', 'Hive'] = 'Spark'*, *wait: bool = True*, *configuration_overrides: dict[str, Any] | None = None*, *tags: dict[str, str] | None = None*, *execution_timeout: int | None = None*, *name: str | None = None*, *emr_serverless_job_wait_polling_delay: float = 5*, *boto3_session: Session | None = None*) → str | dict[str, Any]

Run an EMR serverless job.

https://docs.aws.amazon.com/emr/latest/EMR-Serverless-UserGuide/emr-serverless.html

Parameters

- **application_id** (str) – The id of the application on which to run the job.
- **execution_role_arn** (str) – The execution role ARN for the job run.
- **job_driver_args** (dict[str, Any] | SparkSubmitJobArgs | HiveRunJobArgs) – The job driver arguments for the job run.
- **job_type** (Literal['Spark', 'Hive']) – Type of the job: "Spark" or "Hive". Defaults to "Spark".
- **wait** (bool) – Whether to wait for the job completion or not. Defaults to true.
- **configuration_overrides** (dict[str, Any] | None) – The configuration overrides for the job run.
- **tags** (dict[str, str] | None) – Key/Value collection to put tags on the application. e.g. {"foo": "boo", "bar": "xoo"})
- **execution_timeout** (int | None) – The maximum duration for the job run to run. If the job run runs beyond this duration, it will be automatically cancelled.
- **name** (str | None) – Name of the job.
- **emr_serverless_job_wait_polling_delay** (float) – Time to wait between polling attempts.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str | dict[str, Any]

Returns
Job Id if wait=False, or job run details.

awswrangler.emr_serverless.wait_job

awswrangler.emr_serverless.**wait_job**(*application_id: str*, *job_run_id: str*, *emr_serverless_job_wait_polling_delay: float = 5*, *boto3_session: Session | None = None*) → dict[str, Any]

Wait for the EMR Serverless job to finish.

https://docs.aws.amazon.com/emr/latest/EMR-Serverless-UserGuide/emr-serverless.html

Parameters

- **application_id** (str) – The id of the application on which the job is running.
- **job_run_id** (str) – The id of the job.
- **emr_serverless_job_wait_polling_delay** (float) – Time to wait between polling attempts.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

Returns
Job run details.

1.6.19 Amazon CloudWatch Logs

read_logs(query, log_group_names[, ...])	Run a query against AWS CloudWatchLogs Insights and convert the results to Pandas DataFrame.
run_query(query, log_group_names[, ...])	Run a query against AWS CloudWatchLogs Insights and wait the results.
start_query(query, log_group_names[, ...])	Run a query against AWS CloudWatchLogs Insights.
wait_query(query_id[, boto3_session, ...])	Wait query ends.
describe_log_streams(log_group_name[, ...])	List the log streams for the specified log group, return results as a Pandas DataFrame.
filter_log_events(log_group_name[, ...])	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 | None = None*, *end_time: datetime | None = None*, *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** (list[str]) – The list of log group names or ARNs to be queried. You can include up to 50 log groups.
- **start_time** (datetime | None) – The beginning of the time range to query.
- **end_time** (datetime | None) – The end of the time range to query.
- **limit** (int | None) – The maximum number of log events to return in the query.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
DataFrame

Returns
Result as a 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 | None = None*, *end_time: datetime | None = None*, *limit: int | None = None*, *boto3_session: Session | None = None*) → list[list[dict[str, str]]]

Run a query against AWS CloudWatchLogs 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** (list[str]) – The list of log group names or ARNs to be queried. You can include up to 50 log groups.
- **start_time** (datetime | None) – The beginning of the time range to query.
- **end_time** (datetime | None) – The end of the time range to query.
- **limit** (int | None) – The maximum number of log events to return in the query.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[list[dict[str, str]]]

Returns
Result.

Examples

```
>>> import awswrangler as wr
>>> result = wr.cloudwatch.run_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
```

awswrangler.cloudwatch.start_query

awswrangler.cloudwatch.**start_query**(*query: str*, *log_group_names: list[str]*, *start_time: datetime | None = None*, *end_time: datetime | None = None*, *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** (list[str]) – The list of log group names or ARNs to be queried. You can include up to 50 log groups.
- **start_time** (datetime | None) – The beginning of the time range to query.
- **end_time** (datetime | None) – The end of the time range to query.
- **limit** (int | None) – The maximum number of log events to return in the query.

- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Query ID.

Examples

```
>>> import awswrangler as wr
>>> query_id = wr.cloudwatch.start_query(
...     log_group_names=["loggroup"],
...     query="fields @timestamp, @message | sort @timestamp desc | limit 5",
... )
```

awswrangler.cloudwatch.wait_query

awswrangler.cloudwatch.**wait_query**(*query_id: str*, *boto3_session: Session | None = None*, *cloudwatch_query_wait_polling_delay: float = 1.0*) → dict[str, Any]

Wait query ends.

https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html

Parameters

- **query_id** (str) – Query ID.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **cloudwatch_query_wait_polling_delay** (float) – Interval in seconds for how often the function will check if the CloudWatch query has completed.

Return type
dict[str, Any]

Returns
Query result payload.

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 | None) – The prefix to match log streams' name
- **order_by** (str | None) – 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 | None) – 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** (int | None) – The maximum number of items returned. The default is up to 50 items.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
DataFrame

Returns
Result as a 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 | None`) – Filters the results to include only events from log streams that have names starting with this prefix.
- **log_stream_names** (`list[str] | None`) – Filters the results to only logs from the log streams in this list.
- **filter_pattern** (`str | None`) – The filter pattern to use. If not provided, all the events are matched.
- **start_time** (`datetime | None`) – Events with a timestamp before this time are not returned.
- **end_time** (`datetime | None`) – Events with a timestamp later than this time are not returned.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `None`.

Return type
`DataFrame`

Returns
Result as a 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",
... )
```

1.6.20 Amazon QuickSight

cancel_ingestion(ingestion_id[, ...])	Cancel an ongoing ingestion of data into SPICE.
create_athena_data_source(name[, workgroup, ...])	Create a QuickSight data source pointing to an Athena/Workgroup.
create_athena_dataset(name[, database, ...])	Create a QuickSight dataset.
create_ingestion([dataset_name, dataset_id, ...])	Create and starts a new SPICE ingestion on a dataset.
delete_all_dashboards([account_id, ...])	Delete all dashboards.
delete_all_data_sources([account_id, ...])	Delete all data sources.
delete_all_datasets([account_id, ...])	Delete all datasets.
delete_all_templates([account_id, ...])	Delete all templates.
delete_dashboard([name, dashboard_id, ...])	Delete a dashboard.
delete_data_source([name, data_source_id, ...])	Delete a data source.
delete_dataset([name, dataset_id, ...])	Delete a dataset.
delete_template([name, template_id, ...])	Delete a template.
describe_dashboard([name, dashboard_id, ...])	Describe a QuickSight dashboard by name or ID.
describe_data_source([name, data_source_id, ...])	Describe a QuickSight data source by name or ID.
describe_data_source_permissions([name, ...])	Describe a QuickSight data source permissions by name or ID.
describe_dataset([name, dataset_id, ...])	Describe a QuickSight dataset by name or ID.
describe_ingestion(ingestion_id[, ...])	Describe a QuickSight ingestion by ID.
get_dashboard_id(name[, account_id, ...])	Get QuickSight dashboard ID given a name and fails if there is more than 1 ID associated with this name.
get_dashboard_ids(name[, account_id, ...])	Get QuickSight dashboard IDs given a name.
get_data_source_arn(name[, account_id, ...])	Get QuickSight data source ARN given a name and fails if there is more than 1 ARN associated with this name.
get_data_source_arns(name[, account_id, ...])	Get QuickSight Data source ARNs given a name.
get_data_source_id(name[, account_id, ...])	Get QuickSight data source ID given a name and fails if there is more than 1 ID associated with this name.
get_data_source_ids(name[, account_id, ...])	Get QuickSight data source IDs given a name.
get_dataset_id(name[, account_id, boto3_session])	Get QuickSight Dataset ID given a name and fails if there is more than 1 ID associated with this name.
get_dataset_ids(name[, account_id, ...])	Get QuickSight dataset IDs given a name.
get_template_id(name[, account_id, ...])	Get QuickSight template ID given a name and fails if there is more than 1 ID associated with this name.
get_template_ids(name[, account_id, ...])	Get QuickSight template IDs given a name.
list_dashboards([account_id, boto3_session])	List dashboards in an AWS account.
list_data_sources([account_id, boto3_session])	List all QuickSight Data sources summaries.
list_datasets([account_id, boto3_session])	List all QuickSight datasets summaries.
list_groups([namespace, account_id, ...])	List all QuickSight Groups.
list_group_memberships(group_name[, ...])	List all QuickSight Group memberships.
list_iam_policy_assignments([status, ...])	List IAM policy assignments in the current Amazon QuickSight account.
list_iam_policy_assignments_for_user(user_nan	List all the IAM policy assignments.
list_ingestions([dataset_name, dataset_id, ...])	List the history of SPICE ingestions for a dataset.
list_templates([account_id, boto3_session])	List all QuickSight templates.
list_users([namespace, account_id, ...])	Return a list of all of the Amazon QuickSight users belonging to this account.
list_user_groups(user_name[, namespace, ...])	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 | None) – Dataset name.
- **dataset_id** (str | None) – Dataset ID.
- **account_id** (str | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is 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] | _QuicksightPrincipalList | None = None*, *allowed_to_manage: List[str] | _QuicksightPrincipalList | 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] | None) – Key/Value collection to put on the Cluster. e.g. `{"foo": "boo", "bar": "xoo"})`

- **allowed_to_use** (List[str] | _QuicksightPrincipalList | None) – Dictionary containing usernames and groups that will be allowed to see and use the data. e.g. `{"users": ["john", "Mary"], "groups": ["engineering", "customers"]}` Alternatively, if a list of string is passed, it will be interpreted as a list of usernames only.
- **allowed_to_manage** (List[str] | _QuicksightPrincipalList | None) – Dictionary containing usernames and groups that will be allowed to see, use, update and delete the data source. e.g. `{"users": ["Mary"], "groups": ["engineering"]}` Alternatively, if a list of string is passed, it will be interpreted as a list of usernames only.
- **account_id** (str | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.
- **namespace** (str) – The namespace. Currently, you should set this to default.

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 | None = None*, *data_source_name: str | None = None*, *data_source_arn: str | None = None*, *import_mode: Literal['SPICE', 'DIRECT_QUERY'] = 'DIRECT_QUERY'*, *allowed_to_use: List[str] | _QuicksightPrincipalList | None = None*, *allowed_to_manage: List[str] | _QuicksightPrincipalList | 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 | None`) – Athena's database name.
- **table** (`str | None`) – Athena's table name.
- **sql** (`str | None`) – Use a SQL query to define your table.
- **sql_name** (`str | None`) – Query name.
- **data_source_name** (`str | None`) – QuickSight data source name.
- **data_source_arn** (`str | None`) – QuickSight data source ARN.
- **import_mode** (`Literal['SPICE', 'DIRECT_QUERY']`) – Indicates whether you want to import the data into SPICE.
- **tags** (`dict[str, str] | None`) – Key/Value collection to put on the Cluster. e.g. {"foo": "boo", "bar": "xoo"}
- **allowed_to_use** (`List[str] | _QuicksightPrincipalList | None`) – Dictionary containing usernames and groups that will be allowed to see and use the data. e.g. `` `{"users": ["john", "Mary"], "groups": ["engineering", "customers"]}` `` Alternatively, if a list of string is passed, it will be interpreted as a list of usernames only.
- **allowed_to_manage** (`List[str] | _QuicksightPrincipalList | None`) – Dictionary containing usernames and groups that will be allowed to see, use, update and delete the data source. e.g. `` `{"users": ["Mary"], "groups": ["engineering"]}` `` Alternatively, if a list of string is passed, it will be interpreted as a list of usernames only.
- **logical_table_alias** (`str`) – A display name for the logical table.
- **rename_columns** (`dict[str, str] | None`) – Dictionary to map column renames. e.g. {"old_name": "new_name", "old_name2": "new_name2"}
- **cast_columns_types** (`dict[str, str] | None`) – 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]]] | None`) – 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 | None`) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `None`.
- **namespace** (`str`) – The namespace. Currently, you should set this to default.

Return type
`str`

Returns
Dataset ID.

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 | None) – Dataset name.
- **dataset_id** (str | None) – Dataset ID.
- **ingestion_id** (str | None) – Ingestion ID.
- **account_id** (str | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Ingestion ID

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **regex_filter** (str | None) – Regex regex_filter that will delete all dashboards with a match in their Name
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is 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 | None) – If None, the account ID will be inferred from your boto3 session.
- **regex_filter** (str | None) – Regex regex_filter that will delete all data sources with a match in their Name
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is 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 | None) – If None, the account ID will be inferred from your boto3 session.
- **regex_filter** (str | None) – Regex regex_filter that will delete all datasets with a match in their Name
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is 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 | None) – If None, the account ID will be inferred from your boto3 session.
- **regex_filter** (str | None) – Regex regex_filter that will delete all templates with a match in their Name
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is 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 | None`) – Dashboard name.
- **dashboard_id** (`str | None`) – The ID for the dashboard.
- **version_number** (`int | None`) – 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 | None`) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `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 | None`) – Dashboard name.
- **data_source_id** (`str | None`) – The ID for the data source.
- **account_id** (`str | None`) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `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 | None`) – Dashboard name.
- **dataset_id** (`str | None`) – The ID for the dataset.
- **account_id** (`str | None`) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `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 | None`) – Dashboard name.
- **template_id** (`str | None`) – The ID for the dashboard.
- **version_number** (`int | None`) – 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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is 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 | None) – Dashboard name.
- **dashboard_id** (str | None) – Dashboard ID.
- **account_id** (str | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

Returns
Dashboard Description.

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 | None`) – Data source name.
- **data_source_id** (`str | None`) – Data source ID.
- **account_id** (`str | None`) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `None`.

Return type
`dict[str, Any]`

Returns
Data source Description.

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 | None`) – Data source name.
- **data_source_id** (`str | None`) – Data source ID.
- **account_id** (`str | None`) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `None`.

Return type
`dict[str, Any]`

Returns
Data source Permissions Description.

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 | None`) – Dataset name.
- **dataset_id** (`str | None`) – Dataset ID.
- **account_id** (`str | None`) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (`Session | None`) – The default boto3 session will be used if **boto3_session** is `None`.

Return type
`dict[str, Any]`

Returns
Dataset Description.

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 | None) – Dataset name.
- **dataset_id** (str | None) – Dataset ID.
- **account_id** (str | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

Returns
Ingestion Description.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Dashboard ID.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
Dashboard IDs.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Data source ARN.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
Data source ARNs.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Dataset ID.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
Data source IDs.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Dataset ID.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
Datasets IDs.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Template ID.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[str]

Returns
Template IDs.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
Dashboards.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
Data sources summaries.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
Datasets summaries.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
Groups.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
Group memberships.

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 | None) – The status of the assignments. 'ENABLED'|'DRAFT'|'DISABLED'
- **namespace** (str) – The namespace. Currently, you should set this to default .
- **account_id** (str | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
IAM policy assignments.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
IAM policy assignments.

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 | None) – Dataset name.
- **dataset_id** (str | None) – The ID of the dataset used in the ingestion.
- **account_id** (str | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
IAM policy assignments.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
Templates summaries.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
Groups.

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 | None) – If None, the account ID will be inferred from your boto3 session.
- **boto3_session** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
list[dict[str, Any]]

Returns
Groups.

Examples

```
>>> import awswrangler as wr
>>> groups = wr.quicksight.list_user_groups()
```

1.6.21 AWS STS

get_account_id([boto3_session])	Get Account ID.
get_current_identity_arn([boto3_session])	Get current user/role ARN.
get_current_identity_name([boto3_session])	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 (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
Account ID.

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 (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
str

Returns
User/role ARN.

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 (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type

str

Returns

User/role name.

Examples

```
>>> import awswrangler as wr
>>> name = wr.sts.get_current_identity_name()
```

1.6.22 AWS Secrets Manager

get_secret(name[, boto3_session])	Get secret value.
get_secret_json(name[, boto3_session])	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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type

str | bytes

Returns

Secret value.

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** (Session | None) – The default boto3 session will be used if **boto3_session** is None.

Return type
dict[str, Any]

Returns
Secret JSON value parsed as a dictionary.

Examples

```
>>> import awswrangler as wr
>>> value = wr.secretsmanager.get_secret_json("my-secret-with-json-content")
```

1.6.23 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.

Parameters

- **webhook** (str) – Contains all the authentication information to send the message
- **message** (str) – The actual message which needs to be posted on Slack channel

Return type
Any | None

Returns
The response from Chime

1.6.24 Typing

GlueTableSettings	Typed dictionary defining the settings for the Glue table.
AthenaCTASSettings	Typed dictionary defining the settings for using CTAS (Create Table As Statement).
AthenaUNLOADSettings	Typed dictionary defining the settings for using UNLOAD.
AthenaCacheSettings	Typed dictionary defining the settings for using cached Athena results.
AthenaPartitionProjectionSettings	Typed dictionary defining the settings for Athena Partition Projection.
TimestreamBatchLoadReportS3Configuration	Report configuration for a batch load task.
ArrowDecryptionConfiguration	Configuration for Arrow file decrypting.
ArrowEncryptionConfiguration	Configuration for Arrow file encrypting.
RaySettings	Typed dictionary defining the settings for distributing calls using Ray.
RayReadParquetSettings	Typed dictionary defining the settings for distributing reading calls using Ray.
_S3WriteDataReturnValue	Typed dictionary defining the dictionary returned by S3 write functions.
_ReadTableMetadataReturnValue(columns_types, ...)	Named tuple defining the return value of the `read_*_metadata` functions.

GlueTableSettings

class awswrangler.typing.**GlueTableSettings**

Bases: `TypedDict`

Typed dictionary defining the settings for the Glue table.

Attributes

table_type	The type of the Glue Table.
description	Table description
parameters	Key/value pairs to tag the table.
columns_comments	Columns names and the related comments (e.g.
columns_parameters	Columns names and the related parameters (e.g.
regular_partitions	Create regular partitions (Non projected partitions) on Glue Catalog.

Attributes Documentation

table_type: Literal['EXTERNAL_TABLE']

The type of the Glue Table. Set to EXTERNAL_TABLE if None.

description: str

Table description

Type
Glue/Athena catalog

parameters: dict[str, str]

Key/value pairs to tag the table.

Type
Glue/Athena catalog

columns_comments: dict[str, str]

Columns names and the related comments (e.g. {'col0': 'Column 0.', 'col1': 'Column 1.', 'col2': 'Partition.'}).

columns_parameters: dict[str, dict[str, str]]

Columns names and the related parameters (e.g. {'col0': {'par0': 'Param 0', 'par1': 'Param 1'}}).

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.

AthenaCTASSettings

class awswrangler.typing.**AthenaCTASSettings**

Bases: TypedDict

Typed dictionary defining the settings for using CTAS (Create Table As Statement).

Attributes

database	The name of the alternative database where the CTAS temporary table is stored.
temp_table_name	The name of the temporary table and also the directory name on S3 where the CTAS result is stored.
bucketing_info	Tuple consisting of the column names used for bucketing as the first element and the number of buckets as the second element.
compression	Write compression for the temporary table where the CTAS result is stored.

Attributes Documentation

database: str

The name of the alternative database where the CTAS temporary table is stored. If None, the default *database* is used.

temp_table_name: str

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}/"*.

bucketing_info: Tuple[List[str], int]

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

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.

AthenaUNLOADSettings

class awswrangler.typing.**AthenaUNLOADSettings**

Bases: TypedDict

Typed dictionary defining the settings for using UNLOAD.

Attributes

file_format	Specifies the file format of the output.
compression	This option is specific to the ORC and Parquet formats.
field_delimiter	Specifies a single-character field delimiter for files in CSV, TSV, and other text formats.
partitioned_by	A list of columns by which the output is partitioned.

Attributes Documentation

file_format: str

Specifies the file format of the output. Only *PARQUET* is currently supported.

compression: 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

Specifies a single-character field delimiter for files in CSV, TSV, and other text formats.

partitioned_by: list[str]

A list of columns by which the output is partitioned.

AthenaCacheSettings

class awswrangler.typing.**AthenaCacheSettings**

Bases: TypedDict

Typed dictionary defining the settings for using cached Athena results.

Attributes

max_cache_seconds	awswrangler can look up in Athena's history if this table has been read before.
max_cache_query_inspections	Max number of queries that will be inspected from the history to try to find some result to reuse.
max_remote_cache_entries	Max number of queries that will be retrieved from AWS for cache inspection.
max_local_cache_entries	Max number of queries for which metadata will be cached locally.

Attributes Documentation

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.

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.

AthenaPartitionProjectionSettings

class awswrangler.typing.**AthenaPartitionProjectionSettings**

Bases: TypedDict

Typed dictionary defining the settings for Athena Partition Projection.

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

Attributes

projection_types	Dictionary of partitions names and Athena projections types.
projection_ranges	Dictionary of partitions names and Athena projections ranges.
projection_values	Dictionary of partitions names and Athena projections values.
projection_intervals	Dictionary of partitions names and Athena projections intervals.
projection_digits	Dictionary of partitions names and Athena projections digits.
projection_formats	Dictionary of partitions names and Athena projections formats.
projection_storage_location_template	Value which is allows Athena to properly map partition values if the S3 file locations do not follow a typical *.../column=value/...* pattern.

Attributes Documentation

projection_types: dict[str, Literal['enum', 'integer', 'date', 'injected']]

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: 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: 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: 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: 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: 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: 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}/)

TimestreamBatchLoadReportS3Configuration

class awswrangler.typing.**TimestreamBatchLoadReportS3Configuration**

Bases: TypedDict

Report configuration for a batch load task. This contains details about where error reports are stored.

https://docs.aws.amazon.com/timestream/latest/developerguide/API_ReportS3Configuration.html

Attributes

BucketName	The name of the bucket where the error reports are stored.
ObjectKeyPrefix	Optional S3 prefix for the error reports.
Encryption	Optional encryption type for the error reports.
KmsKeyId	Optional KMS key ID for the error reports.

Attributes Documentation

BucketName: str

The name of the bucket where the error reports are stored.

ObjectKeyPrefix: str

Optional S3 prefix for the error reports.

Encryption: Literal['SSE_S3', 'SSE_KMS']

Optional encryption type for the error reports. SSE_S3 by default.

KmsKeyId: str

Optional KMS key ID for the error reports.

ArrowDecryptionConfiguration

class awswrangler.typing.**ArrowDecryptionConfiguration**

Bases: TypedDict

Configuration for Arrow file decrypting.

Attributes

crypto_factory	Crypto factory for encrypting and decrypting columns.
kms_connection_config	Configuration of the connection to the Key Management Service (KMS).

Attributes Documentation

crypto_factory: pyarrow.parquet.encryption.CryptoFactory

Crypto factory for encrypting and decrypting columns. see: https://arrow.apache.org/docs/python/generated/pyarrow.parquet.encryption.CryptoFactory.html

kms_connection_config: pyarrow.parquet.encryption.KmsConnectionConfig

Configuration of the connection to the Key Management Service (KMS). see: https://arrow.apache.org/docs/python/generated/pyarrow.parquet.encryption.KmsClient.html

ArrowEncryptionConfiguration

class awswrangler.typing.**ArrowEncryptionConfiguration**

Bases: TypedDict

Configuration for Arrow file encrypting.

Attributes

crypto_factory	Crypto factory for encrypting and decrypting columns.
kms_connection_config	Configuration of the connection to the Key Management Service (KMS).
encryption_config	Configuration of the encryption, such as which columns to encrypt see: https://arrow.apache.org/docs/python/generated/pyarrow.parquet.encryption.EncryptionConfiguration.html

Attributes Documentation

crypto_factory: pyarrow.parquet.encryption.CryptoFactory

Crypto factory for encrypting and decrypting columns. see: https://arrow.apache.org/docs/python/generated/pyarrow.parquet.encryption.CryptoFactory.html

kms_connection_config: pyarrow.parquet.encryption.KmsConnectionConfig

Configuration of the connection to the Key Management Service (KMS). see: https://arrow.apache.org/docs/python/generated/pyarrow.parquet.encryption.KmsClient.html

encryption_config: pyarrow.parquet.encryption.EncryptionConfiguration

Configuration of the encryption, such as which columns to encrypt see: https://arrow.apache.org/docs/python/generated/pyarrow.parquet.encryption.EncryptionConfiguration.html

RaySettings

class awswrangler.typing.**RaySettings**

Bases: TypedDict

Typed dictionary defining the settings for distributing calls using Ray.

Attributes

parallelism	The requested parallelism of the read.
override_num_blocks	Override the number of output blocks from all read tasks.

Attributes Documentation

parallelism: int

The requested parallelism of the read. Parallelism may be limited by the number of files of the dataset. Auto-detect by default.

override_num_blocks: int

Override the number of output blocks from all read tasks. By default, the number of output blocks is dynamically decided based on input data size and available resources. You shouldn't manually set this value in most cases.

RayReadParquetSettings

class awswrangler.typing.**RayReadParquetSettings**

Bases: *RaySettings*

Typed dictionary defining the settings for distributing reading calls using Ray.

Attributes

parallelism	
override_num_blocks	
bulk_read	True to enable a faster reading of a large number of Parquet files.

Attributes Documentation

parallelism: int

override_num_blocks: int

bulk_read: bool

True to enable a faster reading of a large number of Parquet files. Offers improved performance due to not gathering the file metadata in a single node. The drawback is that it does not offer schema resolution, so it should only be used when the Parquet files are all uniform.

_S3WriteDataReturnValue

class awswrangler.typing.**_S3WriteDataReturnValue**

Bases: TypedDict

Typed dictionary defining the dictionary returned by S3 write functions.

Attributes

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.

Attributes Documentation

paths: list[str]

List of all stored files paths on S3.

partitions_values: dict[str, list[str]]

Dictionary of partitions added with keys as S3 path locations and values as a list of partitions values as str.

_ReadTableMetadataReturnValue

class awswrangler.typing.**_ReadTableMetadataReturnValue**(*columns_types: dict[str, str]*, *partitions_types: dict[str, str] | None*)

Bases: NamedTuple

Named tuple defining the return value of the `read_*_metadata` functions.

Attributes

columns_types	Dictionary containing column names and types.
partitions_types	Dictionary containing partition names and types, if partitioned.

Attributes Documentation

columns_types: dict[str, str]

Dictionary containing column names and types.

partitions_types: dict[str, str] | None

Dictionary containing partition names and types, if partitioned.

1.6.25 Global Configurations

reset()	Reset one or all (if None is received) configuration values.
to_pandas()	Load all configurations on a Pandas DataFrame.

awswrangler.config.reset

config.**reset**() → None

Reset one or all (if None is received) configuration values.

Parameters

item (str | None) – Configuration item name.

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.

Return type

DataFrame

Returns

Configuration DataFrame.

Examples

```
>>> import awswrangler as wr
>>> wr.config.to_pandas()
```

1.6.26 Engine and Memory Format

Engine()	Execution engine configuration class.
MemoryFormat()	Memory format configuration class.

awswrangler._distributed.Engine

class awswrangler._distributed.**Engine**

Execution engine configuration class.

__init__()

Methods

__init__()	
dispatch_func(source_func[, value])	Dispatch a func based on value or the distribution engine and the source function.
dispatch_on_engine(func)	Dispatch on engine function decorator.
get()	Get the configured distribution engine.
get_installed()	Get the installed distribution engine.
initialize([name])	Initialize the distribution engine.
is_initialized([name])	Check if the distribution engine is initialized.
register([name])	Register the distribution engine dispatch methods.
register_func(source_func, destination_func)	Register a func based on the distribution engine and source function.
set(name)	Set the distribution engine.

awswrangler._distributed.MemoryFormat

class awswrangler._distributed.**MemoryFormat**

Memory format configuration class.

__init__()

Methods

__init__()	
get()	Get the configured memory format.
get_installed()	Get the installed memory format.
set(name)	Set the memory format.

1.6.27 Distributed - Ray

initialize_ray([address, redis_password, ...])	Connect to an existing Ray cluster or start one and connect to it.

awswrangler.distributed.ray.initialize_ray

awswrangler.distributed.ray.**initialize_ray**(*address: str | None = None*, *redis_password: str | None = None*, *ignore_reinit_error: bool = True*, *include_dashboard: bool | None = False*, *configure_logging: bool = True*, *log_to_driver: bool = False*, *logging_level: int = 20*, *object_store_memory: int | None = None*, *cpu_count: int | None = None*, *gpu_count: int | None = None*) → None

Connect to an existing Ray cluster or start one and connect to it.

Parameters

- **address** (str | None) – Address of the Ray cluster to connect to, by default None
- **redis_password** (str | None) – Password to the Redis cluster, by default None
- **ignore_reinit_error** (bool) – If true, Ray suppress errors from calling ray.init() twice, by default True
- **include_dashboard** (bool | None) – Boolean flag indicating whether or not to start the Ray dashboard, by default False
- **configure_logging** (bool) – Boolean flag indicating whether or not to enable logging, by default True
- **log_to_driver** (bool) – Boolean flag to enable routing of all worker logs to the driver, by default False
- **logging_level** (int) – Logging level, defaults to logging.INFO. Ignored unless "configure_logging" is True
- **object_store_memory** (int | None) – The amount of memory (in bytes) to start the object store with, by default None
- **cpu_count** (int | None) – Number of CPUs to assign to each raylet, by default None
- **gpu_count** (int | None) – Number of GPUs to assign to each raylet, by default None

Return type

None

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

O

P

Q

R

S

T

U

W